

Hands-On Software Architecture with C# 8 and .NET Core 3

Architecting software solutions using microservices, DevOps, and design patterns for Azure Cloud



Packt

www.packt.com

Gabriel Baptista and Francesco Abbruzzese

WOW! eBook

www.wowebook.org

Hands-On Software Architecture with C# 8 and .NET Core 3

Architecting software solutions using microservices, DevOps,
and design patterns for Azure Cloud

**Gabriel Baptista
Francesco Abbruzzese**

Packt

BIRMINGHAM - MUMBAI

Hands-On Software Architecture with C# 8 and .NET Core 3

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: Kunal Chaudhari
Acquisition Editor: Alok Dhuri
Content Development Editor: Ruvika Rao
Senior Editor: Afshaan Khan
Technical Editor: Ketan Kamble
Copy Editor: Safis Editing
Project Coordinator: Francy Puthiry
Proofreader: Safis Editing
Indexer: Rekha Nair
Production Designer: Jyoti Chauhan

First published: November 2019

Production reference: 1291119

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

ISBN 978-1-78980-093-7

www.packtpub.com



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.

Contributors

About the authors

Gabriel Baptista is a software architect who technically leads a team in the most diverse projects for retail and industry, using a dozen varieties of Microsoft products. He has become a specialist in Azure **Platform-as-a-Service (PaaS)** solutions since designing a **Software-as-a-Service (SaaS)** platform in partnership with Microsoft. Besides all that, he is also a college computing professor who has published many papers and teaches different subjects related to software engineering, development, and architecture. He is also a speaker on *Channel 9*, one of the most prestigious and active community websites for the .NET stack. As well as that, he is a cofounder of a start-up for developing mobile applications, where Scrum, design thinking, and DevOps philosophy are the keys to delivering user needs.

To my incredible kids, Murilo and Heitor, and my dear wife, Denise, who have always allowed me to move forward.

Francesco Abbruzzese is the author of the book *MVC Controls Toolkit*. He has also contributed to the diffusion and evangelization of the Microsoft web stack since the first version of ASP.NET MVC through tutorials, articles, and tools. He writes about .NET and client-side technologies on his blog, *Dot Net Programming*, and in various online magazines. His company, Mvcct Team, implements and offers web applications, AI software, SAS products, tools, and services for web technologies associated with the Microsoft stack. He has moved from AI systems, where he implemented one of the first decision support systems for banks and financial institutions, to the video games arena, with top-10 titles such as Puma Street Soccer.

To my beloved parents, to whom I owe everything.

About the reviewers

Efraim Kyriakidis has almost 20 years of experience in software development. He got his diploma as an electrical and software engineer from Aristotle University of Thessaloniki in Greece. He has used .NET since its beginnings with version 1.0. In his career, he has mainly focused on Microsoft technologies. He is currently employed by Siemens AG in Germany as a senior software engineer.

Fabio Claudio Ferracchiat is a senior consultant and a senior analyst/developer who uses Microsoft technologies. He works for React Consulting. He is a Microsoft Certified Solution Developer for .NET, Microsoft Certified Application Developer for .NET, and Microsoft Certified Professional. He is also a prolific author and technical reviewer. Over the last 10 years, he's written articles for Italian and international magazines and has co-authored more than 10 books on a variety of computer topics.

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.

Table of Contents

Preface	1
<hr/>	
Section 1: Transforming Customer Needs in Real-World Applications	
<hr/>	
Chapter 1: Understanding the Importance of Software Architecture	9
Technical requirements	10
What is software architecture?	10
Creating an Azure account	12
Software development process models	14
Reviewing traditional software development process models	14
Understanding the waterfall model principles	14
Analyzing the incremental model	15
Understanding agile software development process models	16
Getting into the Scrum model	17
Enabling aspects to be gathered to design high-quality software	18
Understanding the requirements gathering process	18
Practicing the elicitation of user needs	19
Analyzing requirements	20
Writing the specifications	21
Reviewing the specification	21
Using design thinking as a helpful tool	22
Understanding the principles of scalability, robustness, security, and performance	23
Some cases where the requirements gathering process impacted system results	24
Case 1 – my website is too slow to open that page!	24
Understanding caching	25
Applying asynchronous programming	25
Dealing with object allocation	25
Getting better database access	26
Case 2 – the user's needs are not properly implemented	27
Case 3 – the usability of the system does not meet user needs	28
Case study – detecting user needs	28
Book case study – introducing World Wild Travel Club	29
Book case study – understanding user needs and system requirements	30
Summary	32
Questions	32
Further reading	33
Chapter 2: Functional and Nonfunctional Requirements	34

Technical requirements	35
How does scalability interact with Azure and .NET Core?	35
Creating a scalable web app in Azure	36
Vertical scaling (Scale up)	39
Horizontal scaling (Scale out)	41
Creating a scalable web app with .NET Core	42
Performance issues that need to be considered when programming in C#	46
String concatenation	47
Exceptions	48
Multithreading environments for better results – do's and don'ts	49
Usability – why inserting data takes too much time	52
Designing fast selection logic	53
Selecting from a huge amount of items	57
The fantastic world of interoperability with .NET Core	58
Creating a service in Linux	60
Book use case – understanding the main types of .NET Core projects	61
Summary	63
Questions	63
Further reading	64
Chapter 3: Documenting Requirements with Azure DevOps	65
Technical requirements	65
Introducing Azure DevOps	65
Organizing your work using Azure DevOps	70
Azure DevOps repository	71
Package feeds	73
Test plans	75
Pipelines	76
Managing system requirements in Azure DevOps	77
Epics work items	78
Features work items	78
Product Backlog items/User Story work items	79
Book use case – presenting use cases in Azure DevOps	80
Summary	84
Questions	85
Further reading	85
Section 2: Architecting Software Solutions in a Cloud-Based Environment	
Chapter 4: Deciding the Best Cloud-Based Solution	88
Technical requirements	88
Different software deployment models	89

Infrastructure as a service and Azure opportunities	89
Security responsibility in IaaS	91
PaaS – a world of opportunities for developers	92
Web apps	93
Azure SQL Server	94
Azure Cognitive Services	96
SaaS – just sign in and get started!	100
Understanding what serverless means	100
Why are hybrid applications so useful in many cases?	101
Use case – a hybrid application	102
Book use case – which is the best cloud platform for this use case?	103
Summary	104
Questions	104
Further reading	105
Chapter 5: Applying a Microservice Architecture to Your Enterprise Application	106
Technical requirements	107
What are microservices?	107
Microservices and the evolution of the concept of modules	109
Microservice design principles	110
Containers and Docker	114
When do microservices help?	116
Layered architectures and microservices	116
When is it worth considering microservice architectures?	119
How does .NET Core deal with microservices?	120
.NET Core communication facilities	121
Resilient task execution	123
Using generic hosts	125
Visual Studio support for Docker	129
Azure and Visual Studio support for microservice orchestration	134
Which tools are needed to manage microservices?	137
Defining your private Docker registry in Azure	137
Azure Service Fabric	139
Step 1: Basic information	140
Step 2: Cluster configuration	141
Step 3: Security configuration	143
Azure Kubernetes Service (AKS)	146
Use case – logging microservices	150
Ensuring message idempotency	154
The Interaction library	157
Implementing the receiving side of communication	159
Implementing service logic	162
Defining the microservice's host	168
Communicating with the service	169

Testing the application	171
Summary	171
Questions	172
Further reading	172
Chapter 6: Interacting with Data in C# - Entity Framework Core	174
Technical requirements	175
Understanding ORM basics	175
Configuring Entity Framework Core	178
Defining DB entities	179
Defining the mapped collections	182
Completing the mapping configuration	183
Entity Framework Core migrations	184
Understanding stored procedures and direct SQL commands	188
Querying and updating data with Entity Framework Core	189
Returning data to the presentation layer	192
Issuing direct SQL commands	193
Handling transactions	195
Deploying your data layer	195
Understanding Entity Framework Core advanced feature – global filters	196
Summary	197
Questions	198
Further reading	198
Chapter 7: How to Choose Your Data Storage in the Cloud	199
Technical requirements	200
Understanding the different repositories for different purposes	200
Relational databases	201
NoSQL databases	203
Redis	204
Disk memory	206
Choosing between structured or NoSQL storage	206
Azure Cosmos DB – an opportunity to manage a multi-continental database	208
Cosmos DB client	215
Cosmos DB Entity Framework Core provider	216
Use case – storing data	217
Implementing the destinations/packages database with Cosmos DB	218
Summary	223
Questions	223
Further reading	224
Chapter 8: Working with Azure Functions	225
Technical requirements	225

Understanding the Azure Functions App	226
Consumption Plan	227
App Service Plan	227
Programming Azure Functions using C#	228
Listing Azure Functions templates	232
Maintaining Azure Functions	233
Use case – implementing Azure Functions to send emails	235
First Step – creating Azure Queue Storage	237
Summary	243
Questions	243
Further reading	244
Section 3: Applying Design Principles for Software Delivered in the 21st Century	
Chapter 9: Design Patterns and .NET Core Implementation	247
Technical requirements	247
Understanding design patterns and their purpose	248
Builder pattern	249
Factory pattern	251
Singleton pattern	252
Proxy pattern	255
Command pattern	257
Publisher/Subscriber pattern	259
Dependency Injection pattern	260
Understanding the available design patterns in .NET Core	262
Summary	263
Questions	263
Further reading	264
Chapter 10: Understanding the Different Domains in Software Solutions	265
Technical requirements	266
What are software domains?	266
Understanding domain-driven design	269
Entities and value objects	273
Using SOLID principles to map your domains	277
Aggregates	279
The repository and Unit of Work patterns	280
DDD entities and Entity Framework Core	282
Command Query Responsibility Segregation (CQRS) pattern	283
Command handlers and domain events	286
Event sourcing	289
Use case – understanding the domains of the use case	290
Summary	293

Questions	294
Further reading	294
Chapter 11: Implementing Code Reusability in C# 8	295
Technical requirements	295
Understanding the principles of code reusability	296
What is not code reuse?	296
What is code reuse?	298
Inserting reusability into your development cycle	299
Using .NET Standard for code reuse	300
Creating a .NET Standard library	300
How does C# deal with code reuse?	302
Object-oriented analysis	302
Generics	304
Use case – reusing code as a fast track to deliver good and safe software	305
Summary	306
Questions	307
Further reading	307
Chapter 12: Applying Service-Oriented Architectures with .NET Core	308
Technical requirements	309
Understanding the principles of the SOA approach	309
SOAP web services	313
REST web services	315
The OpenAPI standard	321
REST services authorization and authentication	321
How does .NET Core deal with SOA?	324
A short introduction to ASP.NET Core	326
Implementing REST services with ASP.NET Core	330
ASP.NET Core service authorization	334
ASP.NET Core support for OpenAPI	337
.NET Core HTTP clients	341
Use case – exposing WWTravelClub packages	344
Summary	350
Questions	350
Further reading	351
Chapter 13: Presenting ASP.NET Core MVC	352
Technical requirements	353
Understanding the presentation layers of web applications	353
Understanding the ASP.NET Core MVC structure	354
How ASP.NET Core pipeline works	354
Loading configuration data and using it with the options framework	358
Defining the ASP.NET Core MVC pipeline	363

Defining controllers and ViewModels	368
Understanding Razor Views	373
Learning Razor flow of control statements	374
Understanding Razor View properties	376
Using Razor tag helpers	377
Reusing view code	381
What is new in .NET Core 3.0 for ASP.NET Core?	385
Understanding the connection between ASP.NET Core MVC and design principles	387
Advantages of the ASP.NET Core pipeline	388
Server-side and client-side validation	388
ASP.NET Core globalization	389
The MVC pattern	394
Use case – implementing a web app in ASP.NET Core MVC	395
Defining application specifications	395
Defining the application architecture	396
Defining the domain layer	399
Defining the data layer	402
Defining the application layer	407
Controllers and views	412
Summary	418
Questions	418
Further reading	419
Section 4: Programming Solutions for an Unavoidable Future Evolution	
Chapter 14: Best Practices in Coding C# 8	421
Technical requirements	421
The more complex your code is, the worse a programmer you are	422
Maintainability index	423
Cyclomatic complexity	423
Depth of inheritance	427
Class coupling	428
Lines of code	430
Using a version control system	430
Dealing with version control systems in teams	431
Writing safe code in C#	431
try-catch	431
try-finally and using	432
The IDisposable interface	434
.NET Core tips and tricks for coding	434
WWTravelClub – dos and don'ts in writing code	436
Summary	437
Questions	437

Further reading	437
Chapter 15: Testing Your Code with Unit Test Cases and TDD	439
Technical requirements	440
Understanding automated tests	440
Writing automated (unit and integration) tests	442
Writing acceptance and performance tests	444
Understanding test-driven development (TDD)	445
Defining C# test projects	448
Using the xUnit test framework	449
Advanced test preparation and tear-down scenarios	451
Mocking interfaces with Moq	453
Use case – automating unit tests in DevOps Azure	455
Summary	463
Questions	464
Further reading	464
Chapter 16: Using Tools to Write Better Code	465
Technical requirements	466
Identifying a well-written code	466
Understanding and applying tools that can evaluate C# code	468
Applying extension tools to analyze code	472
Using Microsoft Code Analysis 2019	472
Applying SonarLint for Visual Studio 2019	474
Getting Code Cracker for Visual Studio 2017 as a helper to write better code	475
Checking the final code after analysis	475
Use case – evaluating the C# code before publishing the application	477
Summary	479
Questions	479
Further reading	480
Section 5: Delivering Software Continuously and at a High Quality Level	
Chapter 17: Deploying Your Application with Azure DevOps	483
Technical requirements	484
Understanding SaaS	484
Adapting your organization to a service scenario	484
Developing software in a service scenario	485
Technical implications of a service scenario	485
Adopting a SaaS solution	486
Preparing a solution for a service scenario	487
Use case – deploying our package-management application with Azure Pipelines	490

Creating the Azure Web App and the Azure database	490
Configuring your Visual Studio solution	492
Configuring Azure Pipelines	493
Adding a manual approval for the release	496
Creating a release	498
Summary	500
Questions	501
Further reading	501
Chapter 18: Understanding DevOps Principles	502
Technical requirements	503
Describing DevOps	503
Understanding DevOps principles	504
Defining continuous integration	504
Understanding continuous delivery and multistage environment with Azure DevOps	505
Defining continuous feedback and the related DevOps tools	508
Monitoring your software with Application Insights	509
Using the Test and Feedback tool to enable feedback	515
The WWTravelClub project approach	520
Summary	521
Questions	521
Further Reading	522
Chapter 19: Challenges of Applying CI Scenarios in DevOps	523
Technical requirements	524
Understanding CI	524
Understanding the risks and challenges when using CI	525
Disabling continuous production deployment	526
Incomplete features	527
Unstable solution for testing	530
The WWTravelClub project approach	534
Summary	534
Questions	535
Further reading	535
Chapter 20: Automation for Software Testing	536
Technical requirements	536
Understanding the purpose of functional tests	537
Using unit testing tools to automate functional tests in C#	539
Testing the staging application	540
Testing a controlled application	541
Use case – automating functional tests	543
Summary	547
Questions	547

Table of Contents

Further reading	547
Appendix A: Assessments	548
Chapter 1	548
Chapter 2	548
Chapter 3	549
Chapter 4	549
Chapter 5	550
Chapter 6	550
Chapter 7	551
Chapter 8	551
Chapter 9	552
Chapter 10	553
Chapter 11	553
Chapter 12	554
Chapter 13	554
Chapter 14	555
Chapter 15	555
Chapter 16	556
Chapter 17	556
Chapter 18	557
Chapter 19	557
Chapter 20	558
Other Books You May Enjoy	559
Index	562

Preface

This book covers the most common design patterns and frameworks involved in software architecture. It discusses when and how to use each pattern by providing you with practical real-world scenarios. This book also presents techniques and processes such as DevOps, microservices, continuous integration, and cloud computing so that you can have a best-in-class software solution developed and delivered for your customers.

This book will help you to understand the product that your customer wants from you. It will guide you to deliver and solve the biggest problems you could face during development. It also covers the do's and don'ts that you need to follow when you manage your application in a cloud-based environment. You will learn about different architectural approaches, such as layered architectures, service-oriented architecture, microservices, and cloud architecture, and understand how to apply them to specific business requirements. Finally, you will deploy code in remote environments or on the cloud using Azure.

All the concepts in this book will be explained with the help of real-world practical use cases where design principles make the difference when creating safe and robust applications. By the end of the book, you will be able to develop and deliver highly scalable enterprise-ready applications that meet the end customers' business needs.

It is worth mentioning that this book will not only cover the best practices that a software architect should follow for developing C# and .NET Core solutions, but it will also discuss all the environments that we need to master in order to develop a software product according to the latest trends.

Who this book is for

This book is for engineers and senior developers who are aspiring to become architects or wish to build enterprise applications with the .NET stack. Experience with C# and .NET is required.

What this book covers

Chapter 1, *Understanding the Importance of Software Architecture*, explains the basics of software architecture. This chapter will give you the right mindset to face customer requirements, and then select the right tools, patterns, and frameworks.

Chapter 2, *Functional and Nonfunctional Requirements*, guides you in the first stage of application development, that is, collecting user requirements and accounting for all other constraints and goals that the application must fulfill.

Chapter 3, *Documenting Requirements with Azure DevOps*, describes techniques for documenting requirements, bugs, and other information about your applications. While most of the concepts are general, the chapter focuses on the usage of Azure DevOps.

Chapter 4, *Deciding the Best Cloud-Based Solution*, gives you a wide overview of the tools and resources available in the cloud, and in particular on Microsoft Azure. Here, you will learn how to search for the right tools and resources and how to configure them to fulfill your needs.

Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*, offers a broad overview of microservices and Docker containers. Here, you will learn how the microservices-based architecture takes advantage of all the opportunities offered by the cloud and you will see how to use microservices to achieve flexibility, high throughput, and reliability in the cloud. You will learn how to use containers and Docker to mix different technologies in your architecture as well as make your software platform-independent.

Chapter 6, *Interacting with Data in C# - Entity Framework Core*, explains in detail how your application can interact with various storage engines with the help of **Object-Relational Mappings (ORMs)** and Entity Framework Core 3.0.

Chapter 7, *How to Choose Your Data Storage in the Cloud*, describes the main storage engines available in the cloud and, in particular, in Microsoft Azure. Here, you will learn how to choose the best storage engines to achieve the read/write parallelism you need and how to configure them.

Chapter 8, *Working with Azure Functions*, describes the serverless model of computation and how to use it in the Azure cloud. Here, you will learn how to allocate cloud resources just when they are needed to run some computation, thus paying only for the actual computation time.

Chapter 9, *Design Patterns and .NET Core Implementation*, describes common software patterns with .NET Core 3 examples. Here, you will learn the importance of patterns and best practices for using them.

Chapter 10, *Understanding the Different Domains in a Software Solution*, describes the modern domain-driven design software production methodology, how to use it to face complex applications that require several knowledge domains, and how to use it to take advantage of cloud- and microservices-based architectures.

Chapter 11, *Implementing Code Reusability in C# 8*, describes patterns and best practices to maximize code reusability in your C# .NET Core applications.

Chapter 12, *Applying Service-Oriented Architectures with .NET Core*, describes service-oriented architecture, which enables you to expose the functionalities of your applications as endpoints on the web or on a private network so that users can interact with them through various types of clients. Here, you will learn how to implement service-oriented architecture endpoints with ASP.NET Core, and how to self-document them with existing OpenAPI packages.

Chapter 13, *Presenting ASP.NET Core MVC*, describes in detail the ASP.NET Core framework. Here, you will learn how to implement web applications based on the Model-View-Controller (**MVC**) pattern and how to organize them according to the prescriptions of domain-driven design, described in Chapter 10, *Understanding the Different Domains in a Software Solution*.

Chapter 14, *Best Practices in Coding C# 8*, describes best practices to be followed when developing .NET Core applications with C# 8.

Chapter 15, *Testing Your Code with Unit Test Cases and TDD*, describes how to test your applications. Here, you will learn how to test .NET Core applications with xUnit, and see how easily you can develop and maintain code that satisfies your specifications with the help of test-driven design.

Chapter 16, *Using Tools to Write Better Code*, describe metrics that evaluate the quality of your software and how to measure them with the help of all the tools included in Visual Studio.

Chapter 17, *Deploying Your Application with Azure DevOps*, describes how to automate the whole deployment process, from the creation of a new release in your source repository, through various testing and approval steps, to the final deployment of the application in the actual production environment. Here, you will learn how to use Azure Pipelines to automate the whole deployment process.

Chapter 18, *Understanding DevOps Principles*, describes the basics of the DevOps software development and maintenance methodology. Here, you will learn how to organize your application's continuous integration/continuous delivery cycle.

Chapter 19, *Challenges of Applying CI Scenarios in DevOps*, complements the description of DevOps with continuous integration scenarios.

Chapter 20, *Automation for Software Testing*, is dedicated to automatic acceptance tests – that is, tests that verify automatically whether a version of a whole application conforms with the agreed specifications. Here, you will learn how to simulate user operations with automation tools and how to use these tools together with xUnit to write your acceptance tests.

To get the most out of this book

Do not forget to have Visual Studio Community 2019 or higher installed.

Be sure that you understand C# .NET principles.

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.packtpub.com/support and register to have the files emailed directly to you.

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 Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Hands-On-Software-Architecture-with-CSharp-8>. In case there's an update to the code, it will be updated on the existing GitHub repository.

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

Code in Action

You can see the code in action videos at <http://bit.ly/20ld2IG>.

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/9781789800937_ColorImages.pdf.

Conventions used

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

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "They are copied in the final string just once, when you call `sb.ToString()` to get the final result."

A block of code is set as follows:

```
[Fact]
public void Test1()
{
    var myInstanceToTest = new ClassToTest();
    Assert.Equal(5, myInstanceToTest.MethodToTest(1));
}
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "In the **Solution Explorer**, you have the option to **Publish...** by right-clicking."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, 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.packtpub.com/support/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.

1

Section 1: Transforming Customer Needs in Real-World Applications

This section includes the first three chapters of the book. The idea is to make sure you understand how to transform user requirements into the actual architectural needs that are essential for project success, using the various architectural aspects and design considerations involved in developing enterprise applications with C# and .NET Core.

In Chapter 1, *Understanding the Importance of Software Architecture*, we will discuss the importance of software architecture and aspects related to .NET Core and C#. The chapter will also discuss the importance of analyzing software requirements and designing for principles such as scalability and robustness. No matter what software development cycle you decide to have in your project, analyzing requirements will help you adhere to the goal of the project. Without this, the success of your project is at risk. This chapter will present some use cases where a lack of understanding of the requirements gathered led to project failures. Besides this, in each case, we will provide practical advice that could help to protect your software from the scenarios presented.

Once you have understood the process of gathering system requirements, in Chapter 2, *Functional and Nonfunctional Requirements*, we will prompt you to think about the impacts that the requirements have on the architectural design. Scalability, performance, multithreading, interoperability, and other subjects will be discussed, both their theory and their practice.

To finish the first section, in Chapter 3, *Documenting Requirements with Azure DevOps*, we will present Azure DevOps, which is the tool currently being provided by Microsoft to enable an application development life cycle that follows the principles of the DevOps philosophy. There are a variety of good features that can help you document and organize your software, and the purpose of the chapter is to present an overview of those features.

This section includes the following chapters:

- Chapter 1, *Understanding the Importance of Software Architecture*
- Chapter 2, *Functional and Nonfunctional Requirements*
- Chapter 3, *Documenting Requirements with Azure DevOps*

1

Understanding the Importance of Software Architecture

Nowadays, software architecture is one of the most discussed topics in the software industry, and for sure, its importance will grow more in the future. The more we build complex and fantastic solutions, the more we need great software architectures to maintain them. That is the reason why you decided to read this book. That is the reason why we decided to write it.

For sure, it is not an easy task to write about this important topic, which offers so many alternative techniques and solutions. The main objective of this book is not just to build an exhaustive and never-ending list of available techniques and solutions, but also to show how various families of techniques are related and how they impact, in practice, the construction of a maintainable and sustainable solution.

The attention on how to create actual efficacious enterprise solutions increases as users always need more new features in their applications. Moreover, the need to deliver frequent application versions (due to a quickly changing market) increases the obligation to have sophisticated software architecture and development techniques.

The following topics will be covered in this chapter:

- The history of software development and the definition of software architecture
- Software processes currently used by success enterprises
- The process for gathering requirements

By the end of this chapter, you will be able to understand exactly what the mission of a software architecture is. You will also learn what Azure is and how to create your account in the platform. Besides considering this is an introductory chapter, you will get an overview of software processes, models, and other techniques that will enable you to conduct your team.

Technical requirements

This chapter will guide you on how to create an account in Azure, hence no code will be provided.

What is software architecture?

If you are reading this book today, you should thank the computer scientists who decided to consider software development as an engineering area. This happened in the last century and, more specifically, at the end of the sixties, when they proposed that the way we develop software is quite similar to the way we construct buildings. That is why we have the name **software architecture**. Like in the design of a building, the main goal of a software architect is to ensure that the software application is implemented well. But a good implementation requires the design of a great solution. Hence, in a professional development project, you have to do the following things:

- Define the customer requirements for the solution.
- Design a great solution to meet those requirements.
- Implement the designed solution.
- Validate the solution with your customer.
- Deliver the solution in the working environment.

Software engineering defines these activities as the software development life cycle. All of the theoretical software development process models (waterfall, spiral, incremental, agile, and so on) are somehow related to this cycle. No matter which model you use, if you do not work with the essential tasks presented earlier during your project, you will not deliver acceptable software as a solution.

The main point about designing great solutions is totally connected to the purpose of this book. You have to understand that great real-world solutions bring with them a few fundamental constraints:

- The solution needs to meet user requirements.
- The solution needs to be delivered on time.
- The solution needs to adhere to the project budget.
- The solution needs to deliver good quality.
- The solution needs to guarantee a safe and efficacious future evolution.

Great solutions need to be sustainable and you have to understand that there is no sustainable software without great software architecture. Nowadays, great software architectures depend on both tools and environments to perfectly fit users' requirements. To explain this, this book will use some great tools provided by Microsoft:

- **Azure:** This is the cloud platform from Microsoft, where you will find all of the components it provides to build advanced software architecture solutions.
- **Azure DevOps:** This is the application life cycle management environment where you can build solutions using the latest approach for developing software, that is, DevOps.
- **C#:** This is one of the most used programming languages in the world. C# runs on small devices up to huge servers in different operating systems and environments.
- **.NET Core:** This is an open source development platform that is maintained by the Microsoft and .NET community on GitHub.
- **ASP.NET Core:** This is an open source multi-platform environment developed using .NET Core to build web applications and is hosted in the cloud or even on standard servers (on-premises).

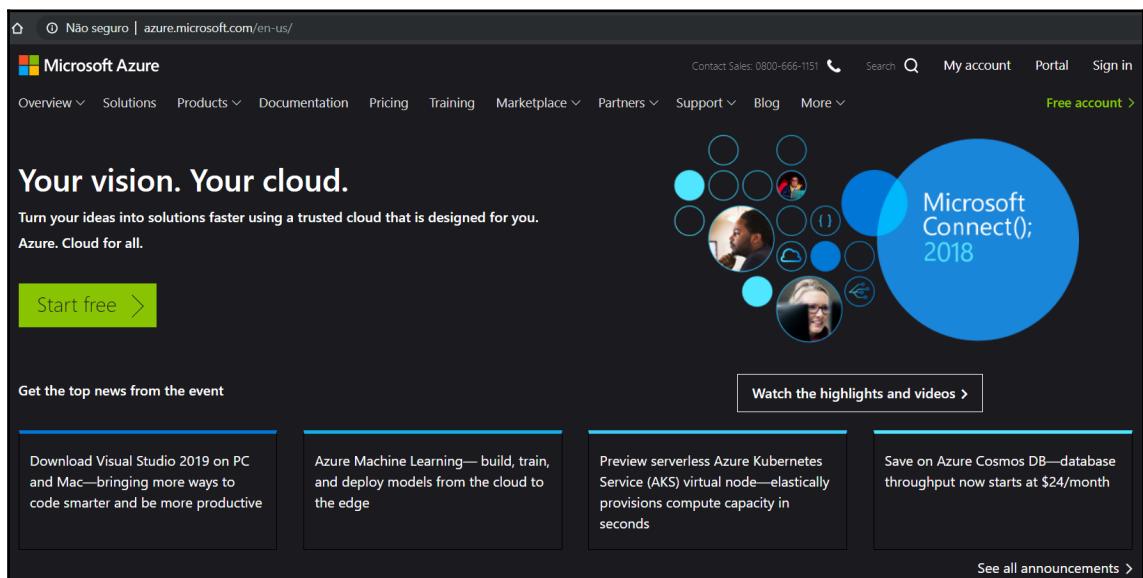
Being a software architect means understanding the aforementioned and a lot of other technologies. This book will guide you on a journey where you, as a software architect working in a team, will provide optimal solutions with the tools listed. Let's start this journey by creating your Azure account.

Creating an Azure account

Microsoft Azure is one of the best cloud solutions currently available on the market. It is important to know that, inside Azure, we will find a bunch of components that can help us in the architecture of twenty-first century solutions.

This subsection will guide you in creating an Azure account. If you already have one, you can skip this part:

1. You can access the Azure portal using this URL: <https://azure.microsoft.com>. Here, you will find a website, as follows. The translation to your native language will probably be set automatically:



2. Once you have accessed this portal, it is possible to sign up. If you have never done this before, it is possible to sign up for free, so you will be able to use some Azure features without spending any money.

3. Once you finish the form, you will be able to access the Azure panel. As you can see in the following screenshot, the panel shows a dashboard that you can customize, and a menu on the left, where you can set up the Azure components you are going to use in your solution. Throughout this book, we will come back to this screenshot to set up the components that create great opportunities for modern software architecture:

The screenshot shows the Microsoft Azure portal at portal.azure.com/#allservices. The interface includes a top navigation bar with back, forward, and search icons. Below the navigation is the Microsoft Azure logo and a search bar. On the left, there's a sidebar titled 'Categories' with a tree view of service categories like All, General, Compute, Networking, Storage, Web, Mobile, Containers, Databases, Analytics, Blockchain, AI + machine learning, Internet of things, Mixed reality, Integration, Identity, Security, DevOps, Migrate, and Monitor. The main content area features a 'Featured' section with tiles for Virtual machines, App Services, Storage accounts, SQL databases, Azure Database for, Azure Cosmos DB, Kubernetes services, and Function App. Below this is a 'Free training from Microsoft' section with three cards: 'Core Cloud Services - Azure architecture and service guarantees', 'Core Cloud Services - Manage services with the Azure portal', and 'Cloud Concepts - Principles of cloud computing'. At the bottom, there's a 'Get to know Azure' section with links to the Quickstart center, Free offerings, and Work with an expert.

Once you have your Azure account created, you are ready to understand how a software architect can conduct a team to develop software taking advantage of all of the opportunities offered by Azure. However, it is important to keep in mind that a software architect needs to understand something more than specific technologies because, nowadays, this role is played by people who are expected to define how the software will be delivered. A software architect not only architects the base of software, but they also determine how the whole software development and deployment process is conducted.

Software development process models

As a software architect, it is really important for you to understand some of the common development processes that are currently used in most enterprises. A software development process defines how people in a team produce and deliver software. In general, this process is connected with a software engineering theory, called **software development process models**. From the time software development was defined as an engineering process, many process models for developing software have been proposed. Let's take a look at the ones that are currently common.

Reviewing traditional software development process models

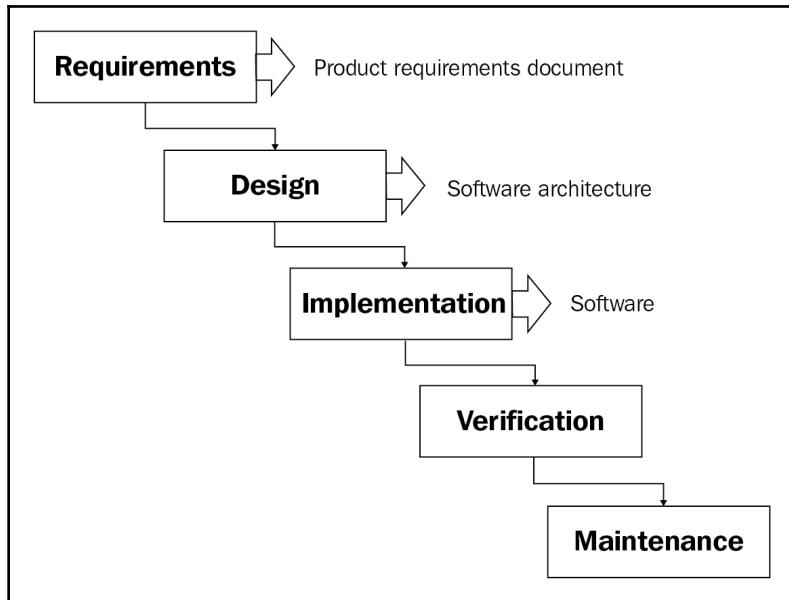
Some of the models introduced in the software engineering theory are already considered traditional and quite obsolete. This book does not aim to cover all of them, but here, we will give a brief explanation of the ones that are still used in some companies.

Understanding the waterfall model principles

This topic may appear strange in a software architecture book of 2019, but yes, you may still find companies where the most traditional software process model still remains the guideline for software development. This process executes all fundamental tasks in sequence. Any software development project consists of the following steps:

- Requirements specification
- Software design
- Programming
- Tests and delivery

Let's look at a diagrammatic representation of this:



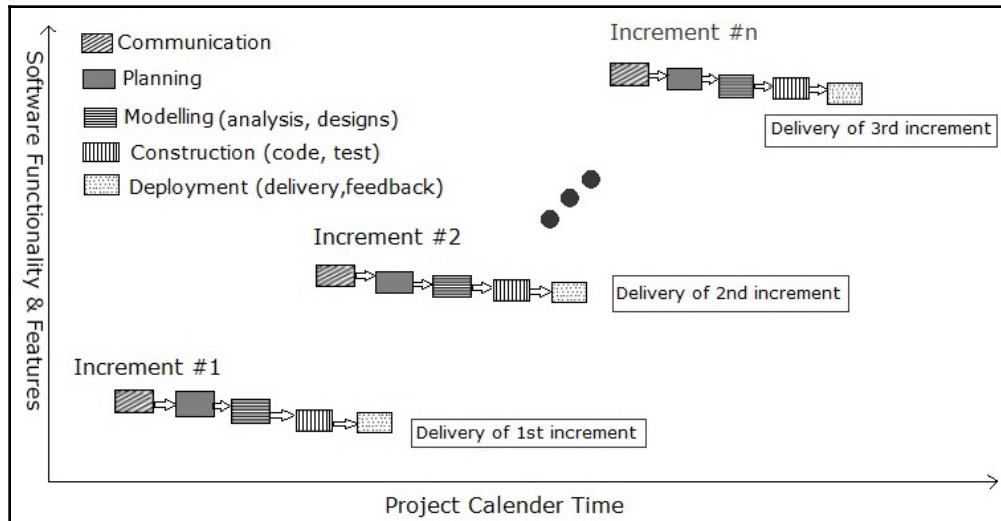
The waterfall development cycle (https://en.wikipedia.org/wiki/Waterfall_model)

Often, the use of waterfall models causes problems related to delays in the delivery of a functional version of the software and user dissatisfaction due to the poor quality of the final product.

Analyzing the incremental model

Incremental development is an approach that tries to overcome the biggest problem of the waterfall model: the user can test the solution only at the end of the project. The idea of this model is to give the users opportunities to interact with the solution as early as possible so that they can give useful feedback, which will help during the development of the software.

However, also in this model, the limited number of increments and the project's bureaucracy can causes problems in the interaction between developers and customers:



The incremental development cycle (https://en.wikipedia.org/wiki/Incremental_build_model)

The incremental model was introduced as an alternative to the waterfall approach and it mitigated the problems related to the lack of communication with the customer. For big projects, fewer increments is still a problem. Besides, at the time the incremental approach was used on a large scale, mainly at the end of the last century, many problems related to project bureaucracy were reported, due to the large amount of documentation required. This scenario caused the rise of a very important movement in the software development industry—**agile**.

Understanding agile software development process models

At the beginning of this century, developing software was considered one of the most chaotic activities in engineering. The number of software projects that failed was incredibly high and this fact proved the need for a different approach to deal with the flexibility required by software development projects. So, in 2001, the Agile Manifesto was introduced to the world and, from that time, various agile process models were proposed. Some of them have survived up till now and are still very common.



Please check out this link for the Agile Manifesto: <https://agilemanifesto.org/>.

One of the biggest differences between agile models and traditional models is the way developers interact with the customer. The message that all agile models transmit is that the faster you deliver software to the user, the better. This idea is sometimes confusing for software developers who understand this as—*let's try coding and that's all folks!* However, there is an important observation of the Agile Manifesto that many people do not read when they start working with agile:

"That is, while there is value in the items on the right, we value the items on the left more."

— Agile Manifesto, 2001

A software architect always needs to remember this. Agile processes do not mean a lack of discipline. Moreover, when you use the agile process, you understand that there is no way to have good software developed without discipline. On the other hand, as a software architect, you need to understand that *soft* means flexibility. A software project that does not deal with flexibility tends to get ruined over time.

Getting into the Scrum model

Scrum is an agile model for the management of software development projects. The model comes from lean principles and is definitely one of the widely used approaches for developing software nowadays.



Please check this link for more information about the Scrum framework: <https://docplayer.net/78853722-Scrum-insights-for-practitioners.html>.

The basis of Scrum is that you have a flexible backlog of user needs that needs to be discussed in each agile cycle, called a **Sprint**. The Sprint Goal is determined by the Scrum Team, composed by the Product Owner, the Scrum Master, and the Development Team. The **Product Owner** is responsible for prioritizing what will be delivered in that sprint. During the sprint, this person will help the team to develop the required features. The person who leads the team in the Scrum process is called **Scrum Master**. All of the meetings and processes are conducted by this person.

It is important to notice that the Scrum process does not discuss how the software needs to be implemented and which activities will be done. So, again, you have to remember the software development basis, discussed at the beginning of this chapter. That means Scrum needs to be implemented together with a process model. DevOps is one of the approaches that may help you with the use of a software development process model together with Scrum. We will discuss this later in this book, in *Chapter 18, Understanding DevOps Principles*.

Enabling aspects to be gathered to design high-quality software

Fantastic! You just started a software development project. Now, it is time to use all of your knowledge to deliver the best software you can. Probably, your next question is—*how do I start?* Well, as a software architect, you are going to be the one to answer it. And be sure your answer is going to evolve in each software project you lead:

1. Defining a software development process is obviously the first thing to do. This is generally done during the project planning process.
2. Besides, another very important thing to do is to gather the software requirements. No matter which software development process you decide to use, collecting real user needs is a part of a very difficult and continuous job. Of course, there are techniques to help you with this. And be sure that gathering requirements will help you to detect important aspects of software architecture.

These two activities are considered by most experts in software development as the key to having success at the end of the development project journey. As a software architect, you need to enable them to happen so that you will not have problems while guiding your team.

Understanding the requirements gathering process

There are different ways to represent the requirements. The most traditional approach consists of you having to write a perfect specification before the beginning of the analysis. Agile methods suggest that you need to write stories as soon as you are ready to start a development cycle.



Remember: you do not write requirements for the user, you write them for you and your team. The user just needs the job done!

The truth is that no matter the approach you decide to adopt in your projects, you will have to follow some steps to gather requirements. This is what we call **requirements engineering**.



Please check out this image of the requirements engineering process for more information: <https://www.slideshare.net/MohammedRomi/ian-sommerville-software-engineering-9th-edition-ch-4>.

During this process, you need to be sure that the solution is feasible. In some cases, the feasibility analysis is a part of the project planning process too, and by the time you start the requirements elicitation, you will have the feasibility report already done. So, let's check the other parts of this process, which will give you a lot of important information for the software architecture.

Practicing the elicitation of user needs

There are a lot of ways to detect what exactly the user needs for a specific scenario. In general, this can be done using techniques that will help you to understand what we call user requirements. Here, you have a list of common techniques:

- **The power of imagination:** If you are an expert in the area where you are providing solutions, you may use your own imagination to find new user requirements. Brainstorming can be conducted together so that a group of experts can define user needs.
- **Questionnaires:** This tool is useful for detecting common and important requirements such as the number and kind of users, peak system usage, and the commonly-used **operating system (OS)** and web browser.
- **Interviews:** Interviewing the users helps you as an architect to detect user requirements that perhaps questionnaires and your imagination will not cover.
- **Observation:** There is no better way to understand the daily routine of a user than being with them for a day.

As soon as you apply one or more of these techniques, you will have great and valuable information, that is, the user's needs. At that moment, you will be able to analyze them and detect the user and system requirements.

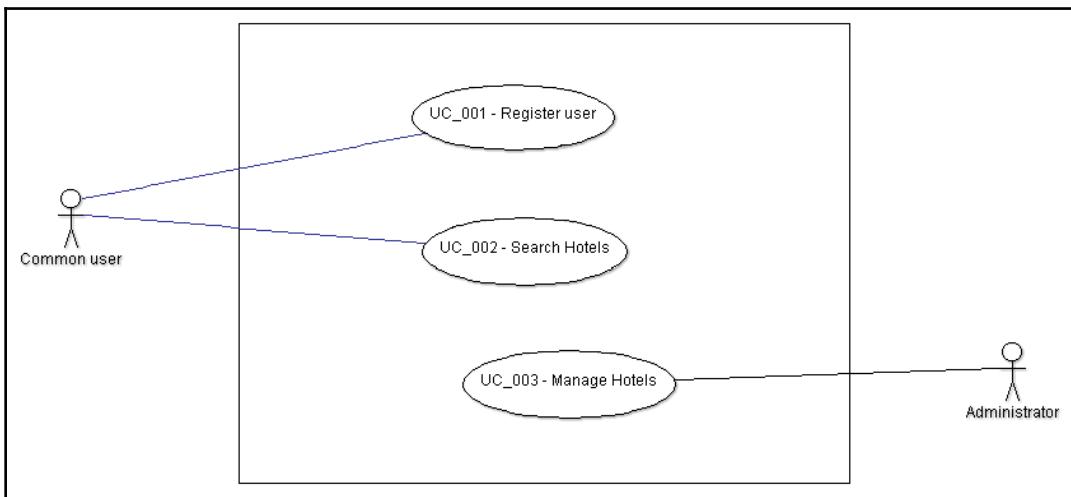


Remember: You can use these techniques in any situation where the real need is to gather requirements, no matter if it is for the whole system or for a single story.

Analyzing requirements

As soon as you detect user needs, it is time to begin the analysis of the requirements. At that time, you can use techniques such as the following:

- **Prototyping:** Prototypes are really good to clarify and to materialize the system requirements. Today, we have many tools that can help you to mock interfaces. A really nice open source tool is the **Pencil Project**. You will find further information about it at <https://pencil.evolus.vn/>.
- **Use cases:** The **Unified Modeling Language (UML)** use case model is an option if you need detailed documentation. The model is composed of a detailed specification and a diagram. **Argo UML** is another open source tool that can help you out with this:



While you are analyzing the requirements of the system, you will be able to clarify exactly what the users' needs are. This is really helpful when you are not sure about the real problem you will solve and is pretty much better than just starting to program the system. It is time that you will invest in having better code in the near future.

Writing the specifications

After you finish the analysis, it is important to register it as a specification. This document can be written using traditional requirements or user stories, which are commonly used in agile projects.

Requirements specification represents the technical contract between the user and the team. There are some basic rules that this document needs to follow:

- All stakeholders need to understand exactly what is written in the technical contract, even if they are not technicians.
- The document needs to be clear.
- You need to classify each requirement.
- Use a simple feature to represent each requirement.
- Ambiguity and controversy need to be avoided.

Besides, some information can help the team to understand the context of the project they are going to work on. Here, you have some tips about it:

- Write an introductory chapter to give a full idea of the solution.
- Create a glossary to make understanding easier.
- Describe the kind of user the solution will cover.
- Write functional and non-functional requirements.
- Attach documents that can help the user to understand rules.

If you decide to write user stories, a good tip to follow is to write short sentences representing each moment in the system with each user, as follows:

As <user>, I want <feature>, so that <reason>

This approach will explain exactly the reason why that feature will be implemented. Besides that, you will have a good tool to later analyze the stories that are more critical and prioritize the success of the project.

Reviewing the specification

Once you have the specification written, it is time to confirm with the stakeholders whether they agree with it. This can be conducted in a review meeting or can be done online using collaboration tools.

This is when you present all of the prototypes, documents, and information you have gathered. As soon as everybody agrees with the specification, you are ready to start studying the best way to implement this part of your project.

Using design thinking as a helpful tool

During your career as a software architect, you will find many projects where your customer will bring you a solution *ready for development*. This is quite complicated once you consider that as the correct solution and, most of the time, there will be architectural and functional mistakes that will cause problems in the solution in the future. There are some cases where the problem is worse—when the customer does not know the best solution for the problem. Design thinking can help us with this.

Design thinking is a process that allows you to collect data directly from the users, focusing on achieving the best results to solve a problem. During this process, the team will have the opportunity to discover all *personas* that will interact with the system. This will have a wonderful impact on the solution since you can develop the software by focusing on the user experience, which can have a fantastic impact on the results.

The process is based on the following steps:

- **Empathize:** In this step, you have to execute field research to discover the user's concerns. This is where you find out about the users of the system. The process is good for making you understand why and for whom you are developing this software.
- **Define:** Once you have the users' concerns, it is time to define their needs to solve them.
- **Ideate:** The needs will provide an opportunity to brainstorm some possible solutions.
- **Prototype:** These solutions can be developed as prototypes to confirm whether they are good ones.
- **Test:** Testing the prototypes will help you to understand the prototype that is most connected to the real needs of the users.

What you have to understand is that design thinking can be a fantastic option to discover real requirements. As a software architect, you are committed to helping your team to use the correct tools at the correct time.

Understanding the principles of scalability, robustness, security, and performance

Detecting requirements is a task that will let you understand the software you are going to develop. However, as a software architect, you don't have to only pay attention to the functional requirements for that system. Understanding the non-functional requirements is really important and one of the primordial activities for a software architect.

We are going to discuss this more in [Chapter 2, Functional and Nonfunctional Requirements](#), but at this point, it is good to know that the principles of scalability, robustness, security, and performance need to be applied for the requirements gathering process. Let's take a look at each concept:

- **Scalability:** As a software developer, globalization gives you the opportunity to have your solution running all over the world. This is fantastic, but you, as a software architect, need to design a solution that provides that possibility. Scalability is the possibility for an application to increase its processing power as soon as it is necessary, due to the number of resources that are being consumed.
- **Robustness:** No matter how scalable your application is, if it is not able to guarantee a stable and always-on solution, you are not going to get any peace. Robustness is really important for critical solutions, where you do not have the opportunity for maintenance at any time, due to the kind of problem that the application solves. In many industries, the software cannot stop and lots of routines run when nobody is available (overnight, holidays, and so on). Designing a robust solution will give you the freedom to live while your software is running well.
- **Security:** This is another really important area that needs to be discussed after the requirements stage. Everybody is worried about security and laws dealing with it are being proposed in different parts of the world. You, as a software architect, have to understand that security needs to be provided by design. This is the only way to cope with all of the needs that the security community is discussing right now.
- **Performance:** The process of understanding the system you are going to develop will probably give you a good idea of what your efforts will need to be to get the desired performance from the system. This topic needs to be discussed with the user to identify most of the bottlenecks you will face during the development stage.

It is worth mentioning that all these concepts are requirements for this new generation of solutions that the world needs. What will differentiate good software for incredible software surely is the amount of work done to meet the project requirements.

Some cases where the requirements gathering process impacted system results

All of the information discussed up to this point in the chapter is useful if you want to design software following the principles of good engineering. This discussion is not related to developing by using traditional or agile methods but focuses on building software professionally or as an amateur.

Besides, it is good to know about some cases where the lack of activities you read about caused some trouble for the software project. The following cases intend to describe what went wrong and how the preceding techniques could have helped the development team to solve the problems. In most cases, simple action could guarantee better communication between the team and the customer and this easy communication flow could transform a big problem into a real solution.

Case 1 – my website is too slow to open that page!

Performance is one of the biggest problems that you as a software architect will live through during your career. The reason why this aspect of any software is so problematic is that we do not have infinite computational resources to solve problems. Besides, the cost of computation is still high, especially if you are talking about software with a high number of simultaneous users.

You cannot solve performance problems by writing requirements. However, you won't end up in trouble if you write them correctly. The idea here is that requirements have to present the desired performance of a system. A simple sentence, describing this, can help the entire team that works on the project:

Non-functional requirement: Performance – any web page of this software will respond in at least 2 seconds.

The preceding sentence just makes everybody (users, testers, developers, architects, managers, and so on) sure that any web page has a target to achieve. This is a good start, but it is not enough. With this, a great environment to both develop and deploy your application is important. This is where .NET Core can help you a lot. Especially if you are talking about web apps, ASP.NET Core is considered one of the fastest options to deliver solutions today.

If you talk about performance, you, as a software architect, should consider the use of the techniques listed in the following sections. It is good to mention that ASP.NET Core will help you to use them easily, together with some **Platform as a Service (PaaS)** solutions delivered by Microsoft Azure.

Understanding caching

Caching is a great technique to avoid queries that can consume time and, in general, give the same result. For instance, if you are fetching the available car models in a database, the number of cars in the database can increase but they will not change. Once you have an application that constantly accesses car models, a good practice is to cache that information.

It is important to understand that a cache is stored in the backend and that cache is shared by the whole application (*in-memory caching*). A single point of attention here is when you are working on a scalable solution, you can configure a *distributed cache* to solve it using the Azure platform. In fact, ASP.NET Core provides both of them, so you can decide on the one that bests fits your needs.

Applying asynchronous programming

When you develop ASP.NET Core applications, you need to keep in mind that this app needs to be designed for simultaneous access by many users. Asynchronous programming lets you do this simply, giving you the keywords `async` and `await`.

The basic concept behind these keywords is that `async` enables any method to run in a different thread from the one that calls it. On the other hand, `await` lets you synchronize the call of an asynchronous method without blocking the thread that is calling it. This easy-to-develop pattern will make your application run without performance bottlenecks and better responsiveness. This book will cover more about this subject in Chapter 2, *Functional and Nonfunctional Requirements*.

Dealing with object allocation

One very good tip to avoid a lack of performance is to understand how the Garbage Collector works. The Garbage Collector is the engine that will free memory automatically when you finish using it. There are some very important aspects of this topic, due to the complexity that the GC has.

Some types of objects are not collected by the GC. The list includes any object that interacts with I/O, such as files and streaming. If you do not correctly use the C# syntax to create and destroy this kind of object, you will have memory leaks, which will deteriorate your application performance.

The incorrect way of working with I/O objects:

```
System.IO.StreamWriter file = new System.IO.StreamWriter(@"C:\sample.txt");
file.WriteLine("Just writing a simple line");
```

The correct way of working with I/O objects:

```
using (System.IO.StreamWriter file = new
System.IO.StreamWriter(@"C:\sample.txt"))
{
    file.WriteLine("Just writing a simple line");
}
```

Even though the preceding practice is mandatory for I/O objects, it is totally recommended that you keep doing this in all disposable objects. This will help the GC and will keep your application running with the right amount of memory.

Another important aspect that you need to know about is that the time spent by the GC to collect objects that will interfere with the performance of your app. Because of this, avoid allocating large objects. This can cause you trouble waiting for the GC to finish its task.

Getting better database access

One of the most common performance Achilles' heel is database access. The reason why this is still a big problem is the lack of attention while writing queries or lambda expressions to get information from the database. This book will cover Entity Framework Core in Chapter 6, *Interacting with Data in C# – Entity Framework Core*, but it is important to know what to choose, the correct data information to read from a database, and filtering columns and lines is imperative for an application that wants to deliver performance.

The good thing is that best practices related to caching, asynchronous programming, and object allocation fit completely in the environment of databases. It is only a matter of choosing the correct pattern to get better-performance software.

Case 2 – the user's needs are not properly implemented

The more technology is used in a wide variety of areas, the more difficult it is to deliver exactly what the user needs. Maybe this sentence sounds weird to you, but you have to understand that developers, in general, study to develop software, but they rarely study to deliver the needs of a specific area. Of course, it is not easy to learn how to develop software, but it is even more difficult to understand a need in a particular area. Software development nowadays delivers software to all possible types of industries. The question here is *how can a developer, being a software architect or not, evolve enough to deliver software in the area they are responsible for?*

Gathering software requirements definitely will help you in this tough task. Moreover, writing them will make you understand and organize the architecture of the system. There are several ways to minimize the risks of implementing something different from what the user really needs:

- Prototyping the interface to achieve the understanding of the user interface faster
- Designing the data flow to detect gaps between the system and the user operation
- Frequent meetings to be updated on the current needs and aligned to the incremental deliveries

Again, as a software architect, you will have to define how the software will be implemented. Most of the time, you are not going to be the one who programs it, but you will always be the one responsible for this. For this reason, some techniques can be useful to avoid the wrong implementation:

- Requirements are reviewed with the developers to guarantee that they understand what they need to develop
- Code inspection to validate a predefined code standard
- Meetings to eliminate impediments

Case 3 – the usability of the system does not meet user needs

Usability is a key point for the success of a software project. The way the software is presented and how it solves a problem can help the user to decide whether they want to use it or not. As a software architect, you have to keep in mind that delivering software with good usability is mandatory nowadays.

There are basic concepts of usability that this book does not intend to cover. But a good way to meet the correct user needs when it comes to usability is by understanding who is going to use the software. Design thinking can help you a lot with that, as was discussed earlier in this chapter.

Understanding the user will help you to decide whether the software is going to run on a web page, or a cell phone, or even in the background. This understanding is very important to a software architect because the elements of a system will be better presented if you correctly map who will use them.

On the other hand, if you do not care about that, you will just deliver software that works. This can be good for a short time, but it will not exactly meet the real needs that made a person ask you to architect software. You have to keep in mind the options and understand that good software is designed to run on many platforms and devices.

You will be happy to know that C# is an incredible cross-platform option for that. So, you can develop solutions to run your apps in Linux, Windows, Android, and iOS. You can run your applications on big screens, tablets, cell phones, and even drones! You can embed apps on boards for automation or in HoloLens for mixed reality. Software architects have to be open-minded to design exactly what their users need.

Case study – detecting user needs

The case study of this book will take you on a journey of creating the software architecture for a travel agency called **World Wild Travel Club (WWTravelClub)**. The purpose of this case study is to make you understand the theory explained in each chapter, plus to provide the during the process of reading this book to develop an enterprise application with Azure, Azure DevOps, C#, .NET Core, ASP.NET Core, and other technologies that will be introduced in this book.

Book case study – introducing World Wild Travel Club

World Wild Travel Club (WWTravelClub) is a travel agency that was created to change the way people make decisions about their vacations and other trips around the world. To do so, they are developing an online service where every detail of a trip experience will be assisted by a club of experts specifically selected for each destination.

The concept of this platform is that you can be both a visitor and a destination expert at the same time. The more you participate as an expert in a destination, the higher the points you will score. These points can be exchanged for tickets that people buy online using the platform.

The customer came with the following requirements for the platform. It is important to know that, in general, customers do not bring the requirements ready for development. That is why the requirements gathering process is so important:

- Common user view:
 - Promotional packages on the home page
 - Search for packages
 - Details for each package:
 - Buy a package
 - Buy a package with a club of experts included:
 - Comment on your experience
 - Ask an expert
 - Evaluate an expert
 - Register as a common user
- Destination expert view:
 - The same view as the common user view
 - Answer the questions asking for your destination expertise
 - Manage the points you scored answering questions:
 - Exchange points for tickets
- Administrator view:
 - Manage packages
 - Manage common users
 - Manage destination experts

To finish this, it is important to note that WWTravelClub intends to have more than 100 Destination Experts per package and will offer around 1,000 different packages all over the world.

Book case study – understanding user needs and system requirements

To summarize the user needs of WWTravelClub, you can read the following user stories:

- US_001: As a common user, I want to view promotional packages on the home page, so that I can easily find my next vacation.
- US_002: As a common user, I want to search for packages I cannot find on the home page so that I can explore other trip opportunities.
- US_003: As a common user, I want to see the details of a package, so that I can decide which package to buy.
- US_004: As a common user, I want to register myself, so that I can start buying the package.
- US_005: As a registered user, I want to buy a package, so that I can process the payment.
- US_006: As a registered user, I want to buy a package with a club of experts included, so that I can have an exclusive trip experience.
- US_007: As a registered user, I want to ask for an expert, so that I can get the best of my trip.
- US_008: As a registered user, I want to comment on my experience, so that I can give feedback from my trip.
- US_009: As a registered user, I want to evaluate an expert who helps me, so that I can share with others how fantastic they were.
- US_010: As a registered user, I want to register as a Destination Expert View, so that I can help people who travel to my city.
- US_011: As an expert user, I want to answer questions about my city, so that I can score points to be exchanged in the future.
- US_012: As an expert user, I want to exchange points for tickets, so that I can travel around the world more.
- US_013: As an administrator user, I want to manage packages, so that users can have fantastic opportunities to travel.

- US_014: As an administrator user, I want to manage registered users, so that WWTravelClub can guarantee good service quality.
- US_015: As an administrator user, I want to manage expert users, so that all of the questions regarding our destinations are answered.
- US_016: As an administrator user, I want to offer more than 1,000 packages around the world, so that different countries can experience WWTravelClub service.
- US_017: As an administrator user, I want to have more than 1,000 users simultaneously accessing the website, so that I can support all of the needs of my users.
- US_018: As a user, I want to access WWTravelClub in my native language, so that I can easily understand the package offered.
- US_019: As a user, I want to access WWTravelClub in the Chrome, Firefox, and Edge web browsers, so that I can use the web browser of my preference.
- US_020: As a user, I want to buy packages safely, so that only WWTravelClub will have my credit card information.

Notice that while you start writing the stories, information related to non-functional requirements such as security, environment, performance, and scalability can be included.

However, some system requirements may be omitted when you write user stories and need to be included in the software specification. These requirements can be related to legal aspects, hardware and software prerequisites, or even points of attention for the correct system delivery. They need to be mapped and listed as well as user stories. The list of WWTravelClub system requirements is presented in the following. Notice that requirements are written in the future because the system does not exist yet:

- SR_001: The system will use Microsoft Azure components to deliver the scalability required.
- SR_002: The system will respect **General Data Protection Regulation (GDPR)** requirements.
- SR_003: The system will run on the Windows, Linux, iOS, and Android platforms.
- SR_004: Any web page of this system will respond in at least 2 seconds.

Summary

In this chapter, you learned the purpose of a software architect in a software development team. Also, this chapter covered the basics of software development process models and the requirements gathering process. You also had the opportunity to learn about how to create your Azure account, which will be used during the case study of this book, which was presented to you in the previous section. Moreover, you even learned about functional and non-functional requirements and how to create them using user stories. These techniques will surely help you deliver a better software project.

In the next chapter, you will have the opportunity to understand how functional and non-functional requirements are important for software architecture.

Questions

1. What is the expertise that a software architect needs to have?
2. How can Azure help a software architect?
3. How does a software architect decide the best software development process model to use in a project?
4. How does a software architect contribute to gathering requirements?
5. What kind of requirements does a software architect need to check in a requirement specification?
6. How does design thinking help a software architect in the process of gathering requirements?
7. How do user stories help a software architect in the process of writing requirements?
8. What are good techniques to develop very good performance software?
9. How does a software architect check whether a user requirement is correctly implemented?

Further reading

Here, you have some books and links you may consider reading to gather more information about this chapter:

- <https://www.packtpub.com/virtualization-and-cloud/hands-azure-developers>
- <https://azure.microsoft.com/en-us/overview/what-is-azure/>
- <https://azure.microsoft.com/en-us/services/devops/>
- <https://docs.microsoft.com/en-us/dotnet/core/about>
- <https://docs.microsoft.com/en-us/aspnet/core/>
- <https://www.packtpub.com/web-development/hands-full-stack-web-development-aspnet-core>
- <https://agilemanifesto.org/>
- <https://www.amazon.com/Software-Engineering-10th-Ian-Sommerville/dp/0133943038>
- <https://www.amazon.com/Software-Engineering-Practitioners-Roger-Pressman/dp/0078022126/>
- <https://scrumguides.org/>
- <https://www.packtpub.com/application-development/professional-scrummasters-handbook>
- <https://docs.microsoft.com/en-us/aspnet/core/performance/performance-best-practices>
- <https://www.microsoft.com/en-us/hololens>
- https://en.wikipedia.org/wiki/Incremental_build_model
- https://en.wikipedia.org/wiki/Waterfall_model

2

Functional and Nonfunctional Requirements

Once you have gathered the system requirements, it is time to think about the impact they have on the architectural design. Scalability, performance, multithreading, interoperability, and other subjects need to be analyzed so that we can meet user needs.

The following topics will be covered in this chapter:

- What is scalability and how does it interact with Azure and .NET Core?
- Good tips for writing better code when it comes to performance improvement
- Creating a safe and useful multithreading software
- Software usability, that is, how to design effective user interfaces
- .NET Core and interoperability

Technical requirements

The samples provided in this chapter will require Visual Studio 2019 Community Edition or Visual Studio Code.

You can find the sample code for this chapter here: <https://github.com/PacktPublishing/Hands-On-Software-Architecture-with-CSharp-8/tree/master/ch02>.

How does scalability interact with Azure and .NET Core?

A short search on scalability returns a definition such as *the ability of a system to keep working well when there's an increase in demand*. Once developers read this, many of them incorrectly conclude *that scalability only means add more hardware to keep things working without stopping the app*.

Scalability relies on technologies involving hardware solutions. However, as a software architect, you have to be aware that good software will keep scalability in a sustainable model, which means that a well-architected software can save a lot of money. Hence, it is not just a matter of hardware but also a matter of overall software design.

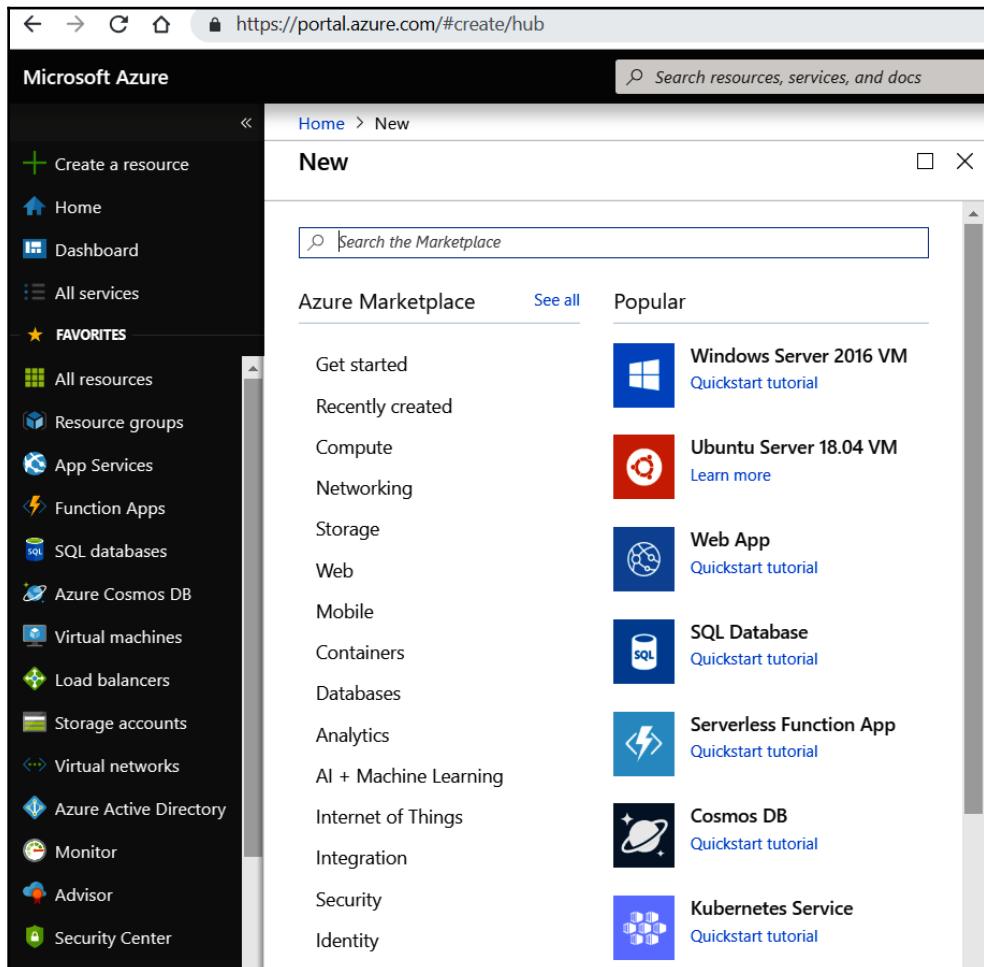
In Chapter 1, *Understanding the Importance of Software Architecture*, while discussing software performance, we proposed some good tips to overcome bad performance issues. The same tips will help you with scalability too. The fewer resources we spend on each process, the more users the application can handle.

It is worth knowing that Azure and .NET Core web apps can be configured to handle scalability too. Let's check this out in the following subsections.

Creating a scalable web app in Azure

It is pretty simple to create a web app in Azure, ready for scaling. The reason why you have to do so is to be able to maintain different amounts of users during different seasons. The more users you have, the more hardware you will need. The following steps will show you how to create a scalable web application in Azure:

1. As soon as you log in to your Azure account, you will be able to create a new resource (web app, database, virtual machine, and so on), as you can see in the following screenshot:



2. After that, you can select **Web App**. This tutorial will take you to the following screen:

The screenshot shows the Microsoft Azure portal interface. On the left, there is a dark sidebar with various service icons and links: 'Create a resource', 'Home', 'Dashboard', 'All services', 'FAVORITES' (with 'All resources', 'Resource groups', 'App Services', 'Function Apps', 'SQL databases', 'Azure Cosmos DB', 'Virtual machines', 'Load balancers', 'Storage accounts', 'Virtual networks', 'Azure Active Directory', 'Monitor', 'Advisor', and 'Security Center'). The main content area has a title 'Web App' with a 'Create' button. It contains several input fields with validation stars: 'App name' (text input with placeholder 'Enter a name for your App' and suffix '.azurewebsites.net'), 'Subscription' (dropdown menu showing 'Pay-As-You-Go'), 'Resource Group' (radio buttons for 'Create new' or 'Use existing' with an input field), 'OS' (radio buttons for 'Windows' or 'Linux' with 'Windows' selected), 'Publish' (radio buttons for 'Code' or 'Docker Image' with 'Docker Image' selected), and 'App Service plan/Location' (dropdown menu showing 'wwtravelclub-plan(South Central ...)'). At the bottom are 'Create' and 'Automation options' buttons.

The required details are as follows:

- **App name:** As you can see, this is the URL that your web app will assume after its creation. The name is checked to ensure it is available.
- **Subscription:** This is the account that will be charged for all application costs.
- **Resource Group:** This is the collection of resources you can define to organize policies and permissions. You may specify a new resource group name or add the web app to a group specified during the definition of other resources.
- **OS:** This is the operating system that will host the web app. Both Windows and Linux may be used for ASP.NET Core projects.
- **Publish:** This parameter indicates whether the web app will be delivered directly or whether it is going to use Docker technology to publish content. Docker will be discussed in more detail in *Chapter 5, Applying a Microservice Architecture to Your Enterprise Application*.
- **App Service Plan/Location:** This is where you define the hardware plan that's used to handle the web app and the location of the servers. This choice defines application scalability, performance, and costs.
- **Application Insights:** This is a useful Azure toolset for monitoring and troubleshooting web apps.

Applications may be scaled in two conceptually different ways:

- Vertically (**Scale up**)
- Horizontally (**Scale out**)

Both of them are available in the web app settings, as you can see in the following screenshot:

The screenshot shows the Azure portal interface for an App Service named 'scalability-sample'. The left sidebar contains navigation links like Home, App Services, scalability-sample, Deployment slots (Preview), Deployment slots, Deployment options (Classic), Deployment Center, Settings, Application settings, Authentication / Authorizati..., Application Insights, Identity, Backups, Custom domains, SSL settings, Networking, Scale up (App Service plan) (highlighted with a red box), Scale out (App Service plan) (highlighted with a red box), and WebJobs. The main content area displays details about the app service, including Resource group (scalability-sample), Status (Running), Location (Central US), Subscription (Pay-As-You-Go), Subscription ID (846d4a74-e1b1-4896-b3b6-38d500f0e432), Tags, URL (https://scalability-sample.azurewebsites.net), App Service Plan (samples (Free: 0 Small)), FTP/deployment username (No FTP/deployment user set), FTP hostname (ftp://waws-prod-dm1-079.ftp.azurewebsites.windows.net), and FTTPS hostname (ftps://waws-prod-dm1-079.ftp.azurewebsites.windows.net). Below the details are three cards: 'Diagnose and solve problems' (Our self-service diagnostic and troubleshooting experience helps you identify and resolve issues with your web app.), 'Application Insights' (Application Insights helps you detect and diagnose quality issues in your apps, and helps you understand what your users actually do with it.), and 'App Service Advisor' (App Service Advisor provides insights for improving app experience on the App platform. Recommendations are sorted by freshness, priority and impact to your app.). At the bottom, there are two buttons: 'Http 5xx' and 'Data In'.

Let's checkout the two types of scaling.

Vertical scaling (Scale up)

Scale up means changing the type of hardware that will sustain your application. In Azure, you have the opportunity of starting with free-shared hardware and moving to an isolated machine in a few clicks.

By selecting this option, you have the opportunity to select more powerful hardware (machines with more CPUs, storage, and RAM). The following screenshot shows the user interface for scaling up a web app:

scalability-sample - Scale up (App Service plan)
App Service

»

Dev / Test
For less demanding workloads

Production
For most production workloads

Isolated
Advanced networking and scale

Recommended pricing tiers

S1
100 total ACU
1.75 GB memory
A-Series compute equivalent
247.01 BRL/Month (Estimated)

P1V2
Premium V2 is not supported for this scale unit. Please consider redeploying or cloning your app.
[Click to learn more.](#)

P2V2
Premium V2 is not supported for this scale unit. Please consider redeploying or cloning your app.
[Click to learn more.](#)

P3V2
Premium V2 is not supported for this scale unit. Please consider redeploying or cloning your app.
[Click to learn more.](#)

▼ See additional options s

Included features
Every app hosted on this App Service plan will have access to these features:

Custom domains / SSL
Configure and purchase custom domains with SNI and IP SSL bindings

Included hardware
Every instance of your App Service plan will include the following hardware configuration:

Azure Compute Units (ACU)
Dedicated compute resources used to run applications deployed in the App Service Plan. [Learn more](#)

Apply

Horizontal scaling (Scale out)

Scaling out means splitting all requests among more servers with the same capacity instead of using more powerful machines. The load on all the servers is automatically balanced by the Azure infrastructure. This solution is advised when the overall load may change considerably in the future since horizontal scaling can be automatically adapted to the current load. The following screenshot shows an automatic **Scale out** strategy defined by two simple rules, which is triggered by CPU usage:

The screenshot shows the Azure portal interface for managing an App Service plan named "scalability-sample". The left sidebar lists various settings like Application settings, Authentication / Authorization, and Scale up/out options. The "Scale out (App Service plan)" option is currently selected. The main pane displays the following configuration:

- Autoscale setting name:** Sample Autoscale
- Resource group:** scalability-sample
- Instance count:** 1
- Default Auto created scale condition:** (No specific condition shown)
- Delete warning:** The very last or default recurrence rule cannot be deleted. Instead, you can disable autoscale to.
- Scale mode:** Scale based on a metric Scale to a specific instance count
- Rules:**
 - Scale out:** When samples (Average) CpuPercentage > 70 Increase instance count by 1
 - Scale in:** When samples (Average) CpuPercentage < 40 Decrease instance count by 1
- Add a rule:** + Add a rule
- Instance limits:** Minimum 1, Maximum 2, Default 1

A complete description of all the available auto scale rules is beyond the purpose of this book. However, they are quite self-explanatory and the *Further reading* section contains links to the full documentation.



The **Scale out** feature is only available in paid service plans.

Creating a scalable web app with .NET Core

Among all the available frameworks for implementing web apps, ASP.NET Core ensures good performance, together with low production and maintenance costs. ASP.NET Core performance is comparable with the performance of Node.js, but production and maintenance costs are lower because of the usage of C# (which is a strongly typed and advanced pure object language) instead of JavaScript.

The steps that follow will guide you through the creation of an ASP.NET Core-based web app. All the steps are quite simple, but some details require particular attention.

First of all, during the web app's creation, you can choose between .NET Core Framework and .NET Framework. Pay attention, because only .NET Core can run on both Windows and cheaper Linux servers, while classic .NET runs only on Windows servers. On the other hand, with classic .NET, you will have access to a larger code base of legacy libraries that include both Microsoft and third-party packages.

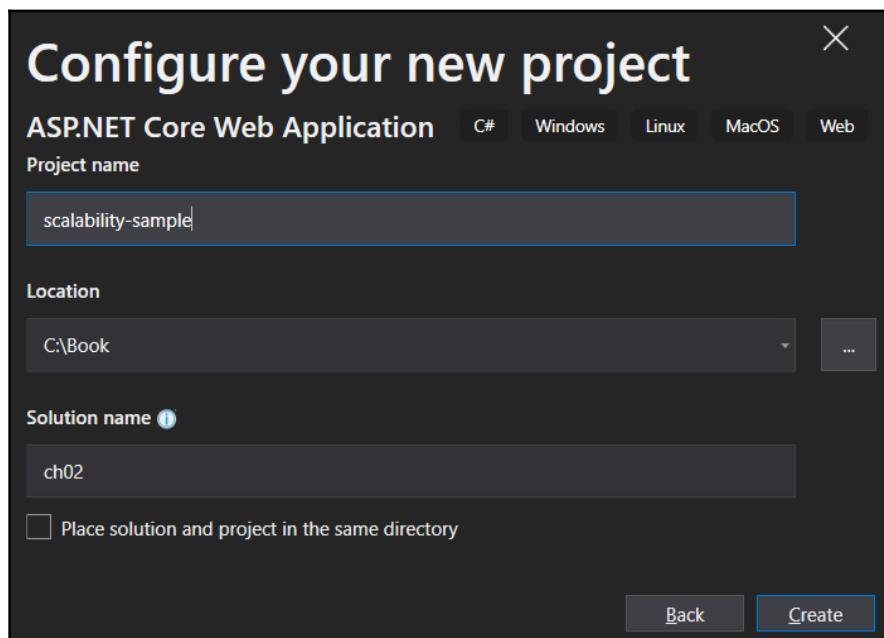
Nowadays, Microsoft recommends classic .NET, just in case the features you need are not available in .NET Core, or even when you deploy your web app in an environment that does not support .NET Core. In any other case, you should prefer .NET Core Framework because it allows you to do the following:

- Run your web app in Windows, Linux, or Docker containers
- Design your solution with microservices
- Have high performance and scalable systems

Containers and microservices will be covered in [Chapter 5, Applying a Microservice Architecture to Your Enterprise Application](#). There, you'll get a better understanding of the advantages of these technologies. For now, it is enough to say that .NET Core and microservices were designed for performance and scalability, which is why you should prefer .NET Core in all of your new projects.

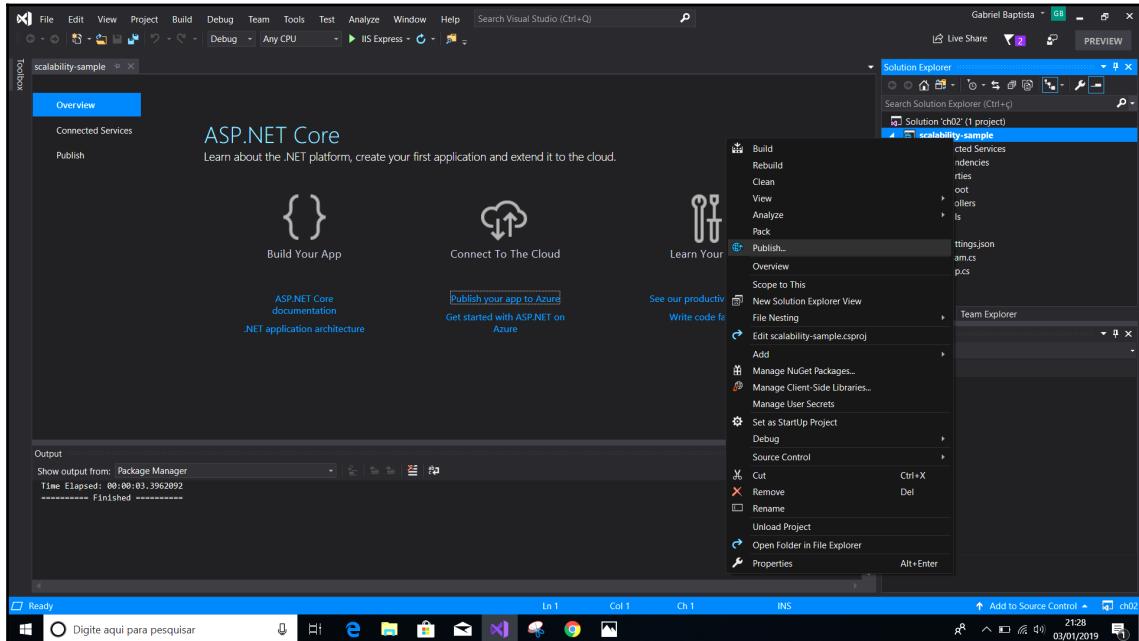
The following steps will show you how to create an ASP.NET Core web app in Visual Studio 2019 with .NET Core 3.0:

1. Once you select **ASP.NET Core Web Application**, you will be directed to a screen where you will be asked to set up the **Project name**, **Location**, and **Solution name**:

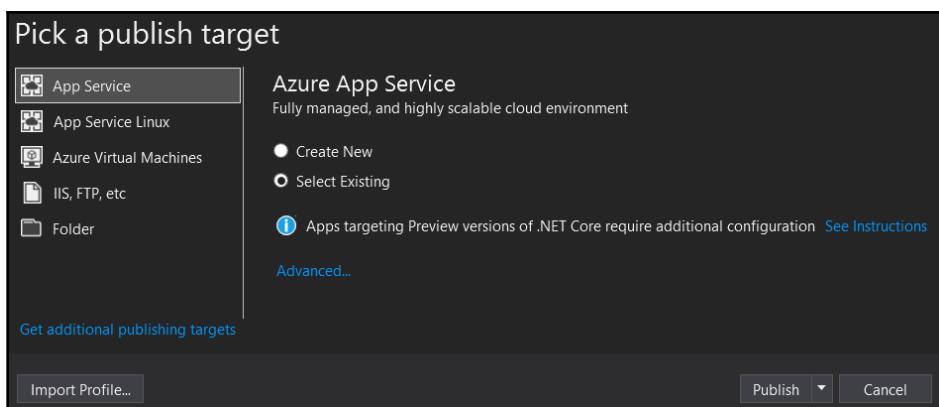


2. After that, you will be able to select the .NET Core version to use. At the time of writing, .NET Core 3.0 was still in its Preview 1 version.
3. Now that we are done with adding the basic details, you can connect your web app project to your Azure account and have it published.

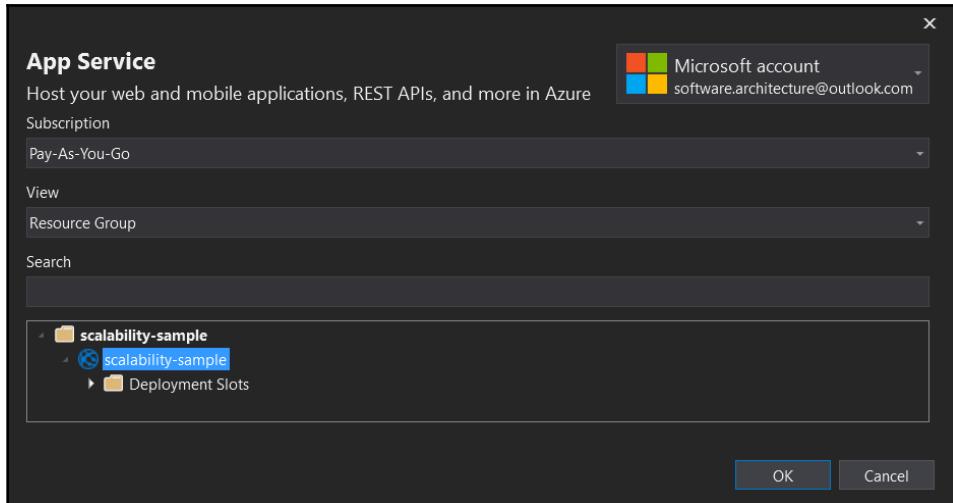
4. In the **Solution Explorer**, you have the option to **Publish...** if you right-click anywhere in there:



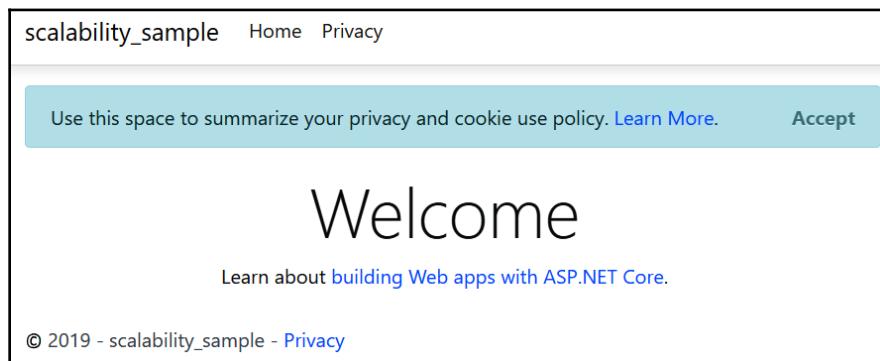
5. After you select the **Publish...** menu item, you will be able to connect your Azure account and then select the web app you wish to deploy:



6. There is full integration between Visual Studio and Azure. This gives you the opportunity to view all the resources you created in the Azure Portal in your development environment:



7. Once you've decided on your publish settings, that is, your publish profile, the web app is automatically published when you click **OK**:



For publishing .NET Core Preview versions, you have to add an extension in the web app setup panel in Azure portal, as shown in the following screenshot:

NAME	VERSION	UPDATE AVAILABLE
ASP.NET Core 3.0 (x64) Runtime	3.0.0-preview-1857...	No



For more information on deploying ASP.NET Core 3.0 to Azure App Service, please take a look at this link: <https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/azure-apps/?view=aspnetcore-2.2#deploy-aspnet-core-preview-release-to-azure-app-service>.

Here, we described the simplest ways to deploy a web app. Chapter 17, *Deploying Your Application with Azure DevOps*, will introduce you to the Azure DevOps **Continuous Integration/Continuous Delivery (CI/CD)** pipeline. This pipeline is a further Azure toolset that automates all the required steps to get the application in production, that is, build, testing, deployment in staging, and deployment in production.

Performance issues that need to be considered when programming in C#

Nowadays, C# is one of the most used programming languages all over the world, so good tips about C# programming are fundamental for the design of good architectures that satisfy the most common non-functional requirements.

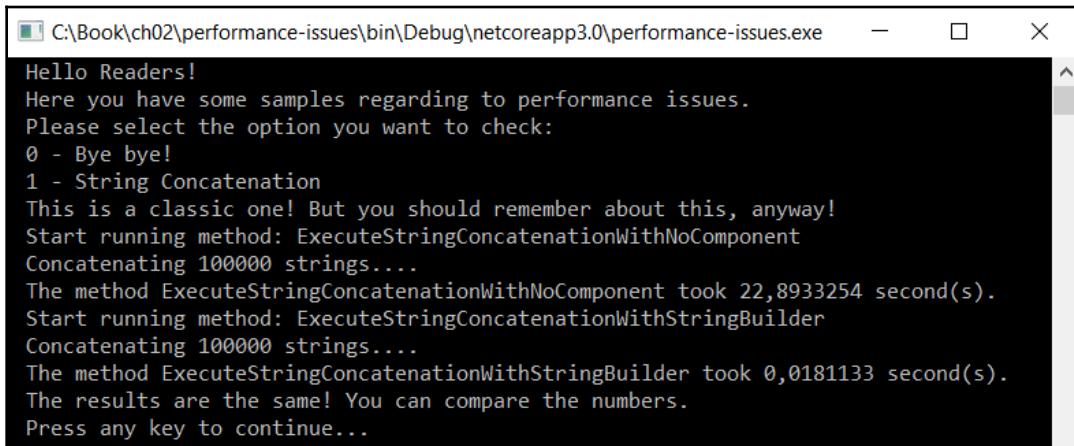
The following sections mention a few simple but efficacious tips—the associated code samples are available in the GitHub repository of this book.

String concatenation

This is a classic one! A naive concatenation of strings with the + string operator may cause serious performance issues since each time two strings are concatenated, their contents are copied into a new string.

So, if we concatenate, say, 10 strings that have an average length of 100, the first operation has a cost of 200, the second one has a cost of $200+100=300$, the third one has a cost of $300+100=400$, and so on. It is not difficult to convince yourself that the overall cost grows like $m*n^2$, where n is the number of strings and m is their average length. n^2 isn't too big for small n (say, $n < 10$), but it becomes quite big when n reaches the magnitude of 100-1,000, and actually unacceptable for magnitudes of 10,000-100,000.

Let's take a look at this with some test code, which compares naive concatenation with the same operation that's performed with the help of the `StringBuilder` class (the code is available in this book's GitHub repository):



The screenshot shows a terminal window with the following text output:

```
C:\Book\ch02\performance-issues\bin\Debug\netcoreapp3.0\performance-issues.exe
Hello Readers!
Here you have some samples regarding to performance issues.
Please select the option you want to check:
0 - Bye bye!
1 - String Concatenation
This is a classic one! But you should remember about this, anyway!
Start running method: ExecuteStringConcatenationWithNoComponent
Concatenating 100000 strings....
The method ExecuteStringConcatenationWithNoComponent took 22,8933254 second(s).
Start running method: ExecuteStringConcatenationWithStringBuilder
Concatenating 100000 strings....
The method ExecuteStringConcatenationWithStringBuilder took 0,0181133 second(s).
The results are the same! You can compare the numbers.
Press any key to continue...
```

If you create a `StringBuilder` class with something like `var sb = new System.Text.StringBuilder()`, and then you add each string to it with `sb.Append(currString)`, the strings are not copied; instead, their pointers are queued in a list. They are copied in the final string just once, when you call `sb.ToString()` to get the final result. Accordingly, the cost of `StringBuilder`-based concatenation grows simply as $m*n$.

Of course, you will probably never find a piece of software with a function like the preceding one that concatenates 100,000 strings. However, you need to recognize pieces of code similar to these ones where the concatenation of some 20-100 strings, say, in a web server that handles several requests simultaneously, might cause bottlenecks that damage your non-functional requirements for performance.

Exceptions

Always remember—exceptions take too much time to be handled! So, the usage of try-catch needs to be concise and essential; otherwise, you will create big performance issues.

The following two samples compare the usage of try-catch and `Int32.TryParse` to check whether a string can be converted into an integer, as follows:

```
private static string ParseIntWithTryParse()
{
    string result = String.Empty;
    int value;
    if (Int32.TryParse(result, out value))
        result = value.ToString();
    else
        result = "There is no int value";
    return $"Final result: {result}";
}

private static string ParseIntWithException()
{
    string result = String.Empty;
    try
    {
        result = Convert.ToInt32(result).ToString();
    }
    catch (Exception)
    {
        result = "There is no int value";
    }
    return $"Final result: {result}";
}
```

The second function doesn't look dangerous, but it is thousands of times slower than the first one:

C:\Book\ch02\performance-issues\bin\Debug\netcoreapp3.0\performance-issues.exe

```
Hello Readers!
Here you have some samples regarding to performance issues.
Please select the option you want to check:
0 - Bye bye!
1 - String Concatenation
2 - Exceptions
Always remember! Exceptions take too much time to handle!
Start running method: ParseIntWithException
The method ParseIntWithException took 0,0450462 second(s) (45,0462 ms)
Start running method: ParseIntWithTryParse
The method ParseIntWithTryParse took 5,6E-06 second(s) (0,00056 ms)
The results are the same! You can compare the numbers.
Press any key to continue...
```

To sum this up, exceptions must be used to deal with exceptional cases that break the normal flow of control, for instance, situations when operations must be aborted for some unexpected reasons, and control must be returned several levels up in the call stack.

Multithreading environments for better results – do's and don'ts

If you want to take advantage of all of the hardware that the system you're building provides, you have to use multithreading. This way, when a thread is waiting for an operation to complete, it can leave the CPU and other resources to other threads instead of wasting CPU time.

On the other hand, no matter how hard Microsoft is working to help with this, parallel code is not as simple as eating a piece of cake: it is error-prone and difficult to test and debug. The most important thing to remember as a software architect when you start considering using threads: *does your system require them?* Non-functional and some functional requirements will definitely answer this question for you.

As soon as you are sure that you need a multithreading system, you should decide on which technology is more adequate. There are a few options here, as follows:

- **Creating an instance of a System.Threading.Thread:** This is a classic way of creating threads in C#. The entirety of the thread life cycle will be in your hands. This is good when you are sure about what you are going to do, but you need to worry about every single detail of the implementation. The resulting code is hard to conceive and debug/test/maintain. So, to keep development costs acceptable, this approach should be confined to a few fundamental performance critique modules.
- **Programming using System.Threading.Tasks.Parallel and System.Threading.Tasks.Task classes:** In the .NET Framework 4.0 versions, you can use parallel classes to enable threads in a simpler way. This is good because you don't need to worry about the life cycle of the threads you create, but it will give you less control about what is happening in each thread.
- **Develop using asynchronous programming:** This is for sure the easiest way to develop multithreading applications since you don't need to care about thread coordination and deadlocks are not possible. When an asynchronous method calls another asynchronous method, it goes in sleeping mode to avoid wasting resources until the called task returns. This way, asynchronous code mimics the behavior of classical synchronous code while keeping most of the performance advantages of general parallel programming.

The overall behavior is deterministic and doesn't depend on the time taken by each task to complete, so non-reproducible bugs are not possible and the resulting code is easy to test/debug/maintain. Defining a method as an asynchronous task or not is the only choice left to the programmer; everything else is automatically handled by the runtime. The only thing you should be concern about is which methods should have asynchronous behavior.

Later on in this book, we will provide some simple examples of asynchronous programming. For more information about asynchronous programming and its related patterns, please check *Task-Based Asynchronous Patterns* in the Microsoft documentation (<https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap>).

No matter the option you choose, there are some do's and don'ts that, as a software architect, you have to pay attention to. These are as follows:

- **Do use concurrent collections** (`System.Collections.Concurrent`): As soon as you start a multithreading application, you have to use these collections. The reason for this is that your program will probably manage the same list, dictionary, and so on from different threads. The use of concurrent collections is the only option for developing thread-safe programs.
- **Do worry about static variables**: It is not possible to say that static variables are prohibited in multithreading development, but you should pay attention to them. Again, multiple threads taking care of the same variable can cause a lot of trouble. If you decorate a static variable with the `[ThreadStatic]` attribute, each thread will see a different copy of that variable, hence solving the problem of several threads competing on the same value. However, `ThreadStatic` variables can't be used for extra-thread communications since values written by a thread can't be read by other threads.
- **Do test system performance after multithreading implementations**: Threads give you the ability to take full advantage of your hardware, but in some cases, badly written threads can waste CPU time just doing nothing! Similar situations may result in almost 100% CPU usage and unacceptable system slowdowns. In some cases, the problem can be mitigated or solved by adding a simple `Thread.Sleep(1)` call in the main loop of some threads to prevent them from wasting too much CPU time, but you need to test this.
- **Do not consider multithreading easy**: Multithreading is not as simple as it seems in some syntax implementations. While writing a multithreading application, you should consider things such as the synchronization of the user interface, threading termination, and coordination. In many cases, programs just stop working well due to a bad implementation of multithreading.
- **Do not forget to plan the number of threads your system should have**: This is really important for 32-bit programs. There is a limitation regarding how many threads you can have in 32-bit environments. You should consider this when you are designing your system.
- **Do not forget to end your threads**: If you do not have the correct termination procedure for each thread, you will probably have trouble with memory and handles leaks.

Usability – why inserting data takes too much time

Scalability, performance tips, and multithreading are the main tools we can use to tune machine performance. However, the effectiveness of the system you design depends on the overall performance of the whole processing pipeline, which includes both humans and machines.

As a software architect, you can't improve the performance of humans, but you can improve the performance of man-machine interaction by designing an effective **user interface (UI)**, that is, user interfaces that ensure a fast interaction with humans, which, in turn, means the following:

- The UI must be easy to learn to reduce the time that's needed for learning and time waste before the target users learn to operate quickly. This constraint is fundamental if UI changes are frequent, and for public websites that need to attract the greatest possible number of users.
- The UI must not cause any kind of slowdown in data insertion; data insertion speed must be limited just by the user's ability to type, not by system delays or by additional gestures that could be avoided.

The following are a few simple tips when it comes to designing *easy to learn* user interfaces:

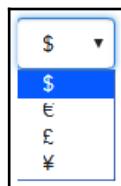
- Each input screen must state its purpose clearly.
- Use the language of the user, not the language of developers.
- Avoid complications. Design the UI with the average case in mind; more complicated cases can be handled with extra inputs that appear only when needed. Split complex screens into more input steps.
- Use past inputs to understand user intentions and to put users on the right paths with messages and automatic UI changes; for instance, cascading drop-down menus.
- Error messages are not bad notes the system gives to the user, but they must explain how to insert correct input.

Fast user interfaces result from efficacious solutions to the following three requirements:

- Input fields must be placed in the order they are usually filled, and it should be possible to move to the next input with the *Tab* or *Enter* key. Moreover, fields that often remain empty should be placed at the bottom of the form. Simply put, the usage of the mouse while filling a form should be minimized. This way, the number of user gestures is kept to a minimum. In a web application, once the optimal placement of input fields has been decided, it is enough to use the `tabindex` attribute to define the right way users move from one input field to the next with the *Tab* key.
- System reactions to user inputs must be as fast as possible. In particular, error (or information) messages must appear as soon as the user leaves the input field. The simplest way to achieve this is to move most of the help and input validation logic to the client side so that system reactions don't need to pass through both communication lines and servers.
- Efficacious selection logic. Selecting an existing item should be as easy as possible; for example, selecting one of some thousands of products in an offer must be possible with a few gestures and with no need to remember the exact product name or its barcode. The next subsection analyzes techniques we can use to increase complexity to achieve fast selection.

Designing fast selection logic

When all the possible choices are in the order of magnitude of 1-50, the usual drop-down menu is enough. For instance, check the currency selection drop-down menu:



When the order of magnitude is higher but less than a few thousand, an autocomplete that shows the names of all the items that start with the characters typed by the user is usually a good choice:



A similar solution can be implemented with a low computational cost since all the main databases can efficiently select strings that start with a given substring.

When names are quite complex, when searching for the characters that were typed in by the user, they should be extended inside each item string. This operation can't be performed efficiently with usual databases and requires ad hoc data structures.

Finally, when we are searching inside descriptions composed of several words, more complex search patterns are needed. This is the case, for instance, of product descriptions. If the chosen database supports full-text search, the system can search for the occurrence of several words that have been typed by the user inside all the descriptions efficiently.

However, when descriptions are made up of names instead of common words, it might be difficult for the user to remember a few exact names contained in the target description. This happens, for instance, with multi-country company names. In these cases, we need algorithms that find the best match for the character that was typed by the user. Substrings of the string that was typed by the user must be searched in different places of each description. In general, similar algorithms can't be implemented efficiently with databases based on indexes but require all the descriptions to be loaded in memory and ranked somehow against the string that was typed by the user.

The most famous algorithm in this class is probably the **Levenshtein** algorithm, which is used by most spell checkers to find a word that best fits the mistyped one by the user. This algorithm minimizes the Levenshtein distance between the description and the string typed by the user, that is, the minimum number of character removals and additions needed to transform one string into another.

The Levenshtein algorithm works great but has a very high computational cost. Now, we give a faster algorithm that works well for searching character occurrences in descriptions. Characters typed by the user don't need to occur consecutively in the description but must occur in the same order. Some characters may miss. Each description is given a penalty that depends on the missing characters and on how the occurrences of the characters typed by the user are far from the others. More specifically, the algorithm ranks each description with two numbers:

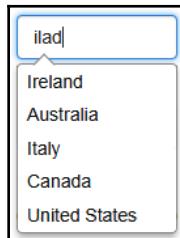
- The number of characters typed by the user that occurs in the description: The more characters contained in the description, the higher its rank.
- Each description is given a penalty equal to the total distance among the occurrences of the characters typed by the user in the description.

The following screenshot shows how the word **Ireland** is ranked against the string **ilad**, which was typed by the user:



The number of occurrences is four, while the total distance among characters occurrences is three.

Once all the descriptions have been rated they are sorted according to the number of occurrences. Descriptions with the same number of occurrences are sorted according to the lowest penalties. The following is an autocomplete that implements the preceding algorithm:



The C# code that ranks each description against the string typed by the user is as follows:

```
public class SmartDictionary<T>
{
    ...
    private Func<T, string> keyAccessor;
    protected class Rater
    {
        public T Item;
        public double Penalty=0;
        public int FoundChars=0;
    }
    ...
    protected Rater RateItem(string search, Rater x)
    {
        var toSearch = search.ToLower();
        var destination = keyAccessor(x.Item).ToLower();
        bool firstMatch = true;
        for (var j = 0; j < toSearch.Length; j++)
        {
            if (destination == string.Empty) return x;
            var currChar = toSearch[j];
            var index = destination.IndexOf(currChar);
            if (index == -1) continue;
            x.FoundChars++;
            if (firstMatch)
            {
                x.Penalty += index;
                firstMatch = false;
            }
            else x.Penalty += index*1000;
            if (index + 1 < destination.Length)
                destination = destination.Substring(index + 1);
            else
                destination = string.Empty;
        }
        return x;
    }
    ...
}
```

The item to rank is inserted in a `Rater` instance. Then, its string description is extracted by a `keyAccessor` function. After that, the code computes both character occurrences and occurrences in the total distance.

The full class code, along with a test console project, is available in this book's GitHub repository.

Selecting from a huge amount of items

Here, huge doesn't refer to the amount of space needed to store the data, but to the difficulty the user has in remembering the features of each item. When an item must be selected from among more than 10,000-100,000 items, there is no hope to find it by searching for character occurrences inside a description. Here, the user must be driven toward the right item through a hierarchy of categories.

In this case, several user gestures are needed to perform a single selection. In other word, each selection requires interaction with several input fields. Once it's decided that the selection can't be done with a single input field, the simplest option is cascading drop-down menus, that is, a chain of drop-down menus whose selection list depends on the values that were selected in the previous drop-down menus.

For example, if the user needs to select a town located anywhere in the world, we may use the first drop-down menu to select the country, and once the country has been chosen, we may use this choice to populate a second one with all the towns in the selected country. A simple example is as follows:

A screenshot of a web form illustrating a cascading dropdown menu. The form has two main sections: 'Country:' and 'City:'. The 'Country:' section contains a dropdown menu with 'Brazil' selected. The 'City:' section contains a dropdown menu with 'Select City' highlighted in blue. Below the dropdowns is a blue button labeled '✓ Submit'. The dropdown menu also lists 'Rio de Janeiro' and 'Salvador'.

Clearly, each drop-down menu can be replaced by an autocomplete when required due to having a high number of options.

If making the right selection can be done by intersecting several different hierarchies, cascading drop-down menus become inefficient too, and we need a filter form, as follows:

The dialog box is titled "Filter". It has four sections: "Name" (dropdown "starts with" and input field), "Package Type" (dropdown "ends with" and input field "Syrup"), "Unit Price" (dropdown "<" and input field "10"), and "Supplier" (dropdown "=" and dropdown menu showing "Exo" and "Exotic Liquids"). At the bottom are "apply filter" and "reset" buttons.

Now, let's understand interoperability with .NET Core.

The fantastic world of interoperability with .NET Core

.NET Core brought Windows developers the ability to deliver their software into various platforms. And you, as a software architect, need to pay particular attention to this. Linux and macOS are no longer a problem for C# lovers—it's much better than that—they are really good opportunities to deliver to new customers. Therefore, we need to ensure performance and multi-platform support, two common non-functional requirements in several systems.

Both console applications and web apps designed with .NET Core in Windows are almost completely compatible with Linux and macOS, too. This means you do not have to build the app again to run it on these platforms. Also, very platform-specific behaviors now have multi-platform support, as shown, for instance, by the `System.IO.Ports.SerialPort` class, which, starting from .NET Core 3.0, is on Linux.

Microsoft offers scripts to help you install .NET Core on Linux and macOS. You can find them at <https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-install-script>. Once you have the SDK installed, you just need to call **dotnet** the same way you do in Windows.

However, you must be aware of some features that are not fully compatible with Linux and macOS systems. For instance, no equivalent to the Windows Registry exists in these OSes and you have to develop an alternative yourself. If needed, an encrypted JSON config file can be a good option.

Another important point is that Linux is case-sensitive, while Windows is not. Please, remember this when you work with files. Another important thing is that the Linux path separator is different from the Windows separator. You can use the `Path.PathSeparator` property and all the other `Path` class methods to ensure your code is actually multi-platform.

Besides, you can also adapt your code to the underlying OS by using the runtime checks provided by .NET Core, as follows:

```
using System;
using System.Runtime.InteropServices;

namespace CheckOS
{
    class Program
    {
        static void Main()
        {
            if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
                Console.WriteLine("Here you have Windows World!");
            else if (RuntimeInformation.IsOSPlatform(OSPlatform.Linux))
                Console.WriteLine("Here you have Linux World!");
            else if (RuntimeInformation.IsOSPlatform(OSPlatform.OSX))
                Console.WriteLine("Here you have macOS World!");
        }
    }
}
```

Creating a service in Linux

The following script can be used to encapsulate a command-line .NET Core app in Linux. The idea is that this service works like a Windows Service. This can be really useful, considering that most Linux installations are command-line only and run without a user logged in:

1. The first step is to create file that will run the command-line app. The name of the app is `app.dll` and it is installed in `appfolder`. The application will be checked every 5,000 milliseconds. This service was created on a CentOS 7 system. Using a Linux Terminal, you can type this:

```
cat > sample.service <<EOF
[Unit]
Description=Your Linux Service
After=network.target
[Service]
ExecStart=/usr/bin/dotnet $(pwd)/appfolder/app.dll 5000
Restart=on-failure
[Install]
WantedBy=multi-user.target
EOF
```

2. Once the file has been created, you have to copy the service file to a system location. After that, you have to reload `systemd` and enable the service so that it will restart on reboots:

```
sudo cp sample.service /lib/systemd/system
sudo systemctl daemon-reload
sudo systemctl enable sample
```

3. Done! Now, you can start, stop, and check the service using the following commands. The whole input that you need to provide in your command-line app is as follows:

```
# Start the service
sudo systemctl start sample

# View service status
sudo systemctl status sample

# Stop the service
sudo systemctl stop sample
```

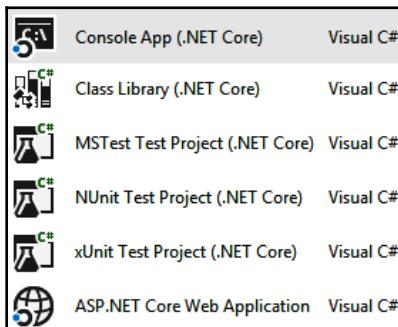
Now that we've learned about a few concepts, let's learn how to implement them in our use case.

Book use case – understanding the main types of .NET Core projects

The development of this book's use case will be based on various kinds of .NET Core Visual Studio projects. This section describes all of them. Let's select **New project** in the Visual Studio file menu. In the window that opens, all the .NET Core projects will be located under the **.NET Core**, **.NET Standard**, and **Cloud** items in the left-hand menu:



Most of them are available under **.NET Core**:



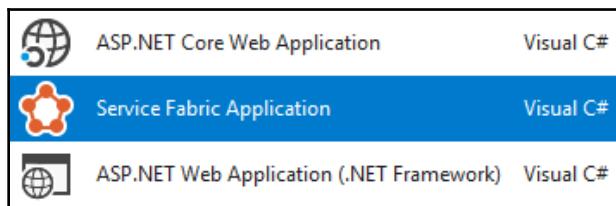
Here, we have a console project, a class library project, and various types of test projects, each based on a different test framework: xUnit, nUnit, and MSTest. Choosing among the various testing frameworks is just a matter of preference since all of them offer comparable features. Adding tests to each piece of software that composes a solution is a common practice and allows software to be modified frequently without jeopardizing its reliability.

Testing will be discussed in detail in Chapter 15, *Testing Your Code with Unit Test Cases and TDD*, and Chapter 20, *Automation for Software Testing*. Finally, we have the ASP.NET Core application we already described in the *Creating a scalable web app with .NET Core* subsection. There, we defined an ASP.NET MVC application, but Visual Studio also contains project templates for projects based on RESTful APIs and the most important single-page application frameworks such as Angular, React, Vue.js, and the new Blazor framework based on WebAssembler. Some of them are available with the standard Visual Studio installation; others require the installation of a SPA package.

For each project type, we can choose the .NET Core version we would like to use. Under the **.NET Standard** menu item, we have only a class library project. .NET Standard class libraries are based on .NET standards instead of a specific .NET Core version so they are compatible with several .NET Core versions. For instance, libraries based on 2.0 standards are compatible with all .NET Core versions greater than or equal to 2.0, and with all .NET Framework versions greater than 4.6.

This compatibility advantage comes at the price of having less available features. However, features that are not a part of a standard can be added as references to additional library packages.

Finally, under the cloud menu, we have several more project types, but the only new project related to .NET Core is the **Service Fabric Application**:



This allows us to define microservices. Microservice-based architectures allow an application to be split into several independent microservices. Several instances of the same microservice can be created and distributed across several machines to fine-tune the performance of each application part. Microservices will be described in Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*.

Summary

Functional requirements that describe system behavior must be completed with non-functional requirements that constrain system performance, scalability, interoperability, and usability. Performance requirements come from response-time and system load requirements. As a software architect, you should ensure you have the required performance with the minimum cost building efficient algorithms and taking full advantage of the available hardware resources with multithreading.

Scalability is the capability of a system to be adapted to an increasing load. Systems can be scaled vertically by providing more powerful hardware, or horizontally by replicating and load balancing the same hardware. The cloud, in general, and Azure, in particular, can help us implement strategies dynamically, with no need to stop your application.

Tools such as .NET Core that run on several platforms can ensure interoperability, that is, the capability of your software to run on different target machines and with different operating systems (Windows, Linux, macOS, Android, and so on).

Usability is ensured by taking care of the input field's order, the effectiveness of the item selection logic, and how easy your system is to learn.

In the next chapter, you will learn how Azure DevOps tools can help us when it comes to collecting, defining, and documenting our requirements.

Questions

1. Which are the two conceptual ways to scale a system?
2. Can you deploy your web app automatically from Visual Studio to Azure?
3. What is multithreading useful for?
4. What are the main advantages of the asynchronous pattern over other multithreading techniques?
5. Why is the order of input fields so important?
6. Why is the .NET Core Path class so important for interoperability?
7. What is the advantage of a .NET standard class library over a .NET Core class library?
8. List the various types of .NET Core Visual Studio projects.

Further reading

The following are some books and links you may consider reading to gather more information about this chapter:

- <https://www.packtpub.com/virtualization-and-cloud/hands-azure-developers>
- <https://docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling>
- <https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/azure-apps/?view=aspnetcore-2.2#deploy-aspnet-core-preview-release-to-azure-app-service>
- <https://docs.microsoft.com/en-us/dotnet/standard/parallel-processing-and-concurrency>

3

Documenting Requirements with Azure DevOps

Azure DevOps is the evolution of Visual Studio Team Services and there is a variety of new features that can help developers to document and organize their software. The purpose of this chapter is to present an overview of this tool provided by Microsoft.

The following topics will be covered in this chapter:

- Creating an Azure DevOps project using your Azure account
- Understanding the functionalities offered by Azure DevOps
- Organizing and managing requirements using Azure DevOps
- Presenting use cases in Azure DevOps

Technical requirements

This chapter requires you to create a new free Azure account or use an existing one. The *Creating an Azure account* section of Chapter 1, *Understanding the Importance of Software Architecture*, explains how to create one.

Introducing Azure DevOps

Azure DevOps is a Microsoft **Software as a Service (SaaS)** platform that enables you to deliver continuous value to your customers. By creating an account there, you will be able to easily plan your project, store your code safely, test it, publish the solution to a staging environment, and then publish the solution to the actual production infrastructure.

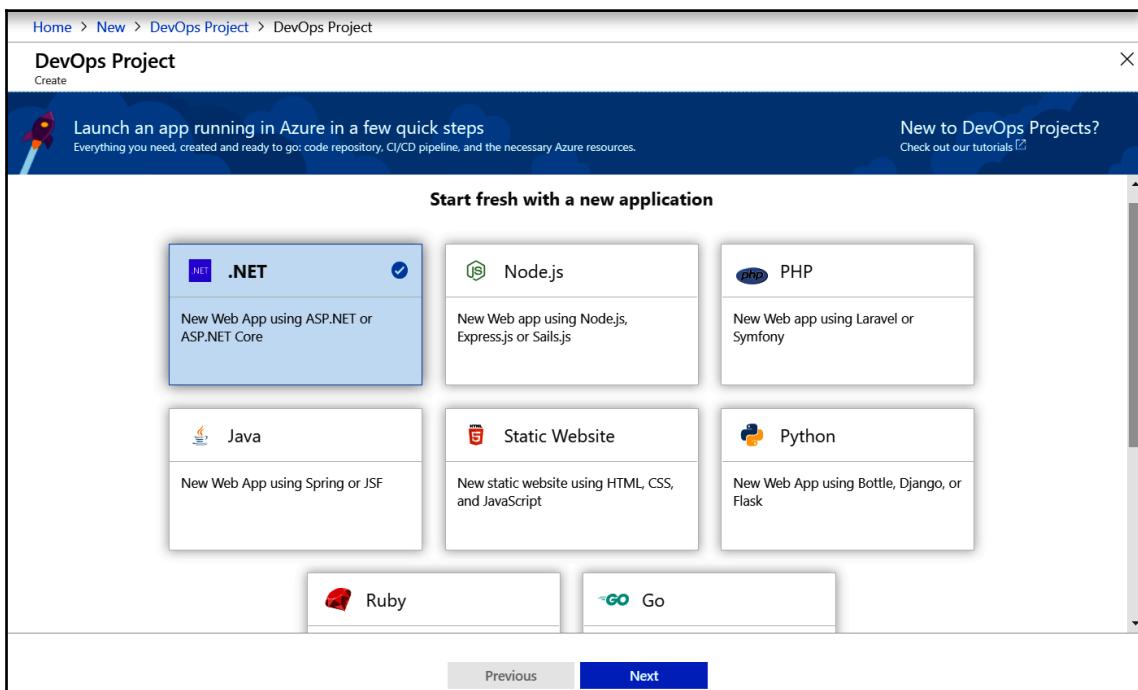
Of course, Azure DevOps is a complete framework and the ecosystem that it provides for software development is currently available. The automation of all the steps involved in software production ensures the continuous enhancement and improvement of an existing solution in order to adapt it to market needs.

You can start the process moving in your Azure portal. If you don't know how to create an Azure portal account, then please check [Chapter 1, Understanding the Importance of Software Architecture](#). The steps to create an Azure DevOps account are quite simple:

1. Select **Create a resource** and then **DevOps Project**:

The screenshot shows the Microsoft Azure portal interface. On the left, there is a dark sidebar with various service icons and links: Create a resource, All services, FAVORITES, Dashboard, All resources, Resource groups, App Services, Function Apps, SQL databases, Azure Cosmos DB, Virtual machines, Load balancers, Storage accounts, Virtual networks, Azure Active Directory, and Monitor. The main content area has a light background. At the top, it says "DevOps Project - Microsoft Azure" and the URL "https://portal.azure.com/?l=en-us#create/hub". Below that, it says "Microsoft Azure" and "Search resources, services, and docs". The main title is "DevOps Project" by Microsoft. A sub-section titled "Launch an app running in Azure in a few quick steps" is shown. It describes what a DevOps Project is and lists its benefits. The "Key benefits of DevOps Project:" section includes a bulleted list: Get up and running with a new app and a full DevOps pipeline in just a few minutes; Support for a wide range of popular frameworks such as .NET, Java, PHP, Node, and Python; Start fresh or bring your own application from Github; Built-in Application Insights integration for instant analytics and actionable insights; Cloud-powered CI/CD using Azure Pipelines and Azure Repos. Another section below discusses how DevOps Projects are powered by Azure Pipelines. At the bottom of the main content area is a blue "Create" button.

2. As soon as you start the wizard for creating the project, you can choose how you want to deliver your system from several different platforms. This is one of the greatest advantages of Azure DevOps as you are not limited to Microsoft tools and products, but you can from all common platforms, tools, and products available on the market:



3. The options available will depend on the platform chosen in the first step. In some cases, you can choose from several deployment options, as you can see in the following screenshot, which appears if you select the .NET platform:

The screenshot shows the 'DevOps Project' creation wizard at step 3, titled 'Select an Azure service to deploy the application'. The steps are represented by a horizontal bar with four circles: 'Runtime' (green checkmark), 'Framework' (green checkmark), 'Service' (blue circle with '3'), and 'Create' (dark grey circle with '4'). Below the bar, the text 'Select an Azure service to deploy the application' is displayed.

The 'Service' step displays six deployment options:

- Kubernetes Service**: Fully managed Kubernetes container orchestration service for managing containers without container expertise.
- Service Fabric Cluster**: A distributed systems platform to deploy and manage scalable and reliable microservices and containers.
- Windows Web App**: Fully managed compute platform on Windows for web applications and websites. This option is selected, indicated by a checked checkbox icon.
- Linux Web App**: Fully managed compute platform on Linux for web applications and websites.
- Web App for Containers**: Fully managed compute platform on Linux for deploying and running containerized web applications.
- Virtual machine**: Windows virtual machine to run your app.

At the bottom, a message says 'Don't see a service you're looking for? We're continuously adding support for more Azure services and app frameworks.' followed by a 'Learn more' link. Navigation buttons 'Previous' and 'Next' are also present.

- Once the setup is complete, you will be able to manage the project using the project portal according to the information you provided. It is worth mentioning that this wizard will create an Azure DevOps Service if you do not have one. The Azure DevOps organization is where you can organize all of your Azure DevOps projects. The whole process takes less than 20 minutes:

Home > New > DevOps Project > DevOps Project

DevOps Project

Create

Runtime Framework Service Create

Almost there!
Ready to deploy ASP.NET Core web app to a new Windows Web App.

* Project name
 ✓

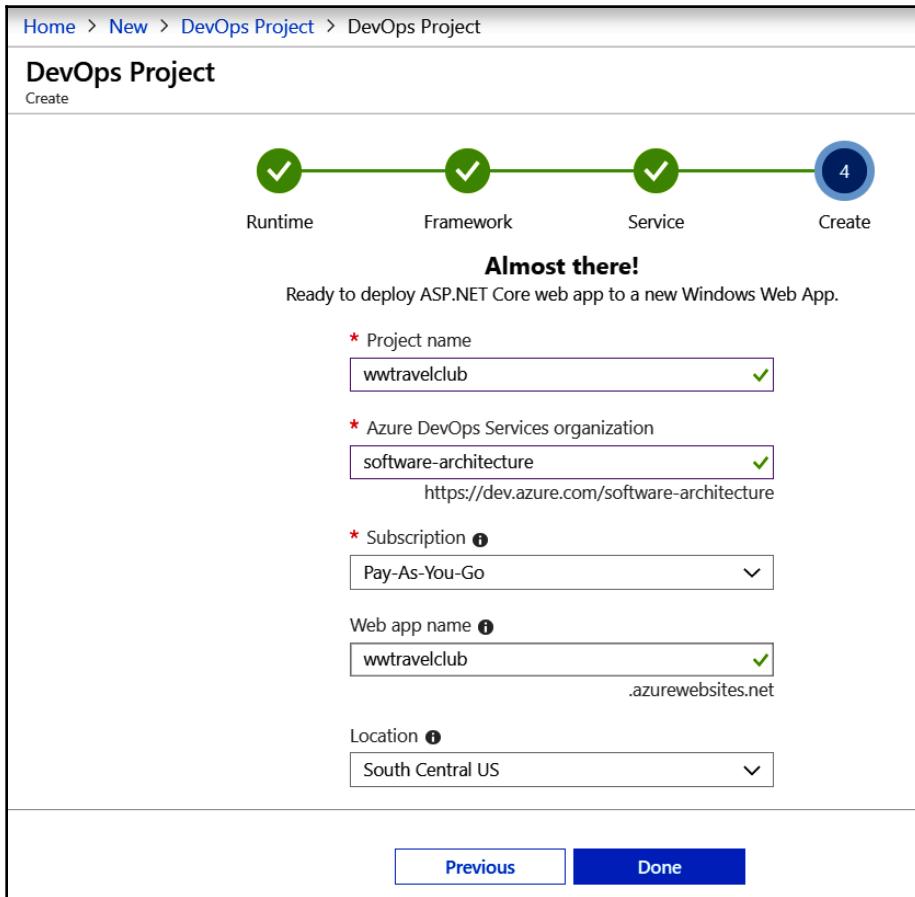
* Azure DevOps Services organization
 ✓
<https://dev.azure.com/software-architecture>

* Subscription ⓘ
 ▾

Web app name ⓘ
 ✓
.azurewebsites.net

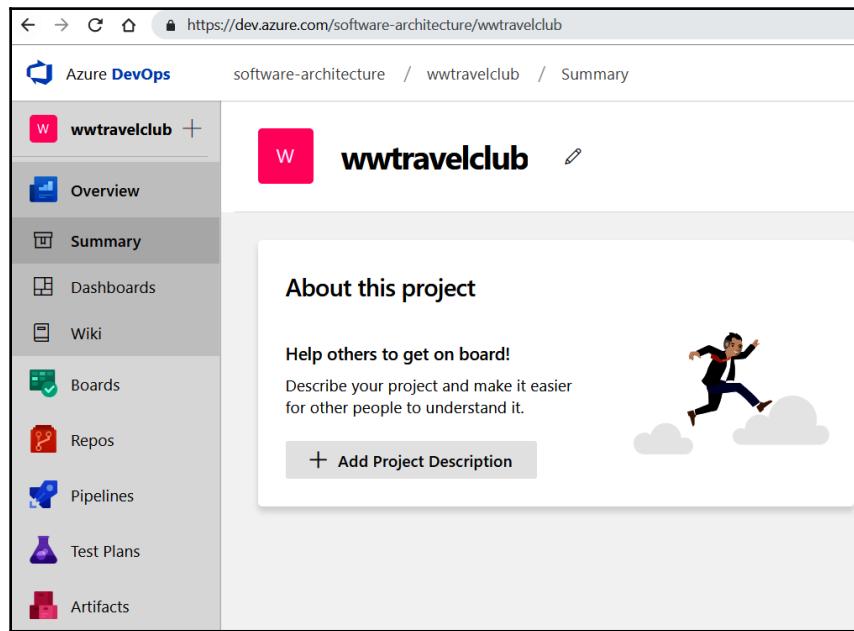
Location ⓘ
 ▾

[Previous](#) [Done](#)



The screenshot shows the 'Create' step of the DevOps Project wizard. A progress bar at the top indicates three steps are completed (Runtime, Framework, Service) and one step remains (Create). Below the progress bar, a message says 'Almost there!' and 'Ready to deploy ASP.NET Core web app to a new Windows Web App.' The form fields are filled with 'wwtravelclub' for the project name, 'software-architecture' for the organization, 'Pay-As-You-Go' for the subscription, and 'wwtravelclub' for the web app name. The location is set to 'South Central US'. At the bottom, there are 'Previous' and 'Done' buttons.

5. After that, you will be able to start planning your project. The following screenshot shows the page that appears once the Azure DevOps project creation is complete. In the remainder of this book, we will come back to this page several times to introduce and describe various useful features that ensure a faster and efficacious deployment:

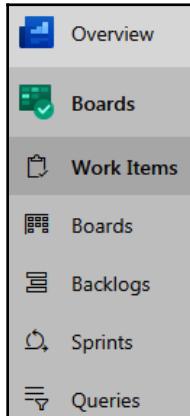


As you can see from the preceding screenshot, the process for creating an Azure DevOps account and starting to develop the best-in-class DevOps tool is quite simple. It is worth mentioning that you can start using this fantastic tool at no cost, considering you have up to five developers on your team, plus any number of stakeholders.

Organizing your work using Azure DevOps

DevOps is a **Continuous Integration/Continuous Deployment (CI/CD)** methodology, that is, a set of best practices on how to apply continuous improvements to a software application and how to deliver them to the production environment. Azure DevOps is a powerful tool whose range of applications encompasses all the steps involved in both the initial development of an application and in its subsequent CI/CD process.

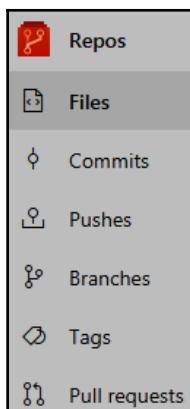
Azure DevOps contains tools for collecting requirements and for organizing the whole development process. They can be accessed by clicking the **Boards** menu in the Azure DevOps page and will be described in more detail in the next two sections:



All other functionalities available in Azure DevOps are briefly reviewed in the following subsections. They will be discussed in detail in [Chapter 15, Testing Your Code with Unit Test Cases and TDD](#), to [Chapter 20, Automation for Software Testing](#).

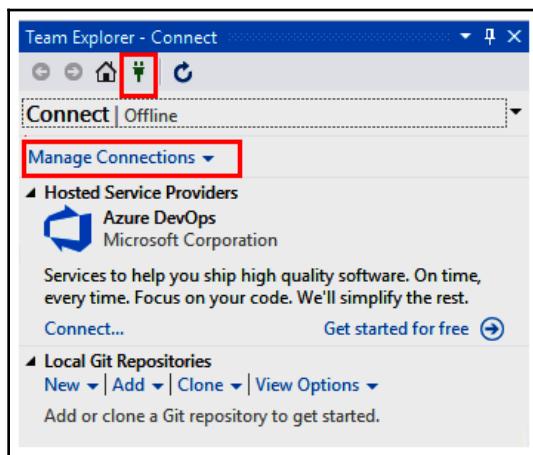
Azure DevOps repository

The **Repos** menu item gives you access to a Git repository in order to place the project code:

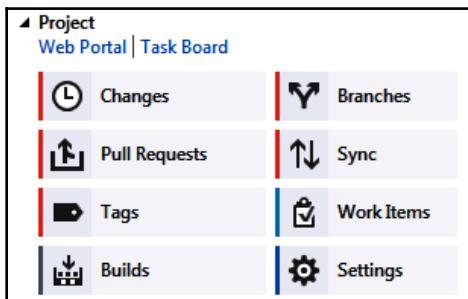


You can connect to this repository from inside Visual Studio in a very simple way:

1. Start Visual Studio and ensure you are logged in to Visual Studio with the same Microsoft account used to define your DevOps project (or used to add you as a team member).
2. Prepare a Visual Studio solution containing the Visual Studio projects you need in your application (further projects can be added during development) if your DevOps project repository is empty.
3. Select the **Team Explorer** tab and then click the connection button:



4. Clicking the **Connect...** link of **Azure DevOps**, you will be driven to connect with one of your Azure DevOps projects.
5. Click the Team Explorer **Home** button. Now, you will see commands for performing Git operations and for interaction with other Azure DevOps areas:

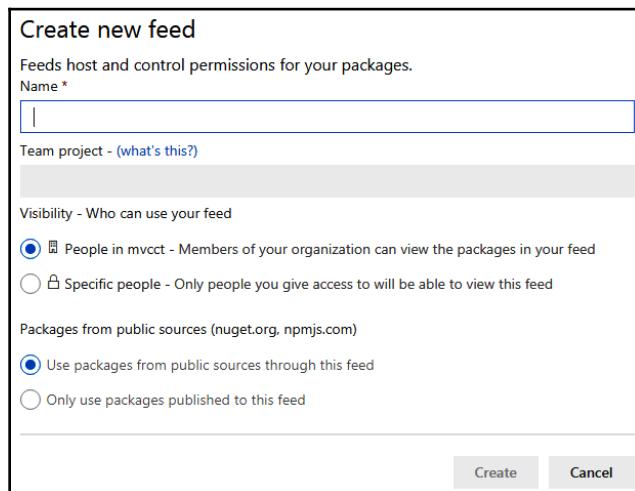


6. Commit the solution you just created by clicking the **Changes** button if the Azure DevOps repository is empty and then following the subsequent instructions.
7. Click the **Sync** button to synchronize your local repository with the remote Azure DevOps repository. If the remote repository is empty and you just created a solution, this action will initialize the remote Azure DevOps repository with this solution; otherwise, this action will download the remote repository on your local machine.
8. Once all team members have initialized both their local machine repositories and the Azure DevOps repository with the preceding steps, it is enough to open Visual Studio. The solution created in your local repository will appear in the bottom area of the **Team Explorer** window.
9. Click the window to open the solution on your local machine. Then, synchronize with the remote repository to ensure the code you are modifying is up to date.

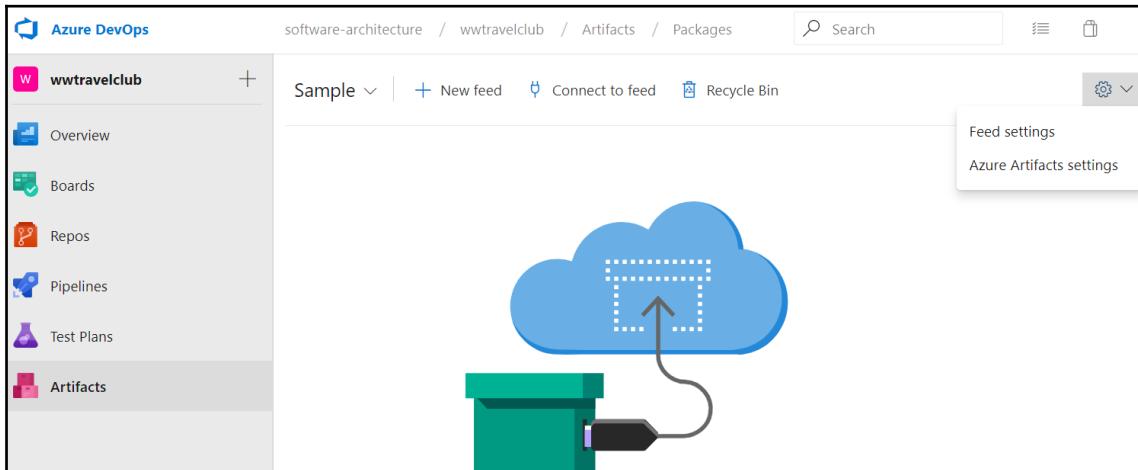
The **Team Explorer** menu enables you to execute most Git commands, to launch remote builds (**Builds** button) and to interact with other Azure DevOps areas (see, for instance, the **Work Items** button).

Package feeds

The **Artifacts** menu handles the software packages used by the project. There, you may define feeds for basically all types of packages such as NuGet, Node.js, and Python. Once in the **Artifacts** area, you may create several feeds by clicking the **+ New Feed** link, where each feed can handle several kinds of packages:

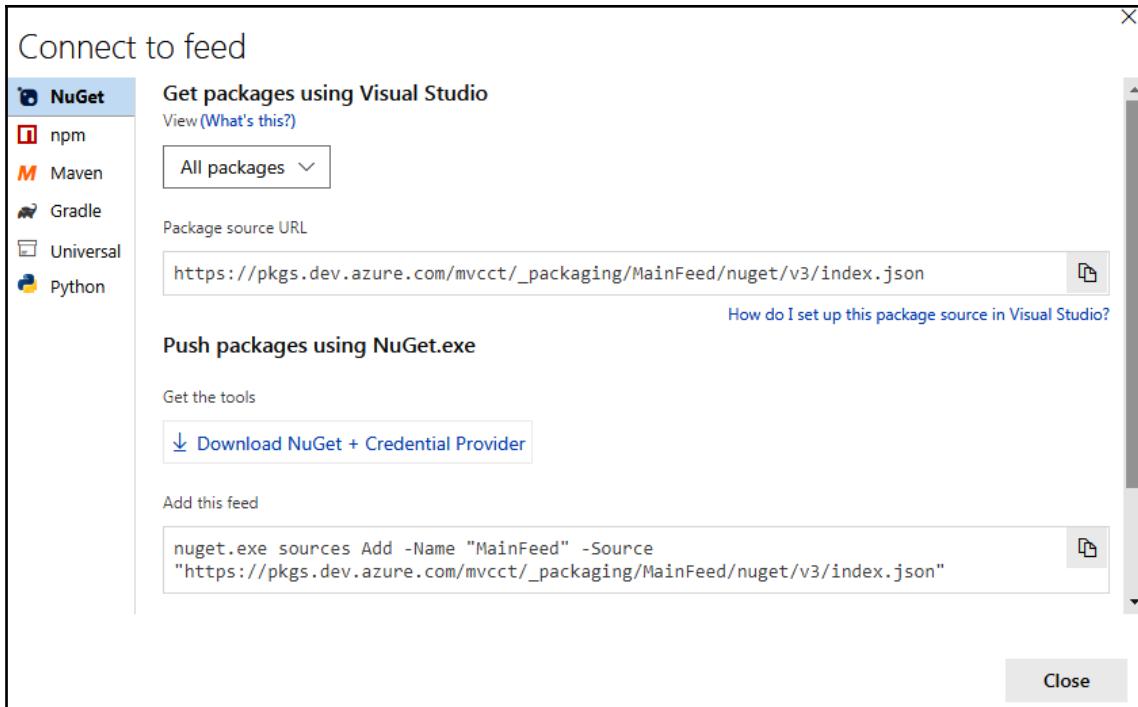


If you select the option to connect to packages from public sources, as a default, the feed connects to npmjs, nuget.org, and pypi.org. However, you can go to the upstream sources tab in the **Feed settings** section through the menu in the upper-right corner and remove/add package sources:



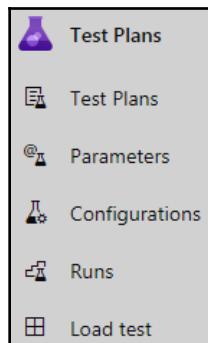
The **Connect to feed** link of each feed shows a window that, for each package type, explains how to do the following things:

1. Upload private packages to the feed. This way each team can use its private package's code base.
2. Connect to the feed to use its packages from Visual Studio. You should add all the project feeds to your Visual Studio feeds in order to also use the private team packages uploaded in the feed; otherwise, your local build will fail.
3. Manage credentials to access the feed:



Test plans

The **Test Plans** section allows you to define the test plans you want to use and their settings. It can be accessed through the **Test Plans** menu item:



Here, you may define, execute, and track test plans made of the following:

- Manual acceptance tests
- Automatic unit tests
- Load tests

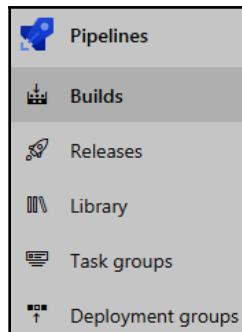
Automatic unit tests must be defined in test projects contained in the Visual Studio solution and based on a framework such as NUnit, xUnit, and MSTest (Visual Studio has project templates for all of them). **Test Plans** gives you the opportunity to execute these tests on Azure and to define the following:

- A number of configuration settings
- When to execute them
- How to track them and where to report their results in the overall project documentation

For manual tests, you may define complete instructions for the operator, the environment in which to execute them (for example, an operating system), and where to report their results in the project documentation. You can also define how to execute load tests, how to measure results, and where to report them.

Pipelines

Pipelines are automatic action plans that specify all steps from the code build until the software deployment is in production. They can be defined in the pipelines area, which is accessible through the **Pipelines** menu item:



There, you can define a complete pipeline of tasks to execute alongside their triggering events, which encompasses steps such as code building, launching test plans, and what to do after the tests are passed.

Typically, after the tests are passed, the application is automatically deployed in a staging area where it can be beta-tested. You can also define the criteria for the automatic deployment in production. Such criteria include, but are not limited to, the following:

- Number of days the application was beta-tested
- Number of bugs found during beta-testing and/or removed by the last code change
- Manual approval by one or more managers/team members

The criteria decision will depend on the way the company wants to manage the product that is being developed. You, as a software architect, have to understand that when it comes to moving code to production, the safer, the better.

Managing system requirements in Azure DevOps

Azure DevOps enables you to document system requirements using *work items*. Work items are stored in your project as a piece of information that can be assigned to a person. They are classified into various types and may contain a measure of the development effort required, a status, and the development stage (iteration) they belong to.

In fact, DevOps methodology, as an Agile methodology, is made of several iterations and the whole development process is organized as a set of sprints. The work items available depends on the *Working Item Process* you select while creating the Azure DevOps project. The following subsections contain a description of the most common work item types.

Epics work items

Imagine you are developing a system made of various subsystems. Probably, you are not going to conclude the whole system in a single iteration. Therefore, we need an umbrella spanning several iterations to encapsulate all features of each subsystem. Each **Epics** work item represents one of these umbrellas that can contain several features to be implemented in various development iterations.

In the **Epics** work item, you can define the state and acceptance criteria as well as the start date and target date. Besides, you can also provide a priority and an effort estimate. All of this detailed information helps the stakeholders to follow the development process. This is really useful as a macro view of the project.

Epics are not available as a default. They must be enabled in the project's **Team Settings** page, which can be reached by clicking the project settings link in the bottom-left corner of the project page and then selecting **Team Settings**:

The screenshot shows the 'Boards' settings page in Azure DevOps. On the left, there is a sidebar with 'General', 'Overview', 'Teams', 'Security', 'Notifications', 'Service hooks', and 'Dashboards'. The 'General' tab is selected. At the top right, there is a note: 'This project is currently using the Agile process. To customize your work item types, go to the process customization page.' Below this, there are tabs for 'General', 'Iterations', 'Areas', and 'Templates'. The main content area is titled 'Backlogs' with the sub-instruction 'See only the backlogs your team manages.' Under 'Backlog navigation levels', there is a list with three items: 'Epics' (unchecked), 'Features' (checked), and 'Stories' (checked). The entire screenshot is enclosed in a black border.

Features work items

All of the information that you provide in an **Epics** work item can also be placed in a **Features** work item. So, the difference between these two types of work items is not related to the kind of information they contain, but to their roles and the focus; your team will get to conclude them. **Epics** may span several iterations and are hierarchically above **Features**, that is, each **Epics** work items is linked to several children **Features**, while each **Features** work items must be implemented in a single iteration and is part of a single **Epics** work items.

It is worth mentioning that all work items have sections for team discussions. There, you will be able to find a team member in the discussion area by typing the @ character (like in several forum/social applications). Inside each work item, you can link and attach various information. You may also check the history of the current work item in a specific section.

Features work items are the places to start recording user requirements. For instance, you can write a **Features** work item called **Access Control** to define the complete functionality needed to implement the system access control.

Product Backlog items/User Story work items

After selecting the *Working Item Process*, you will know which of these two work items is available. There are minor differences between them, but their purpose is substantially the same. They contain detailed requirements for the **Features**, described by the **Features** work items they are connected to. More specifically, each Product Backlog/User Story work item specifies the requirements of a single functionality that is a part of the behavior described in its father **Features** work items. For instance, in a **Features** of system access control, the maintenance of the users and the login interface should be two different User Stories/Product Backlog items. These requirements will guide the creation of other children work items:

- **Tasks:** They are important work items that describe the job that needs to be done to meet the requirements stated in the father Product Backlog items/User Story work item. Task work items can contain time estimates that help team capacity management and overall scheduling.
- **Test cases:** These items describe how to test the functionality described by the requirements.

The number of tasks and test cases you will create for each Product Backlog/User Story work item will vary according to the development and testing scenario you use.

Book use case – presenting use cases in Azure DevOps

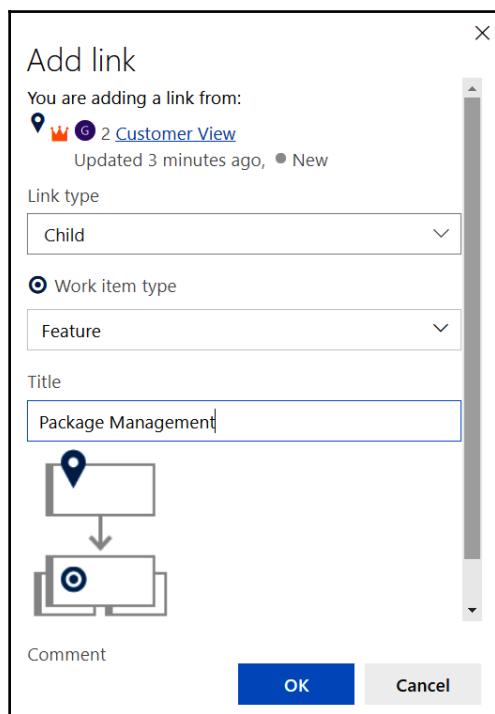
This section clarifies the concepts exposed in the previous section with the practical example of the **wwtravelclub**. Considering the scenario described in Chapter 1, *Understanding the Importance of Software Architecture*, we decided to define three **Epics** work items, as follows:

The screenshot shows the Azure DevOps interface for the 'software-architecture' project under the 'wwtravelclub' organization. The left sidebar navigation includes 'Work Items' (selected), 'Boards', 'Backlogs', 'Sprints', 'Queries', 'Repos', 'Pipelines', 'Test Plans', and 'Artifacts'. The main content area displays a table of work items with columns: ID, Title, Assigned To, State, and Area Path. The three listed work items are:

ID	Title	Assigned To	State	Area Path
2	Customer View	G Gabriel	New	wwtravelclub
3	Destination Expert View	G Gabriel	New	wwtravelclub
4	Administrator View	G Gabriel	New	wwtravelclub

The creation of these work items is quite simple:

1. Inside each work item, link the different types of work items, as you can see in the following screenshot.
2. It is really important to know that the connection between work items are really useful during software development. Hence, as a software architect, you have to provide this knowledge to your team and, more than that, you have to incentive them to make these connections:



3. As soon as you create a **Feature** work item, you will be able to connect it to several Product Backlog work items that detail its specifications. The following screenshot shows the details of a Product Backlog work item:

The screenshot shows the Azure DevOps interface for a project named "wwtravelclub". The left sidebar is collapsed, and the main area displays a "PRODUCT BACKLOG ITEM 6*" titled "6 US_001: As a common user, I want to view promotional packages in the homepage, so that I can easily find my next vacation." The item was created by "Gabriel" and has "0 comments" and "Add tag" options. It is set to "New" state, "Area: wwwtravelclub", and "Reason: New backlog item". The "Iteration" is "wwwtravelclub\Sprint 1". The item was updated 22 minutes ago by Gabriel.

Description: Promotional packages need to be presented with the promotional price in red so that common users can easily check the advantages.

Acceptance Criteria: A list containing a good picture of each package + price in red.

Discussion: Add a comment. Use # to link a work item, ! to link a pull request, or @ to mention a person.

Details: Priority: 2, Effort: 10, Business Value: Value area Business.

Development: + Add link, Development hasn't started on this item. Create a new branch.

Related Work: + Add link, Parent: 5 Package Management (Updated 22 minutes ago, ● New), Tested By: 9 View promotional packages (Updated 23 minutes ago, ● Design), Child (2): 7 Design Database (Updated 22 minutes ago, ● To Do).

4. After that, Task and Test Case work items can be created for each Product Backlog work item. The user interface provided by Azure DevOps is really efficacious because it enables you to track the chain of functionalities and the relations among them:

The screenshot shows the Azure DevOps interface for the 'wwtravelclub' team. The left sidebar is the navigation menu with options like Overview, Boards, Work Items, Boards, Backlogs, Sprints, Queries, Repos, Pipelines, Test Plans, and Artifacts. The 'Boards' option is selected. The main area is titled 'wwtravelclub Team' and shows a backlog item titled '6 US_001: As a common user, I want to view promotional packages in the homepage, so that I can easily find my next vacation.' The item is assigned to 'Gabriel'. It has a state of 'New' with 0/2 tasks completed. Below the backlog item, there are two tasks listed: 'Design Database' and 'Implement MVC Structure', both marked as completed.

5. As soon as you complete the input for the Product Backlog and Tasks work items, you will be able to plan the project sprints together with your team. The plan view enables you to drag and drop Product Backlog work items to each planned **Sprint**:

The screenshot shows the Azure DevOps interface for managing a software project named "wwtravelclub". The left sidebar has a "Backlogs" section selected. The main area displays a backlog item titled "US_001: As a common user, I want to view promotional ...". This item is categorized as a "Product Backlog" and contains two tasks: "Design Database" and "Implement MVC Structure". To the right, a "Planning" pane is open, showing three sprints: Sprint 1 (with 1 task), Sprint 2 (empty), and Sprint 3 (empty). A tooltip indicates that work items can be "Drag and drop work items to include them in a sprint".

This is how these work items are created. Once you understand this mechanism, you will be able to create and plan any software project. It is worth mentioning that the tool itself will not solve problems related to team management. However, the tool is a great way to incentive the team to update the project status, so you can keep a clear vision of how the project is evolving.

Summary

This chapter covered how you can create an Azure DevOps account for a software development project, and how to start managing your projects with Azure DevOps. It also gave a short review of all Azure DevOps functionalities, explaining how to access them through the Azure DevOps main menu. This chapter described in more detail how to manage system requirements and how to organize the job with various kinds of work items, and how to plan and organize sprints that will deliver **Epics** solutions with many **Features**.

The next chapter discusses the different models of software architecture. We will also learn about the fundamental hints and criteria for choosing among the options offered by a sophisticated cloud platform such as Azure while developing the infrastructure.

Questions

1. Is Azure DevOps available only for .NET Core projects?
2. May Azure DevOps trigger automatic builds after a commit in a specific branch? May Azure DevOps automatically trigger deployment in production?
3. What kind of test plans are available in Azure DevOps?
4. Can DevOps projects use private NuGet packages?
5. Why do we use work items?
6. What is the difference between **Epics** and **Features** work items?
7. What kind of relation exists between Tasks and Product Backlog items/User Story work items?

Further reading

Here are some books and links you may consider reading with a view to gathering more information about this chapter:

- <https://go.microsoft.com/fwlink/?LinkID=825688>
- <https://www.packtpub.com/virtualization-and-cloud/hands-devops-azure-video>
- <https://www.packtpub.com/application-development/mastering-non-functional-requirements>

2

Section 2: Architecting Software Solutions in a Cloud-Based Environment

This section introduces you to the tools included in the main modern cloud platforms. The focus is on Microsoft Azure, which has the most flexible and diverse offerings.

Chapter 4, *Deciding the Best Cloud-Based Solution*, is a general introduction to the cloud and Azure. There, you find all relevant cloud concepts and a description of the overall Azure offering, together with examples that show how you can configure resources in the cloud to meet your needs. Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*, describes the microservice computational model, which is the most efficacious way to achieve flexibility, high throughput, and reliability in the cloud. There, you can learn also about containers and Docker, which will enable you to mix different technologies in your architecture and to make your software platform-independent.

Chapter 6, *Interacting with Data in C# - Entity Framework Core*, and Chapter 7, *How to Choose Your Data Storage in the Cloud*, describes Azure's main storage offerings and how to use them. There, you can learn the following:

- How to choose the best storage solutions for each subsystem of your architecture
- How to configure storage solutions to achieve the read/write parallelism you need
- How to integrate all these into your software

Finally, in Chapter 8, *Working with Azure Functions*, you will learn about the serverless model of computation, which is included in all main cloud offerings, and understand how to use it in the Azure cloud. Thanks to serverless, you can run your computations just when they are needed without preallocating cloud resources, and you can do so while paying only for the actual computation time.

This section includes the following chapters:

- Chapter 4, *Deciding the Best Cloud-Based Solution*
- Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*
- Chapter 6, *Interacting with Data in C# - Entity Framework Core*
- Chapter 7, *How to Choose Your Data Storage in the Cloud*
- Chapter 8, *Working with Azure Functions*

4

Deciding the Best Cloud-Based Solution

To design your application so that it's cloud-based, you have to understand different architectural designs—from the simplest to the most sophisticated. This chapter discusses different software architecture models and teaches you how to take advantage of the opportunities offered by the cloud in your solutions. This chapter will also discuss the different types of cloud service that we can consider while developing our infrastructure, what the ideal scenarios are, and where we can use each of them.

The following topics will be covered in this chapter:

- Infrastructure as a Service solutions
- Platform as a Service solutions
- Software as a Service solutions
- Serverless solutions
- How to use hybrid solutions and why they are so useful

Technical requirements

For the practical content in this chapter, you have to create or use an Azure account. I've explained the account creation process in Chapter 1, *Understanding the Importance of Software Architecture*, in the *Creating an Azure account* section.

Different software deployment models

Cloud solutions can be deployed with different models. The way you decide to deploy your applications depends on the kind of team you work with. In companies where you have infrastructure engineers, you will probably find more people working with **Infrastructure as a Service (IaaS)**. On the other hand, in companies where IT is not the core business, you will find a bunch of **Software as a Service (SaaS)** systems. It is really common for developers to decide to use the **Platform as a Service (PaaS)** option, or go serverless, as they have no need to deliver infrastructures in this scenario.

As a software architect, you have to cope with this environment and be sure that you are optimizing the cost and work factors, not only during the initial development of the solution but also during its maintenance. Also, as an architect, you have to understand the needs of your system and work hard to connect those needs to best-in-class peripheral solutions to speed up delivery and keep the solution as close as possible to the customer's specifications.

Infrastructure as a service and Azure opportunities

IaaS was the first generation of cloud services provided by many different cloud players. Its definition is easily found in many places, but we can summarize it as "your computing infrastructure delivered on the internet". In the same way that we have virtualization of services in a local data center, IaaS will also give you virtualized components, such as servers, storage, and firewalls in the cloud.

In Azure, several services are provided with an IaaS model. Most of them are paid for and you should pay attention to this when it comes to testing. It is worth mentioning that this book does not set out to describe all of the IaaS services that Azure provides in detail. However, as a software architect, you just need to understand that you will find services such as the following:

- **Virtual machines:** Windows Server, Linux, Oracle, and Data Science - Machine Learning
- **Network:** Virtual networks, load balancers, and DNS zones.
- **Storage:** Files, tables, databases, and Redis.

Perform the following steps to create any service in Azure:

1. You have to find the service that best fits your needs and then create a resource.
The following screenshot shows a Windows Server virtual machine being configured:

The screenshot shows the Microsoft Azure portal interface. On the left, there is a sidebar with various service icons and links: Create a resource, Home, Dashboard, All services, FAVORITES (with All resources, Resource groups, App Services, Function Apps, SQL databases, Azure Cosmos DB, Virtual machines, Load balancers, Storage accounts, Virtual networks, Azure Active Directory, Monitor, Advisor, and Security Center), and a search bar at the top.

The main content area is titled "Create a virtual machine". At the top, there are tabs for Basics, Disks, Networking, Management, Guest config, Tags, and Review + create. The Basics tab is selected.

The Basics tab contains instructions: "Create a virtual machine that runs Linux or Windows. Select an image from Azure marketplace or use your own customized image. Complete the Basics tab then Review + create to provision a virtual machine with default parameters or review each tab for full customization. Looking for classic VMs? [Create VM from Azure Marketplace](#)".

PROJECT DETAILS:
Subscription: Pay-As-You-Go
Resource group: wwtravelclub-rg (with a "Create new" button)

INSTANCE DETAILS:
Virtual machine name: WWTravelClub
Region: Central US
Availability options: No infrastructure redundancy required
Image: Windows Server 2016 Datacenter (with a "Browse all images and disks" link)

At the bottom of the screen, there are buttons for "Review + create", "Previous", and "Next : Disks >".

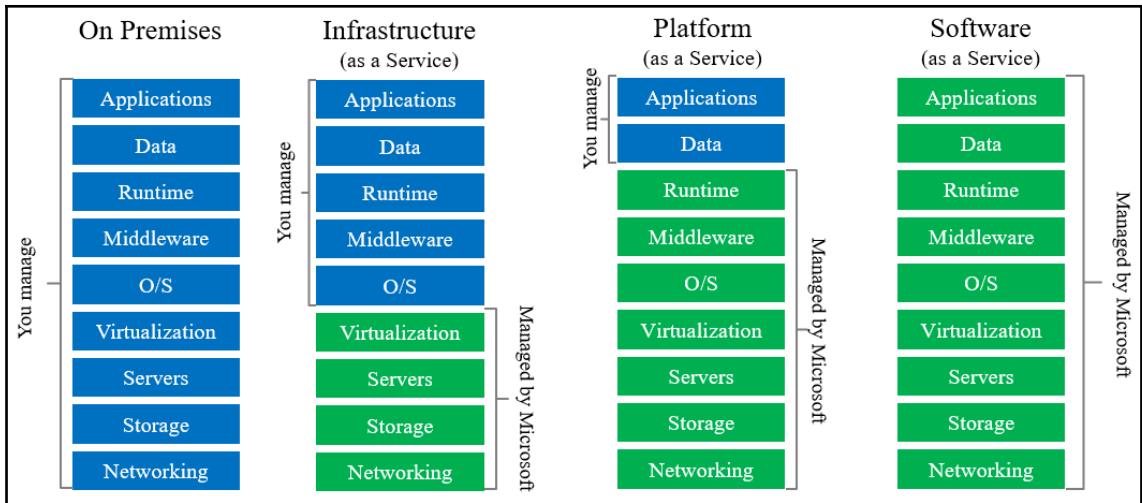
- Follow the wizard provided by Azure to set up your virtual machine and then connect to it using **Remote Desktop Protocol (RDP)**. A great curiosity from this kind of subscription is related to the capacity of hardware that you can have in some minutes. The following screenshot exemplifies this:

VM SIZE	OFFERING	FAMILY	vCPUS	RAM (GB)	DATA DISKS	MAX IOPS	TEMPORARY STOR...	PREMIUM DISK SUP...
B1ms	Standard	General purpose	1	2	800	4 GB	Yes	
B1s	Standard	General purpose	1	1	2	400	4 GB	Yes
B2ms	Standard	General purpose	2	8	4	2400	16 GB	Yes
B2s	Standard	General purpose	2	4	4	1600	8 GB	Yes
B4ms	Standard	General purpose	4	16	8	3600	32 GB	Yes
B8ms	Standard	General purpose	8	32	16	4320	64 GB	Yes
D16s_v3	Standard	General purpose	16	64	32	25600	128 GB	Yes
D2s_v3	Standard	General purpose	2	8	4	3200	16 GB	Yes
D32s_v3	Standard	General purpose	32	128	32	51200	256 GB	Yes
D4s_v3	Standard	General purpose	4	16	8	6400	32 GB	Yes
D64s_v3	Standard	General purpose	64	256	32	80000	512 GB	Yes

If you compare the on-premise velocity to deliver hardware and cloud velocity, you will realize that there is nothing better than the cloud when it comes to time-to-market. For instance, the **D64s_v3** machine presented at the bottom of the screenshot with 64 CPUs, 256 GB of RAM, and temporary storage of 512 GB is something you probably will not find in an on-premise data center. Besides, in some use cases, this machine will just be used for some hours during the month so it will be impossible to justify its purchase in an on-premise scenario. This is why cloud computing is so amazing!

Security responsibility in IaaS

Security responsibility is another important thing to know about an IaaS platform. Many people think that once you decide to go on the cloud, all of the security is done by the provider. However, this is not true as you can see in the following screenshot:



IaaS will force you to take care of security from the operating system to the application. In some cases, this is inevitable, but you have to understand that this will increase your system cost.

IaaS can be a good option if you just want to move an already existing on-premise structure to the cloud. This enables scalability, due to the tools that Azure gives you along with all of the other services. However, if you are planning to develop an application from scratch, you should also consider other options available on Azure.

Let's check one of the fastest systems in the next section, that is, PaaS.

PaaS – a world of opportunities for developers

If you are studying or have studied software architectures, you will probably understand perfectly the meaning of the next sentence: the World demands high speed when it comes to software development! If you agree with this, you will love PaaS.

As you can see in the preceding screenshot, PaaS allows you to worry about security only in terms of aspects that are closer to your business: your data and applications. As a developer, this represents freedom from having to implement a bunch of configurations that make your solution work safely.

Besides, security handling is not the only advantage of PaaS. As a software architect, you can introduce these services as an opportunity to deliver richer solutions faster. Time-to-market can surely justify the cost of many applications that run on a PaaS basis.

There are lots of services delivered as PaaS nowadays in Azure and, again, it is not the purpose of this book to list all of them. However, some of them do need to be mentioned. The list keeps growing and the recommendation here is: use and test services as much as you can! Make sure that you will deliver better-designed solutions with this thought in mind.

On the other hand, it is worth mentioning that, with PaaS solutions, you will not have full control of the operating system. In fact, in many situations, you do not even have a way to connect to it. This is good most of the time, but in some debugging situations, you may miss this feature. The good thing is that PaaS components are evolving every single day and one of the biggest concerns from Microsoft is making them widely visible.

The following sections present the most common PaaS components delivered by Microsoft for .NET Core web apps such as Azure web apps and Azure SQL Server. We also describe Azure Cognitive Services, a very powerful PaaS platform that demonstrates how wonderful development is in the PaaS world. We will explore some of them on greater depth in the remainder of this book.

Web apps

A web app is a PaaS option you can use to deploy your web app. You can deploy different types of application, such as .NET, .NET Core, Java, PHP, Node JS, and Python. A sample of this was presented in Chapter 1, *Understanding the Importance of Software Architecture*.

The good thing is that creating a web app doesn't require any structure and/or IIS web server setup. In some cases, where you are using Linux to host your .NET Core application, you do not have IIS at all.

Moreover, web apps have a plan option where you don't need to pay for usage. Of course, there are limitations, such as only running 32-bit apps and failing to enable scalability, but this can be a wonderful scenario for prototyping.

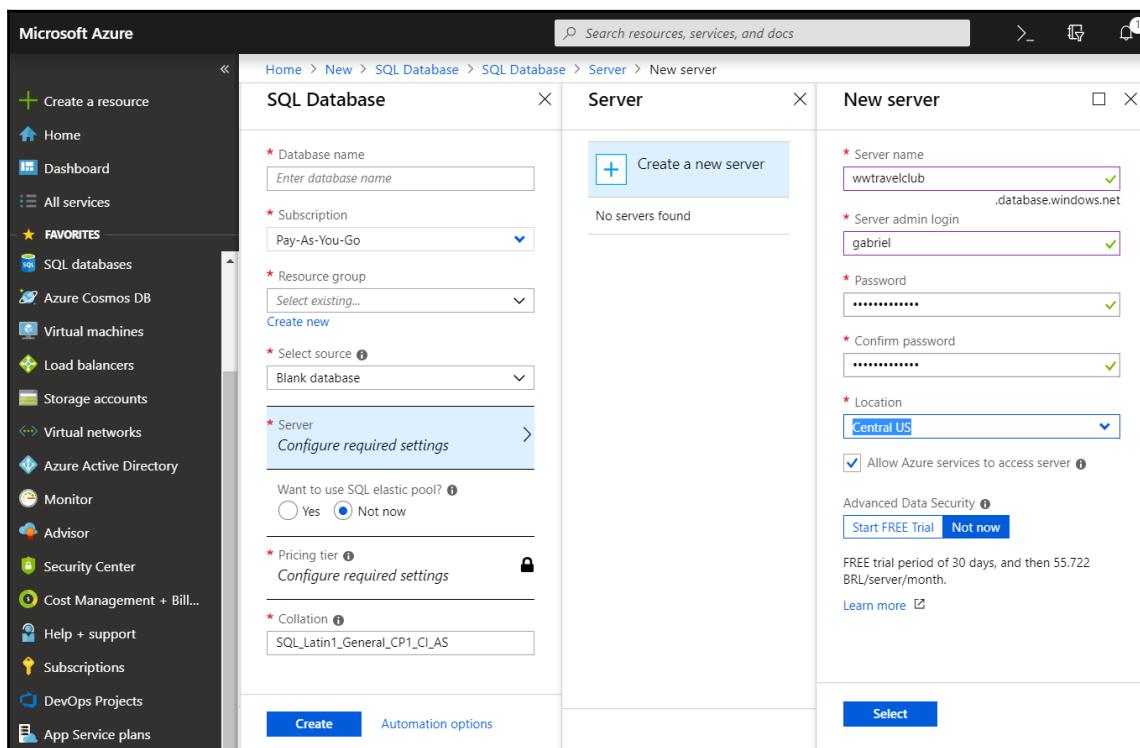
Azure SQL Server

Imagine how fast you can deploy a solution if you have all of the power of an SQL Server without needing to pay for a big server to deploy this database. This applies to Azure SQL Server. With Azure SQL Server, you have the opportunity to use Microsoft SQL Server to perform what you need the most—storage and data processing. In this scenario, Azure assumes responsibility for backing up the database.

Azure SQL Server even gives you the option to manage performance by itself. This is called automatic tuning. Again, with PaaS components, you will be able to focus on what is really important to your business: a very fast time-to-market.

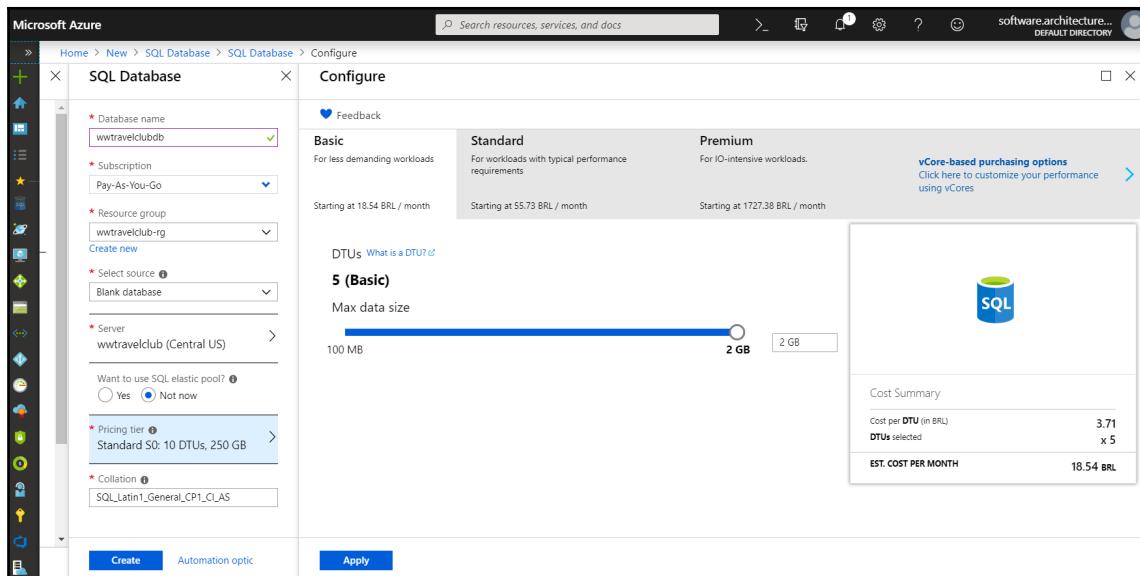
The steps for creating an Azure SQL Server database are quite simple, like what we checked before for other components. However, there are two things you need to pay attention to: the creation of the server itself and how you will be charged.

When you search Azure SQL Server about creating a database, you will find this wizard to help you:



As you can see, you have to create (at least for the first database) a database.windows.net server, where your databases will be hosted. This server will provide all of the parameters you need to access the SQL Server database using current tools such as Visual Studio or SQL Server Management Studio. It is worth mentioning that you have a bunch of features regarding security, such as transparency encryption and IP firewall.

As soon as you decide on the name of your database server, you will be able to choose the pricing tier on which your system will be charged. Especially in Azure SQL Server databases, there are several different pricing options, as you can see in the following screenshot. You should study each of them carefully because, depending on your scenario, you may save money by optimizing a pricing tier:





For more information about SQL configuration, you can check this link: <https://azure.microsoft.com/en-us/services/sql-database/>.

Once you have the configuration done, you will be able to connect to this server database in the same way you do when your SQL Server is installed on-premise. The only detail that you have to pay attention to is the configuration of the Azure SQL Server firewall, but this is quite simple to set up and a good demonstration of how safe the PaaS service is.

Azure Cognitive Services

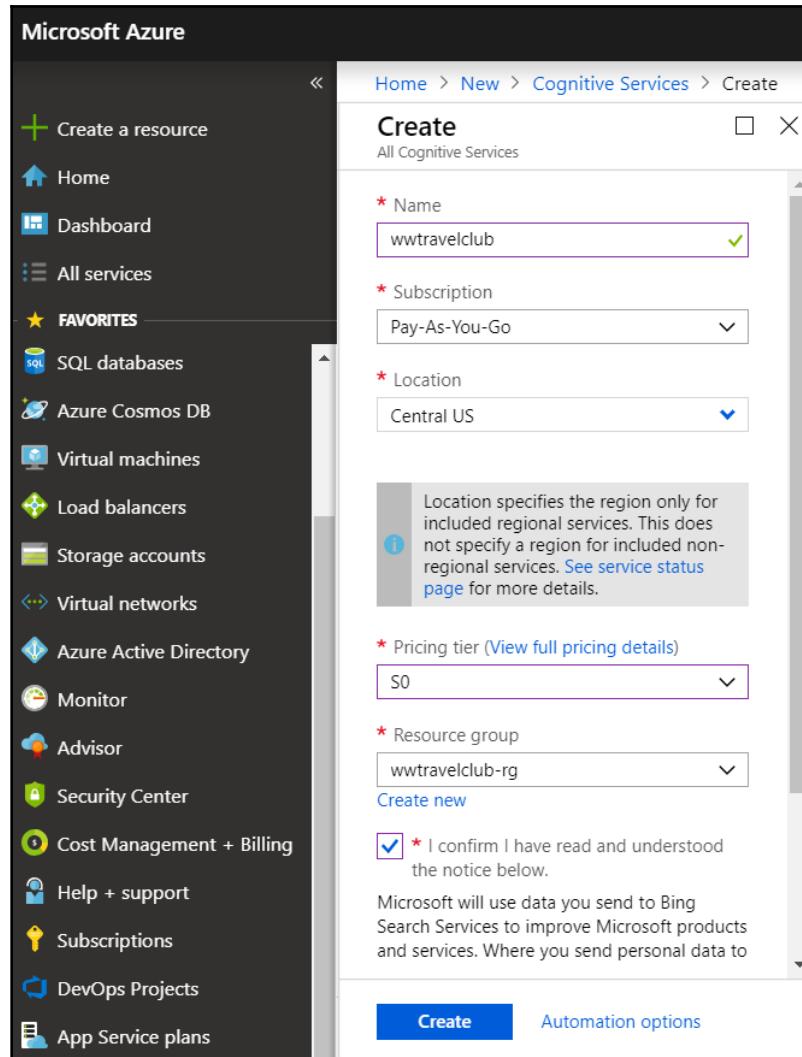
Artificial Intelligence (AI) is one of the most frequently discussed topics in software architecture. We are a step away from a really great world where AI will be everywhere. To make this last sentence come true, as a software architect you cannot think about AI as software you need to reinvent every time from scratch.

Azure Cognitive Services can help you with this. In this set of APIs, you will find various ways to develop vision, knowledge, speech, searches, and language solutions. Some of them need to be trained to make things happen, but these services provide APIs for that too.

The great thing about PaaS is evident from this scenario. The number of jobs you will have to perform to prepare your application in an on-premise or IaaS environment is enormous. In PaaS, you just do not need to worry about this. You're totally focused on what really matters to you as a software architect: the solution to your business problem.

Setting up Azure Cognitive Services in your Azure account is also quite simple:

1. First, will need to add Cognitive Services like any other Azure component, as you can see in the following screenshot:



2. As soon as you have done this, you will be able to use the APIs provided by the server. You will find two important features in the service you've created: endpoints and access keys. They are going to be used in your code to access APIs:

The screenshot shows the Azure portal interface for the 'wwtravelclub' Cognitive Services resource. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Resource Management (Keys, Quick start, Pricing tier, Billing By Subscription, Properties, Locks, Automation script), Monitoring (Alerts, Metrics), and Help. The main content area displays the service details: Resource group (wwtravelclub-rg), Status (Active), Location (Central US), Subscription (Pay-As-You-Go), Subscription ID (846d4a74-e1b1-4896-b3b6-38d500f0e432), and Tags (Click here to add tags). It also shows API type (All Cognitive Services), Pricing tier (Standard), Endpoint (https://centralus.api.cognitive.microsoft.com/), and a link to Manage keys (Show access keys ...).

The following code sample shows how you can use Cognitive Services to translate sentences. The main concept underlying this translation service is that you can post the sentence you want to translate, according to the key and region where the service was set. The following code enables you to post a request to the service API:

```
private static async Task<string> PostAPI(string api, string key, string region,
    string textToTranslate)
{
    string result = String.Empty;
    using (var client = new HttpClient())
    {
        using (var request = new HttpRequestMessage(HttpMethod.Post, api))
        {
            request.Headers.Add("Ocp-Apim-Subscription-Key", key);
            request.Headers.Add("Ocp-Apim-Subscription-Region", region);
    }
}
```

```
// five seconds for timeout
client.Timeout = new TimeSpan(0, 0, 5);
var body = new object[] { new { Text = textToTranslate } };
var requestBody = JsonConvert.SerializeObject(body);

request.Content = new StringContent(requestBody, Encoding.UTF8,
    "application/json");

var response = await client.SendAsync(request);

if (response.IsSuccessStatusCode)
    result = await response.Content.ReadAsStringAsync();
}
}

return result;
}
```

It is worth mentioning that the preceding code will allow you to post requests to translate any text into any language provided you define it in the parameters. The following is the main program that calls the previous method:

```
/// <summary>
/// Check this content at: https://docs.microsoft.com/en-us/azure/cognitive
/// services/translator/reference/v3-0-reference
/// </summary>
static void Main()
{
    var host = "https://api.cognitive.microsofttranslator.com";
    var route = "/translate?api-version=3.0&to=es";
    var subscriptionKey = "[YOUR KEY HERE]";
    var region = "[YOUR REGION HERE]";

    var translatedSentence = PostAPI(host + route, subscriptionKey, region,
        "Hello World!").Result;
    Console.WriteLine(translatedSentence);
}
```

This is a perfect example of how easily and quickly you can use services such as this to architect your projects. Also, this kind of approach to development is really good, since you are using a piece of code already tested and used by other solutions.

SaaS – just sign in and get started!

Software as a Service is probably the easiest way to use cloud-based services. Cloud players provide many good options that solve common problems in a company for their end users.

A good example of this type of service is Office 365. The key point with these platforms is that you don't need to worry about application maintenance. This is particularly convenient in scenarios where your team is totally focused on developing the core business of the application. For example, if your solution needs to deliver good reports, maybe you can design them using Power BI (which is included in Office 365).

Another pretty good example of an SaaS platform is Azure DevOps. As a software architect, before Azure DevOps or **Visual Studio Team Services (VSTS)**, you needed to install and configure Team Foundation Server (or even older tools like this one) to make your team work with a common repository and an application life cycle management tool.

We used to spend a lot of time just working either on preparing the server for **Team Foundation Server (TFS)** installation or on upgrading and continuously maintaining the TFS already installed. This is no longer needed due to the simplicity of SaaS Azure DevOps.

Understanding what serverless means

The word explains itself: a serverless solution means a solution without a server. But how can this be possible in a cloud architecture? It is pretty simple. You do not have to worry about anything related to the server in such a solution.

You may now be thinking that serverless is just another option—of course, this is true as this architecture does not deliver a complete solution. But the key point here is that, in a serverless solution, you have a very fast, simple, and agile application life cycle since all serverless code is stateless and loosely coupled with the remainder of the system. Some authors refer to this as **Function as a Service (FaaS)**.

Of course, the server runs somewhere. The key point here is that you don't need to worry about this, or even scalability. This will enable you to focus completely on your app business logic. Again, the World needs fast development and good customer experiences at the same time. The more you focus on customer needs, the better!

In Chapter 8, *Working with Azure Functions*, you will explore one of the best serverless implementations that Microsoft provides in Azure—Azure Functions. There, we will focus on how you can develop serverless solutions and learn about their advantages and disadvantages.

Why are hybrid applications so useful in many cases?

Hybrid solutions are solutions whose parts do not share a uniform architectural choice; each part makes a different architectural choice. In the cloud, the word hybrid refers mainly to solutions that mix cloud subsystems with on-premise subsystems. However, it can refer also to mixing web subsystems with device-specific subsystems.

Due to the number of services Azure can provide and the number of design architectures that can be implemented, hybrid applications are probably the best answer to the main question addressed in this chapter, that is, how to use the opportunities offered by the cloud in your projects. Nowadays, many current projects are moving from an on-premise solution to a cloud architecture and, depending on where you are going to deliver these projects, you will still find many bad preconceptions regarding moving to the cloud. Most of them are related to cost, security, and service availability.

You need to understand that there is some truth in these preconceptions, but not in the way people think. For sure, you as a software architect cannot ignore them. Especially when you develop a critical system, you have to decide whether everything can go on the cloud or whether it is better to deliver part of the system on the edge.

Mobile solutions can be considered a classic example of hybrid applications since they mix a web-based architecture with a device-based architecture to offer a better user experience. There are lots of scenarios where you can replace a mobile application with a responsive website. However, when it comes to interface quality and performance, maybe a responsive web site will not give the end user what they really need.

In the next section, we will discuss the practical example of the book use case.

Use case – a hybrid application

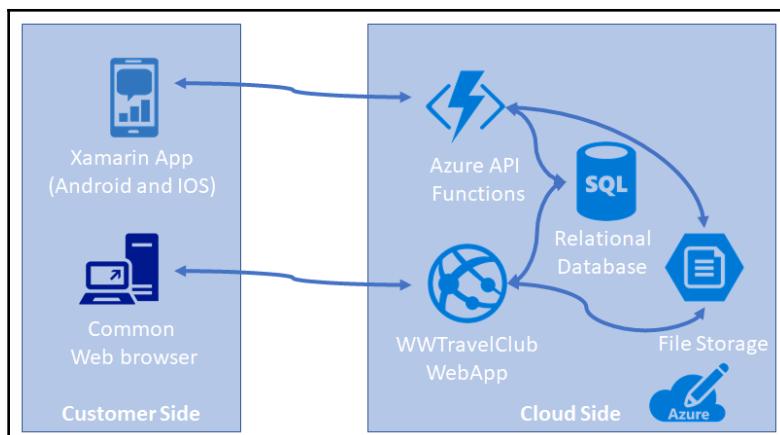
If you go back to Chapter 1, *Understanding the Importance of Software Architecture*, you will find a system requirement that describes the system environments where our WWTravelClub example application is supposed to run:

SR_003: The system shall run in Windows, Linux, iOS and Android platforms.

At first sight, any developer would respond by saying: web apps. However, the iOS and Android platforms will also need your attention as a software architect. In this scenario, as in several situations, user experience is the key to the success of the project. The decision needs to be driven not only by development speed but again by the benefits gained by delivering a great user experience.

Another decision that the software architect must make in this project is related to the technology for the mobile application if they decide to develop one. Again, this is going to be a choice between hybrid and native apps since, in this case, a hybrid solution such as Xamarin can be used. So, with mobile applications, you also have the option to keep writing the code in C#.

The following screenshot represents our first choice for the WWTravelClub architecture. The decision to rely on Azure components is related to cost and maintenance considerations. Each of the following items will be discussed later on in this book, in Chapter 6, *Interacting with Data in C# - Entity Framework Core*, Chapter 7, *How to Choose Your Data Storage in the Cloud*, and Chapter 8, *Working with Azure Functions*, together with the reasons for the choice. For now, it is enough to know that WWTravelClub is a hybrid application, running Xamarin Apps on mobiles and a .NET Core web app on the server side:



There will be an Azure SQL Server database connected from the web app by Entity Framework Core, which will be discussed in Chapter 6, *Interacting with Data in C# - Entity Framework Core*. Later on, in Chapter 7, *How to Choose Your Data Storage in the Cloud*, we will also add NoSQL databases for performance and cost reasons. For picture storage, file storage is chosen. To finish, Xamarin Apps will get information from the system through Azure Functions.

Book use case – which is the best cloud platform for this use case?

As you can verify in the screenshot in the last section, the WWTravelClub architecture was designed mainly with Platform as a Service and serverless components provided by Azure. All of the development will be conducted inside the Azure DevOps SaaS Microsoft Platform.

In the imaginary scenario we have in WWTravelClub, the sponsors have indicated that no one in the WWTravelClub team specializes in infrastructure. This is why the software architecture uses PaaS services. Considering this scenario and the required development speed, these components will surely perform well.

While we fly through the chapters and technologies discussed in this book, this architecture will change and evolve without being constrained by any earlier choices. This is a great opportunity offered by Azure and by modern architecture design. You can easily change components and structures as your solution evolves.

Summary

In this chapter, you learned how to take advantage of the services offered by the cloud in your solutions, and the various options you can choose from.

This chapter covered different ways to deliver the same application in a cloud-based structure. We also noted how rapidly Microsoft is delivering all of these options to its customers, because you can experience all of these options in actual applications and choose the one that best fits your needs since there is no *silver bullet* that works in all situations. As a software architect, you need to analyze your environment and your team, and then decide on the best cloud architecture to implement in your solution.

The next chapter is dedicated to how to build a flexible architecture made up of small scalable software modules called microservices.

Questions

1. Why should you use IaaS in your solution?
2. Why should you use PaaS in your solution?
3. Why should you use SaaS in your solution?
4. Why should you use serverless in your solution?
5. What is the advantage of using an Azure SQL Server database?
6. How can you accelerate AI in your application with Azure?
7. How can hybrid architectures help you to design a better solution?

Further reading

You can checkout these web links to decide which topics covered in this chapter you should study in greater depth:

- <https://visualstudio.microsoft.com/xamarin/>
- <https://www.packtpub.com/application-development/xamarin-cross-platform-application-development>
- <https://www.packtpub.com/virtualization-and-cloud/learning-azure-functions>
- <https://azure.microsoft.com/overview/what-is-iaas/>
- <https://docs.microsoft.com/en-us/azure/security/azure-security-iaas>
- <https://azure.microsoft.com/services/app-service/web/>
- <https://azure.microsoft.com/services/sql-database/>
- <https://azure.microsoft.com/en-us/services/virtual-machines/data-science-virtual-machines/>
- <https://docs.microsoft.com/azure/sql-database/sql-database-automatic-tuning>
- <https://azure.microsoft.com/en-us/services/cognitive-services/>
- <https://docs.microsoft.com/en-us/azure/architecture/>
- <https://powerbi.microsoft.com/>
- <https://office.com>
- <https://azure.microsoft.com/en-us/overview/what-is-serverless-computing/>
- <https://azure.microsoft.com/en-us/pricing/details/sql-database/>
- <https://www.packtpub.com/virtualization-and-cloud/professional-azure-sql-database-administration>

5

Applying a Microservice Architecture to Your Enterprise Application

This chapter is dedicated to describing highly scalable architectures based on small modules called microservices. The microservices architecture allows for fine-grained scaling operations where every single module can be scaled as required without it affecting the remainder of the system. Moreover, they allow for better **Continuous Integration/Continuous Deployment (CI/CD)** by permitting every system subpart to evolve and be deployed independently of the others.

In this chapter, we will cover the following topics:

- What are microservices?
- When do microservices help?
- How does .NET Core deal with microservices?
- Which tools are needed to manage microservices?
- Use case – logging a microservice

By the end of this chapter, you will have learned how to implement a microservice in .NET Core based on this chapter's use case.

Technical requirements

In this chapter, you will require the following:

- Visual Studio 2017 or 2019 free Community Edition or better with all the database tools installed.
- A free Azure account. The *Creating an Azure account* section in Chapter 1, *Understanding the Importance of Software Architecture*, explains how to create one.
- A local emulator for Azure Service Fabric to debug your microservices in Visual Studio. It is free and can be downloaded from <https://www.microsoft.com/web-handlers/webpi.ashx?command=getinstallerredirect&appid=MicrosoftAzure-ServiceFabric-CoreSDK>. To avoid installation issues, ensure your version of Windows is up to date. Moreover, the emulator uses PowerShell high-privilege-level commands that, by default, are blocked by PowerShell. To enable them, you need to execute the following command in the Visual Studio Package Manager Console or in any PowerShell console. Visual Studio or an external PowerShell console must be started as an *administrator* for the following command to be successful:

```
Set-ExecutionPolicy -ExecutionPolicy Unrestricted -Force -Scope CurrentUser
```

- Docker CE for Windows if you want to debug Docker containerized microservices in Visual Studio (<https://store.docker.com/editions/community/docker-ce-desktop-windows?tab=description>).

What are microservices?

Microservice architectures allow each module that makes up a solution to be scaled independently from the others to achieve the maximum throughput with minimal cost. In fact, scaling whole systems instead of their current bottlenecks inevitably results in a remarkable waste of resources, so a fine-grained control of subsystem scaling has a considerable impact on the system's overall cost.

However, microservices are more than scalable components – they are software building blocks that can be developed, maintained, and deployed independently of each other. Splitting development and maintenance among modules that can be independently developed, maintained, and deployed improves the overall system's CI/CD cycle (the CI/CD concept was explained in more detail in the *Organizing your work using Azure DevOps* section in Chapter 3, *Documenting Requirements with Azure DevOps*).

The CI/CD improvement is due to microservice *independence* because it enables the following:

- Scaling and distributing microservices on different types of hardware.
- Since each microservice is deployed independently from the others, there can't be binary compatibility or database structure compatibility constraints. Therefore, there is no need to align the versions of the different microservices that compose the system. This means that each of them can evolve, as needed, without being constrained by the others.
- Assigning their development to completely separate smaller teams, thus simplifying job organization and reducing all the inevitable coordination inefficiencies that arise when handling large teams.
- Implementing each microservice with more adequate technologies and in a more adequate environment, since each microservice is an independent deployment unit. This means choosing tools that best fit your requirements and an environment that minimizes development efforts and/or maximizes performance.
- Since each microservice can be implemented with different technologies, programming languages, tools, and operating systems, enterprises can use all available human resources by matching environments with developers' competences. For instance, underused Java developers can also be involved in .NET projects if they implement microservices in Java with the same required behavior.
- Legacy subsystems can be embedded in independent microservices, thus enabling them to cooperate with newer subsystems. This way, companies may reduce the time to market of new system versions. Moreover, this way, legacy systems can evolve slowly toward more modern systems with an acceptable impact on costs and the organization.

The next subsection explains how the concept of microservices was conceived. Then, we will continue this introductory section by exploring basic microservice design principles and analyzing why microservices are often designed as Docker containers.

Microservices and the evolution of the concept of modules

For a better understanding of the advantages of microservices, as well as their design techniques, we must keep the two-folded nature of software modularity, and of software modules, in mind:

- Code modularity refers to a code organization that makes it easy for us to modify a chunk of code without it affecting the remainder of the application. It is usually enforced with object-oriented design, where *modules* can be identified with classes.
- Deployment modularity depends on what your deployment units are and which properties they have. The simplest deployment units are executable files and libraries. Thus, for instance, **dynamic link libraries (DLL)** are, for sure, more modular than static libraries since they must not be linked with the main executable before being deployed.

While the fundamental concepts of code modularity have reached stasis, the concept of deployment modularity is still evolving and microservices are currently state of the art along this evolution path.

As a short review of the main milestones on the path that led to microservices, we can say that, first, monolithic executables were broken into static libraries. Later on, dynamic link libraries replaced static libraries.

A great change took place when .NET (and other analogous frameworks, such as Java) improved the modularity of executables and libraries. In fact, with .NET, they can be deployed on different hardware and on different operating systems since they are deployed in an intermediary language that's compiled when the library is executed for the first time. Moreover, they overcome some versioning issues of previous DLLs since any executable brings with it a DLL with a version that differs from the version of the same DLL that is installed in the operating system.

However, .NET can't accept two referenced DLLs – let's say, A and B – using two different versions of a common dependency – let's say, C. For instance, suppose there is a newer version of A with a lot of new features we would like to use that, in turn, rely on a newer version of C that's not supported by B. In a similar situation, we should renounce the newer version of A because of the incompatibility of C with B. This difficulty has led to two important changes:

- The development world moved from DLLs and/or single files to package management systems such as NuGet and npm, which automatically check version compatibility with the help of *semantic versioning*.
- **Service-Oriented Architecture (SOA)**. Deployment units started being implemented as XML and then as REST web services. This solves the version compatibility problem since each web service runs in a different process and can use the most adequate version of each library with no risk of causing incompatibilities with other web services. Moreover, the interface that's exposed by each web service is platform-agnostic, that is, web services can connect with applications using any framework and run on any operating system since web service protocols are based on universally accepted standards. SOAs and protocols will be discussed in more detail in Chapter 12, *Applying Service-Oriented Architectures with .NET Core*.

Microservices are an evolution of SOA and add more features and more constraints that improve scalability and the modularity of services to improve the overall CI/CD cycle. It's sometimes said that *microservices are SOA done well*.

Microservice design principles

To sum things up, the microservice architecture is an SOA that maximizes independence and fine-grained scaling. Now that we've clarified all the advantages of microservice independence and fine-grained scaling, as well as the very nature of independence, we are in a position to look at microservice design principles.

Let start with principles that arise from the independence constraint:

- **Independence of design choices:** The design of each microservice must not depend on the design choices that were made in the implementation of other microservices. This principle enables the full independence of each microservice CI/CD cycle and leaves us with more technological choices on how to implement each microservice. This way, we can choose the best available technology to implement each microservice.

Another consequence of this principle is that different microservices can't connect to the same shared storage (database or filesystem) since sharing the same storage also means sharing all the design choices that determined the structure of the storage subsystem (database table design, database engine, and so on). Thus, either a microservice has its own data storage or it has no storage at all and communicates with other microservices that take care of handling storage.

Here, having dedicated data storage doesn't mean that the physical database is distributed within the process boundary of the microservice itself, but that the microservice has exclusive access to a database or set of database tables that are handled by an external database engine. In fact, for performance reasons, database engines must run on dedicated hardware and with OS and hardware features that are optimized for their storage functionalities. Usually, *independence of design choices* is interpreted in a lighter form by distinguishing between logical and physical microservices. More specifically, a logical microservice is implemented with several physical microservices that use the same data storage but that are load-balanced independently. That is, the logical microservice is designed as a logical unity and then split into more physical microservices to achieve better load balance.

- **Independence from the deployment environment:** Microservices are scaled out on different hardware nodes and different microservices can be hosted on the same node. Therefore, the less a microservice relies on the services offered by the operating system and on other installed software, the more available hardware nodes it can be deployed on. More node optimization can also be performed. This is the reason why microservices are usually containerized and use Docker. Containers will be discussed in more detail in the *Containers and Docker* subsection of this chapter, but basically, containerization is a technique that allows each microservice to bring its dependencies with it so that it can run anywhere.

- **Loose coupling:** Each microservice must be loosely coupled with all the other microservices. This principle has a two-folded nature. On the one hand, this means that, according to object-oriented programming principles, the interface that's exposed by each microservice must not be too specific, but as general as possible. However, it also means that communications among microservices must be minimized in order to reduce communication costs since microservices don't share the same address space and run on different hardware nodes.
- **No chained requests/responses:** When a request reaches a microservice, it must not cause a recursive chain of nested requests/responses to other microservices since a similar chain would result in an unacceptable response time. Chained requests/responses can be avoided if the private data models of all the microservices synchronize with push notifications each time they change. In other words, as soon as the data that's handled by a microservice changes, those changes are sent to all the microservices that may need them to serve their requests. This way, each microservice has all the data it needs to serve all its incoming requests in its private data storage, with no need to ask other microservices for the data that it lacks.

In conclusion, every microservice must contain all the data it needs to serve incoming requests and ensure fast responses. To keep their data models up to date and ready for incoming requests, microservices must communicate their data changes as soon as they take place. These data changes should be communicated through asynchronous messages since synchronous nested messages cause unacceptable performance because they block all the threads involved in the call tree until a result is returned.

It is worth pointing out that the first constraint we mentioned is substantially the Bounded Context principle of domain-driven design, which we will talk about in detail in Chapter 10, *Understanding the Different Domains in Software Solutions*. In this chapter, we will see that, often, a full domain-driven design approach is useful for the *update* subsystem of each microservice.



It's not trivial that the opposite is also true, that is, that systems that have been developed according to the Bounded Context principle are better implemented with a microservice architecture. In fact, once a system has been decomposed into several completely independent and loosely coupled parts, it is very likely that these different parts need to be scaled independently because of different traffic and different resources requirements.

The preceding constraints are some best practices for building a reusable SOA. More details on these best practices will be given in *Chapter 12, Applying Service-Oriented Architectures with .NET Core*, but nowadays, most SOA best practices are automatically enforced by tools and frameworks that are used to implement web services.

Fine-grained scaling requires that microservices are small enough to isolate well-defined functionalities, but this also requires a complex infrastructure that takes care of automatically instantiating microservices, allocating instances on nodes, and scaling them as needed. These kinds of structure will be discussed in the *Which tools are needed to manage Microservices?* section of this chapter.

Moreover, fine-grained scaling of distributed microservices that communicate through asynchronous communication requires each microservice to be resilient. In fact, communication that's directed to a specific microservice instance may fail due to a hardware fault or for the simple reason that the target instance was killed or moved to another node during a load balancing operation.

Temporary failures can be overcome with exponential retries. This is where we retry the same operation after each failure with a delay that increases exponentially until a maximum number of attempts is reached. For instance, first, we would retry after 10 milliseconds, and if this retried operation results in a failure, a new attempt is done after 20 milliseconds, then after 40 milliseconds, and so on.

On the other hand, long-term failures often cause an explosion of retry operations that may saturate all system resources in a way that is similar to a Denial Of Service Attack. Therefore, usually, exponential retries are used together with a *circuit break strategy*: after a given number of failures, a long-term failure is assumed and access to the resource is prevented for a given time by returning an immediate failure without attempting the communication operation.

It is also fundamental that the congestion of some subsystems, due to either failure or to a requests peak, does not propagate to other system parts, in order to prevent overall system congestion. **Bulkhead isolation** avoids congestion propagation in the following ways:

- Only a maximum number of similar simultaneous outbound requests are allowed, let's say, 10. This is similar to putting an upper bound on thread creation.
- Requests exceeding the previous bound are queued.
- If the maximum queue length is reached, any further requests result in exceptions being thrown to abort them.

Retry policies may make it so that the same message is received and processed several times because the sender has received no confirmation that the message has been received or simply because it has timed-out the operation, while the receiver actually received the message. The only possible solution to this problem is designing all messages so that they're idempotent, that is, designing messages in such a way that processing the same message several times has the same effect as processing it once.

Updating a database table field to a value, for instance, is an idempotent operation since repeating it once or twice has exactly the same effect. However, incrementing a decimal field is not an idempotent operation. Microservice designers should make an effort to design the overall application with as many idempotent messages as possible. The remaining non-idempotent messages must be transformed into idempotent ones in the following ways, or with some other similar technique:

- Attach both a time and some identifier that uniquely identify each message.
- Store all the messages that have been received in a dictionary that's been indexed by the unique identifier attached to the message mentioned in the previous point.
- Reject old messages.
- When a message that may be a duplicate is received, verify whether it's contained in the dictionary. If it is, then it has already been processed, so reject it.
- Since old messages are rejected, they can be periodically removed from the dictionary to avoid it growing exponentially.

We will use this technique in the example at the end of this chapter.

In the next subsection, we will talk about microservice containerization based on Docker.

Containers and Docker

We've already discussed the advantages of having microservices that don't depend on the environment where they run: better hardware usage, the ability to mix legacy software with newer modules, the ability to mix several development stacks in order to use the best stack for each module implementation, and so on. Independence on the hosting environment can be easily achieved by deploying each microservice with all its dependencies on a private virtual machine.

However, starting a virtual machine with its private copy of the operating system takes a lot of time, and microservices must be started and stopped quickly to reduce load balancing and fault recovery costs. In fact, new microservices may be started either to replace faulty ones or because they were moved from one hardware node to another to perform load balancing. Moreover, adding a whole copy of the operating system to each microservice instance would be an excessive overhead.

Luckily, microservices can rely on a lighter form of technology: containers. Containers are a kind of light virtual machine. They do not virtualize a full machine – they just virtualize the **operating system (OS)** filesystem level that sits on top of the OS kernel. They use the OS of the hosting machine (kernel, DLLs, and drivers) and rely on the OS's native features to isolate processes and resources to ensure an isolated environment for the images they run.

As a consequence, containers are tied to a specific operating system but they don't suffer the overhead of copying and starting a whole OS in each container instance.

On each host machine, containers are handled by a runtime that takes care of creating them from *images* and creating an isolated environment for each of them. The most famous container runtime is Docker, which is a *de facto* standard for containerization.

Images are files that specify what is put in each container and which container resources, such as communication ports, to expose outside the container. None of the images need to explicitly specify their full content, but they can reference other images. This way, images are built by adding new software and configuration information on top of existing images.

For instance, if you want to deploy a .NET Core application as a Docker image, it is enough to just add your software and files to your Docker image and then reference an already existing .NET Core Docker image.

To allow for easy image referencing, images are grouped into registries that may be either public or private. They are similar to NuGet or npm registries. Docker offers a public registry (https://hub.docker.com/_/registry) where you can find most of the public images you may need to reference in your own images. However, each company can define private registries. For instance, Azure offers a private container registry service:
<https://azure.microsoft.com/en-us/services/container-registry/>.

Before instantiating each container, the Docker runtime must solve all the recursive references. This cumbersome job is not performed each time a new container is created since the Docker runtime has a cache where it stores the fully assembled images that correspond to each input image and that it's already processed.

Since each application is usually composed of several modules to be run in different containers, Docker also allows .yml files, also known as composition files, that specify the following information:

- Which images to deploy.
- How the internal resources that are exposed by each image must be mapped to the physical resources of the host machine. For instance, how communication ports that are exposed by Docker images must be mapped to the ports of the physical machine.

We will analyze Docker images and .yml files in the *How does .NET Core deal with Microservices?* section of this chapter.

The Docker runtime handles images and containers on a single machine but, usually, containerized microservices are deployed and load-balanced on clusters that are composed of several machines. Clusters are handled by pieces of software called **Orchestrators**. Orchestrators will be discussed in the *Which tools are needed to manage microservices?* section of this chapter.

Now that we have understood what microservices are, what problems they can solve, and their basic design principles, we are ready to analyze when and how to use them in our system architecture. The next section analyzes when we should use them.

When do microservices help?

The answer to this question requires us to understand the roles microservices play in modern software architectures. We will look at this in the following subsections.

Layered architectures and microservices

Enterprise systems are usually organized in logical independent layers. The first layer is the one that interacts with the user and is called the presentation layer, while the last layer takes care of storing/retrieving data and is called the data layer. Requests originate in the presentation layer and pass through all the layers until they reach the data layer, and then come back, traversing all the layers in reverse until they reach the presentation layer, which takes care of presenting the results to the user/client. Layers can't be *jumped*.

Each layer takes data from the previous layer, processes it, and passes it to the next layer. Then, it receives the results from its next layer and sends them back to its previous layer. Also, thrown exceptions can't jump layers – each layer must take care of intercepting all the exceptions and either *solving them* somehow or transforming them into other exceptions that are expressed in the language of its previous layer. The layer architecture ensures the complete independence of the functionalities of each layer from all the other layers of their functionalities.

For instance, we can change the database engine without affecting all the layers that are above the data layer. In the same way, we can completely change the user interface, that is, the presentation layer, without affecting the remainder of the system.

Moreover, each layer implements a different kind of system specification. The data layer takes care of what the system *must remember*, the presentation layer takes care of the system-user interaction protocol, and all the layers that are in the middle implement the domain rules, which specify how data must be processed (for instance, how an employed paycheck must be computed). Typically, the data and presentation layers are separated by just one domain rule layer, called the business or application layer.

Each layer *speaks* a different language: the data layer *speaks* the language of the chosen storage engine, the business layer speaks the language of domain experts, and the presentation layer speaks the language of users. So, when data and exceptions pass from one layer to another, they must be translated into the language of the destination layer.

A detailed example of how to build a layered architecture will be given in the *Use case - Logging Microservices* section in Chapter 10, *Understanding the Different Domains in Software Solutions*, which is dedicated to domain-driven design.

That being said, how do microservices fit into a layered architecture? Are they adequate for the functionalities of all the layers or of just some layers? Can a single microservice span several layers?

The last question is the easiest to answer: yes! In fact, we've already stated that microservices should store the data they need within their logical boundaries. Therefore, there are microservices that span the business and data layers. Some others take care of encapsulating shared data and remain confined in the data layer. Thus, we may have business layer microservices, data layer microservices, and microservices that span both layers. So, what about the presentation layer?

The presentation layer can also fit into a microservice architecture if it is implemented on the server-side. Single-page applications and mobile applications run the presentation layer on the client machine, so they either connect directly to the business microservices layer or, more often, to an *API Gateway* that exposes the public interface and takes care of routing requests to the right microservices.

In a microservices architecture, when the presentation layer is a website, it can be implemented with a set of microservices. However, if it requires heavy web servers and/or heavy frameworks, containerizing them may not be convenient. This decision must also consider the loss of performance that happens when containerizing the web server and the possible need for hardware firewalls between the web server and the remainder of the system.

ASP.NET Core is a lightweight framework that runs on the light Kestrel web server, so it can be containerized efficiently and used in a microservice for intranet applications.

However, public high-traffic websites require dedicated hardware/software components that prevent them from being deployed together with other microservices. In fact, while Kestrel is an acceptable solution for an intranet website, public websites need a more complete web server such as IIS. In this case, security requirements are more compelling and require specialized hardware/software components.

Monolithic websites can be easily broken into load-balanced smaller subsites without microservice-specific technologies, but a microservice architecture can bring all the advantages of microservices into the construction of a single HTML page. More specifically, different microservices may take care of different areas of each HTML page. Unfortunately, at the time of writing, such a similar scenario is not easy to implement with the available .NET and .NET Core technology.

A proof of concept that implements a website with ASP.NET Core-based microservices that cooperate in the construction of each HTML page can be found here: <https://github.com/Particular/Workshop/tree/master/demos/asp-net-core>. The main limit of this approach is that microservices cooperate just to generate the data that's needed to generate the HTML page and not to generate the actual HTML page. Instead, this is handled by a monolithic gateway. In fact, at the time of writing, frameworks such as ASP.NET Core MVC don't provide any facilities for the distribution of HTML generation. We will return to this example in Chapter 13, *Presenting ASP.NET Core MVC*.

Now that we've clarified which parts of a system can benefit from the adoption of microservices, we are ready to state the rules when it comes to deciding how they're adopted.

When is it worth considering microservice architectures?

Microservices can improve the implementation of both the business and data layer, but their adoption has some costs:

- Allocating instances to nodes and scaling them has a cost in terms of cloud fees or internal infrastructures and licenses.
- Splitting a unique process into smaller communicating processes increases communication costs and hardware needs, especially if the microservices are containerized.
- Designing and testing software for a microservice requires more time and increases human resources costs. In particular, making microservices resilient and ensuring that they adequately handle all possible failures, as well as verify these features with integration tests, can increase the development time by more than one order of magnitude.

So, when are microservices worth the cost of using them? Are there functionalities that must be implemented as microservices?

A rough answer to the first question is: yes, when the application is big enough in terms of traffic and/or software complexity. In fact, as an application grows in complexity and its traffic increases, it's recommended that we pay the costs connected to scaling it since this allows for more scaling optimization and better handling when it comes to the development team. The costs we pay for these would soon exceed the cost of microservice adoption.

Thus, if fine-grained scaling makes sense for our application, and if we are able to estimate the savings that fine-grained scaling and development give us, we can easily compute an overall application throughput limit that makes the adoption of microservices convenient.

Microservice costs can also be justified by the market value of our products/services increasing. Since the microservice architecture allows us to implement each microservice with a technology that has been optimized for its use, the quality that's added to our software may justify all or part of the microservice costs.

However, scaling and technology optimizations are not the only parameters to consider. Sometimes, we are forced to adopt a microservice architecture without being able to perform a detailed cost analysis.

If the size of the team that takes care of the CI/CD of the overall system grows too much, the organization and coordination of this big team cause difficulties and inefficiencies. In this type of situation, it is desirable to move to an architecture that breaks the whole CI/CD cycle into independent parts that can be taken care of by smaller teams.

Moreover, since these development costs are only justified by a high volume of requests, we probably have high traffic being processed by independent modules that have been developed by different teams. Therefore, scaling optimizations and the need to reduce interaction between development teams makes the adoption of a microservice architecture very convenient.

From this, we may conclude that, if the system and the development team grows too much, it is necessary to split the development team into smaller teams, each working on an efficient Bounded Context subsystem. It is very likely that, in a similar situation, a microservices architecture is the only possible option.

Another situation that forces the adoption of a microservice architecture is the integration of newer subparts with legacy subsystems based on different technologies since containerized microservices are the only way to implement an efficient interaction between the legacy system and the new subparts in order to gradually replace the legacy subparts with newer ones. Similarly, if our team is composed of developers with experience in different development stacks, an architecture based on containerized microservices may become a *must*.

In the next section, we will analyze building blocks and tools that are available so that we can implement .NET Core-based microservices.

How does .NET Core deal with microservices?

.NET Core was conceived as a multi-platform framework that was light and fast enough to implement efficient microservices. In particular, ASP.NET Core is the ideal tool for implementing REST APIs to communicate with a microservice, since it can run efficiently with light web servers such as Kestrel and is itself light and modular.

The whole .NET Core framework evolved with microservices as a strategic deployment platform in mind and has facilities and packages for building efficient and light HTTP communication to ensure service resiliency and to handle long-running tasks. The following subsections describe some of the different tools or solutions that we can use to implement a .NET Core-based microservice architecture.

.NET Core communication facilities

Microservices need two kinds of communication channel:

- A communication channel to receive external requests, either directly or through an API Gateway. HTTP is the usual protocol for external communication due to available web services standards and tools. .NET Core's main HTTP communication facility is ASP.NET Core since it's a lightweight HTTP framework, which makes it ideal for implementing Web APIs in small microservices. We will describe ASP.NET Core App in detail in Chapter 12, *Applying Service-Oriented Architectures with .NET Core*, which is dedicated to HTTP services. .NET Core also offers an efficient and modular HTTP client solution that is able to pool and reuse heavy connection objects. Also, the `HttpClient` class will be described in more detail in Chapter 12, *Applying Service-Oriented Architectures with .NET Core*.
- A different type of communication channel to push updates to other microservices. In fact, we have already mentioned that intra-microservice communication cannot be triggered by an on-going request since a complex tree of blocking calls to other microservices would increase request latency to an unacceptable level. As a consequence, updates must not be requested immediately before they're used and should be pushed whenever state changes take place. Ideally, this kind of communication should be asynchronous to achieve acceptable performance. In fact, synchronous calls would block the sender while they are waiting for the result, thus increasing the idle time of each microservice. However, synchronous communication that just puts the request in a processing queue and then returns confirmation of the successful communication instead of the final result is acceptable if communication is fast enough (low communication latency and high bandwidth). A publisher/subscriber communication would be preferable since, in this case, the sender and receiver don't need to know each other, thus increasing the microservices' independence. In fact, all the receivers that are interested in a certain type of communication merely need to register to receive a specific *event*, while senders just need to publish those events. All the wiring is performed by a service that takes care of queuing events and dispatching them to all the subscribers. The publisher/subscriber pattern will be described in more detail in Chapter 9, *Design Patterns and .NET Core Implementation*, along with other useful patterns.

While .NET Core doesn't directly offer tools that may help in asynchronous communication or client/server tools that implement a publisher/subscriber communication, Azure offers a similar service with *Azure Service Bus*. Azure Service Bus handles both queued asynchronous communication through Azure Service Bus *queues* and publisher/subscriber communication through Azure Service Bus *topics*.

Once you've configured an Azure Service Bus on the Azure portal, you can connect to it in order to send messages/events and to receive messages/events through a client contained in the `Microsoft.Azure.ServiceBus` NuGet package.

Azure Service Bus has two types of communication: queue-based and topic-based. In queue-based communication, each message that's placed in the queue by a sender is removed from the queue by the first receiver that pulls it from the queue. Topic-based communication, on the other hand, is an implementation of the publisher/subscriber pattern. Each topic has several subscriptions and a different copy of each message sent to a topic can be pulled from each topic subscription.

The design flow is as follows:

1. Define an Azure Service Bus private namespace.
2. Get the root connection strings that were created by the Azure portal and/or define new connection strings with fewer privileges.
3. Define queues and/or topics where the sender will send their messages in binary format.
4. For each topic, define names for all the required subscriptions.
5. In the case of queue-based communication, the sender sends messages to a queue and the receivers pull messages from the same queue. Each message is delivered to one receiver. That is, once a receiver gains access to the queue, it reads and removes one or more messages.
6. In the case of topic-based communication, each sender sends messages to a topic, while each receiver pulls messages from the private subscription associated with that topic.

There are also other commercial alternatives to Azure Service Bus, such as NServiceBus, MassTransit, Brighter, and ActiveMQ. There is also a free open source option: RabbitMQ. RabbitMQ can be installed locally, on a virtual machine, or in a Docker container. Then, you can connect with it through the client contained in the `RabbitMQ.Client` NuGet package.

The functionalities of RabbitMQ are similar to the ones offered by Azure Service Bus but you have to take care of all the implementation details, confirmations of performed operations, and so on, while Azure Service Bus takes care of all the low-level tasks and offers you a simpler interface. Azure Service Bus and RabbitMQ will be described alongside Publisher/Subscriber-based communication in Chapter 9, *Design Patterns and .NET Core Implementation*.

If microservices are published to Azure Service Fabric, which will be described in the next section, we can use a built-in reliable binary communication. Communication is resilient since communication primitives automatically use a retry policy. This communication is synchronous, but this is not a big limitation since microservices in Azure Service Fabric have built-in queues; thus, once the receiver has received a message, they can just put it in a queue and return it immediately, without blocking the sender.

The messages in the queue are then processed by a separate thread. The main limitation of this built-in communication is that it is not based on the publisher/subscriber pattern; the senders and receivers must know each other. When this is not acceptable, you should use Azure Service Bus. We will learn how to use Service Fabric's built-in communication in the *Use case - logging microservices* section of this chapter.

Resilient task execution

Resilient communication and, in general, resilient task execution can be implemented easily with the help of a .NET Core library called Polly, which is maintained by the .NET Foundation. Polly is available through the Polly NuGet package.

In Polly, you define policies, and then execute tasks in the context of that policy, as follows:

```
var myPolicy = Policy
    .Handle<HttpRequestException>()
    .Or<OperationCanceledException>()
    .Retry(3);
    ....
    ....
myPolicy.Execute(()=>{
    //your code here
});
```

The first part of each policy specifies the exceptions that must be handled. Then, you specify what to do when one of those exceptions is captured. In the preceding code, the Execute method is retried up to three times if a failure is reported either by an `HttpRequestException` exception or by an `OperationCanceledException` exception.

The following is the implementation of an exponential retry policy:

```
var erPolicy= Policy
...
//Exceptions to handle here
.WaitAndRetry(6,
    retryAttempt => TimeSpan.FromSeconds(Math.Pow(2,
        retryAttempt)));
```

The first argument of `WaitAndRetry` specifies that a maximum of six retries is performed in case of failure. The lambda function passed as second argument specifies how much time to wait before the next attempt. In the specific example, this time grows exponentially with the number of the attempt with a power of 2 (2 seconds for the first retry, 4 seconds for the second retry, and so on).

The following is a simple Circuit Breaker policy:

```
var cbPolicy=Policy
    .Handle<SomeExceptionType>()
    .CircuitBreaker(6, TimeSpan.FromMinutes(1));
```

After six failures, the task can't be executed for 1 minute since an exception is returned.

The following is the implementation of the Bulkhead Isolation policy (see the *Microservices design principles* section for more information):

```
Policy
    .Bulkhead(10, 15)
```

A maximum of 10 parallel executions is allowed in the `Execute` method. Further tasks are inserted in an execution queue. This has a limit of 15 tasks. If the queue limit is exceeded, an exception is thrown.



For the Bulkhead policy to work properly and, in general, for every strategy to work properly, task executions must be triggered through the same policy instance; otherwise, Polly is unable to count how many executions of a specific task are active.

Policies can be combined with the `Wrap` method:

```
var combinedPolicy = Policy
    .Wrap(erPolicy, cbPolicy);
```

Polly offers several more options, such as generic methods for tasks that return a specific type, timeout policies, task result caching, the ability to define custom policies, and so on. The link to the official Polly documentation is in the *Further reading* section.

Using generic hosts

Each microservice may need to run several independent threads, each performing a different operation on requests received. Such threads need several resources, such as database connections, communication channels, specialized modules that perform complex operations, and so on. Moreover, all processing threads must be adequately initialized when the microservice is started and gracefully stopped when the microservice is stopped as a consequence of either load balancing or errors.

All of these needs led the .NET Core team to conceive and implement *hosted services* and *hosts*. A host creates an adequate environment for running several tasks, known as **hosted services**, and provides them with resources, common settings, and graceful start/stop.

The concept of a web host was mainly conceived to implement the ASP.NET Core web framework, but, with effect from .NET Core 2.1, the host concept was extended to all .NET applications. All features related to the concept of "host" are contained in the `Microsoft.Extensions.Hosting` NuGet package.

First, you need to configure the host with a fluent interface, starting with a `HostBuilder` instance. The final step of this configuration is calling the `Build` method, which assembles the actual host with all the configuration information we provided:

```
var myHost=new HostBuilder()
    //Several chained calls
    //defining Host configuration
    .Build();
```

Host configuration includes defining the common resources, defining the default folder for files, loading the configuration parameters from several sources (JSON files, environment variables, and any arguments that are passed to the application), and declaring all the hosted services.

Then, the host can be started, which causes all the hosted services to be started:

```
host.Start();
```

The program remains blocked on the preceding instruction until the host is shutdown. The host can be shutdown either by one of the hosted services or externally by calling `await host.StopAsync(timeout)`. Here, `timeout` is a time span defining the maximum time to wait for the hosted services to stop gracefully. After this time, all the hosted services are aborted if they haven't been terminated.

Often, the fact that a microservice is being shutdown is signaled by a `CancellationToken` being passed when the microservice is started by the orchestrator. This happens when microservices are hosted in Azure Service Fabric.

In this case, instead of using `host.Start()`, we can use the `RunAsync` method and pass it the `CancellationToken` that we received from the orchestrator:

```
await host.RunAsync(cancellationToken)
```

This way of shutting down is triggered as soon as the `cancellationToken` enters a canceled state. By default, the host has a 5-second timeout for shutting down; that is, it waits 5 seconds before exiting once a shutdown has been requested. This time can be changed within the `ConfigureServices` method, which is used to declare *hosted services* and other resources:

```
var myHost = new HostBuilder()
    .ConfigureServices((hostContext, services) =>
{
    services.Configure<HostOptions>(option =>
    {
        option.ShutdownTimeout = System.TimeSpan.FromSeconds(10);
    });
    ...
    ...
    //further configuration
})
.Build();
```

However, increasing the host timeout doesn't increase the orchestrator timeout, so if the host waits too long, the whole microservice is killed by the orchestrator.

Hosted services are implementations of the `IHostedService` interface, whose only methods are `StartAsync(CancellationToken)` and `StopAsync(CancellationToken)`. Both methods are passed a `CancellationToken`. The `CancellationToken` in the `StartAsync` method signals that a shutdown was requested. The `StartAsync` method periodically checks this `CancellationToken` while performing all operations needed to start the host, and if it is signaled the host start process is aborted. On the other hand, the `CancellationToken` in the `StopAsync` method signals that the shutdown timeout expired.

Hosted services must be declared in the same `ConfigureServices` method that's used to define host options, as follows:

```
services.AddHostedService<MyHostedService>();
```

Most declarations inside `ConfigureServices` require the addition of the following namespace:

```
using Microsoft.Extensions.DependencyInjection;
```

Usually, the `IHostedService` interface isn't implemented directly but can be inherited from the `BackgroundService` abstract class, which exposes the easier-to-implement `ExecuteAsync(CancellationToken)` method, which is where we can place the whole logic of the service. A shutdown is signaled by passing `CancellationToken` as an argument, which is easier to handle. We will look at an implementation of `IHostedService` in the example at the end of this chapter.

To allow a hosted service to shutdown the host, we need to declare an `IAplicationLifetime` interface as its constructor parameter:

```
public class MyHostedService : BackgroundService
{
    private applicationLifetime;
    public MyHostedService(IAplicationLifetime applicationLifetime)
    {
        this.applicationLifetime=applicationLifetime;
    }
    protected Task ExecuteAsync(CancellationToken token)
    {
        ...
        applicationLifetime.StopApplication();
        ...
    }
}
```

When the hosted service is created, it will be automatically passed an implementation of `IAplicationLifetime`, whose `StopApplication` method will trigger the host shutdown. This implementation is handled automatically, but we can declare custom resources whose instances will be automatically passed to all the host service constructors that declare them as parameters. There are several ways to define these resources:

```
services.AddTransient<MyResource>();
services.AddTransient<IResourceInterface, MyResource>();
services.AddSingleton<MyResource>();
services.AddSingleton<IResourceInterface, MyResource>();
```

When we use `AddTransient`, a different instance is created and passed to all the constructors that require an instance of that type. On the other hand, with `AddSingleton`, a unique instance is created and passed to all the constructors that require the declared type. The overload with two generic types allows you to pass an interface and a type that implements that interface. This way, a constructor requires the interface and is decoupled from the specific implementation of that interface.

If resource constructors contain parameters, they will be automatically instantiated with the types declared in `ConfigureServices` in a recursive fashion. This pattern of interaction with resources is called **dependency injection (DI)** and will be discussed in detail in Chapter 9, *Design Patterns and .NET Core Implementation*.

`HostBuilder` also has a method we can use to define the default folder:

```
.UseContentRoot("c:\\<deault path>")
```

It also has methods that we can use to add logging targets:

```
.ConfigureLogging((hostContext, configLogging) =>
{
    configLogging.AddConsole();
    configLogging.AddDebug();
})
```

The preceding example shows a console-based logging source, but we can also log into Azure targets with adequate providers. The *Further reading* section contains links to some Azure logging providers that can work with microservices that have been deployed in Azure Service Fabric. Once you've configured logging, you can enable your hosted services and log custom messages by adding an `ILoggerFactory` parameter in their constructors.

Finally, `HostBuilder` has methods we can use to read configuration parameters from various sources:

```
.ConfigureHostConfiguration(configHost =>
{
    configHost.AddJsonFile("settings.json", optional: true);
    configHost.AddEnvironmentVariables(prefix: "PREFIX_");
    configHost.AddCommandLine(args);
})
```

The way parameters can be used from inside the application will be explained in more detail in Chapter 13, *Presenting ASP.NET Core MVC*, which is dedicated to ASP.NET Core.

Visual Studio support for Docker

Visual Studio offers support for creating, debugging, and deploying Docker images. Docker deployment requires us to install *Docker CE for Windows* on our development machine so that we can run Docker images. The download link can be found in the *Technical requirements* section at the beginning of this chapter. Before we start any development activity, we must ensure it is installed and running (you should see a Docker icon in the window notification bar when the Docker runtime is running).

Docker support will be described with a simple ASP.NET Core MVC project. Let's create one. To do so, follow these steps:

1. Name the project MvcDockerTest.
2. For simplicity, disable authentication.
3. You are given the option to add Docker support when you create the project, but please don't check the Docker support checkbox. You can test how Docker support can be added to any project after it has been created.

Once you have your ASP.NET Core MVC application scaffolded and running, right-click on its project icon in the **Solution Explorer** and select **Container Orchestrator Support | Docker Compose**. This will enable not only the creation of a Docker image but also the creation of a Docker Compose project, which helps you configure Docker Compose files so that they run and deploy several Docker images simultaneously. In fact, if you add another MVC project to the solution and enable container orchestrator support for it, the new Docker image will be added to the same Docker Compose file.

The advantage of enabling Docker Compose instead of just `docker` is that you can manually configure how the image is run on the development machine, as well as how Docker image ports are mapped to external ports by editing the Docker Compose files that are added to the solution.

If your Docker runtime has been installed properly and is running, you should be able to run the Docker image from Visual Studio.

Let's analyze the Docker file that was created by Visual Studio. It is a sequence of image creation steps. Each step enriches an existing image with something else with the help of the `FROM` instruction, which is a reference to an already existing image. The following is the first step:

```
FROM microsoft/dotnet:x.x-aspnetcore-runtime AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443
```

The first step uses the `microsoft/dotnet:x.x-aspnetcore-runtime` ASP.NET Core runtime that was published by Microsoft in the Docker public repository (where `x.x` is the ASP.NET Core version that was selected in your project).

The `WORKDIR` command creates the directory that follows the current directory within the image that is going to be created. If the directory doesn't exist yet, it is created in the image. The two `EXPOSE` commands declare which ports of the image ports will be exposed outside the image and mapped to the actual hosting machine. Mapped ports are decided in the deployment stage either as command-line arguments of a Docker command or within a Docker Compose file. In our case, there are two ports: one for HTTP (80) and another for HTTPS (443).

This intermediate image is cached by Docker, which doesn't need to recompute it since it doesn't depend on the code we write on the selected version of the ASP.NET Core runtime.

The second step produces a different image that will not be used to deploy. Instead, it will be used to create application-specific files that will be deployed:

```
FROM microsoft/dotnet:x.x-sdk AS build
WORKDIR /src
COPY MvcDockerTest/MvcDockerTest.csproj MvcDockerTest/
RUN dotnet restore MvcDockerTest/MvcDockerTest.csproj
COPY . .
WORKDIR /src/MvcDockerTest
RUN dotnet build MvcDockerTest.csproj -c Release -o /app

FROM build AS publish
RUN dotnet publish MvcDockerTest.csproj -c Release -o /app
```

This step starts from the ASP.NET SDK image, which contains parts we don't need to add for deployment; these are needed to process the project code. The new `src` directory is created in the `build` image and made the current image directory. Then, the project file is copied into `/src/MvcDockerTest`.

The `RUN` command executes an operating system command on the image. In this case, it calls the `dotnet` runtime, asking it to restore the NuGet packages that were referenced by the previously copied project file.

Then, the `COPY . .` command copies the whole project file tree into the `src` image directory. Finally, the project directory is made the current directory and the `dotnet` runtime is asked to build the project in release mode and copy all the output files into the new `/app` directory. Finally, a new image called `publish` executes the `publish` command on the output files.

The final step starts from the image that we created in the first step, which contains the ASP.NET Core runtime, and adds all the files that were published in the previous step:

```
FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "MvcDockerTest.dll"]
```

The `ENTRYPOINT` command specifies the operating system command that's needed to execute the image. It accepts an array of strings. In our case, it accepts the `dotnet` command and its first command-line argument, that is, the DLL we need to execute.

If we right-click on our project and click **Publish**, we are presented with several options:

- Publish the image to an existing or new web app (automatically created by Visual Studio)
- Publish to one of several Docker registries, including a private Azure Container Registry that, if it doesn't already exist, can be created from within Visual Studio
- Publish to an Azure Virtual machine

Docker Compose support allows you to run and publish a multi-container application and add further images, such as a containerized database that is available everywhere.

The following Docker Compose file adds two ASP.NET Core applications to the same Docker image:

```
version: '3.4'

services:
  mvcdockertest:
    image: ${DOCKER_REGISTRY-}mvcdockertest
    build:
      context: .
      dockerfile: MvcDockerTest/Dockerfile

  mvcdockertest1:
    image: ${DOCKER_REGISTRY-}mvcdockertest1
    build:
      context: .
      dockerfile: MvcDockerTest1/Dockerfile
```

The preceding code references existing Docker files. Any environment-dependent information is placed in the `docker-compose.override.yml` file, which is merged with the `docker-compose.yml` file when the application is launched from Visual Studio:

```
version: '3.4'

services:
  mvcdockertest:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=https://+:443;http://+:80
      - ASPNETCORE_HTTPS_PORT=44355
    ports:
      - "3150:80"
      - "44355:443"
    volumes:
      - ${APPDATA}/Asp.NET/Https:/root/.aspnet/https:ro
      - ${APPDATA}/Microsoft/UserSecrets:/root/.microsoft/usersecrets:ro
  mvcdockertest1:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=https://+:443;http://+:80
      - ASPNETCORE_HTTPS_PORT=44317
    ports:
      - "3172:80"
      - "44317:443"
    volumes:
      - ${APPDATA}/Asp.NET/Https:/root/.aspnet/https:ro
      - ${APPDATA}/Microsoft/UserSecrets:/root/.microsoft/usersecrets:ro
```

For each image, the file defines some environment variables, which will be defined in the image when the application is launched, the port mappings, and some host files.

The files in the host are directly mapped into the images, so if the image isn't projected to a host containing those files, the image won't run properly. Each declaration contains the path in the host, how the path is mapped in the image, and the desired access rights. In our case, `volumes` are used to map the self-signed https certificate that's used by Visual Studio and the user secrets (encrypted settings) that are used by ASP.NET Core.

Now, suppose we want to add a containerized SQL Server instance. We would need something like the following instructions split between `docker-compose.yml` and `docker-compose.override.yml`:

```
sql.data:
  image: mssql-server-linux:latest
  environment:
```

```
- SA_PASSWORD=Pass@word  
- ACCEPT_EULA=Y  
ports:  
- "5433:1433"
```

Here, the preceding code specifies the properties of the SQL Server container, as well as the SQL server's configuration and installation parameters. More specifically, the preceding code contains the following information:

- `sql.data` is the name that's given to the container.
- `image` specifies where to take the image from. In our case, the image is contained in a public Docker registry.
- `environment` specifies the environment variables that are needed by SQL Server, that is, the administrator password and the acceptance of a SQL Server license.
- As usual, `ports` specifies the port mappings.

`docker-compose.override.yml` is used to run the images from within Visual Studio. If you need to specify parameters for either the production environment or the testing environment, you can add further `docker-compose-xxx.override.yml` files, such as `docker-compose-staging.override.yml` and `docker-compose-production.override.yml`, and then launch them manually in the target environment with something like the following code:

```
docker-compose -f docker-compose.yml -f docker-compose-staging.override.yml
```

Then, you can destroy all the containers with the following code:

```
docker-compose -f docker-compose.yml -f docker-compose.test.staging.yml  
down
```

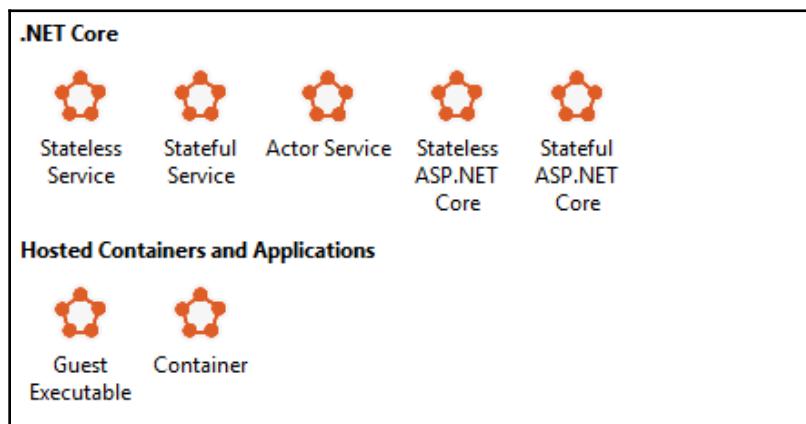
While `docker-compose` has a limited capability when it comes to handling node clusters, it is mainly used in testing and development environments. For production environments, more sophisticated tools are needed, as we will see in the *Which tools are needed to manage microservices?* section.

Azure and Visual Studio support for microservice orchestration

Visual Studio has a specific project template for microservice applications, based on the Service Fabric platform, where you can define various microservices, configure them, and deploy them to Azure Service Fabric, which is a microservice orchestrator. Azure Service Fabric will be described in more detail in the next section.

In this section, we will describe the various types of microservice you can define within a Service Fabric Application. A complete code example will be provided in the last section of this chapter. If you want to debug microservices on your development machine, you need to install the Service Fabric emulator listed in this chapter's technical requirements.

Service Fabric Applications can be found by selecting *cloud in Visual Studio project type dropdown filter*. Once you've selected the project, you can choose from a variety of services:



All projects under .NET Core use a microservice model that is specific to Azure Service Fabric. The Guest executable adds a wrapper around an existing Windows application to turn it into a microservice that can run in Azure Service Fabric. The Container application enables the addition of any Docker image in the Service Fabric Application. All the other choices scaffold a template that allows you to code a microservice with a Service Fabric-specific pattern.

Once you select any of the choices in the preceding screenshot and you fill in all the request information, Visual Studio creates two projects: an application project that contains configuration information for the overall application and a project for the specific service you have chosen that contains both the service code and service-specific configuration. If you want to add more microservices to your application, right-click on the application project and select **Add | New Service Fabric Service**.



If you right-click on the solution and select **Add | New project**, a new Service Fabric application will be created instead of a new service being added to the already existing application.

If you select **Guest Executable**, you need to provide the following:

- A folder containing the main executable file, along with all the files it needs to work properly. You need this if you want to create a copy of this folder in your project or simply to link to the existing folder.
- The main executable file.
- Arguments to pass on the command line to that executable.
- Which folder to use as a working folder on Azure. You want to use the folder containing the main executable (`CodeBase`), the folder where Azure Service Fabric will package the whole microservice (`CodePackage`), or a new subfolder named `Work`.

If you select **Container**, you need to provide the following:

- The complete name of a Docker image in your private Azure Container Registry.
- The username that will be used to connect to Azure Container Registry. The password will be specified manually in the same `RepositoryCredentials` XML element of the application configuration file that was automatically created for the username.
- The port where you can access your service (Host Port) and the port inside the container the Host Port must be mapped to (Container Port). The Container Port must be the same port that was exposed in the Docker file and used to define the Docker image.

Afterward, you may need to add further manual configuration to ensure that your Docker application works properly. The *Further reading* section contains links to the official documentation where you can find more details.

There are five types of .NET Core native Service Fabric services. The Actor service pattern is an opinionated pattern that was conceived several years ago by Carl Hewitt. We will not discuss it here, but the *Further reading* section contains some links that provide more information on this.

The remaining four patterns refer to the usage (or not) of ASP.NET Core as the main interaction protocol and to the fact that the service has or hasn't got an internal state. In fact, Service Fabric allows microservices to use distributed queues and dictionaries that are globally accessible to all instances of the microservice that declares them, independent of the hardware node where they are running (they are serialized and distributed to all available instances when they're needed).

Stateful and stateless templates differ mainly in terms of their configuration. All native services are classes that specify just two methods:

```
protected override IEnumerable<ServiceReplicaListener>
CreateServiceReplicaListeners()

protected override async Task RunAsync(CancellationToken cancellationToken)
```

The `CreateServiceReplicaListeners` method specifies a list of listeners that are used by the microservice to receive messages and the code that handles those messages. Listeners may use any protocol, but they are required to specify an implementation of the relative socket.

`RunAsync` contains the code for background threads that asynchronously run tasks that are triggered by received messages. Here, you can build a host that runs several hosted services.

ASP.NET Core templates follow the same pattern; however, they use a unique ASP.NET Core-based listener and no `RunAsync` implementation since background tasks can be launched from inside ASP.NET Core. However, you may add further listeners to the array of listeners returned by the `CreateServiceReplicaListeners` implementation created by Visual Studio, and also a custom `RunAsync` override.

More details on Service Fabric's native services pattern will be provided in the *Which tools are needed to manage microservices?* section, while a complete code example will be provided in the *Testing the application* section of this chapter, which is dedicated to this book's use case.

While this section presented the tools we can use to build the code for our microservices, the next section describes the tools we can use to define and manage the clusters where our microservices will be deployed.

Which tools are needed to manage microservices?

Effectively handling microservices in your CI/CD cycles requires both a private Docker image registry and a state of-the-art microservice orchestrator that's capable of doing the following:

- Allocating and load-balancing microservices on available hardware nodes
- Monitoring the health state of services and replacing faulty services if hardware/software failures occur
- Logging and presenting analytics
- Allowing the designer to dynamically change requirements such as hardware nodes allocated to a cluster, the number of service instances, and so on

The following subsection describes the Azure facilities we can use to store Docker images and to orchestrate microservices.

Defining your private Docker registry in Azure

Defining your private Docker registry in Azure is easy. Just type `Container registries` into the Azure search bar and select **Container registries**. On the page that appears, click on the **Add** button.

The following form will appear:

The screenshot shows the 'Create container registry' dialog box. It contains the following fields:

- Registry name:** MyCompanyRegistry (highlighted with a purple border)
- Subscription:** Visual Studio Ultimate con MSDN
- Resource group:** Default-Web-WestEurope (with a 'Create new' link)
- Location:** West Europe
- Admin user:** A button with two options: **Enable** (selected) and **Disable**.
- SKU:** Standard

The name you select is used to compose the overall registry URI: <name>.azurecr.io. As usual, you can specify the subscription, resource group, and location. The **SKU** dropdown lets you choose from various levels of offerings that differ in terms of performance, available memory, and a few other auxiliary features.

If you enable **Admin user**, an admin user will be created whose username is <name> and whose password is created automatically by the portal; otherwise, the user will log in with your Azure portal credentials. Once **Admin user** has been selected, their login information will be available under the resource *Access key* menu item.

Whenever you mention image names in Docker commands or in a Visual Studio publish form, you must prefix its name with the registry URI: <name>.azurecr.io/<my imagename>.

If images are created with Visual Studio, then they can be published by following the instructions that appear once you've published the project. Otherwise, you must use docker commands to push them into your registry.

Let's say you have the image in another registry. The first step pulls the image onto your local machine:

```
docker pull other.registry.io/samples/myimage
```

If there are several versions of the preceding image, the latest will be pulled since no version was specified. The version of the image can be specified as follows:

```
docker pull other.registry.io/samples/myimage:version1.0
```

Using the following command, you should see `myimage` within the list of local images:

```
docker images
```

Now, log in to Azure by typing in the following command and providing your credentials:

```
docker login myregistry.azurecr.io
```

Then, tag the image with the path you want to assign in the Azure registry:

```
docker tag myimage myregistry.azurecr.io/testpath/myimage
```

Both the name and destination tag may have versions (`:<version name>`).

Finally, push it:

```
docker push myregistry.azurecr.io/testpath/myimage
```

In this case, you can specify a version; otherwise, the latest version is pushed.

By doing this, you can remove the image from your local computer using the following command:

```
docker rmi myregistry.azurecr.io/testpath/myimage
```

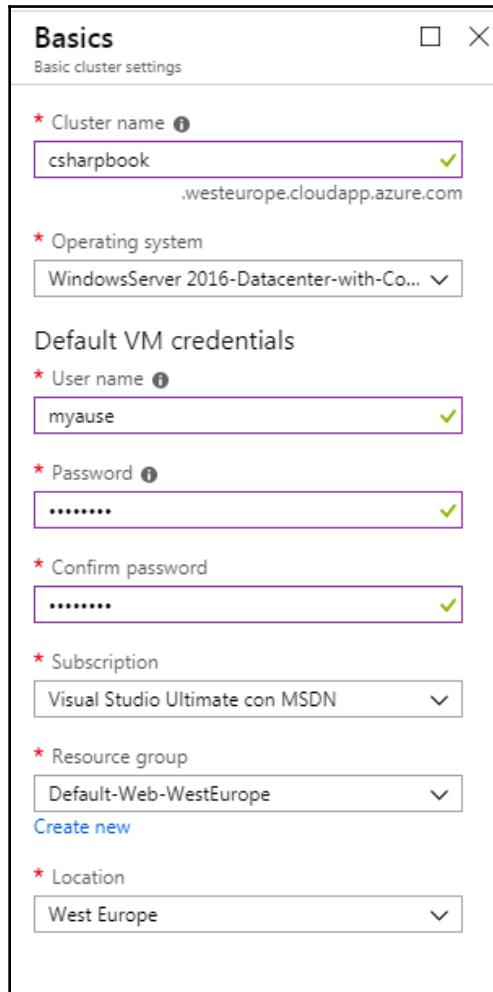
Azure Service Fabric

Azure Service Fabric is the main Microsoft orchestrator that can host Docker containers, native .NET applications, and a distributed computing model called **reliable services**. We've already explained how we can create and publish applications that contain these three types of service in the *Azure and Visual Studio support for microservice orchestration* subsection. In this section, we will explain how to create an Azure Service Fabric cluster in the Azure portal and provide some more details on **reliable services**. More practical details regarding *reliable services* will be provided in the example described in the *Use case - logging microservices* section.

You can enter the Service Fabric section of Azure by typing `Service Fabric` into the Azure search bar and selecting **Service Fabric Cluster**. A multi-step wizard will appear. The following subsections describe the available steps.

Step 1: Basic information

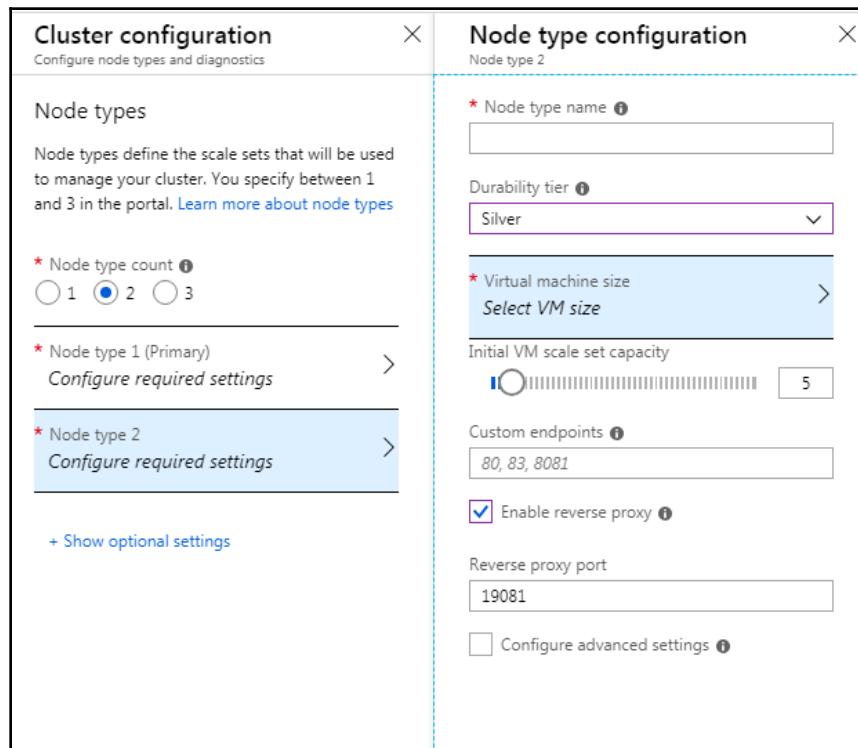
The following screenshot shows the creation of Azure Service Fabric:



Here, you can choose the operating system, resource group, subscription, location, and username and password you want to use to connect the remote desktop to all the cluster nodes. You are required to choose a cluster name, which will be used to compose the cluster URI as <cluster name>. <location>.cloudapp.azure.com, where location is a name associated with the datacenter location you have chosen.

Step 2: Cluster configuration

In the second step, you can configure the number of nodes and their features:



You can specify up to three node types. Nodes of a different node type can be scaled independently, and node type 1, called the **primary node** type, is where Azure Service Fabric runtime services will be hosted. For each node type, you can specify the type of machine (durability tier), machine dimensions (CPU and RAM), and the initial number of nodes.

You can also specify all the ports that will be visible from outside the cluster (**Custom endpoints**).



The services that are hosted on the different nodes of a cluster can communicate through any port since they are part of the same local network. Therefore, *Custom endpoints* must declare the ports that need to accept traffic from outside the cluster. The port that's exposed in *Custom endpoints* is the cluster's public interface, which can be reached through the cluster URI, that is, <cluster name>. <location>. cloudapp.azure.com. Their traffic is automatically redirected to all the microservices that have had the same ports opened by the cluster load balancer.

To understand the *enable reverse proxy* option, we must explain how communications are sent to several instances of services whose physical addresses change during their lifetimes. From within the cluster, services are identified with a URI such as fabric://<application name>/<service name>. That is, this name allows us to access one of the several load-balanced instances of <service name>. However, these URIs can't be used directly by communication protocols. Instead, they are used to get the physical URI of the required resource, along with all its available ports and protocols from the Service Fabric naming service.

Later, we will learn how to perform this operation with *reliable services*. However, this model is not adequate for Dockerized services that weren't conceived to run specifically on Azure Service Fabric since they are not aware of Service Fabric-specific naming services and APIs.

Therefore, Service Fabric provides two more options that we can use to standardize URLs instead of interacting directly with its naming service:

- **DNS:** Each service can specify its hostname (also known as its **DNS name**). The DNS service takes care of translating it into the actual service URL. For example, if a service specifies an `order.processing` DNS name and it has an HTTP endpoint on port 80 and a `/purchase` path, we can reach this endpoint with `http://order.processing:80/purchase`.

- **Reverse proxy:** Service Fabric's Reverse Proxy intercepts all the calls that have been directed to the cluster address and uses the name service to send them to the right application and service within that application. Addresses that are resolved by the reverse proxy service have the following structure: <cluster name>.<location>.cloudapp.azure.com: <port>//<app name>/<service name>/<endpoint path>?PartitionKey=<value> & PartitionKind=value. Here, partition keys are used to optimize state, fully reliable services and will be explained at the end of this subsection. This means that stateless services lack the query string part of the previous address. Thus, a typical address that's solved by reverse proxy may be something similar to myCluster.eastus.cloudapp.azure.com:
80//myapp/myservice/<endpoint path>?PartitionKey=A & PartitionKind=Named. If the preceding endpoint is called from a service hosted on the same cluster, we can specify localhost instead of the complete cluster name (that is, from the same cluster, not from the same node):
localhost: 80//myapp/myservice/<endpoint path>?PartitionKey=A & PartitionKind=Named.

By default, the DNS service is activated but the reverse proxy isn't. Therefore, we must enable it by checking the *Enable reverse proxy* checkbox in the second step of Service Fabric's configuration.

Step 3: Security configuration

Once we've submitted the second step, we come to a security page:

The screenshot shows a configuration dialog for 'Configuration Type'. A radio button for 'Basic' mode is selected. Below it, a note states: 'In 'Basic' mode, a certificate will be created for you in the key vault of your choice. If you would like to use an existing certificate or would like additional certificate options, choose 'Custom' instead.' Two fields are present: 'Key vault' containing 'mvccvault' and 'Certificate name' containing 'testcertificate'.

If we choose the **basic** option, the wizard creates an X509 certificate to secure our communication with the cluster. Otherwise, we can select an existing one from the Azure Key Vault. If you don't have a Key Vault, the wizard will make you create one so that you can store the newly created certificate. In the certificate options, locate the **certificate usage** option and select **publishing/deploying**. If you don't, you will receive an error message, along with some instructions telling you what to do to fix the issue.

Once the certificate is ready, download it onto your machine (by following the wizard's instructions) and double-click on it to install it in your local machine. The certificate will be used to deploy applications from your machine. Specifically, you are required to insert the following information into the Cloud Publish Profile of your Visual Studio Service Fabric applications (see this chapter's *Use case – logging microservices* section for more details):

```
<ClusterConnectionParameters  
    ConnectionEndpoint="<cluster name>.<location  
    code>.cloudapp.azure.com:19000"  
    X509Credential="true"  
    ServerCertThumbprint="<server certificate thumbprint>"  
    FindType="FindByThumbprint"  
    FindValue="<client certificate thumbprint>"  
    StoreLocation="CurrentUser"  
    StoreName="My" />
```

Since both the client (Visual Studio) and the server use the same certificate for authentication, the server and client thumbprint are the same. The certificate thumbprint can be copied from your Azure Key Vault. It is worth mentioning that you can add also client-specific certificates with the main server certificate by selecting the **Custom** option in step 3.

Once you submit your certificate, you are presented with a summary of your configuration. Submitting your approval will create the cluster. Pay attention to this: a cluster may spend your Azure free credit in a short time, so just keep your cluster on when you're testing. After, you should delete it.

As we mentioned in the *Azure and Visual Studio support for microservices orchestration* subsection, Azure Service Fabric supports two kinds of *reliable service*: stateless and stateful. Stateless services either don't store permanent data or they store it in external supports such as the Redis Cache or databases (see Chapter 7, *How to Choose Your Data Storage in the Cloud*, for the main storage options offered by Azure).

Stateful services, on the other hand, use Service Fabric-specific distributed dictionaries and queues. Each distributed data structure is accessible from all the *identical* replicas of a service, but only one replica, called the primary replica, is allowed to write on them to avoid synchronized access to those distributed resources, which may cause bottlenecks. All the other replicas, known as secondary replicas, can only be read from these distributed data structures.

You can check if a replica is primary by looking at the context object your code receives from the Azure Service Fabric runtime, but usually, you don't need to do this. In fact, when you declare your service endpoints, you are required to declare those that are read-only. A read-only endpoint is supposed to receive requests so that it can read data from the shared data structures. Therefore, since only read-only endpoints are activated for secondary replicas, if you implement them correctly, write/update operations should be automatically prevented on stateful secondary replicas with no need to perform further checks.

In stateful services, secondary replicas enable parallelism on read operations, so in order to get parallelism on write/update operations, stateful services are assigned different data partitions. More specifically, for each stateful service, Service Fabric creates a primary instance for each partition. Then, each partition may have several secondary replicas.

Distributed data structures are shared between the primary instance of each partition and its secondary replicas. The whole extent of data that can be stored in a stateful service is split among the chosen number of partitions, according to a partition key that is generated by a hashing algorithm on the data to be stored.

Typically, partition keys are integers that belong to a given interval that is split among all the available partitions. For instance, a partition key can be generated by calling the .NET `GetHashCode()` method on one or more string fields to get integers that are then processed to get a unique integer (using, for instance, an exclusive or operation on the integer bits). Then, this integer can be constrained to the integer interval that was chosen for the partition key by taking the remainder of an integer division (for instance, the remainder of a division for 1,000 will be an integer in the 0-999 interval).

Let's say we want four partitions, which will be selected with an integer key in the 0-999 interval. Here, Service Fabric will automatically create four primary instances of our stateful service and assign them the following four partition key subintervals: 0-249, 250-499, 500-749, 750-999.

From within your code, you are required to compute the partition key of the data you send to a stateful service. Then, Service Fabric's runtime will select the right primary instance for you. The *Use case – logging microservices* section at the end of this chapter provides more practical details on this and how to use reliable services in practice.

Azure Kubernetes Service (AKS)

Kubernetes is an advanced open source orchestrator that you can install locally on your private machine's cluster. At the time of writing, it is the most widespread orchestrator, so Microsoft offers it as an alternative to Azure Service Fabric. Also, if you prefer Azure Service Fabric, you may be forced to use the **Azure Kubernetes Service (AKS)** since some advanced solutions (for instance, some big data solutions) are built on top of Kubernetes. This subsection provides a short introduction to AKS, but more details can be found in the official documentation, which is referenced in the *Further reading* section.

To create an AKS cluster, type `AKS` into Azure search, select **Kubernetes services**, and then click the **Add** button. The following form will appear:

Create Kubernetes cluster

* Subscription ⓘ A resource group is a collection of resources that share the same lifecycle, permissions, and policies.

* Resource group ⓘ

CLUSTER DETAILS

* Kubernetes cluster name ⓘ

* Region ⓘ

* Kubernetes version ⓘ

* DNS name prefix ⓘ The DNS name must contain between 3 and 15 characters. The name can contain numbers, and hyphens. The name must start with a letter.

SCALE

The number and size of nodes in your cluster. For production workloads, at least 3 nodes are recommended for resiliency. For development or test workloads, only one node is required. You will not be able to change the node size after cluster creation, but you will be able to change the number of nodes in your cluster after creation. [Learn more about scaling in Azure Kubernetes Service](#)

* Node size ⓘ
2 vcpus, 7 GB memory
[Change size](#)

* Node count ⓘ

Virtual nodes (preview) ⓘ

As usual, you are required to specify a subscription, resource group, and location. Then, you can choose a unique name (Kubernetes cluster name), the prefix of the cluster URI (DNS name prefix), and the version of Kubernetes you would like to use. For computational power, you are asked to select a machine template for each node (Node size) and the number of nodes. If you click **Next**, you can provide security information, namely a *service principal*, and specify whether you wish to enable role-based access control. In Azure, service principals are accounts that are associated with services you may use to define resource access policies. If you have no experience with this concept and/or if you don't have any preexisting service principals, you can let the wizard create one for you.

There are other settings you can change too, but the default values work well.

Once you've created the Kubernetes cluster, you can interact with it through the `kubectl` command-line tool. `kubectl` is integrated into the Azure console, so you need just to activate your cluster credentials. Select the Azure console at the top of the page portal and then type in the following command:

```
az aks get-credentials --resource-group <resource group> --name <cluster name>
```

The preceding command downloads the credentials that were automatically created to enable your interaction with the cluster and configures the Kubernetes CLI so that it can use them.

Then, if you write `kubectl get nodes`, you should get a list of available Kubernetes nodes.

Docker images can be loaded into the cluster and configured by writing a `.yaml` configuration file, such as `myClusterConfiguration.yaml`, and typing the following:

```
kubectl apply -f myClusterConfiguration.yaml
```

You can create and edit this file by writing `nano` on the Azure console to launch the `nano` editor. Once you're in the editor, you can paste content from a local file and then save it.

The preceding command deploys the application and runs it. The deployment state can be monitored with the following command:

```
kubectl get service MyDeployment --watch
```

Here, `MyDeployment` is the name that's given to deployment in the `.yaml` file.

When the cluster is no longer needed, you can delete it with the following command:

```
az aks delete --resource-group <resource group> --name <cluster name> --no-wait
```

The application state can be monitored by selecting **Insights** from the resource Azure menu. Here, you can apply filters and select the information you need.

.yaml files have the same structure as JSON files but they have a different syntax. You have objects and lists but object properties are not surrounded by {} and lists are not surrounded by []. Instead, nested objects are declared by simply indenting their content with spaces. The number of spaces can be freely chosen, but once they've been chosen, they must be used coherently.

List items can be distinguished from object properties by preceding them with a hyphen (-). .yaml files can contain several sections that are separated by a line containing the --- string. Typically, you define a Deployment that describes which images to deploy and how many replicas they must have. Each deployment groups a set of images to be deployed together on the same node, which means that each replica that's deployed in any node must have all those images installed. A set of images to be deployed together is called a pod.

For instance, the following configuration deploys two replicas of a single image pod:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: MyDeployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: MyApplication
  template:
    ...
    ...
```

The initial header declares the Kubernetes API version to use and the kind of object we are going to define (a deployment), and assigns a name to the object. The deployment name can be used at a later time to modify the cluster with deployment edit commands.

```
template:
  metadata:
    labels:
      app: MyApplication
  spec:
    containers:
```

```
- name: MyContainerName
  image: myregistry.azurecr.io/testpath/myimage
  resources:
    requests:
      cpu: 100m
      memory: 128Mi
    limits:
      cpu: 250m
      memory: 256Mi
  ports:
    - containerPort: 80
    - name: http
  env:
    - name: MyEnvironmentVariable
      value: "MyEnvironmentVariable"
```

On the other hand, the `spec` attribute under the template lists all the containers that will compose each replica of the pod.

In turn, each container has a name and specifies the Docker image to be used to instantiate it. Then, it specifies the average computational resources that are needed and their maximum limits. Finally, it specifies the ports that are exposed externally. These ports are not forwarded to a different port and are exposed as they are. This port setting overrides the `EXPOSE` Docker file setting.

Finally, we can specify some environment variables to set inside each container.

Since there are several replicas of the same services located on different nodes, and since allocating services to nodes may change dynamically, there is a problem when it comes to internal communication among pods and internal-to-external communication. This problem is solved by defining services that offer a unique entry point for all instances of a pod. A service definition can be added to the same `.yaml` file, separated by `---`:

```
---
apiVersion: v1
kind: Service
metadata:
  name: MyApplication-service
spec:
  ports:
    - port: 8080
      targetPort: 80
      protocol: TCP
      name: http
  selector:
    app: MyApplication
```

The preceding definition creates a service that's exposed on port 8080, which redirects all requests to port 80 of a MyApplication replica. The pod that's served by the service is selected by the selector property. The service IP is internally visible, but client pods don't need to know the service IP since the services can be reached through their names, just like hosts in a classic network. Thus, in this case, MyApplication-service:8080 does the job.

If we need a publicly accessible IP, we need to add type: LoadBalancer under spec before ports. AKS will select a public IP for you. We can get the chosen public IP by watching the deployment process with kubectl get service MyDeployment --watch until the IP is selected. If we bought an IP address in the same resource group as AKS, we can specify this IP address by adding clusterIP: <your IP> under the service spec.

Pods can be organized into namespaces if we create namespaces in our .yaml files:

```
apiVersion: v1
kind: Namespace
metadata:
  name: my-namespace
  labels:
    name: my-namespace
```

Then, you can target objects (services or deployments) in a namespace by adding namespace: <your namespace> after its name in its definition metadata. Similarly, you can target kubectl commands in a specific namespace by adding them with the --namespace=<your namespace> option.

The use case in the next section will provide more details when it comes to defining a Service Fabric application. More details on Kubernetes clusters can be found in the references listed in the *Further reading* section.

Use case – logging microservices

In this section, we will take a look at a microservice-based system that logs data about purchases relating to various destinations in our WWTravelClub use case. In particular, we will design microservices that takes care of computing daily revenues per location. Here, we're assuming that these microservices receive data from other subsystems hosted in the same Azure Service Fabric application. More specifically, each purchase log message is composed of the location name, the overall package cost, and the date and time of the purchase.

As a first step, let's ensure that the Service Fabric emulator that we mentioned in the *Technical requirements* section of this chapter has been installed and is running on your development machine. Now, we need so switch it so that it runs **5 nodes**.

Now, we can follow the steps we explained in the *Azure and Visual Studio support for microservice orchestration* section to create a Service Fabric project named PurchaseLogging. Select a .NET Core stateful reliable service and name it LogStore.

The solution that's created by Visual Studio is composed of a PurchaseLogging project, which represents the overall application, and a LogStore project, which will contain the implementation of the first microservice that's included in the PurchaseLogging application.

Under the PackageRoot folder, the LogStore service and each reliable service contain the ServiceManifest.xml configuration file and a Settings.xml folder (under the Config subfolder). The Settings.xml folder contains some settings that you can read from the service code. The initial file contains predefined settings that are needed by the Service Fabric runtime. Let's add a new settings section, as shown in the following code:

```
<?xml version="1.0" encoding="utf-8" ?>
<Settings xmlns:xsd="http://www.w3.org/2001/XMLSchema"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns="http://schemas.microsoft.com/2011/01/fabric">
    <!-- This is used by the StateManager's replicator. -->
    <Section Name="ReplicatorConfig">
        <Parameter Name="ReplicatorEndpoint" Value="ReplicatorEndpoint" />
    </Section>
    <!-- This is used for securing StateManager's replication traffic. -->
    <Section Name="ReplicatorSecurityConfig" />

    <!-- Below the new Section to add -->
    <Section Name="Timing">
        <Parameter Name="MessageMaxDelaySeconds" Value="" />
    </Section>
</Settings>
```

We will use the value of MessageMaxDelaySeconds to configure the system component and ensure message idempotency. The setting value is empty, because most of the settings are overridden by the overall application settings contained in the PurchaseLogging project.

The `ServiceManifest.xml` file contains some configurations tags that are automatically handled by Visual Studio, as well as a list of endpoints. Two endpoints are preconfigured since they are used by the Service Fabric runtime. Here, we must add the configuration details of all the endpoints our microservice will listen to. Each endpoint definition has the following format:

```
<Endpoint Name="<endpoint name>" PathSuffix="<the path of the endpoint URI>" Protocol="<a protocol like Tcp, http, https, etc.>" Port="the exposed port" Type="<Internal or Input>" />
```

If `Type` is `Internal`, the port will be opened just inside the cluster's local network; otherwise, the port will be available from outside the cluster as well. In the preceding case, we must declare that port in the configuration of the Azure Service Fabric cluster as well, otherwise the cluster load balancer/firewall will not forward messages to it.



Public ports can be reached directly from the cluster URI (`<cluster name>. <location code>.cloudapp.azure.com`) since the load balancer that interfaces each cluster will forward the input traffic it receives to them.

In this example, we won't define endpoints since we are going to use remoting-based communication, which has already been defined, for all internal interactions, but we will show you how to use them.

The `PurchaseLogging` project contains a reference to the `LogStore` project under the *services* **Solution Explorer** node and contains various folders with various XML configuration files. More specifically, we have the following folders:

- `ApplicationPackageRoot`, which contains the overall application manifest named `ApplicationManifest.xml`. This file contains some initial parameter definitions and then further configurations. Parameters have the following format:

```
<Parameter Name="<parameter name>" DefaultValue="<parameter definition>" />
```

- Once defined, parameters can replace any value in the remainder of the file. Parameter values are referenced by enclosing the parameter name between square brackets, as shown in the following code:

```
<UniformInt64Partition PartitionCount="[LogStore_PartitionCount]" LowKey="0" HighKey="1000" />
```

Some parameters define the number of replicas and partitions for each service and are automatically created by Visual Studio:

```
<Parameter Name="LogStore_MinReplicaSetSize" DefaultValue="1" />
<Parameter Name="LogStore_PartitionCount" DefaultValue="2" />
<Parameter Name="LogStore_TargetReplicaSetSize" DefaultValue="1" />
```

Let's replace the initial values suggested by Visual Studio with those in the preceding code. We will use just two partitions to show you how partitions work, but you can increase this value to improve write/update parallelism. Each partition of the LogStore service doesn't need several replicas since replicas improve performance on read operations and this service is not designed to offer read services. Therefore, you may choose just one replica, or at most two, to make the system redundant and more robust to failures.

The preceding parameters are used to define the role of the LogStore service inside the overall application. This definition is generated automatically by Visual Studio in the same file, below the initial definition created by Visual studio, with just the partition interval changed to 0-1,000:

```
<Service Name=LogStore
ServicePackageActivationMode="ExclusiveProcess">
    <StatefulService ServiceTypeName="LogStoreType"
        TargetReplicaSetSize=
            "[LogStore_TargetReplicaSetSize]"
        MinReplicaSetSize="[LogStore_MinReplicaSetSize]">
        <UniformInt64Partition PartitionCount=
            "[LogStore_PartitionCount]"
            LowKey="0" HighKey="1000" />
    </StatefulService>
</Service>
```

- ApplicationParameters contains possible overrides for parameters defined in ApplicationManifest.xml for various deployment environments: the cloud (that is, the actual Azure Service Fabric cluster) and local emulators with one or five nodes.

- PublishProfiles contains the settings that are needed to publish the application in the same environments handled by the ApplicationParameters folder. You just need to customize the cloud publish profile with the actual name of your Azure Service Fabric URI and with the authentication certificate you downloaded during the Azure cluster configuration process:

```
<ClusterConnectionParameters  
    ConnectionEndpoint="<cluster name>.<location  
    code>.cloudapp.azure.com:19000"  
    X509Credential="true"  
    ServerCertThumbprint="<server certificate thumbprint>"  
    FindType="FindByThumbprint"  
    FindValue="<client certificate thumbprint>"  
    StoreLocation="CurrentUser"  
    StoreName="My" />
```

The remaining steps that need to be followed in order to complete the application have been organized into several subsections. Let's start by looking at ensuring message idempotency.

Ensuring message idempotency

Messages can become lost because of failures or small timeouts caused by load balancing. Here, we will use a predefined remoting-based communication that performs automatic message retries in the case of failures. However, as we explained in the *Microservice design principles* subsection, this may cause the same messages to be received twice. Since we are summing up the revenues of purchase orders, we must protect ourselves from summing up the same purchase several times.

To do this, we will implement a library containing the necessary tools to ensure that message replicas are discarded.

Let's add a new .NET Standard 2.0 library project called **IdempotencyTools** to our solution. Now, we can remove the initial class scaffolded by Visual studio. This library needs a reference to the same version of the `Microsoft.ServiceFabric.Services` NuGet package referenced by `LogStore`, so let's verify the version number and add the same NuGet package reference to the `IdempotencyTools` project.

The main tool that ensures message idempotency is the `IdempotentMessage` class:

```
using System;
using System.Runtime.Serialization;

namespace IdempotencyTools
{
    [DataContract]
    public class IdempotentMessage<T>
    {
        [DataMember]
        public T Value { get; protected set; }
        [DataMember]
        public DateTimeOffset Time { get; protected set; }
        [DataMember]
        public Guid Id { get; protected set; }

        public IdempotentMessage(T originalMessage)
        {
            Value = originalMessage;
            Time = DateTimeOffset.Now;
            Id = Guid.NewGuid();
        }
    }
}
```

We added the `DataContract` and `DataMember` attributes since they are needed by the remoting communication serializer we are going to use for all internal messages. Basically, the receding class is a wrapper that adds a `Guid` and a time mark to the message class instance that's passed to its constructor.

The `IdempotencyFilter` class uses a distributed dictionary to keep track of the messages it's already received. To avoid the indefinite growth of this dictionary, older entries are periodically deleted. Messages that are too old to be found in the dictionary are automatically discarded.

The time interval entries are kept in the dictionary and are passed in the `IdempotencyFilter` static factory method, which creates new filter instances, along with the dictionary name and the `IReliableStateManager` instance, which are needed to create the distributed dictionary:

```
public class IdempotencyFilter
{
    protected IReliableDictionary<Guid, DateTimeOffset> dictionary;
    protected int maxDelaySeconds;
    protected DateTimeOffset lastClear;
```

```
protected IReliableStateManager sm;
protected IdempotencyFilter() { }
public static async Task<IdempotencyFilter> NewIdempotencyFilter(
    string name,
    int maxDelaySeconds,
    IReliableStateManager sm)
{
    var result = new IdempotencyFilter();
    result.dictionary = await
        sm.GetOrAddAsync<IReliableDictionary<Guid, DateTimeOffset>>
        (name);
    result.maxDelaySeconds = maxDelaySeconds;
    result.lastClear = DateTimeOffset.Now;
    result.sm = sm;
    return result;
}
...
...
```

The dictionary contains each message time mark indexed by the message Guid and is created by invoking the `GetOrAddAsync` method of the `IReliableStateManager` instance with the dictionary type and name. `lastClear` contains the time of the removal of all old messages.

When a new message arrives, the `NewMessage` method checks whether it must be discarded. If the message must be discarded, it returns `null`; otherwise, it adds the new message to the dictionary and returns the message without the `IdempotentMessage` wrapper:

```
public async Task<T> NewMessage<T>(IdempotentMessage<T> message)
{
    DateTimeOffset now = DateTimeOffset.Now;
    if ((now - lastClear).TotalSeconds > 1.5 * maxDelaySeconds)
    {
        await Clear();
    }
    if ((now - message.Time).TotalSeconds > maxDelaySeconds)
        return default(T);
    using (ITransaction tx = this.sm.CreateTransaction())
    {
        ...
        ...
    }
}
```

As a first step, the method verifies whether it's time to clear the dictionary and whether the message is too old. Then, it starts a transaction to access the dictionary. All distributed dictionary operations must be enclosed in a transaction, as shown in the following code:

```
using (ITransaction tx = this.sm.CreateTransaction())
{
    var result = await dictionary.TryGetValueAsync(tx,
        message.Id);
    if (result.HasValue)
    {
        tx.Abort();
        return default(T);
    }
    else
    {
        await dictionary.TryAddAsync(tx, message.Id, message.Time);
        await tx.CommitAsync();
        return message.Value;
    }
}
```

If the message Guid is found in the dictionary, the transaction is aborted since the dictionary doesn't need to be updated and the method returns `default(T)`, which is actually `null` since the message must not be processed. Otherwise, the message entry is added to the dictionary and the unwrapped message is returned.

The code of the `Clear` method can be found in the GitHub repository associated with this book.

The Interaction library

There are some types that must be shared among all microservices. If the internal communication is implemented with either remoting or WCF, each microservice must expose an interface with all the methods other microservices call. Such interfaces must be shared among all microservices. Moreover, with all communication interfaces, the classes that implement the messages must also be shared among all microservices (or among some subsets of them). Therefore, all of these structures are declared in external libraries that are referenced by the microservices.

Now, let's add a new .NET Standard 2.0 library project called `Interactions` to our solution. Since this library must use the `IdempotentMessage` generic class, we must add it as a reference to the `IdempotencyTools` project. We must also add a reference to the remoting communication library contained in the `Microsoft.ServiceFabric.Services.Remoting` NuGet package since all interfaces that are used to expose the microservice's remote methods must inherit from the `IService` interface defined in this package.

`IService` is an empty interface that declares the communication role of the inheriting interface. The `Microsoft.ServiceFabric.Services.Remoting` NuGet package version must match the version of the `Microsoft.ServiceFabric.Services` package declared in the other projects.

The following code shows the declarations of the interface that need to be implemented by the `LogStore` class:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;
using IdempotencyTools;
using Microsoft.ServiceFabric.Services.Remoting;

namespace Interactions
{
    public interface ILogStore: IService
    {
        Task<bool> LogPurchase(IdempotentMessage<PurchaseInfo>
            idempotentMessage);
    }
}
```

The following is the code of the `PurchaseInfo` message class, which is referenced in the `ILogStore` interface:

```
using System;
using System.Collections.Generic;
using System.Runtime.Serialization;
using System.Text;

namespace Interactions
{
    [DataContract]
    public class PurchaseInfo
    {
        [DataMember]
```

```
        public string Location { get; set; }
        [DataMember]
        public decimal Cost { get; set; }
        [DataMember]
        public DateTimeOffset Time { get; set; }
    }
}
```

Now, we are ready to implement our main LogStore microservice.

Implementing the receiving side of communication

To implement the LogStore microservice, we must add a reference to the Interaction library, which will automatically create references to the remoting library and to the IdempotencyTools project. Then, the LogStore class must implement the `ILogStore` interface:

```
internal sealed class LogStore : StatefulService, ILogStore
{
    ...
    ...
    private IReliableQueue<IdempotentMessage<PurchaseInfo>> LogQueue = null;
    public async Task<bool>
        LogPurchase(IdempotentMessage<PurchaseInfo> idempotentMessage)
    {
        if (LogQueue == null) return false;
        using (ITransaction tx = this.StateManager.CreateTransaction())
        {
            await LogQueue.EnqueueAsync(tx, idempotentMessage);
            await tx.CommitAsync();
            return true;
        }
    }
}
```

Once the service receives a `LogPurchase` call from the remoting runtime, it puts the message in the `LogQueue` to avoid the caller remaining blocked, waiting for message processing completion. This way, we achieve both the reliability of a synchronous message passing protocol (the caller knows that the message has been received) and the performance advantages of asynchronous message processing that are typical of asynchronous communication.

`LogQueue`, as a best practice for all distributed collections, is created in the `RunAsync` method, so `LogQueue` may be null if the first call arrives before the Azure Service Fabric runtime has called `RunAsync`. In this case, the method returns `false` to signal that the service isn't ready yet. Otherwise, a transaction is created to enqueue the new message.

However, our service will not receive any communication if we don't furnish an implementation of `CreateServiceReplicaListeners()` that returns all the listeners that the service would like to activate. In the case of remoting communications, there is a predefined method that performs the whole job, so we just need to call it:

```
protected override IEnumerable<ServiceReplicaListener>
    CreateServiceReplicaListeners()
{
    return this.CreateServiceRemotingReplicaListeners<LogStore>();
}
```

Here, `CreateServiceRemotingReplicaListeners` is an extension method defined in the remoting communication library. It creates listeners for both primary replicas and secondary replicas (for read-only operations). When creating the client, we can specify whether its communications are addressed just to primary replicas or also to secondary replicas.

If you would like to use different listeners, you must create an `IEnumerable` of `ServiceReplicaListener` instances. For each listener, you must invoke the `ServiceReplicaListener` constructor with three arguments:

- A function that receives the reliable service context object as its input and returns an implementation of the `ICommunicationListener` interface.
- The name of the listener. This second argument becomes obligatory when the service has more than one listener.
- A Boolean that is true if the listener must be activated on secondary replicas.

For instance, if we would like to add both custom and HTTP listeners, the code becomes something like the following:

```
return new ServiceReplicaListener[]
{
    new ServiceReplicaListener(context =>
        new MyCustomHttpListener(context, "<endpoint name>"),
        "CustomWriteUpdateListener", true),

    new ServiceReplicaListener(serviceContext =>
        new KestrelCommunicationListener(serviceContext, "<endpoint name>",
            url, listener) =>
```

```
    {
        ...
    })
    "HttpReadOnlyListener",
    true)
};
```

MyCustomHttpListener is a custom implementation of `ICommunicationListener`, while KestrelCommunicationListener is a predefined HTTP listener based on Kestrel and ASP.NET Core. The following is the full code that defines the `KestrelCommunicationListener` listener:

```
new ServiceReplicaListener(serviceContext =>
    new KestrelCommunicationListener(serviceContext, "<endpoint name>" (url,
    listener) =>
{
    return new WebHostBuilder()
        .UseKestrel()
        .ConfigureServices(
            services => services
                .AddSingleton<StatefulServiceContext>(serviceContext)
                .AddSingleton<IReliableStateManager>(this.StateManager))
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseStartup<Startup>()
        .UseServiceFabricIntegration(listener,
            ServiceFabricIntegrationOptions.UseUniqueServiceUrl)
        .UseUrls(url)
        .Build();
})
    "HttpReadOnlyListener",
    true)
```

Usually, `ICommunicationListener` implementations accept the node context and an endpoint name in their constructors and are responsible for reading the endpoint data defined in the `ServiceManifest.xml` service, as well as creating a listening endpoint that satisfies the specification contained there. They do this in their `CommunicationListener.OpenAsync` method:

```
public async Task<string> OpenAsync(CancellationToken cancellationToken)
{
    EndpointResourceDescription serviceEndpoint = serviceContext
        .CodePackageActivationContext.GetEndpoint("ServiceEndpoint");
    //create service URI that depend on current Ip
    (FabricRuntime.GetNodeContext().IPAddressOrFQDN)
    //partition id (serviceContext.PartitionId)
    //and replica id (serviceContext.ReplicaOrInstanceId)
    //open the listener
```

```
        return <computedURISchema>;  
    }  
  
<computedURISchema> is the URI with the IP address replaced by a "+". Once returned by OpenAsync, it is published in the Service Fabric naming service and used to compute the actual service address from the cluster node IP address it's been deployed in.
```

ICommunicationListener implementations must also have a `Close` method, which must close the opened communication channel, and an `Abort` method, which must **immediately** close the communication channel (ungracefully, that is, without informing connected clients and so on).

Now that we have turned communications on, we can implement the service logic.

Implementing service logic

Service logic is executed by the tasks that are launched as independent threads when `RunAsync` is invoked by the Service Fabric runtime. It's good practice to create an `IHost` and design all the tasks as `IHostedService` implementations when you only need to implement one task. In fact, `IHostedService` implementations are independent chunks of software that are easier to unit-test. `IHost` and `IHostedService` were discussed in detail in the *Using generic hosts* subsection.

In this section, we will implement the logic that computes daily revenues for each location into an `IHostedService` named `ComputeStatistics`, which uses a distributed dictionary whose keys are the location names and whose values are instances of a class called `RunningTotal`. This class stores the current running total and the day that is being computed:

```
namespace LogStore  
{  
    public class RunningTotal  
    {  
        public DateTime Day { get; set; }  
        public decimal Count { get; set; }  
  
        public RunningTotal  
            Update(DateTimeOffset time, decimal value)  
        {  
            ...  
        }  
    }  
}
```

This class has an `Update` method that updates the instance when a new purchase message is received. First of all, the incoming message time is normalized to universal time. Then, the day part of this time is extracted and compared with the current `Day` of the running total, as shown in the following code:

```
public RunningTotal Update(DateTimeOffset time, decimal value)
{
    var normalizedTime = time.ToUniversalTime();
    var newDay = new DateTime(normalizedTime.Year,
        normalizedTime.Month, normalizedTime.Day);
    ...
    ...
}
```

If it's a new day, we assume that the running total computation of the previous day has finished, so the `Update` method returns it in a new `RunningTotal` instance and resets `Day` and `Count` so that it can compute the new day running total. Otherwise, the new value is added to the running `Count` and the method returns `null`, meaning that the day total isn't ready yet. This implementation can be seen in the following code:

```
public RunningTotal Update(DateTimeOffset time, decimal value)
{
    ...
    ...
    var result = newDay > Day && Day != DateTime.MinValue ?
        new RunningTotal
    {
        Day=Day,
        Count=Count
    }
    : null;
    if(newDay > Day) Day = newDay;
    if (result != null) Count = value;
    else Count += value;
    return result;
}
```

The `IHostedService` implementation of `ComputeStatistics` needs some parameters to work properly, as follows:

- The queue containing all the incoming messages
- The `IReliableStateManager` service, so that it can create the distributed dictionary where it stores data

- The ConfigurationPackage service, so that it can read the settings defined in the Settings.xml service file and possibly those overridden in the application manifest

The preceding parameters must be passed in the ComputeStatistics constructor when a ComputeStatistics instance is created by IHost through dependency injection. We will return to the IHost definition in the next subsection. For now, let's concentrate on the ComputeStatistics constructor and its fields:

```
namespace LogStore
{
    public class ComputeStatistics : BackgroundService
    {
        IReliableQueue<IdempotentMessage<PurchaseInfo>> queue;
        IReliableStateManager stateManager;
        ConfigurationPackage configurationPackage;
        public ComputeStatistics(
            IReliableQueue<IdempotentMessage<PurchaseInfo>> queue,
            IReliableStateManager stateManager,
            ConfigurationPackage configurationPackage)
        {
            this.queue = queue;
            this.stateManager = stateManager;
            this.configurationPackage = configurationPackage;
        }
    }
}
```

All the constructor parameters are stored in private fields so that they can be used when ExecuteAsync is called:

```
protected async override Task ExecuteAsync(CancellationToken stoppingToken)
{
    bool queueEmpty = false;
    var delayString=configurationPackage.Settings.Sections["Timing"]
        .Parameters["MessageMaxDelaySeconds"].Value;
    var delay = int.Parse(delayString);
    var filter = await IdempotencyFilter.NewIdempotencyFilter(
        "logMessages", delay, stateManager);
    var store = await
        stateManager.GetOrAddAsync<IReliableDictionary<string,
    RunningTotal>>("partialCount");
    ....
    ...
}
```

Before entering its loop, the ComputeStatistics service prepares some structures and parameters. It declares that the queue isn't empty so that it can start dequeuing messages. Then, it extracts MessageMaxDelaySeconds from the service settings and turns it into an integer. The value of this parameter was left empty in the Settings.xml file. Now, it's time to override it and define its actual value in ApplicationManifest.xml:

```
<ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="LogStorePkg"
ServiceManifestVersion="1.0.0" />
    <!--code to add start -->
    <ConfigOverrides>
        <ConfigOverride Name="Config">
            <Settings>
                <Section Name="Timing">
                    <Parameter Name="MessageMaxDelaySeconds"
Value="[MessageMaxDelaySeconds]" />
                </Section>
            </Settings>
        </ConfigOverride>
    </ConfigOverrides>
    <!--code to add end-->
</ServiceManifestImport>
```

ServiceManifestImport imports the service manifest in the application and overrides some configuration. Its version number must be changed every time its content and/or the service definition is changed and the application is redeployed in Azure because version number changes tell the Service Fabric runtime what to change in the cluster. Version numbers also appear in other configuration settings. They must be changed every time the entities they refer to change.

MessageMaxDelaySeconds is passed to the instance of the idempotency filter, along with a name for the dictionary of the already received messages, and with the instance of the IReliableStateManager service. Finally, the main distributed dictionary that's used to store running totals is created.

After this, the service enters its loop and finishes when stoppingToken is signaled, that is, when the Service Fabric runtime signals that the service is going to be stopped:

```
while (!stoppingToken.IsCancellationRequested)
{
    while (!queueEmpty && !stoppingToken.IsCancellationRequested)
    {
        RunningTotal total = null;
        using (ITransaction tx = stateManager.CreateTransaction())
        {
            ...
        }
    }
}
```

```
    ...
    ...
}

}
await Task.Delay(100, stoppingToken);
queueEmpty = false;
}

}
```

The inner loop runs until the queue isn't empty and then exits and waits 100 milliseconds before verifying whether new messages have been enqueued:

```
await Task.Delay(100, stoppingToken);
queueEmpty = false;
```

The following is the code for the inner loop, which is enclosed in a transaction:

```
RunningTotal total = null;
using (ITransaction tx = stateManager.CreateTransaction())
{
    var result = await queue.TryDequeueAsync(tx);
    if (!result.HasValue) queueEmpty = true;
    else
    {
        var item = await filter.NewMessage<PurchaseInfo>(result.Value);
        if(item != null)
        {
            var counter = await store.TryGetValueAsync(tx,
                item.Location);
            //counter update
            ...
        }
        ...
        ...
    }
}
```

Here, the service is trying to dequeue a message. If the queue is empty, it sets `queueEmpty` to `true` to exit the loop; otherwise, it passes the message through the idempotency filter. If the message survives this step, it uses it to update the running total of the location referenced in the message. However, correct operation of the distributed dictionary requires that the old counter is replaced with a new counter each time an entry is updated. Accordingly, the old counter is copied into a new `RunningTotal` object. This new object can be updated with the new data if we call the `Update` method:

```
//counter update
var newCounter = counter.HasValue ?
    new RunningTotal
{
    Count=counter.Value.Count,
    Day= counter.Value.Day
}
: new RunningTotal();
total = newCounter.Update(item.Time, item.Cost);
if (counter.HasValue)
    await store.TryUpdateAsync(tx, item.Location,
        newCounter, counter.Value);
else
    await store.TryAddAsync(tx, item.Location, newCounter);
```

Then, the transaction is committed, as shown in the following code:

```
if(item != null)
{
    ...
    ...
}
await tx.CommitAsync();
if(total != null)
{
    await SendTotal(total, item.Location);
}
```

When the `Update` method returns a complete computation result, that is when the `total != null` method is called:

```
protected async Task SendTotal(RunningTotal total, string location)
{
    //Empty, actual application would send data to a service
    //that exposes daily statistics through a public Http endpoint
}
```

The `SendTotal` method sends the total to a service that publicly exposes all the statistics through an HTTP endpoint. After reading Chapter 12, *Applying Service-Oriented Architectures with .NET Core*, which is dedicated to the Web API, you may want to implement a similar service with a stateless ASP.NET Core microservice connected to a database. The stateless ASP.NET Core service template automatically creates an ASP.NET Core-based HTTP endpoint for you.

However, since this service must receive data from the `SendTotal` method, it also needs remote-based endpoints. Therefore, we must create them, just like we did for the `LogStore` microservice, and concatenate the remote-based endpoint array with the preexisting array containing the HTTP endpoint.

Defining the microservice's host

Now, we have everything in place to define the microservice's `RunAsync` method:

```
protected override async Task RunAsync(CancellationToken cancellationToken)
{
    // TODO: Replace the following sample code with your own logic
    // or remove this RunAsync override if it's not needed in your service.
    cancellationToken.ThrowIfCancellationRequested();
    LogQueue = await
        this.StateManager
            .GetOrAddAsync<IReliableQueue<IDempotentMessage<PurchaseInfo>>>("logQueue");
    var configurationPackage = Context
        .CodePackageActivationContext
        .GetConfigurationPackageObject("Config");
    ...
    ...
}
```

Here, the method verifies whether the cancellation token was signaled, in which case we throw an exception to abort the method. Then, the service queue is created, and the service settings are saved in `configurationPackage`.

After that, we can create the `IHost` service, as we explained in the *Using generic hosts* subsection:

```
var host = new HostBuilder()
    .ConfigureServices((hostContext, services) =>
{
    services.AddSingleton(this.StateManager);
    services.AddSingleton(this.LogQueue);
    services.AddSingleton(configurationPackage);
```

```
        services.AddHostedService<ComputeStatistics>();  
    }  
    .Build();  
    await host.RunAsync(cancellationToken);
```

ConfigureServices defines all singletons instances that may be needed by `IHostedService` implementations, so they are injected into the constructor of all the implementations that reference their types. Then, `AddHostedService` declares the unique `IHostedService` of the microservice. Once the `IHost` is built, we run it until the `RunAsync` cancellation token is signaled. When the cancellation token is signaled, the request to shutdown is passed to all `IHostedService` implementations.

Communicating with the service

Since we haven't implemented the whole purchase logic yet, we will implement a stateless microservice that sends random data to the `LogStore` service. Right-click on the `PurchaseLogging` project in the **Solution Explorer** and select **Add | Service Fabric Service**. Then, select the .NET Core stateless template and name the new microservice project `FakeSource`.

Now, let's add a reference to the `Interaction` project. Before moving on to the service code, we need to update the replica count of the newly created service in `ApplicationManifest.xml` and in all the other environment-specific parameter overrides (the cloud, one local cluster node, five local cluster nodes):

```
<Parameter Name="FakeSource_InstanceCount" DefaultValue="2" />
```

This fake service needs no listeners and its `RunAsync` method is straightforward:

```
string[] locations = new string[] { "Florence", "London", "New York",  
"Paris" };  
  
protected override async Task RunAsync(CancellationToken cancellationToken)  
{  
    Random random = new Random();  
    while (true)  
    {  
        cancellationToken.ThrowIfCancellationRequested();  
  
        PurchaseInfo message = new PurchaseInfo  
        {  
            Time = DateTimeOffset.Now,  
            Location= locations[random.Next(0, locations.Length)],  
            Cost= 200m*random.Next(1, 4)
```

```
    };
    //Send message to counting microservices
    ...
    ...
    await Task.Delay(TimeSpan.FromSeconds(1), cancellationToken);
}
}
```

In each loop, a random message is created and sent to the counting microservices. Then, the thread sleeps for a second and starts a new loop. The code that sends the created messages is as follows:

```
//Send message to counting microservices
var partition = new
ServicePartitionKey(Math.Abs(message.Location.GetHashCode()) % 1000);
var client = ServiceProxy.Create<ILogStore>(
    new Uri("fabric:/PurchaseLogging/LogStore"), partition);
try
{
    while (!await client.LogPurchase(new
IdempotentMessage<PurchaseInfo>(message)))
    {
        await Task.Delay(TimeSpan.FromMilliseconds(100),
            cancellationToken);
    }
}
catch
{
}
```

Here, a key in the 0-9,999 interval is computed from the location string. This integer is passed to the `ServicePartitionKey` constructor. Then, a service proxy is created, and the URI of the service to call and the partition key are passed. The proxy uses this data to ask the naming service for a physical URI for a primary instance for the given partition value.

`ServiceProxy.Create` also accepts a third optional argument that specifies whether messages that are sent by the proxy can also be routed to secondary replicas. The default is that messages are routed just to primary instances. If the message target returns `false`, meaning that it's not ready (remember that `LogPurchase` returns `false` when the `LogStore` message queue hasn't been created yet), the same transmission is attempted after 100 milliseconds.

Sending messages to a remoting target is quite easy. However, other communication listeners require that the sender interacts manually with the naming service to get a physical service URI. This can be done with the following code:

```
ServicePartitionResolver resolver = ServicePartitionResolver.GetDefault();  
  
ResolvedServicePartition partition =  
    await resolver.ResolveAsync(new Uri("fabric:/MyApp/MyService"),  
        new ServicePartitionKey(...), cancellationToken);  
//look for a primary service only endpoint  
var finalURI= partition.Endpoints.First(p =>  
    p.Role == ServiceEndpointRole.StatefulPrimary).Address;
```

Moreover, in the case of generic communication protocols, we must manually handle failures and retries with a library such as Polly (see the *Resilient task execution* subsection for more information).

Testing the application

To test that the application actually computes running purchase totals, let's place a breakpoint in the `ComputeStatistics.cs` file:

```
total = newCounter.Update(item.Time, item.Cost);  
if (counter.HasValue) ... //put breakpoint on this line
```

Each time the breakpoint is hit, look at the content of `newCounter` to verify how the running totals of all the locations change.

Summary

In this chapter, we described what microservices are and how they have evolved from the concept of a module. Then, we talked about the advantages of microservices and when it's worth using them, as well as general criteria for their design. We also explained what Docker containers are and analyzed the strong connection between containers and microservice architectures.

Then, we took on a more practical implementation by describing all the tools that are available in .NET Core so that we can implement microservice-based architectures. We also described infrastructures that are needed by microservices and how the Azure cluster offers Azure Kubernetes Services and Azure Service Fabric.

Finally, we put these concepts into practice by implementing a Service Fabric application. Here, we looked at the various ways in which Service Fabric applications can be implemented.

The next chapter focuses on how to use ORMs and Entity Framework Core to interact with various kinds of database while keeping our code independent from the database engine we've selected.

Questions

1. What is the two-folded nature of the module concept?
2. Is scaling optimization the only advantage of microservices? If not, list some further advantages.
3. What is Polly?
4. What is ConfigureServices?
5. What Docker support is offered by Visual Studio?
6. What Docker application method is more powerful: the one based on .yml files or the one based on .yaml files?
7. What kinds of port must be declared during the definition of an Azure Service Fabric cluster?
8. Why are partitions of reliable stateful services needed?
9. How can we declare that a remoting communication must be addressed by secondary replicas? What about for other types of communication?

Further reading

The following are links to the official documentation for Azure Service Bus and RabbitMQ, two event bus technologies:

- **Azure Service Bus:** <https://docs.microsoft.com/en-us/azure/service-bus-messaging/>
- **RabbitMQ:** <https://www.rabbitmq.com/getstarted.html>

The documentation for Polly, a tool for reliable communication/tasks, can be found here: <https://github.com/App-vNext/Polly>.

More information on Docker can be found on Docker's official website: <https://docs.docker.com/>.

The official documentation for Kubernetes and .yaml files can be found here: <https://kubernetes.io/docs/home/>.

The official documentation for Azure Kubernetes can be found here: <https://docs.microsoft.com/en-US/azure/aks/>.

The official documentation for Azure Service Fabric can be found here: <https://docs.microsoft.com/en-US/azure/service-fabric/>.

The official documentation for Azure Service Fabric's reliable services can be found here: <https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-services-introduction>.

More information about the Actor model can be found here: https://www.researchgate.NET/publication/234816174_Actors_A_conceptual.foundation_for.concurrent.object-oriented_programming.

The official documentation for Actor models that can be implemented in Azure Service Fabric can be found here: <https://docs.microsoft.com/en-US/azure/service-fabric/service-fabric-reliable-actors-introduction>.

Microsoft has also implemented an advanced actor model that is independent of Service Fabric. This is known as the Orleans framework. More information about Orleans can be found at the following links:

- **Orleans – Virtual Actors:** <https://www.microsoft.com/en-us/research/project/orleans-virtual-actors/?from=https%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fprojects%2Forleans%2F>
- **Orleans Documentation:** <http://dotnet.github.io/orleans/Documentation/>

6

Interacting with Data in C# - Entity Framework Core

As we mentioned in [Chapter 5, Applying a Microservice Architecture to Your Enterprise Application](#), software systems are organized into layers, and each layer communicates with the previous and next layers through interfaces that don't depend on how the layer is implemented. When the software is a Business/Enterprise system, it usually contains at least three layers: the data layer, the business layer, and the presentation layer. In general, the interface that's offered by each layer and the way the layer is implemented depends on the application.

However, it turns out that the functionalities offered by data layers are quite standard, since they just map data from a data storage subsystem into objects and vice versa. This leads to the conceptions of general-purpose frameworks for implementing data layers in a substantially declarative way. These tools are called **Object-Relational Mapping (ORM)** tools since they are data storage subsystems based on relational databases. However, they also work well with the modern non-relational storages classified as NoSQL databases (such as MongoDB and Azure Cosmos DB) since their data model is closer to the target object model than a pure relational model.

In this chapter, we will cover the following topics:

- Understanding ORM basics
- Configuring Entity Framework Core
- Entity Framework Core migrations
- Querying and updating data with Entity Framework Core
- Deploying your data layer
- Understanding Entity Framework Core advanced features – global filters

This chapter describes ORMs and how to configure them, and then focuses on Entity Framework Core, the ORM included in .NET Core.

Technical requirements

This chapter requires Visual Studio 2017 or 2019 free Community Edition or better with all the database tools installed.

All the concepts in this chapter will be clarified with practical examples based on the WWTravelClub book use case. You will find the code for this chapter at <https://github.com/PacktPublishing/Hands-On-Software-Architecture-with-CSharp-8>.

Understanding ORM basics

ORMs map relational DB tables into in-memory collections of objects where object properties correspond to DB table fields. Types from C#, such as Booleans, numeric types, and strings, have corresponding DB types. If GUIDs are not available in the mapped database, while single characters are mapped to DB single-character strings, then types such as GUIDs are mapped to their equivalent string representations. All date and time types are mapped either to C# `DateTime` when date/time contains no time zone information or to `DateTimeOffset` when date/time also contains explicit time zone information. Any DB time duration is mapped to a `TimeSpan`.

Since the string properties of most object-oriented languages have no length limits associated with them (while DB string fields usually have length limits), the DB limits are taken into account in the DB mapping configuration. In general, when the mapping between DB types and object-oriented language types need options to be specified, these options are declared in the mapping configuration.

The way the whole configuration is defined depends on the specific ORM. Entity Framework Core offers three options:

- Data annotations (property attributes)
- Name conventions
- Fluent configuration interface based on configuration objects and methods

While the fluent interface can be used to specify any configuration option, the data annotations and name conventions can be used for a smaller subset of them.

Each ORM adapts to a specific DB type (Oracle, MySQL, SQL Server, and so on) with DB-specific adapters called **providers** or **connectors**. Entity Framework Core has providers for most of the available DB engines.



A complete list of providers can be found at <https://docs.microsoft.com/en-US/ef/core/providers/>.

Adapters are necessary for the differences in DB types, for the way transactions are handled, and for all the other features that are not standardized by the SQL language.

Relations among tables are represented with object pointers. For instance, in a one-to-many relationship, the class that's mapped to the *one* side of the relationship contains a collection that is populated with the related objects on the *many* side of the relationship. On the other hand, the class mapped to the *many* side of the relationship has a simple property that is populated with the uniquely related object on the *one* side of the relationship.

The whole database (or just a part of it) is represented by an in-memory cache class that contains a property for each collection that's mapped to a DB table. First, the query and update operations are performed on an instance of an in-memory cache class, and then this instance is synchronized with the database. The in-memory cache class that's used by Entity Framework Core is called `DbContext` and it also contains the mapping configuration. More specifically, the application-specific in-memory cache class is obtained by inheriting `DbContext` and adding it to all the mapped collections and all the necessary configuration information.

Summing up, `DbContext` subclass instances contain partial snapshots of the DB that are synchronized with the database to get/update the actual data.

DB queries are performed with a query language made of method calls on the collections of the in-memory cache class. The actual SQL is created and executed during the synchronization stage. For instance, Entity Framework Core performs **Language Integrated Queries (LINQ)** on the collections mapped to the DB tables.

In general, LINQ queries produce `IEnumerable` instances, that is, collections whose elements are not computed when `IEnumerable` is created at the end of the query, but when you actually attempt to retrieve the collection elements from `IEnumerable`. This works as follows:

- LINQ queries that start from a mapped collection of a `DbContext` create a specific subclass of `IEnumerable` called `IQueryable`.
- An `IQueryable` contains all the information that's needed to issue a query to the database, but the actual SQL is produced and executed when the first element of the `IQueryable` is retrieved.

- Thus, in the case of Entity Framework Core, the synchronization with the database is performed when an element is actually retrieved from the final `IQueryable`.
- Typically, each Entity Framework query ends with a `ToListAsync` or `Toarray` operation that transforms the `IQueryable` into a list or array, thereby causing the actual execution of the query on the database.
- In case the query is expected to return just a single element or no element at all, we typically execute a `FirstOrDefault` operation that returns a single element, if any, or `null`.

Also, updates, deletions, and additions of new entities to a DB table are performed by mimicking these operations on a `DbContext` collection property that represents the database table. However, entities may only be updated or deleted this way after they have been loaded in that memory collection by means of a query. An update query requires the in-memory representation of the entity to be modified as needed, while a delete query requires the in-memory representation of the entity to be removed from its in-memory mapped collection. In Entity Framework Core, the removal operation is performed by calling the `Remove(entity)` method of the collection.

The addition of a new entity has no further requirements. It is enough to add the new entity to the in-memory collection. Updates, deletes, and additions that are performed on various in-memory collections are actually passed to the database with an explicit call to a DB synchronization method. For instance, Entity Framework Core passes all the changes that are performed on a `DbContext` instance to the database when you call the `DbContext.SaveChanges()` method.

Changes that are passed to the database during a synchronization operation are executed in a single transaction. Moreover, for ORMs, such as Entity Framework Core, that have an explicit representation of transactions, a synchronization operation is executed in the scope of a transaction, since it uses that transaction instead of creating a new one.

The remaining sections in this chapter explain how to use Entity Framework Core, along with some example code based on this book's WWTravelClub use case.

Configuring Entity Framework Core

Since Database handling is confined within a dedicated application layer, it is good practice to define your Entity Framework Core (`DbContext`) in a separate library. Accordingly, we need to define a .NET Core class library project. As we discussed in the *Book use case – .NET Core in action, Main Types of .NET Core projects* section of Chapter 2, *Functional and Nonfunctional Requirements*, we have two different kinds of library projects: **.NET Standard** and **.NET Core**.

While .NET Core libraries are tied to a specific .NET Core version, .NET Standard 2.0 libraries have a wide range of applications since they work with any .NET version greater than 2.0 and also with the classical .NET Framework.

However, the `Microsoft.EntityFrameworkCore` package (which we need in our DB layer) depends just on .NET Standard 2.0. It is designed to work with a specific .NET Core version (its version numbers are the same as the .NET Core versions). Therefore, if we define our DB layer as .NET Standard 2.0, the

specific `Microsoft.EntityFrameworkCore` package that we add as a dependency may conflict with another version of the same library contained in another system component that's tied to a specific .NET Core version.

Since our library is not a general-purpose library (it's just a component of a specific application), it is preferable to tie it to a specific .NET Core version than to track its version dependencies in the whole design of our application. Therefore, let's choose a .NET Core library project for the latest .NET Core version installed on our machine. Our .NET Core library project can be created and prepared as follows:

1. Open Visual Studio and define a new solution named `WWTravelClubDB` and then select **Class Library (.NET Core)** for the latest .NET Core version available.
2. We must install all Entity Framework Core-related dependencies. The simplest way to have all the necessary dependencies installed is to add the NuGet package for the provider of the database engine we are going to use – in our case, SQL Server – as we mentioned in Chapter 4, *Deciding on the Best Cloud-Based Solution*. In fact, any provider will install all the required packages since it has all of them as dependencies. So, let's add the latest stable version of `Microsoft.EntityFrameworkCore.SqlServer`. If you plan to use several database engines, you can also add other providers since they can work side by side. Later in this chapter, we will install other NuGet packages that contain tools that we need to process our Entity Framework Core. Then, we will explain how to install further tools that are needed to process Entity Framework Core's configuration.

3. Let's rename the default `Class1` class to `MainDbContext`. This was automatically added to the class library.
4. Now, let's replace its content with the following code:

```
using System;
using Microsoft.EntityFrameworkCore;

namespace WWTravelClubDB
{
    public class MainDbContext : DbContext
    {
        public MainDbContext(DbContextOptions options)
            : base(options)
        {
        }

        protected override void OnModelCreating(ModelBuilder
builder)
        {
        }
    }
}
```

5. We inherit from `DbContext` and we are required to pass `DbContextOptions` to the `DbContext` constructor. `DbContextOptions` contains creation options such as the database connection string, which depend on the target DB engine.
6. All the collections that have been mapped to database tables will be added as properties of `MainDbContext`. The mapping configuration will be defined inside of the overridden `OnModelCreating` method with the help of the `ModelBuilder` object passed as a parameter.

The next step is the creation of all the classes that represent all the DB table rows. These are called **entities**. We need an entity class for each DB table we want to map. Let's create a `Models` folder in the project root for all of them. The next subsection explains how to define all the required entities.

Defining DB entities

DB design, like the whole application design, is organized in iterations. Let's suppose that, in the first iteration, we need a prototype with two database tables: one for all the travel packages and another one for all the locations referenced by the packages. Each package covers just one location, while a single location may be covered by several packages, so the two tables are connected by a one-to-many relationship.

So, let's start with the location database table. As we mentioned at the end of the previous section, we need an entity class to represent the rows of this table. Let's call `Destination` the entity class:

```
namespace WWTravelClubDB.Models
{
    public class Destination
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Country { get; set; }
        public string Description { get; set; }
    }
}
```

All the DB fields must be represented by read/write C# properties. Suppose that each destination is something like a town or a region that can be defined by just its name and the country it is in, and that all the relevant information is contained in its `Description`. In future iterations, we will probably add several more fields. `Id` is an auto-generated key.

However, now, we need to add information about how all the fields are mapped to DB fields. In Entity Framework Core, all the primitive types are mapped automatically to DB types by the DB engine-specific provider that's used (in our case, SQL Server provider). Our only preoccupations are as follows:

- **Length limits on the string:** They can be taken into account by applying adequate `MaxLength` and `MinLength` attributes to each string property. All the attributes that are useful for the entity's configuration are contained in the `System.ComponentModel.DataAnnotations` and `System.ComponentModel.DataAnnotations.Schema` namespaces. Therefore, it's good practice to add both of them to all the entity definitions.
- **Specifying which fields are obligatory and which ones are optional:** By default, all the reference types (such as all the strings) are assumed to be optional, while all the value types (numbers and GUIDs, for instance) are assumed to be obligatory. If we want a reference type to be obligatory, then we must decorate it with the `Required` attribute. However, if we want a `T` value type property to be optional, then we must replace it with `T?`.

- **Specifying which property represents the primary key:** The key may be specified by decorating a property with the Key attribute. However, if no Key attribute is found, a property named Id (if there is one) is taken as the primary key. In our case, there is no need for the Key attribute. If the primary key is composed of several properties, it is enough to add the Key attribute to all of them.

Since each destination is on the *one* side of a one-to-many relationship, it must contain a collection for the related package entities; otherwise, we will not be able to refer to the related entities in the clauses of our LINQ queries.

Putting everything together, the final version of the Destination class is as follows:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace WWTravelClubDB.Models
{
    public class Destination
    {
        public int Id { get; set; }
        [MaxLength(128), Required]
        public string Name { get; set; }
        [MaxLength(128), Required]
        public string Country { get; set; }
        public string Description { get; set; }
        public ICollection<Package> Packages { get; set; }
    }
}
```

Since the Description property has no length limits, it will be implemented with a SQL Server ntext field of indefinite length. We can write the code for the Package class in a similar way:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace WWTravelClubDB.Models
{
    public class Package
    {
        public int Id { get; set; }
        [MaxLength(128), Required]
        public string Name { get; set; }
        [MaxLength(128)]
```

```
        public string Description { get; set; }
        public decimal Price { get; set; }
        public int DurationInDays { get; set; }
        public DateTime? StartValidityDate { get; set; }
        public DateTime? EndValidityDate { get; set; }
        public Destination MyDestination { get; set; }
        public int DestinationId { get; set; }
    }
}
```

Each package has a duration in days, as well as optional start and stop dates in which the package offer is valid. `MyDestination` connects packages with their destinations in the many-to-one relationship that they have with the `Destination` entity, while `DestinationId` is the external key of the same relation.

While it is not obligatory to specify the external key, it is good practice to do so since this is the only way to specify some properties of the relationship. For instance, in our case, since `DestinationId` is an `int` (value type), it is obligatory. Therefore, the relationship here is one-to-many and not `(0, 1)`-to-many. Defining `DestinationId` as `int?`, instead of `int`, would turn the one-to-many relationship into a `(0, 1)`-to-many relationship.

In the next section, we will explain how to define the in-memory collection that represents the database tables.

Defining the mapped collections

Once we have defined all the entities that are object-oriented representations of the database rows, we need to define the in-memory collections that represent the database tables themselves. As we mentioned in the *ORM basics* section, all the database operations are mapped to the operations on these collections (the *Querying and updating data with Entity Framework Core* section of this chapter explains *how*). It is enough to add a `DbSet<T>` collection property to our `DbContext` for each entity, `T`. Usually, the name of each of these properties is obtained by pluralizing the entity name. Thus, we need to add the following two properties to our `MainDbContext`:

```
public DbSet<Package> Packages { get; set; }
public DbSet<Destination> Destinations { get; set; }
```

Up until now, we've translated database stuff into properties, classes, and data annotations. However, Entity Framework needs further information to interact with a database. The next subsection explains how to provide them.

Completing the mapping configuration

The mapping configuration information that we couldn't specify in the entity definitions must be added in the `OnModelCreating` `DbContext` method. Each configuration information relative to an entity, `T`, starts with `builder.Entity<T>()` and continues with a call to a method that specifies that kind of constraint. Further nested calls specify further properties of the constraint. For instance, our one-to-many relationship may be configured as follows:

```
builder.Entity<Destination>()
    .HasMany(m => m.Packages)
    .WithOne(m => m.MyDestination)
    .HasForeignKey(m => m.DestinationId)
    .OnDelete(DeleteBehavior.Cascade);
```

The two sides of the relationship are specified through the navigation properties that we added to our entities. `HasForeignKey` specifies the external key. Finally, `OnDelete` specifies what to do with packages when a destination is deleted. In our case, it performs a cascade delete of all the packages related to that destination.

The same configuration can be defined by starting from the other side of the relationship, that is, starting with `builder.Entity<Package>()`:

```
builder.Entity<Package>()
    .HasOne(m => m.MyDestination)
    .WithMany(m => m.Packages)
    .HasForeignKey(m => m.DestinationId)
    .OnDelete(DeleteBehavior.Cascade);
```

The only difference is that the previous statement's `HasMany-WithOne` methods are replaced by the `HasOne-WithMany` methods since we started from the other side of the relationship.

The `ModelBuilder` `builder` object allows us to specify database indexes with something such as the following:

```
builder.Entity<T>()
    .HasIndex(m => m.PropertyName);
```

Multi-property indexes are defined as follows:

```
builder.Entity<T>()
    .HasIndex("propertyName1", "propertyName2", ...);
```

If we add all the necessary configuration information, then our `OnModelCreating` method will look as follows:

```
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.Entity<Destination>()
        .HasMany(m => m.Packages)
        .WithOne(m => m.MyDestination)
        .HasForeignKey(m => m.DestinationId)
        .OnDelete(DeleteBehavior.Cascade);

    builder.Entity<Destination>()
        .HasIndex(m => m.Country);

    builder.Entity<Destination>()
        .HasIndex(m => m.Name);

    builder.Entity<Package>()
        .HasIndex(m => m.Name);

    builder.Entity<Package>()
        .HasIndex("StartValidityDate", "EndValidityDate");
}
```

Once you've configured Entity Framework Core, we can use all the configuration information we have to create the actual database and put all the tools we need in place in order to update the database's structure as the application evolves. The next section explains how.

Entity Framework Core migrations

Now that we've configured Entity Framework and defined our application-specific `DbContext` subclass, we can use the Entity Framework Core design tools to generate the physical database and create the database structure snapshot that's needed by Entity Framework Core to interact with the database.

Entity Framework Core design tools must be installed in each project that needs them as NuGet packages. There are two equivalent options:

- **Tools that work in any Windows console:** These are available through the `Microsoft.EntityFrameworkCore.Design` NuGet package. All Entity Framework Core commands are in `dotnet ef` format since they are contained in the `ef` command line's .NET Core application.
- **Tools that are specific to the Visual Studio Package Manager Console:** These are contained in the `Microsoft.EntityFrameworkCore.Tools` NuGet package. They don't need the `dotnet ef` prefix since they can only be launched from the **Package Manager Console** inside of Visual Studio.

Entity Framework Core's design tools are used within the design/update procedure. This procedure is as follows:

1. We modify `DbContext` and Entities' definitions as needed.
2. We launch the design tools to ask Entity Framework Core to detect and process all the changes we made.
3. Once launched, the design tools update the database structure snapshot and generate a new *migration*, that is, a file containing all the instructions we need in order to modify the physical database to reflect all the changes we made.
4. We launch another tool to update the database with the newly created migration.
5. We test the newly configured DB layer and, if new changes are necessary, we go back to *step 1*.
6. When the data layer is ready, it is deployed in staging or production, where all the migrations are applied once more to the actual staging/production database.

This is repeated several times in the various software project iterations and during the lifetime of the application. If we operate on an already existing database, we need to configure `DbContext` and its models to reflect the existing structure of all the tables we want to map. Then, we call the design tools with an `IgnoreChanges` option so that they generate an empty migration. Also, this empty migration must be passed to the physical database so that it can synchronize a database structure version associated with the physical database with the version that's been recorded in the database snapshot. This version is important because it determines which migrations must be applied to a database and which ones have already been applied.

The whole design process needs a test/design database and, if we operate on an existing database, the structure of this test/design database must reflect the actual database – at least in terms of the tables we want to map. To enable design tools so that we can interact with the database, we must define the `DbContextOptions` options that they pass to the `DbContext` constructor. These options are important at design time since they contain the connection string of the test/design database. The design tools can be informed about our `DbContextOptions` options if we create a class that implements the `IDesignTimeDbContextFactory<T>` interface, where `T` is our `DbContext` subclass:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;

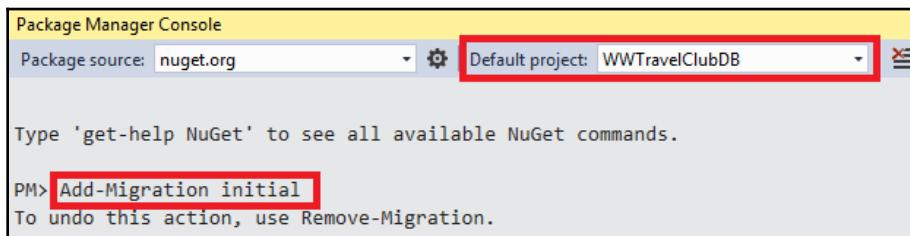
namespace WWTravelClubDB
{
    public class LibraryDesignTimeDbContextFactory
        : IDesignTimeDbContextFactory<MainDbContext>
    {
        private const string connectionString =
            @"Server=(localdb)\mssqllocaldb;Database=wwtravelclub;
            Trusted_Connection=True;MultipleActiveResultSets=true";
        public MainDbContext CreateDbContext(params string[] args)
        {
            var builder = new DbContextOptionsBuilder<MainDbContext>();
            builder.UseSqlServer(connectionString);
            return new MainDbContext(builder.Options);
        }
    }
}
```

`connectionString` will be used by Entity Framework to create a new database in the local SQL Server instance that's been installed in the development machine and connects with Windows credentials. You are free to change it to reflect your needs.

Now, we are ready to create our first migration! Let's get started:

1. Let's go to the Package Manager Console and ensure that `WWTravelClubDB` is selected as our default project.

2. Now, type `Add-Migration initial` and press *Enter* to issue this command:

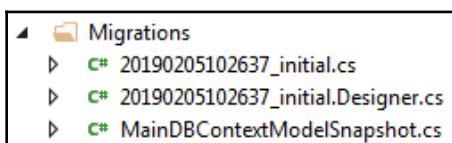


The screenshot shows the Package Manager Console window. At the top, it says "Package Manager Console" and "Package source: nuget.org". A red box highlights the "Default project: WWTravelClubDB" dropdown. Below the header, it says "Type 'get-help NuGet' to see all available NuGet commands.". In the main area, there is a command prompt: "PM> **Add-Migration initial**". Below the command, a note says "To undo this action, use Remove-Migration.".



`initial` is the name we gave our first migration. So, in general, the command is `Add-Migration <migration name>`. When we operate on an existing database, we must add the `-IgnoreChanges` option to the first migration (and just to that) so that an empty migration is created. References to the whole set of commands can be found in the *Further reading* section.

3. If, after having created the migration, but before having applied the migration to the database, we realize we made some errors, we can undo our action with the `Remove-Migration` command. If the migration has already been applied to the database, the simplest way to correct our error is to make all the necessary changes to the code and then apply another migration.
4. As soon as the `Add-Migration` command is executed, a new folder appears in our project:



`20190205102637_initial.cs` is our migration expressed in an easy to understand language.

You may review the code to verify that everything is okay and you may also modify the migration content (only if you are enough of an expert to do it reliably). Each migration contains an `Up` method and a `Down` method. The `Up` method implies the migration, while the `Down` method undoes its changes. Accordingly, the `Down` method contains the reverse actions of all the actions included in the `Up` method in reverse order.

20190205102637_initial.Designer.cs is the Visual Studio designer code you *mustn't* modify, while MainDbContextModelSnapshot.cs is the overall database structure snapshot. If you add further migrations, new migration files and their designer counterparts will appear and the unique MainDbContextModelSnapshot.cs database structure snapshot will be updated to reflect the database's overall structure.

The same command can be issued in a Windows console by typing `dotnet ef migrations add initial`. However, this command must be issued from within the project's root folder (not from within the solution's root folder).

Migrations can be applied to the database by typing `Update-Database` in the Package Manager Console. The equivalent Windows console command is `dotnet ef database update`. Let's try using this command to create the physical database!

The next subsection explains how to create database stuff that Entity Framework is unable to create automatically. After that, in the next section, we will use Entity Framework's configuration and the database we generated with `dotnet ef database update` to create, query, and update data.

Understanding stored procedures and direct SQL commands

Some database structures can't be generated automatically by the Entity Framework Core commands and declarations we described previously. For instance, Entity Framework Core can't generate automatically stored procedures. Stored procedures such as generic SQL strings can be included manually in the `Up` and `Down` methods through the `migrationBuilder.Sql("<sql scommand>")` method.

The safest way to do this is by adding a migration without performing any configuration changes so that the migration is empty when it's created. Then, we can add the necessary SQL commands to the empty `Up` method of this migration and their converse commands in the empty `Down` method. It is good practice to put all the SQL strings in the properties of resource files (`.resx` files).

Now, you are ready to interact with the database through Entity Framework Core.

Querying and updating data with Entity Framework Core

To test our DB layer, we need to add a console project based on the same .NET Core version as our library to the solution. Let's get started:

1. Let's call the new console project WWTravelClubDBTest.
2. Now, we need to add our data layer as a dependency of the console project by right-clicking on the *References* node of the console project and selecting *Add reference*.
3. Remove the content of the `Main` static method in the `program.cs` file and start by writing the following:

```
Console.WriteLine("program start: populate database");
Console.ReadKey();
```

4. Then, add the following namespaces at the top of the file:

```
using WWTravelClubDB;
using WWTravelClubDB.Models;
using Microsoft.EntityFrameworkCore;
using System.Linq;
```

Now that we have finished preparing our test project, we can experiment with queries and data updates. Let's start by creating some database objects, that is, some destinations and packages. Follow these steps to do so:

1. First, we must create an instance of our `DbContext` subclass with an appropriate connection string. We can use the `same LibraryDesignTimeDbContextFactory` class that's used by the design tools to get it:

```
var context = new LibraryDesignTimeDbContextFactory()
    .CreateDbContext();
```

2. New rows can be created by simply adding class instances to the mapped collections of our `DbContext` subclass. If a `Destination` instance has packages associated with it, we can simply add them to its `Packages` property:

```
var firstDestination= new Destination
{
    Name = "Florence",
    Country = "Italy",
    Packages = new List<Package>()
```

```
{  
    new Package  
    {  
        Name = "Summer in Florence",  
        StartValidityDate = new DateTime(2019, 6, 1),  
        EndValidityDate = new DateTime(2019, 10, 1),  
        DurationInDays=7,  
        Price=1000  
    },  
    new Package  
    {  
        Name = "Winter in Florence",  
        StartValidityDate = new DateTime(2019, 12, 1),  
        EndValidityDate = new DateTime(2020, 2, 1),  
        DurationInDays=7,  
        Price=500  
    }  
};  
context.Destinations.Add(firstDestination);  
context.SaveChanges();  
Console.WriteLine(  
    "DB populated: first destination id is "+  
    firstDestination.Id);  
Console.ReadKey();
```

There is no need to specify primary keys since they are auto-generated and will be filled in by the database. In fact, after the `SaveChanges()` operation synchronizes our context with the actual DB, the `firstDestination.Id` property has a non-zero value. The same is true for the primary keys of `Package`.

When we declare that an entity (in our case, a `Package`) is a child of another entity (in our case, a `Destination`) by inserting it in a father entity collection (in our case, the `Packages` collection), there is no need to explicitly set its external key (in our case, `DestinationId`) since it is inferred automatically by Entity Framework Core. Once created and synchronized with the `firstDestination` database, we can add further packages in two different ways:

- Create a `Package` class instance, set its `DestinationId` external key to `firstDestination.Id` and add it to `context.Packages`
- Create a `Package` class instance, with no need to set its external key, and then add it to the `Packages` collection of its father `Destination` instance.

The latter option is the only possibility when a child entity (`Package`) is added with its father entity (`Destination`) and the father entity has an auto-generated principal key since, in this case, the external key isn't available at the time we perform the additions. In most of the other circumstances, the former option is simpler since the second option requires the father `Destination` entity to be loaded in memory, along with its `Packages` collection, that is, together with all the packages associated with the `Destination` object (by default, connected entities aren't loaded by queries).

Now, let's say we want to modify the *Florence* destination and give a 10% increment to all *Florence* packages prices. How do we proceed? Follow these steps to find out how:

1. First, we need to load the entity into memory with a query, modify it, and call `SaveChanges()` to synchronize our changes with the database. If we want to modify, say, just its description, a query such as the following is enough:

```
var toModify = context.Destinations
    .Where(m => m.Name == "Florence").FirstOrDefault();
```

2. We need to load all the related destination packages that are not loaded by default. This can be done with the `Include` clause, as follows:

```
var toModify = context.Destinations
    .Where(m => m.Name == "Florence")
    .Include(m => m.Packages)
    .FirstOrDefault();
```

3. After that, we can modify the description and package prices, as follows:

```
toModify.Description =
    "Florence is a famous historical Italian town";
foreach (var package in toModify.Packages)
    package.Price = package.Price * 1.1m;
context.SaveChanges();

var verifyChanges= context.Destinations
    .Where(m => m.Name == "Florence")
    .FirstOrDefault();

Console.WriteLine(
    "New Florence description: " +
    verifyChanges.Description);
Console.ReadKey();
```

So far, we've performed queries whose unique purpose is to update the retrieved entities. Next, we will explain how to retrieve information that will be shown to the user and/or be used by complex business operations.

Returning data to the presentation layer

To keep the layers separated and to adapt queries to the data that's actually needed by each *use case*, DB entities aren't sent as they are to the presentation layer. Instead, the data is projected into smaller classes that contain the information that's needed by the *use case*. These are implemented by the presentation layer's caller method. Objects that move data from one layer to another are called **Data Transport Objects (DTOs)**. As an example, let's create a DTO containing the summary information that is worth showing when returning a list of packages to the user (we suppose that, if needed, the user can get more details by clicking the package they are interested in):

1. Let's add a DTO to our WWTravelClubDBTest project that contains all the information that needs to be shown in a list of packages:

```
namespace WWTravelClubDBTest
{
    public class PackagesListDTO
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
        public int DurationInDays { get; set; }
        public DateTime? StartValidityDate { get; set; }
        public DateTime? EndValidityDate { get; set; }
        public string DestinationName { get; set; }
        public int DestinationId { get; set; }
        public override string ToString()
        {
            return string.Format("{0}. {1} days in {2}, price: {3}", Name, DurationInDays, DestinationName, Price);
        }
    }
}
```



We don't need to load entities in memory and then copy their data into the DTO, but database data can be projected directly into the DTO, thanks to the LINQ `Select` clause. This minimizes how much data is exchanged with the database.

2. As an example, we can populate our DTOs with a query that checks all the packages that are available around the 10th of August:

```
var period = new DateTime(2019, 8, 10);
var list = context.Packages
    .Where(m => period >= m.StartValidityDate
    && period <= m.EndValidityDate)
```

```
.Select(m => new PackagesListDTO
{
    StartValidityDate=m.StartValidityDate,
    EndValidityDate=m.EndValidityDate,
    Name=m.Name,
    DurationInDays=m.DurationInDays,
    Id=m.Id,
    Price=m.Price,
    DestinationName=m.MyDestination.Name,
    DestinationId = m.DestinationId
})
.ToList();
foreach (var result in list)
    Console.WriteLine(result.ToString());
Console.ReadKey();
```

3. In the `Select` clause, we can also navigate to any related entities to get the data we need. For instance, the preceding query navigates to the related `Destination` entity to get the `Package` destination name.
4. Now, right-click on the `WWTravelClubDBTest` project in the Solution Explorer and set it as the start project. Then, run the solution.
5. The programs stop at each `Console.ReadKey()` method, waiting for you to hit any key. This way, you have time to analyze the output that's produced by all the code snippets that we added to the `Main` method.

Now, we will learn how to handle operations that can't be efficiently mapped to the immediate operations in the in-memory collections that represent the database tables.

Issuing direct SQL commands

Not all database operations can be executed efficiently by querying the database with LINQ and updating in-memory entities. For instance, counter increments can be performed more efficiently with a single SQL instruction. Moreover, some operations can be executed with acceptable performance if we define adequate stored procedures/SQL commands. In these cases, we are forced to either issue direct SQL commands to the database or call database stored procedures from our Entity Framework code. There are two possibilities: SQL statements that perform database operations but do not return entities, and SQL statements that do return entities.

SQL commands that don't return entities can be executed with the `DbContext` method, as follows:

```
int DbContext.Database.ExecuteSqlRaw(string sql, params object []
parameters)
```

Parameters can be referenced in the string as `{0}`, `{1}`, ..., `{n}`. Each `{m}` is filled with the object contained at the `m` index of the `parameters` array, which is converted from a .NET type into the corresponding SQL type. The method returns the number of affected rows.

SQL commands that return collections of entities must be issued through the `FromSqlRaw` method of the mapped collection associated with those entities:

```
context.<mapped collection>.FromSqlRaw(string sql, params object []
parameters)
```

Thus, for instance, a command that returns `Package` instances would look something like this:

```
var results = context.Packages.FromSqlRaw("<some sql>", par1, par2,
...).ToList();
```

SQL strings and parameters work like this in the `ExecuteSqlRaw` method. The following is a simple example:

```
var allPackages = context.Packages.FromSqlRaw(
    "SELECT * FROM Products WHERE Name = {0}",
    myPackageName)
```

It is good practice to put all the SQL strings in resource files and encapsulate all the `ExecuteSqlRaw` and `FromSqlRaw` calls inside the public methods that you defined in your `DbContext` subclasses, in order to keep the dependence from a specific database inside of your Entity Framework Core-based data layer.

Handling transactions

All the changes that are made to a `DbContext` instance are passed in a single transaction at the first `SaveChanges` call. However, sometimes, it is necessary to include queries and updates in the same transaction. In these cases, we must handle the transaction explicitly. Several entity Framework Core commands can be included in a transaction if we put them inside a `using` block associated with a transaction object:

```
using (var dbContextTransaction = context.Database.BeginTransaction())
{
    try{
        ...
        ...
        dbContextTransaction.Commit();
    }
    catch
    {
        dbContextTransaction.Rollback();
    }
}
```

In the preceding code, `context` is an instance of our `DbContext` subclass. Inside of the `using` block, the transaction can be aborted and committed by calling its `Rollback` and `Commit` methods. Any `SaveChanges` calls that are included in the transaction block use the transaction they are already in, instead of creating new ones.

Deploying your data layer

When your database layer is deployed in production or in staging, usually, an empty database already exists, so you must apply all the migrations in order to create all the database objects. This can be done by calling `context.Database.Migrate()`. The `Migrate` method applies the migrations that haven't been applied to the databases yet, so it may be called safely several times during the application's lifetime. `context` is an instance of our `DbContext` class that must be passed through a connection string with enough privileges to create tables and to perform all the operations included in our migrations. Thus, typically, this connection string is different from the string we will use during normal application operations.

During the deployment of a web application on Azure, we are given the opportunity to check migrations with a connection string we provide. We can also check migrations manually by calling the `context.Database.Migrate()` method when the application starts. This will be discussed in detail in Chapter 13, *Presenting ASP.NET Core MVC*, which is dedicated to ASP.NET MVC Web applications.

For desktop applications, we can apply migrations during the installation of the application and of its subsequent updates.

At the first application installation and/or in subsequent application updates, we may need to populate some tables with initial data. For Web applications this operation can be performed at application start, while for desktop application this operation can be included in the installation.

Database tables can be populated with Entity Framework Core commands. First, though, we need to verify whether the table is empty in order to avoid adding the same table rows several times. This can be done with the `Any()` LINQ method, as shown in the following code:

```
if (!context.Destinations.Any())
{
    //populate here the Destinations table
}
```

Let's take a look at a few advanced features that Entity Framework Core has to share.

Understanding Entity Framework Core advanced feature – global filters

Global filters were introduced at the end of 2017. They enable techniques such as soft delete and multi-tenant tables that are shared by several users, where each user just *sees* its records.

Global filters are defined with the `modelBuilder` object, which is available in the `DbContext OnModelCreating` method. The syntax for this method is as follows:

```
modelBuilder.Entity<MyEntity>().HasQueryFilter(m => <define filter
condition here>);
```

For instance, if we add an `IsDeleted` property to our `Package` class, we may soft delete a `Package` without removing it from the database by defining the following filter:

```
modelBuilder.Entity<Package>().HasQueryFilter(m => !m.IsDeleted);
```

However, filters contain `DbContext` properties. Thus, for instance, if we add a `CurrentUserId` property to our `DbContext` subclass (whose value is set as soon as a `DbContext` instance is created), then we can add a filter like the following one to all the entities that refer to a user ID:

```
modelBuilder.Entity<Document>().HasQueryFilter(m => m.UserId == CurrentUserId);
```

With the preceding filter in place, the currently logged user can only access the documents they own (the ones that have their `UserId`). Similar techniques are very useful in the implementation of multi-tenant applications.

Summary

In this chapter, we looked at the essentials of ORM basics and why they are so useful. Then, we described Entity Framework Core. In particular, we discussed how to configure the database mappings with class annotations and other declarations and commands that are included in `DbContext` subclasses.

Then, we discussed how to automatically create and update the physical database structure with the help of migrations, as well as how to query and pass updates to the database through Entity Framework Core. Finally, we learned how to pass direct SQL commands and transactions through Entity Framework Core, as well as how to deploy a data layer based on Entity Framework Core.

This chapter also reviewed some of the advanced features that had been introduced in the latest Entity Framework Core releases.

In the next chapter, we will discuss how Entity Framework Core can be used with NoSQL data models and the various types of storage options that are available in the cloud and, in particular, in Azure.

Questions

1. How does Entity Framework Core adapt to several different database engines?
2. How are primary keys declared in Entity Framework Core?
3. How is a string field's length declared in Entity Framework Core?
4. How are indexes declared in Entity Framework Core?
5. How are relations declared in Entity Framework Core?
6. What are the two important migration commands?
7. By default, are related entities loaded by LINQ queries?
8. Is it possible to return database data in a class instance that isn't a database entity? If yes, how?
9. How are migrations applied in production and staging?

Further reading

- More details about migrations commands can be found at <https://docs.microsoft.com/en-US/ef/core/miscellaneous/cli/index> and in the other links contained there.
- More details about Entity Framework Core can be found in the official Microsoft documentation: <https://docs.microsoft.com/en-us/ef/core/>.
- An exhaustive set of examples of complex LINQ queries can be found here: <https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b>.

7

How to Choose Your Data Storage in the Cloud

Azure, like other clouds, offers a wide range of storage devices. The simplest approach is to define a scalable set of virtual machines hosted in the cloud where we can implement our custom solutions. For instance, we can create a SQL Server cluster on our cloud-hosted virtual machines to increase reliability and computational power. However, usually, custom architectures aren't the optimal solution and don't take full advantage of the opportunities offered by the cloud infrastructure.

Therefore, this chapter will not discuss such custom architectures but will focus mainly on the various **Storage as a Service (SaaS)** offerings that are available in the cloud and, in particular, on Azure. These offers include scalable solutions based on plain disk space, relational databases, NoSQL databases, and in-memory data stores such as Redis.

Choosing a more adequate storage type is based not only on the application's functional requirements but also on performance and scaling-out requirements. In fact, while scaling-out when processing resources causes a linear increase in performance, scaling-out storage resources doesn't necessarily imply an acceptable increase in performance. In a few words, no matter how much you duplicate your data storage devices, if several requests affect exactly the same chunk of data, they will always queue the same amount of time to access it!

Scaling-out data causes linear increases of read operation throughput since each copy can serve a different request, but it doesn't imply the same increase in the throughput for write operations since all copies of the same chunk of data must be updated! Accordingly, more sophisticated techniques are required to scale-out storage devices, and not all storage engines scale equally well.

In particular, relational databases don't scale well in all scenarios. Therefore, scaling needs and the need to distribute data geographically play a fundamental role in the choice of a storage engine, as well as in the choice of a SaaS offering.

In this chapter, we will cover the following topics:

- Understanding the different repositories for different purposes
- Choosing between structure or NoSQL storage
- Azure Cosmos DB – an opportunity to manage a multi-continental database
- Use case – storing data

Let's get started:

Technical requirements

This chapter requires that you have the following:

- Visual Studio 2017 or 2019 free Community Edition or better with all its database tools installed.
- A free Azure account. The *Creating an Azure account* subsection in Chapter 1, *Understanding the Importance of Software Architecture*, explains how to create one.
- For a better development experience, we advise that you also install the local emulator of Cosmos DB, which can be found at <https://aka.ms/cosmosdb-emulator>.

Understanding the different repositories for different purposes

This section describes the functionalities that are offered by the most popular data storage techniques. Mainly, we will focus on the functional requirements they are able to satisfy. Performance and scaling-out features will be analyzed in the next section, which is dedicated to comparing relational and NoSQL databases. In Azure, the various offerings can be found by typing product names into the search bar at the top of all Azure portal pages.

The following subsections describe the various kinds of database that we can use in our C# projects.

Relational databases

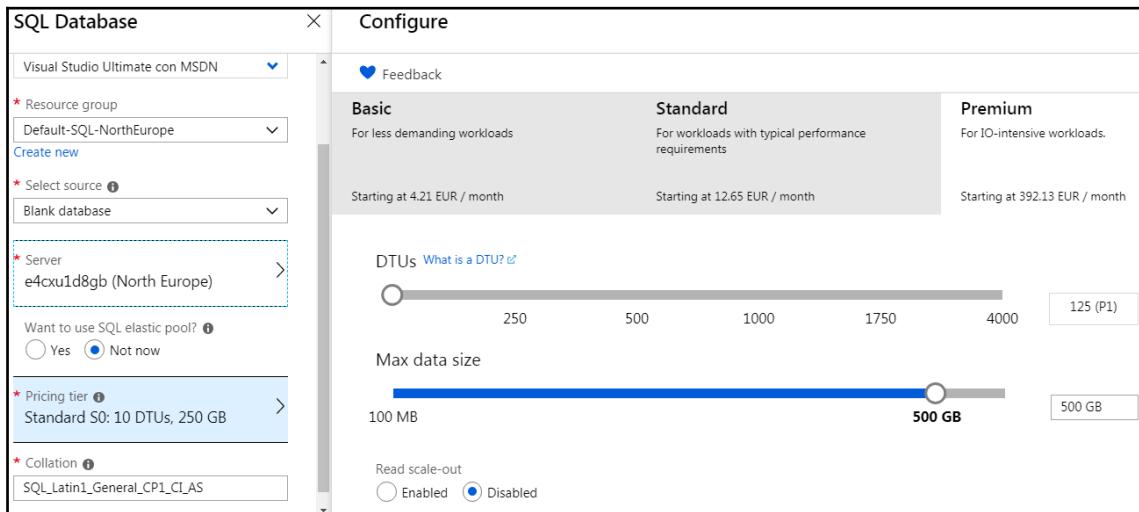
Usually, clouds offer several database engines. Azure offers a variety of popular database engines, such as SQL Server (Azure SQL Server), MySQL, and Oracle.

With regard to the Oracle database engine, Azure offers configurable virtual machines with various Oracle editions installed on them, which you can easily verify by the suggestions you get after typing `Oracle` into the Azure portal search bar. Azure fees don't include Oracle licenses; they just bring computation time, so you must bring your own license to Azure.

With MySQL on Azure, you pay to use a private server instance. The fees you incur depend on the number of cores you have, how much memory has to be allocated, and on backup retention time. MySQL instances are redundant and you can choose between a local or geographically distributed redundancy:

Category	Value
Cost per vCore	65.32
vCores selected	4
General Purpose Storage Cost per GB / month	0.12
Storage amount selected	100
EST. MONTHLY COST	272.83 EUR

Azure SQL Server is the most flexible offer. Here, you can configure resources that are used by every single database. When you create a database, you have the option to place it on an existing server instance or create a new instance. Fees are based on the database memory that's been reserved and on the required **Database Transaction Units (DTUs)**. Here, a DTU is a linear combination of I/O operations, CPU usage, and memory usage that's determined by a reference workload. Roughly, maximal DB performance increases linearly when you increase DTUs:



You can also configure data replication by enabling **Read scale-out**. This way, you can improve the performance of read operations. Backup retention is fixed for each offering level (basic, standard, and premium).

If you select **Yes** for **Want to use SQL elastic pool?**, the database will be added to an elastic pool. Databases that are added to the same elastic pool will share their resources, so resources that aren't used by a database can be used during the *usage peaks* of other databases. Elastic pools can contain databases hosted on different server instances. Elastic pools are an efficient way to optimize resource usage in order to reduce costs.

NoSQL databases

In NoSQL databases, relational tables are replaced with more general collections that can contain heterogeneous JSON objects. That is, collections have no predefined structure and no predefined fields with length constraints (in the case of strings) but can contain any type of object. The only structural constraint associated with each collection is the name of the property that acts as a primary key.

More specifically, each collection entry can contain nested objects and object collections nested in object properties, that is, related entities that, in relational databases, are contained in different tables and connected through external keys. In NoSQL, databases can be nested in their father entities. Since collection entries contain complex nested objects instead of simple property/value pairs, as is the case with relational databases, entries aren't called tuples or rows, but *documents*.

No relations and/or external key constraints can be defined between documents that belong to the same collection or to different collections. If a document contains the primary key of another document in one of its properties, it does so at its own risk. The developer has the responsibility of maintaining and keeping these coherent references.

Finally, since NoSQL storage is quite cheap, whole binary files can be stored as the values of document properties as Base64 strings. The developer can define rules to decide what properties to index in a collection. Since documents are nested objects, properties are actually tree paths. Usually, by default, all the paths are indexed, but you can specify which collection of paths and subpaths to index.

NoSQL databases are queried either with a subset of SQL or with a JSON-based language where queries are JSON objects whose paths represent the properties to query, and whose values represent the query constraints that have been applied to them.

The possibility of nesting children objects inside documents can be simulated in relational databases with the help of one-to-many relationships. However, with relational databases, we are forced to redefine the exact structure of all the related tables, while NoSQL collections don't impose any predefined structure on the objects they contain. The only constraint is that each document must provide a unique value for the primary key property. Therefore, NoSQL databases are the only option when the structure of our objects is extremely variable. However, often they are chosen for the way they scale-out read and write operations and, more generally, for their performance advantages in distributed environments. Their performance features will be discussed in the next section, which compares them to relational databases.

The graph data model is an extreme case of a completely unstructured document. The whole database is a graph where queries can add, change, and delete graph documents.

In this case, we have two kinds of document: nodes and relations. While relationships have a well-defined structure (the primary key of the nodes connected by the relationship, plus the relationship's name), nodes have no structure at all since properties and their values are added together during node update operations. Graph data models were conceived to represent the features of people and the objects they manipulate (media, posts, and so on), along with their relationships in *social applications*. The Gremlin language was conceived specifically to query graph data models. We won't discuss this in this chapter, but references are available in the *Further reading* section.

NoSQL databases will be analyzed in detail in the remaining sections of this chapter, which are dedicated to describing Azure Cosmos DB and comparing it with relational databases.

Redis

Redis is a distributed concurrent in-memory storage based on key-value pairs and supports distributed queuing. It can be used as permanent in-memory storage and as a web application cache for database data. Alternatively, it can render pages whose content doesn't change very often.

Redis can also be used to store a web application's user session data. In fact, ASP.NET MVC, Pages, and WebForms support session data to overcome the fact that the HTTP protocol is stateless. More specifically, user data that's kept between page changes is maintained in server-side stores such as Redis and indexed by a session key stored in cookies.

Interaction with the Redis server in the cloud is typically based on a REST interface; that is, each Redis resource is accessed via HTTP GET at a URI and commands are passed in the query string, while answers are returned in JSON format. However, clients that offer an easy-to-use interface are available in all popular languages. The client for .NET and .NET Core is available through the `StackExchange.Redis` NuGet package. The basic operations of the `StackExchange.Redis` client have been documented at <https://stackexchange.github.io/StackExchange.Redis/Basics>, while the full documentation can be found at <https://stackexchange.github.io/StackExchange.Redis>.

The user interface for defining a Redis server on Azure is quite simple:

The screenshot shows the Azure portal interface for creating a new Redis Cache. On the left, there's a navigation bar with 'Home > Redis Caches > New Redis Cache'. Below it, a list titled 'Redis Caches' shows a single entry with a plus icon for 'Add'. A search bar says 'Filter by name...'. A large icon of two cylinders with a lightning bolt is displayed. A message below says 'No redis caches to display' and 'Try changing your filters if you don't see what you're looking for.' On the right, the 'New Redis Cache' dialog is open, containing fields for 'DNS name' (with placeholder 'enter a name' and suffix '.redis.cache.windows.net'), 'Subscription' (set to 'Visual Studio Ultimate con MSDN'), 'Resource group' (with options 'Select existing...' and 'Create new'), 'Location' (set to 'Central US'), 'Pricing tier' (set to 'Standard C1 (1 GB Cache, Replication)'), and '(PREVIEW) Availability zone' (set to 'Requires Premium tier').

The **Pricing tier** dropdown allows us to select one of the available memory/replication options. A quick-start guide that explains how to use Azure Redis credentials and the URI with the StackExchange.Redis .NET Core client can be found at <https://docs.microsoft.com/en-us/azure/azure-cache-for-redis/cache-dotnet-core-quickstart>.

Disk memory

All clouds offer scalable and redundant general-purpose disk memory that you can use as virtual disks in virtual machines and/or as external file storage. Azure *storage account* disk space can also be structured in **Tables** and **Queues**. However, these two storage options are only supported for backward compatibility since Azure NoSQL databases are a better option than tables and Azure Redis is a better option than Azure storage queues:

The screenshot shows the 'Create storage account' wizard. At the top, there's a breadcrumb navigation: 'Home > Create storage account'. Below it is the title 'Create storage account'. A horizontal navigation bar contains four items: 'Basics' (underlined), 'Advanced', 'Tags', and 'Review + create'. Underneath this bar, a descriptive text block states: 'Azure Storage is a Microsoft-managed service providing cloud storage that is highly available, secure, durable, scalable, and redundant. Azure Storage includes Azure Blobs (objects), Azure Data Lake Storage Gen2, Azure Files, Azure Queues, and Azure Tables. The cost of your storage account depends on the usage and the options you choose below.' It ends with a 'Learn more' link.

In the rest of this chapter, we will focus on NoSQL databases and how they differ from relational databases. Next, we will look at how to choose one over the other.

Choosing between structured or NoSQL storage

In the previous section, we stated that NoSQL databases should be preferred when data has almost no predefined structure. Actually, unstructured data can be represented in relational databases since variable properties of a tuple, t , can be placed in a connected table containing the property name, property value, and the external key of t . However, the problem is performance. In fact, property values that belong to a single object would be spread all over the available memory space. In a small database, *all over the available memory space* means far away but on the same disk; in a bigger database, this means far away but in different disk units; in a distributed cloud environment, this means far away but in different – and possibly geographically distributed – servers.

On the other hand, NoSQL databases not only keep variable attributes close to their owners, but they also keep some related objects close since they allow related objects to be nested inside properties and collections.

Therefore, we can conclude that relational databases perform well when tables that are usually accessed together can be stored close in memory. NoSQL databases, on the other hand, automatically ensure that related data is kept close since each entry keeps most of the data it is related to inside it as nested objects. Therefore, NoSQL databases perform better when they are distributed to a different memory and also to different geographically distributed servers.

Unfortunately, the only way to scale out storage write operations is to split collection entries across several servers according to the values of *shard keys*. For instance, we can place all the records containing usernames that start with **A** in a server, the records containing usernames that start with **B** on another server, and so on. This way, write operations for usernames with different start letters may be executed in parallel, ensuring that the write throughput increases linearly with the number of servers.

However, if a *shard* collection is related to several other collections, there is no guarantee that related records will be placed on the same server. Also, putting different collections on different servers without using collection sharding increases write throughput linearly until we reach the limit of a single collection per server, but it doesn't solve the issue of being forced to perform several operations on different servers to retrieve or update data that's usually processed together.

This issue becomes catastrophic for performance if access to related distributed objects must be transactional and/or must ensure structural constraints (such as external key constraints) aren't violated. In this case, all related objects must be blocked during the transaction, preventing other requests from accessing them during the whole lifetime of a time-consuming distributed operation.

NoSQL databases don't suffer from this problem and perform better with sharding and consequently with write-scaled output. This is because they don't distribute related data to different storage units and instead store them as nested objects of the same database entry.

In NoSQL database design, we always try to put all related objects that are likely to be processed together into a single entry. Related objects that are accessed less frequently are placed in different entries. Since external key constraints aren't enforced automatically and NoSQL transactions are very flexible, the developer can choose the best compromise between performance and coherence.

It is worth mentioning that there are situations where relational databases perform well with sharding. A typical instance is a multi-tenant application. In a multi-tenant application, all entries collections can be partitioned into non-overlapping sets called **tenants**. Only entries belonging to the same tenant can refer to each other, so if all the collections are sharded in the same way according to their object tenants, all related records end up in the same shard, that is, in the same server, and can be navigated efficiently.

Multi-tenant applications aren't rare in the cloud since all applications that offer the same services to several different users are often implemented as multi-tenant applications, where each tenant corresponds to a user subscription. Accordingly, relational databases are conceived to work in the cloud, such as Azure SQL Server, and usually offer sharding options for multi-tenant applications. Typically, sharding isn't a cloud service and must be defined with database engine commands. Here, we won't describe how to define shards with Azure SQL Server, but the *Further reading* section contains a link to the official Microsoft documentation.

In conclusion, relational databases offer a pure, logical view of data that's independent of the way they are actually stored, and use a declarative language to query and update them. This simplifies development and system maintenance, but it may cause performance issues in a distributed environment that requires write scale-out. In NoSQL databases, you must handle more details about how to store data, as well as some procedural details for all the update and query operations, manually, but this allows you to optimize performance in distributed environments that require both read and write scale-out.

In the next section, we will look at Azure Cosmos DB, the main Azure NoSQL offering.

Azure Cosmos DB – an opportunity to manage a multi-continental database

Azure Cosmos DB is Azure's main NoSQL offering. Azure Cosmos DB has its own interface that is a subset of SQL, but it can be configured with a MongoDB interface. It can be also configured as a graph data model that can be queried with Gremlin. Cosmos DB allows replication for fault tolerance and read scale-out, and replicas can be distributed geographically to optimize communication performance. Moreover, you can specify which data center all the replicas are placed in. The user also has the option to write-enable all the replicas so that writes are immediately available in the geographical area where they are done. Write scale-up is achieved with sharding, which the user can configure by defining which properties to use as shard keys.

You can define a Cosmos DB account by typing Cosmos DB into the Azure portal search bar and clicking **Add**. The following page will appear:

Create Azure Cosmos DB Account

Basics Network Tags Review + create

Azure Cosmos DB is a fully managed globally distributed, multi-model database service, transparently replicating your data across any number of Azure regions. You can elastically scale throughput and storage, and take advantage of fast, single-digit-millisecond data access using the API of your choice backed by 99.999 SLA. [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 Ultimate con MSDN

* Resource Group: Default-SQL-NorthEurope

[Create new](#)

INSTANCE DETAILS

* Account Name: Enter account name documents.azure.com

* API: Core (SQL)

* Location: Australia East

Geo-Redundancy: [Enable](#) [Disable](#)

Multi-region Writes: [Enable](#) [Disable](#)

[Review + create](#) [Previous](#) [Next: Network](#)

The account name you choose is used in the resource URI as {account name}.documents.azure.com. The API dropdown lets you choose the kind of interface you prefer (SQL, MongoDB, or Gremlin). Then, you can decide which data center the main database will be placed in and whether you want to enable geographically distributed replication. Once you've enabled geographically distributed replication, you can choose the number of replicas you want to use and where to place them.

Finally, the **Multi-region Writes** toggle lets you enable writes on geographically distributed replicas. If you don't do this, all write operations will be routed to the main data center.

1. **Going to the resource:** Once you've created your account, select **Data Explorer** to create your databases and collections inside of them:



2. **Creating a collection:** Since databases just have a name and no configuration, you can directly add a collection and then the database where you wish to place it:

The screenshot shows the 'Add Collection' dialog box. It has fields for 'Database id' (radio buttons for 'Create new' or 'Use existing', with 'Create new' selected and a text input field 'Type a new database id'), 'Provision database throughput' (checkbox), 'Collection Id' (text input field 'e.g., Collection1'), 'Where did 'fixed' collections go?' (text input field), 'Partition key' (text input field 'e.g., /address/zipCode'), 'Throughput (400 - 5.000 RU/s)' (input field containing '15000' with up/down arrows and minus/plus buttons), and 'Unique keys' (button '+ Add unique key'). At the bottom is an 'OK' button.

Here, you can decide on database and collection names and the property to use for sharding (partition key). Since NoSQL entries are object trees, property names are specified as paths. You can also add properties whose values are required to be unique. However, uniqueness IDs are checked inside each shard, so this option is only useful in certain situations, such as multi-tenant applications (where each tenant is included in a single shard). The fees depend on the collection throughput that you choose.

3. **Targeting all resource parameters to your needs:** Throughput is expressed in **Request Unit per second**, where **Request Unit per second** is defined as the throughput we have when performing a read of 1 KB per second. Hence, if you check the **Provision database throughput** option, the chosen throughput is shared with the whole database, instead of being reserved as a single collection.
4. **Getting connection information:** By selecting the **Keys** menu, you will see all the information you need in order to connect with your Cosmos DB account from your application:



5. **Connection information page:** Here, you will find the account URI and two connection keys, which can be used interchangeably to connect with the account:

Read-write Keys	Read-only Keys
URI <code>https://533a17b5-0ee0-4-231-b9ee.documents.azure.com:443/</code>	
PRIMARY KEY <code>KRQ9j8ffnf3sqicR2vneWa9ZyvmFxCwQepdRri6kPOjtZNlyMesn2POGDof6W1BzOuKk6lxUyKJpVxM0tieyM5w==</code>	
SECONDARY KEY <code>yyb0cRjW1ThgVMupHpBjPNUNUSbn50mVtWDrWl4wQSqkWDFrBEkoTkVXk7mbqDFj3zvg7laloJ8J9oC8v140fw==</code>	
PRIMARY CONNECTION STRING <code>AccountEndpoint=https://533a17b5-0ee0-4-231-b9ee.documents.azure.com:443/;AccountKey=KRQ9j8ffnf3sqicR2vneWa9ZyvmFxCwQepdRri6kPOjtZNlyMesn2POGDof6W1BzOuKk6lxUyKJ... </code>	
SECONDARY CONNECTION STRING <code>AccountEndpoint=https://533a17b5-0ee0-4-231-b9ee.documents.azure.com:443/;AccountKey=yyb0cRjW1ThgVMupHpBjPNUNUSbn50mVtWDrWl4wQSqkWDFrBEkoTkVXk7mbqDFj3zvg... </code>	

There are also keys with read-only privileges. Every key can be regenerated and each account has two equivalent keys so that this operation can be handled efficiently; that is, when a key is changed, the other one is kept. Therefore, existing applications can continue using the other key before upgrading to the new key.

6. **Selecting the default consistency level:** By selecting the **Default consistency**, you can choose the default replication consistency that you wish to apply to all of your collections:



This default can be overridden in each collection, either from the Data Explorer or programmatically. Consistency problems in read/write operations are a consequence of data replication. More specifically, the results of various read operations may be incoherent if the read operations are executed on different replicas that have received different partial updates.

The following are the available consistency levels. These have been ordered from the weakest to the strongest:

- **Eventual:** After enough time has passed, if no further write operations are done, all the reads converge and apply all the writes.
- **Consistent Prefix:** All the writes are executed in the same order on all the replicas. So, if there are n write operations, each read is consistent with the result of applying the first m writes for some m less or equal to n .
- **Session:** This is the same as the consistency prefix but also guarantees that each writer sees the result of its own writes in all subsequent read operations and that subsequent reads of each reader are coherent (either the same database or a more updated version of it).
- **Bounded Staleness:** This is associated either with a delay time, Δt , or with a number of operations, N . Each read sees the results of all the write operations that were performed before a time Δt (or before the last N operations). That is, its reads converge with the result of all the writes with a maximum time delay of Δt (or a maximum operations delay of N).
- **Strong:** This is bounded staleness combined with $\Delta t = 0$. Here, each read reflects the result of all previous write operations.

The strongest consistency can be obtained to the detriment of performance. By default, the consistency is set to **Session**, which is a good compromise between coherence and performance. A lower level of consistency is difficult to handle in applications and is only usually acceptable if sessions are either read-only or write-only.

If you select the **Scale & settings** option in the Data Explorer, you can configure which paths to index and which kind of indexing to apply to each data type of each path. The configuration consists of a JSON object. Let's analyze its various properties:

```
{  
    "indexingMode": "consistent",  
    "automatic": true,  
    ...  
}
```

If you set `indexingMode` to `none` instead of `consistent`, no index is generated and the collection can be used as a key-value dictionary that's indexed by the collection primary key. When `automatic` is set to `true`, all document properties are automatically indexed:

```
{  
    ...  
    "includedPaths": [  
        {  
            "path": "/**",  
            "indexes": [  
                {  
                    "kind": "Range",  
                    "dataType": "Number",  
                    "precision": -1  
                },  
                {  
                    "kind": "Range",  
                    "dataType": "String",  
                    "precision": -1  
                },  
                {  
                    "kind": "Spatial",  
                    "dataType": "Point"  
                }  
            ]  
        }  
    ],  
    ...  
}
```

Each entry in the *Included paths* specifies a path pattern such as `/subpath1/subpath2/?` (settings apply just to the `/subpath1/subpath2/` property) or `/subpath1/subpath2/*` (settings apply to all the paths starting with `/subpath1/subpath2/`).

Patterns contain the `[]` symbol when settings must be applied to child objects contained in collection properties; for example, `/subpath1/subpath2/[]/?`, `/subpath1/subpath2/[]/childpath1/?`, and so on. Settings specify the index type to apply to each data type (string, number, geographic point, and so on). Range indexes are needed for comparison operations, while hash indices are more efficient if we need equality comparisons.

It is possible to specify a precision, that is, the maximum number of characters or digits to use in all the index keys. `-1` means no limit. `-1` is acceptable for strings, while a finite precision should be used for numbers. On the other hand, using finite precision with strings may result in unexpected behavior since string keys are truncated. In hash indexes, precision may vary from 1 to 8, while in range indexes, it may vary from 1 to 100:

```
...
"excludedPaths": [
  {
    "path": "/\\\"_etag\\\"/?"
  }
]
```

Paths contained in `excludedPaths` aren't indexed at all. Index settings can also be specified programmatically.

Here, you have two options to connect to Cosmos DB: use a version of its official client for your preferred programming language or use Cosmos DB's Entity Framework Core provider, which at the time of writing this book, is still in preview. In the following subsections, we will have a look at both options. Then, we will describe how to use Cosmos DB's Entity Framework Core provider with a practical example.

Cosmos DB client

The Cosmos DB client for .NET Core is available through the `Microsoft.Azure.DocumentDB.Core` NuGet package. It offers full control of all Cosmos DB features, while the Cosmos DB Entity Framework provider is easier to use but hides some Cosmos DB peculiarities. Follow these steps to interact with Cosmos DB through the official **Cosmos DB client for .NET Core**:

1. Any operation requires the creation of a client object:

```
var client = new DocumentClient(new Uri("service endpoint"), "auth key")
```

2. Don't forget that the client must be disposed of by calling its `Dispose` method (or by enclosing the code that references it in a `using` statement) when you don't need it anymore.
3. Then, you can get a reference to a database and create it if it doesn't exist with the following code:

```
Database db = client.CreateDatabaseIfNotExistsAsync(new Database { Id = "MyDatabase" }).Result;
```

4. Finally, you can get a reference to a collection or create it if it doesn't exist with the following code:

```
var collection = client.CreateDocumentCollectionIfNotExistsAsync(  
    UriFactory.CreateDatabaseUri("MyDatabase"),  
    new DocumentCollection { Id = "MyCollection" }).Result;
```

5. During collection creation, you can pass an `option` object, where you can specify the consistency level, how to index properties, and all the other collection features.
6. Then, you must define the .NET classes that correspond to the structure of the JSON document you need to manipulate in your collections. You can also use the `JsonProperty` attribute to map class property names to JSON names if they aren't equal.
7. Once you have all the necessary classes, you can use client methods to add, update, and write collection entries, as well as the client `CreateDocumentQuery` method, which returns an `IQueryable` value that you can query with LINQ.

When you read a document, apply some modifications, and then try to upload your modified version of the document, someone else may have modified the same document. Often, you only need to perform an update if no one else has modified the same document. This can be done using the `_etag` property, which Cosmos DB automatically attaches to each document. This property value changes after each update, so you need to follow these steps:

1. Map the `_etag` JSON property to a property on your .NET class so that you get its value when you read a document.
2. Pass the original value of the `_etag` property as the value of the `AccessCondition` property of the `option` object you pass to the `ReplaceDocumentAsync` client method.
3. If the `_etag` has changed `ReplaceDocumentAsync`, abort the operation and return an exception.

There is also the `MvcControlsToolkit.Business.DocumentDB` NuGet package, which simplifies and automates all operations that are required by the `Microsoft.Azure.DocumentDB.Core` library and overcomes some limitations of Cosmos DB SQL. The *Further reading* section contains references to tutorials for `Microsoft.Azure.DocumentDB.Core` and `MvcControlsToolkit.Business.DocumentDB`.

Cosmos DB Entity Framework Core provider

The Cosmos DB provider for Entity Framework Core is contained in the `Microsoft.EntityFrameworkCore.Cosmos` NuGet package. Once you've added this to your project, you can proceed in a similar way to when you used the SQL Server provider in Chapter 6, *Interacting with Data in C# - Entity Framework Core*, but with a few differences. Let's take a look:

1. There are no migrations since Cosmos DB databases have no structure to update. Instead, they have a method that ensures that the database, along with all the necessary collections, is created:

```
context.Database.EnsureCreated();
```

2. `DbSet<T>` `DbContext` properties don't map one-to-one to database collections, but several `DbSet<T>` properties can map to the same collection since collections can contain objects with different structures. Moreover, by default, all `DbSet<T>` properties are mapped to a unique collection since this is the cheapest option, but you can override this default by explicitly specifying which collection you want to map some entities to by using the following configuration instruction:

```
builder.Entity<MyEntity>()
    .ToContainer("collection-name");
```

3. The only useful annotation on entity classes is the `Key` attribute, which becomes obligatory when the principal keys are called `Id`.
4. Principal keys must be strings and can't be auto-incremented to avoid synchronization issues in a distributed environment. The uniqueness of primary keys can be ensured by generating GUIDs and transforming them into strings.
5. When defining relationships between entities, you can specify that an entity or a collection of entities is owned by another entity, in which case it is stored together with the father entity.

We will look at the usage of Cosmos DB's Entity Framework provider in the next section.

Use case – storing data

Now that we've learned how to use NoSQL, we have to decide whether NoSQL databases are adequate for our WWTravelClub application. We need to store the following families of data:

- **Information about available destinations and packages:** Relevant operations for this data are reads since packages and destinations don't change very often. However, they must be accessed as fast as possible from all over the World in order to ensure a pleasant user experience when users browse the available options. Therefore, a distributed relational database with geographically distributed replicas is possible, but not necessary, since packages can be stored inside their destinations in a cheaper NoSQL database.
- **Destination reviews:** In this case, distributed write operations have a non-negligible impact. Moreover, most writes are additions, since reviews aren't usually updated. Additions benefit a lot from sharding and don't cause consistency issues like updates do. Accordingly, the best option for this data is a NoSQL collection.

- **Reservations:** In this case, consistency errors aren't acceptable because they may cause overbooking. Reads and writes have a comparable impact, but we need reliable transactions and good consistency checks. Luckily, data can be organized in a multi-tenant database where tenants are destinations since reservation information belonging to different destinations is completely unrelated.

Accordingly, we may use sharded SQL Azure database instances.

In conclusion, the best option for data in the first and second bullet points is Cosmos DB, while the best option for the third point is Azure SQL Server. Actual applications may require a more detailed analysis of all data operations and their frequencies. In some cases, it is worth implementing prototypes for various possible options and executing performance tests with typical workloads on all of them.

In the remainder of this section, we will migrate the destinations/packages data layer we looked at in [Chapter 6, Interacting with Data in C# - Entity Framework Core](#), to Cosmos DB.

Implementing the destinations/packages database with Cosmos DB

Let's move on to the database example we built in [Chapter 6, Interacting with Data in C# – Entity Framework Core](#), to Cosmos DB by following these steps:

1. First of all, we need to make a copy of the WWTravelClubDB project and make `WWTravelClubDBCosmo` the new root folder.
2. Open the project and delete the migrations folder since migrations aren't required anymore.
3. We need to replace the SQL Server Entity Framework provider with the Cosmos DB provider. To do this, go to **Manage NuGet Packages** and uninstall the `Microsoft.EntityFrameworkCore.SqlServer` NuGet package. Then, install the `Microsoft.EntityFrameworkCore.Cosmos` NuGet package.
4. Then, do the following on the Destination and Package entities:
 - Remove all data annotations.
 - Add the `[Key]` attribute to their `Id` properties since this is obligatory for Cosmos DB providers.

- Transform the type of the `Id` properties of both `Package` and `Destination`, and the `PackagesListDTO` classes from `int` to `string`. We need to turn into `string` also the `DestinationId` external references in the `Package`, and in the `PackagesListDTO` classes. In fact, the best option for keys in distributed databases is a string generated from a GUID, because it is hard to maintain an identity counter when table data is distributed among several servers.
5. In the `MainDbContext` file, we need to specify that packages related to a destination must be stored inside the destination document itself. This can be achieved by replacing the Destination-Package relation configuration in the `OnModelCreating` method with the following code:

```
builder.Entity<Destination>()
    .OwnsMany(m => m.Packages);
```
 6. Here, we must replace `HasMany` with `OwnsMany`. There is no equivalent to `WithOne` since once an entity is owned, it must have just one owner, and the fact that the `MyDestination` property contains a pointer to the father entity is evident from its type. Cosmos DB also allows the use of `HasMany`, but in this case, the two entities aren't nested one in the other. There is also an `OwnOne` configuration method for nesting single entities inside other entities.
 7. Actually, both `OwnsMany` and `OwnsOne` are available for relational databases, but in this case, the difference between `HasMany` and `HasOne` is that children entities are automatically included in all queries that return their father entities, with no need to specify an `Include` LINQ clause. However, child entities are still stored in separate tables.
 8. `LibraryDesignTimeDbContextFactory` must be modified to use Cosmos DB connection data, as shown in the following code:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;

namespace WWTravelClubDB
{
    public class LibraryDesignTimeDbContextFactory
        : IDesignTimeDbContextFactory<MainDbContext>
    {
        private const string endpoint = "<your account endpoint>";
        private const string key = "<your account key>";
        private const string databaseName = "packagesdb";
        public MainDbContext CreateDbContext(params string[] args)
```

```
        {
            var builder = new DbContextOptionsBuilder<Main
                DBContext>();

            builder.UseCosmos(endpoint, key, databaseName);
            return new MainDBContext(builder.Options);
        }
    }
}
```

9. Finally, in our test console, we must explicitly create all entity principal keys using GUIDS:

```
var context = new LibraryDesignTimeDbContextFactory()
    .CreateDbContext();
context.Database.EnsureCreated();
var firstDestination = new Destination
{
    Id = Guid.NewGuid().ToString(),
    Name = "Florence",
    Country = "Italy",
    Packages = new List<Package>()
    {
        new Package
        {
            Id=Guid.NewGuid().ToString(),
            Name = "Summer in Florence",
            StartValidityDate = new DateTime(2019, 6, 1),
            EndValidityDate = new DateTime(2019, 10, 1),
            DurationInDays=7,
            Price=1000
        },
        new Package
        {
            Id=Guid.NewGuid().ToString(),
            Name = "Winter in Florence",
            StartValidityDate = new DateTime(2019, 12, 1),
            EndValidityDate = new DateTime(2020, 2, 1),
            DurationInDays=7,
            Price=500
        }
    }
};
```

Here, we call `context.Database.EnsureCreated()` instead of applying migrations since we only need to create the database. Once the database and collections have been created, we can fine-tune their settings from the Azure Portal. Hopefully, future versions of Cosmos DB Entity Framework Core provider will allow us to specify all collection options.

10. Finally, the final query that starts with `context.Packages.Where...` must be modified since queries can't start from entities that are nested in other documents (in our case, Package entities). Therefore, we must start our query from the unique root `DbSet<T>` property we have in our `DbContext`, that is, `Destinations`. We can move from listing the external collection to listing all the internal collections with the help of the `SelectMany` method, which performs a logical merge of all nested `Packages` collections. However, since CosmosDB SQL doesn't support `SelectMany`, we must force `SelectMany` to be simulated on the client with `AsEnumerable()`, as shown in the following code:

```
var list = context.Destinations
    .AsEnumerable() // move computation on the client side
    .SelectMany(m => m.Packages)
    .Where(m => period >= m.StartValidityDate....)
    ...
```

11. The remainder of the query remains unchanged. If you run the project now, you should see the same outputs that were received in the case of SQL Server (with the exception of the primary key values).

12. After executing the program, go to your Cosmos DB account. You should see something like the following:

```
4   "Description": "Florence is a famous historical Italian town",
5   "Discriminator": "Destination",
6   "Name": "Florence",
7   "id": "7690cdcaa-e772-48ff-af5d-b4175336ca5f",
8   "Packages": [
9     {
10       "Id": "53d8aa59-3d73-4401-8b79-1a5593a3f04f",
11       "Description": null,
12       "DestinationId": "74d5c087-1086-4ec8-9ef8-cfd59c4afa0b",
13       "Discriminator": "Package",
14       "DurationInDays": 7,
15       "EndValidityDate": "2019-10-01T00:00:00",
16       "Name": "Summer in Florence",
17       "Price": 1100,
18       "StartValidityDate": "2019-06-01T00:00:00"
19     },
20     {
21       "Id": "e2f06575-9994-4dfc-86a0-51b92890ad2f",
22       "Description": null,
23       "DestinationId": "74d5c087-1086-4ec8-9ef8-cfd59c4afa0b",
24       "Discriminator": "Package",
25       "DurationInDays": 7,
26       "EndValidityDate": "2020-02-01T00:00:00",
27       "Name": "Winter in Florence",
28       "Price": 550,
29       "StartValidityDate": "2019-12-01T00:00:00"
```

The packages have been nested inside their destinations as required and Entity Framework Core creates a unique collection that has the same name as the `DbContext` class.

If you would like to continue experimenting with Cosmos DB development without wasting all your free Azure Portal credit, you can install the Cosmos DB emulator available at this link: <https://aka.ms/cosmosdb-emulator>.

Summary

In this chapter, we looked at the main storage options available in Azure and learned when to use them. Then, we compared relational and NoSQL databases. We pointed out that relational databases offer automatic consistency checking and transaction isolation, but NoSQL databases are cheaper and offer better performance, especially when distributed writes form a high percentage of the average workload.

Then, we described Azure's main NoSQL option, Cosmos DB, and explained how to configure it and how to connect with a client.

Finally, we learned how to interact with Cosmos DB with Entity Framework Core and looked at a practical example based on the WWTravelClubDB use case. Here, we learned how to decide between relational and NoSQL databases for all families of data involved in an application. This way, you can choose the kind of data storage that ensures the best compromise between data coherence, speed, and parallel access to data in each of your applications.

In the next chapter, we will learn all about Serverless and Azure Functions.

Questions

1. Is Redis a valid alternative to relational databases?
2. Are NoSQL databases a valid alternative to relational databases?
3. What operation is more difficult to scale out in relational databases?
4. What is the main weakness of NoSQL databases? What is their main advantage?
5. Can you list all Cosmos DB consistency levels?
6. Can we use auto-increment integer keys with Cosmos DB?
7. Which Entity Framework configuration method is used to store an entity inside its related father document?
8. Can nested collections be searched efficiently with Cosmos DB?

Further reading

- In this chapter, we didn't talk about how to define sharding with SQL Azure. Here is the link to the official documentation if you want to find out more: <https://docs.microsoft.com/en-us/azure/sql-database/sql-database-elastic-scale-introduction>.
- Cosmos DB was described in detail in this chapter, but further details can be found in the official documentation: <https://docs.microsoft.com/en-us/azure/cosmos-db/>.
- The following is a reference to the Gremlin language, which is supported by Cosmos DB: <http://tinkerpop.apache.org/docs/current/reference/#graph-traversal-steps>.
- The following is a general description of the Cosmos DB Graph Data Model: <https://docs.microsoft.com/en-us/azure/cosmos-db/graph-introduction>.
- Details on how to use Cosmos DB's official .NET client can be found at <https://docs.microsoft.com/en-us/azure/cosmos-db/sql-api-dotnetcore-get-started>. A good introduction to the `MvcControlsToolkit.Business.DocumentDB` NuGet package we mentioned in this chapter is the *Fast Azure Cosmos DB Development with the DocumentDB Package* article contained in Issue 34 of DNCMagazine. This can be downloaded from <http://www.dotnetcurry.net/s/dnc-mag-34th-single>.

8

Working with Azure Functions

As we mentioned in Chapter 4, *Deciding on the Best Cloud-Based Solution*, the serverless architecture is one of the newest ways to provide flexible software solutions. To do so, Microsoft Azure provides Azure Functions, an event-driven, serverless, and scalable technology that accelerates your project development. The main goal of this chapter is to inform you of Azure Functions and the best practices you can implement while using it.

In this chapter, we will cover the following topics:

- Understanding the Azure Functions App
- Programming Azure Functions using C#
- Maintaining Azure Functions
- Use case – implementing Azure Functions to send emails

By the end of this chapter, you will understand how to use Azure Functions in C#.

Technical requirements

This chapter requires that you have the following:

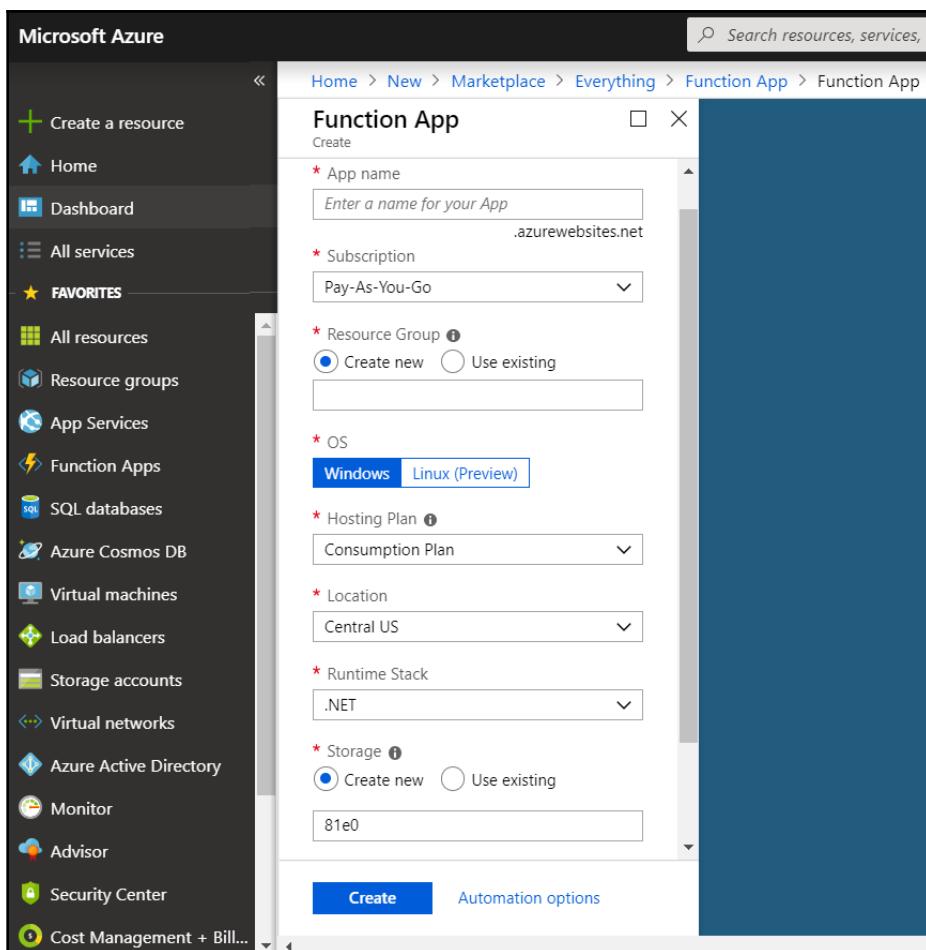
- Visual Studio 2017 or 2019 free Community Edition or better with all the database tools installed.
- A free Azure account. The *Creating an Azure account* section of Chapter 1, *Understanding the Importance of Software Architecture*, explains how to create one.

You can find the sample code for this chapter at <https://github.com/PacktPublishing/Hands-On-Software-Architecture-with-CSharp-8/tree/master/ch08>.

Understanding the Azure Functions App

The Azure Functions App is an Azure PaaS where you can build pieces of code (functions) and connect them to your application and use triggers to start them. The concept is quite simple – you build a function in the language you prefer and decide on the trigger that will start it. You can write as many functions as you want in your system. There are cases where the system is written entirely with functions.

The steps for creating the necessary environment are as simple as the ones we need to follow in order to create the function itself. The following screenshot shows the parameters that you have to decide on when you create the environment. After you select **Create a Resource** in Azure and filter by **Function App**, you will see the following screen:



There are a couple of key points that you should consider while creating the environment. The first one is the Hosting Plan, which is where you will run your functions. There are two options for the Hosting Plan: Consumption Plan and App Service Plan. Let's talk about these now.

Consumption Plan

If you choose a Consumption Plan, your functions will only waste resources when they are executed. This means that you will only be charged while your functions are running. Scalability and memory resources will be automatically managed by Azure.

Something we need to take note of while writing functions in this plan is the **timeout**. By default, after 5 minutes, the function will time out. You can change the timeout value using the `functionTimeout` parameter. The maximum value is 10 minutes.

When you choose a Consumption Plan, the way that you will be charged will depend on what you're executing, their execution time, and their memory usage. More information on this can be found at <https://azure.microsoft.com/en-us/pricing/details/functions/>.

Note that this can be a good option when you don't have App Services in your environment and you are running functions with low periodicity. On the other hand, if you need continuous processing, you may want to consider the App Service Plan.

App Service Plan

App Service Plan is one of the options you can choose when you want to create an Azure Functions App. The following is a list of reasons (suggested by Microsoft) why you should use the App Service Plan instead of the Consumption Plan to maintain your functions:

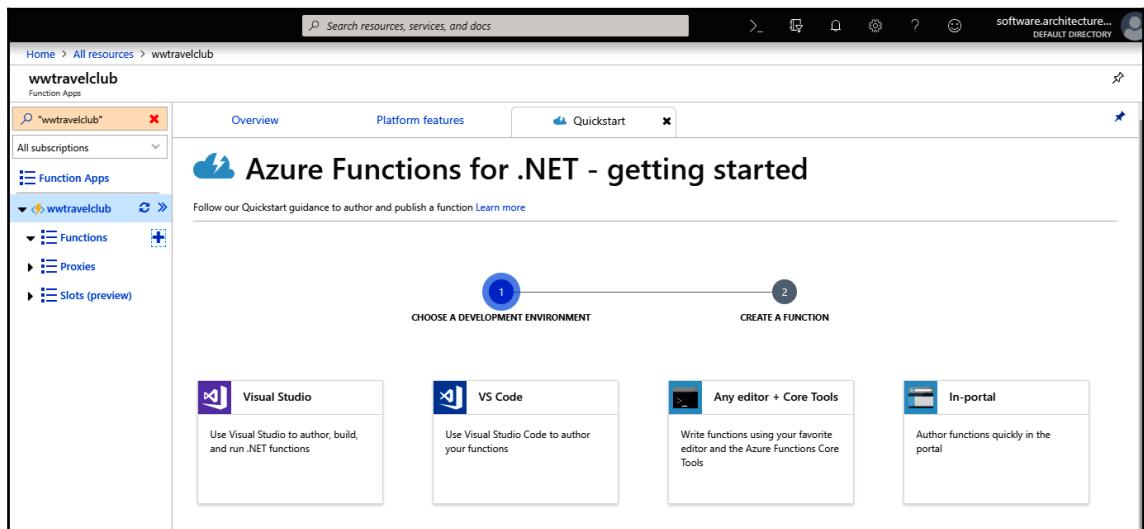
- You can use underutilized existing App Service instances.
- Function apps run continuously or nearly continuously.
- You need more CPU or memory options than what's provided with the Consumption Plan.
- Your code needs to run longer than 10 minutes.
- You require features such as VNET/VPN connectivity.
- You want to run your function app on Linux or on a custom image.

In the scenario of App Service Plan, the `functionTimeout` value varies according to the Azure Function Runtime version. However, the value is at least 30 minutes.

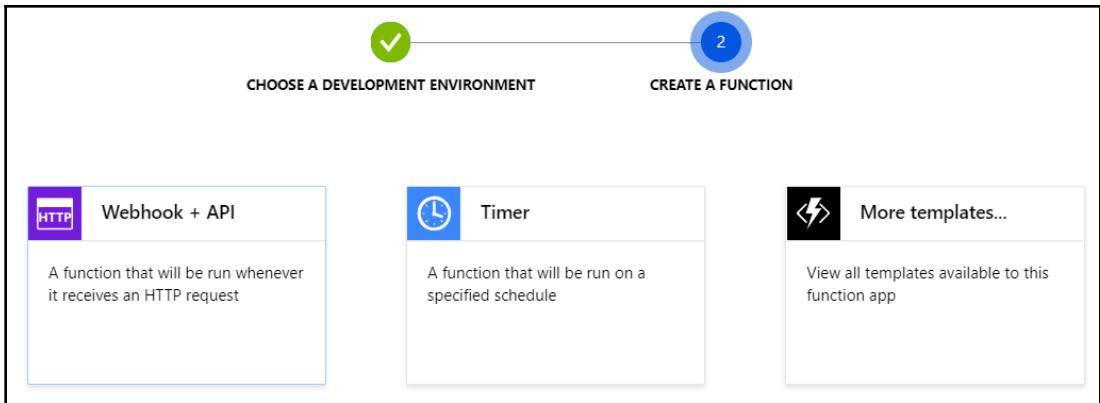
Programming Azure Functions using C#

In this section, you will learn how to create Azure Functions. It is worth mentioning that there are several ways to create them using C#. The first one is by creating the functions and developing them in the Azure Portal itself. To do this, follow these steps:

1. From the **Home** page, go to **All resources**, search for the `wwtravelclub` app, and click it. You will see the following screen:

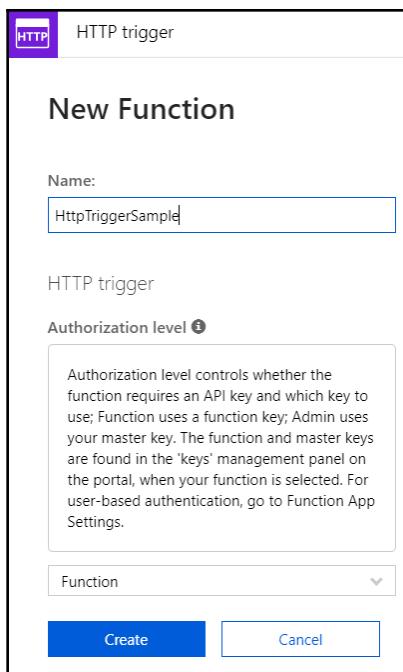


2. Clicking on the **In-portal** creation option. Here, you will be prompted to decide on the kind of trigger that you want to use to start the execution. The most used ones are **HTTP Request** and **Timer Trigger**, as shown in the following screenshot:



When you decide on the trigger you want to use, you have to name it.

3. Depending on the trigger you decide on, you will have to install some extensions and set up other parameters. For instance, HTTP trigger requires that you set up an authorization level. Three options are available, that is, **Function**, **Anonymous**, and **Admin**, out of which we have selected the **Function** option as shown in the following screenshot:





It is worth mentioning that this book doesn't cover all the options that are available when it comes to building functions. As a software architect, you should understand that Azure provides a good service for serverless architectures in terms of functions. This can be useful in several situations. This was discussed in more detail in Chapter 4, *Deciding on the Best Cloud-Based Solution*.

5. The result of this is as follows. Notice that Azure provides an editor that allows us to run the code, check logs, and test the function that we've created. This is a good interface for testing and coding basic functions:

The screenshot shows the Azure Functions portal interface. On the left, there's a sidebar with navigation links: Home, All resources, wwtravelclub - HttpTriggerSample (selected), Function Apps, All subscriptions, Function Apps, wwtravelclub, Functions, HttpTriggerSample (selected), Integrate, Manage, Monitor, Proxies, Slots (preview). The main area is titled "wwtravelclub - HttpTriggerSample". It shows a code editor for "run.csx" with the following C# code:

```
1 #r "Newtonsoft.Json"
2
3 using System.Net;
4 using Microsoft.AspNetCore.Mvc;
5 using Microsoft.Extensions.Primitives;
6 using Newtonsoft.Json;
7
8 public static async Task<ActionResult> Run(HttpContext req, ILogger log)
9 {
10     log.LogInformation("C# HTTP trigger function processed a request.");
11
12     string name = req.Query["name"];
13
14     string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
15     dynamic data = JsonConvert.DeserializeObject(requestBody);
16     name = name ?? data?.name;
17
18     return name != null
19         ? (ActionResult)new OkObjectResult($"Hello, {name}")
20         : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
21 }
```

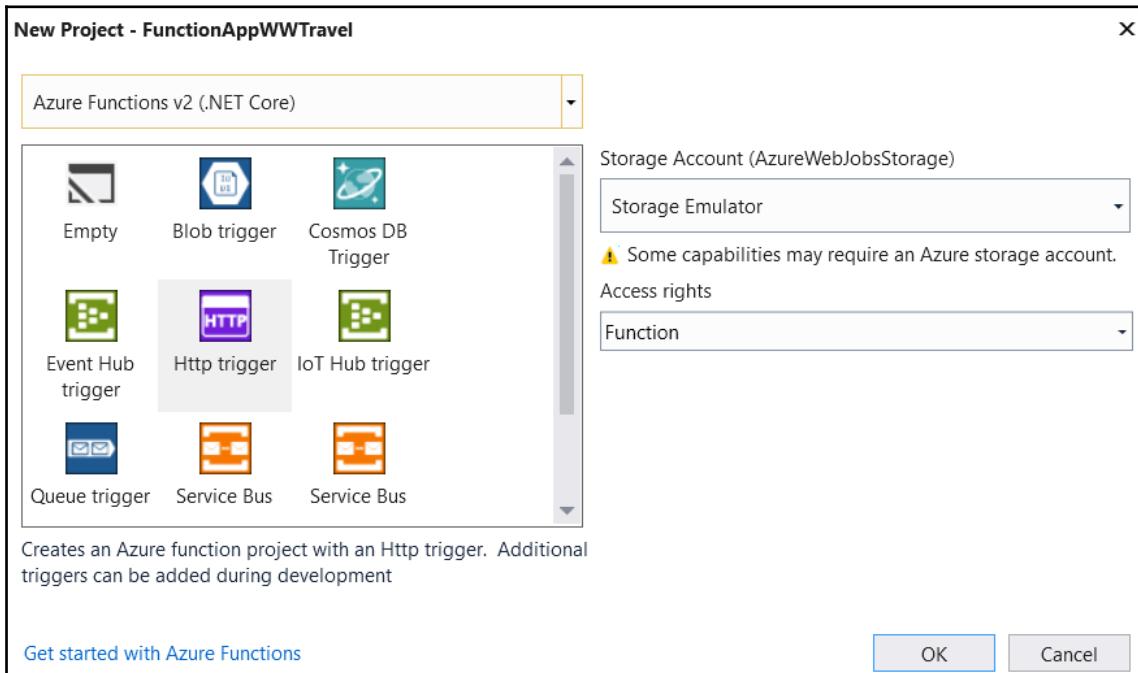
Below the code editor, there are tabs for "Logs" and "Console". To the right of the code editor, there are buttons for "View files" and "Test". At the top of the main area, there are buttons for "Save", "Run", and "Get function URL".

6. However, if you want to create more sophisticated functions, you may need a more sophisticated environment so that you can code and debug them more efficaciously. This is where the Visual Studio Azure Functions Project can help you.



In Visual Studio, you are able to create a project dedicated to Azure Functions by going to the **New Project | Cloud** menu.

- Once you've submitted your project, Visual Studio will ask you for the type of trigger you're using and for the Azure version that your function will run on:



At the time of writing, there are two versions of Azure Functions:

- In the first version, you can create functions that run on .NET Framework.
- In the second version, you can create functions that run on .NET Core.



As a software architect, you always have to keep code reusability in mind. In this case, you should pay attention to which version of Azure Functions Project you will decide to build your functions in.

By default, the code that's generated is similar to the code that's generated when you create Azure Functions in Azure Portal. The publish method follows the same steps as the publish procedure for web apps that we described in Chapter 1, *Understanding the Importance of Software Architecture*.

Listing Azure Functions templates

There are several templates in the Azure Portal that you can use to create Azure Functions. The number of templates that you can choose from is updated continuously. The following are just a few of them:

- **Blob Storage:** You may want to process something for a file as soon as this file is uploaded to your blob storage. This can be a good use case for Azure functions.
- **Cosmos DB:** You may want to synchronize data that arrives in a Cosmos DB database with a processing method. Cosmos DB was discussed in detail in Chapter 7, *How to Choose Your Data Storage in the Cloud*.
- **Event Grid:** This is a good way to manage Azure events. Functions can be triggered so that they manage each event.
- **Event Hubs:** These can be used with Azure Functions to manage data that arrives for each connected device.
- **HTTP:** This trigger is really useful for building serverless APIs and web apps events.
- **Microsoft Graph Events:** The Graph API allows you to deliver functionality associated with Office 365. For example, using this trigger, you can connect a calendar event to a function.
- **Queue storage:** You can handle queue processing using a function as a service solution.
- **Service Bus:** This is another messaging service that can be a trigger for functions. Azure Service Bus will be covered in more detail in Chapter 9, *Design Patterns and .NET Core Implementation*.
- **Timer:** This is commonly used with functions and is where you specify Time Triggers so that you can continuously process data from your system.
- **WebHooks:** WebHooks is a technology that allows your application to avoid polling data from an API. You can connect them to a function to learn how the event you've hooked is being processed.

Maintaining Azure Functions

Once you've created and programmed your function, you need to monitor and maintain it. To do this, you can use a variety of tools – all of which you can find in Azure Portal. These tools will help you solve problems due to the amount of information you will be able to collect with them.

The first option when it comes to monitoring your function is using the **Monitor** menu inside of the Azure Functions interface in Azure Portal. There, you will be able to check all of your function executions, including successful results and failures:

The screenshot shows the Azure Portal interface for the 'HttpTriggerSample' function app under the 'wwtravelclub' subscription. The left sidebar includes links for 'All subscriptions', 'Function Apps', 'wwtravelclub' (selected), 'Functions' (selected), 'Integrate', 'Manage', and 'Monitor' (selected). The main content area displays execution history with columns for DATE (UTC), SUCCESS, RESULT CODE, DURATION (MS), and OPERATION ID. Three successful executions are listed:

DATE (UTC)	SUCCESS	RESULT CODE	DURATION (MS)	OPERATION ID
2019-02-25 22:00:58.732	✓	0	1.6852	CGzCwnLL+bA=
2019-02-25 22:00:42.483	✓	0	3.6808	9eUDmcplXU=
2019-02-25 22:00:32.582	✓	0	99.3548	5miY1jEbXpg=

At the top right, there are links for 'Run in Application Insights' and 'Diagnose and solve problems'. A 'Refresh' button is also present.

It will take about 5 minutes for any results to be available. The date shown in the grid is in UTC time.

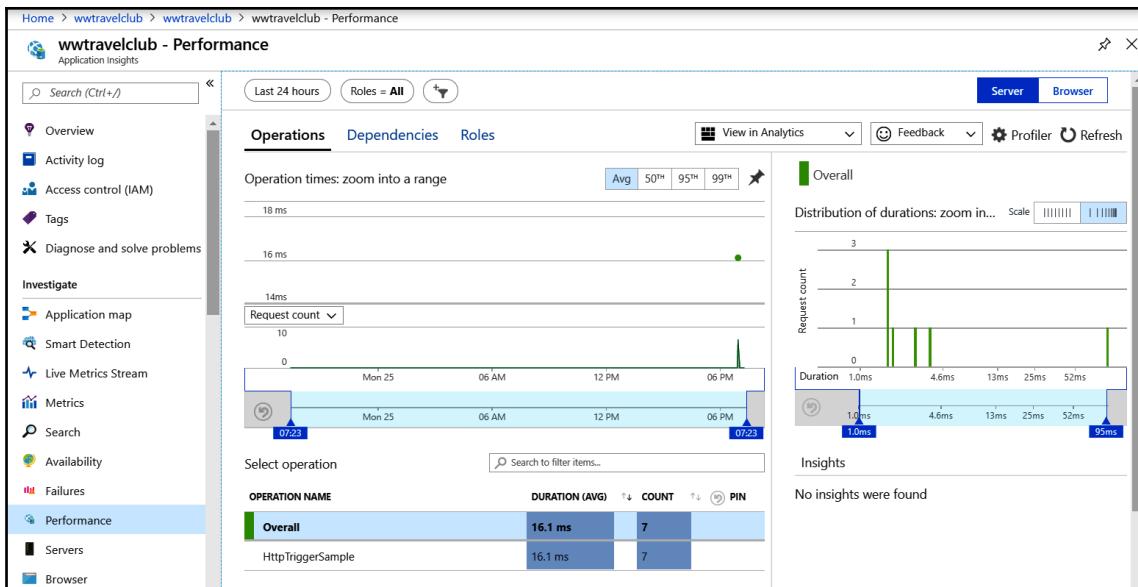
The same interface allows you to connect to Application Insights. This will take you to a world of almost indefinite options that you can use to analyze your function data. Application Insights is one of the best **Application Performance Management (APM)** systems available nowadays:

The screenshot shows the Azure Application Insights Analytics interface. The top navigation bar includes 'Application 1*', 'Run', 'Time range: Set in query', and various export and save options. The main area displays a query results table titled 'Completed'.

Completed

timestamp [UTC]	id	name	success	resultCode	duration	operation_id	cloud_RoleName
2019-02-25T22:01:09.111	0MsGwmRN26c=.d827bec9_	HttpTriggerSample	True	0	1.7053	OnsGwmRN26c=	wwtravelclub
2019-02-25T22:01:08.332	7rpN6Xt1T6E=.d827bec8_	HttpTriggerSample	True	0	1.6907	7rpN6Xt1T6E=	wwtravelclub
2019-02-25T22:01:07.572	CzElIrcRk98=.d827bec7_	HttpTriggerSample	True	0	1.8384	CzElIrcRk98=	wwtravelclub
2019-02-25T22:01:05.787	xIPoyz70898=.d827bec6_	HttpTriggerSample	True	0	2.8504	xIPoyz70898=	wwtravelclub
2019-02-25T22:00:58.732	CGCwmlL+bA=.d827bec5_	HttpTriggerSample	True	0	1.6852	CGCwmlL+bA=	wwtravelclub
2019-02-25T22:00:42.483	9eUDmcplLoXU=.d827bec0_	HttpTriggerSample	True	0	3.6808	9eUDmcplLoXU=	wwtravelclub
2019-02-25T22:00:32.582	5miYtjEbXpg=.d827bebd_	HttpTriggerSample	True	0	99.3548	5miYtjEbXpg=	wwtravelclub

Beyond the query interface, you can also check all the performance issues of your function using the Insights interface in Azure Portal. There, you can analyze and filter all the requests that have been received by your solution and check their performance and dependencies. You can also trigger alerts when something abnormal happens to one of your endpoints:



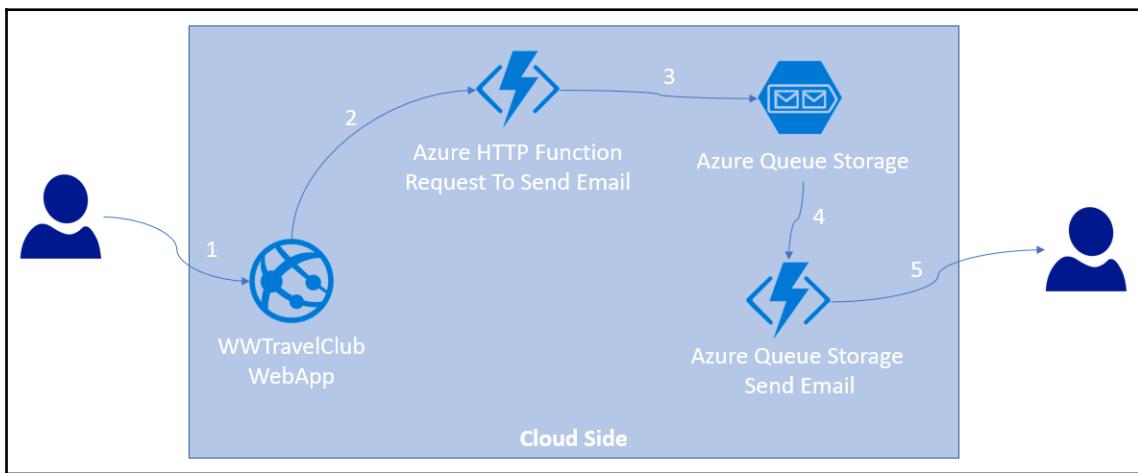
As a software architect, you will find a really good daily helper for your projects in this tool. It is worth mentioning that Application Insights works on several other Azure Services, such as web apps and virtual machines. This means you can monitor the health of your system and maintain it using the wonderful features provided by Azure.

Use case – implementing Azure Functions to send emails

Here, we will use a subset of the Azure components we described previously. The use case from WWTravelClub proposes a worldwide implementation of the service, and there is a chance that this service will need different architecture designs to face all the performance key points that we described in Chapter 1, *Understanding the Importance of Software Architecture*.

If you go back to the user stories that were described in [Chapter 1, Understanding the Importance of Software Architecture](#), you will find that many needs are related to communication. Because of this, it is really common to have some alerts be provided by emails in the solution. This chapter's use case will focus on how to send emails. The architecture will be totally serverless.

The following diagram shows the basic structure of the architecture. To give users a great experience, all the emails that are sent by the application will be queued asynchronously, thus avoiding high delays in the system's responses:



Note that there are no servers that manage Azure Functions for inserting and Azure Functions for getting messages from the Queue Storage. This is exactly what we call serverless. It is worth mentioning that this architecture is not restricted to only sending emails – it can also be used to process any HTTP POST request.

Now, we will learn how to set up security in the API so that only authorized applications can use the given solution.

First Step – creating Azure Queue Storage

It's quite simple to create storage in Azure Portal. Let's learn how:

1. First, you will need to create a storage account and set up its name, security, and network, as shown in the following screenshot:

The screenshot shows the 'Create storage account' wizard in the Azure Portal. The top navigation bar includes 'Home > New > Marketplace > Everything > Storage account > Create storage account'. The main title is 'Create storage account'. Below it, there are tabs: 'Basics' (which is selected), 'Advanced', 'Tags', and 'Review + create'. A descriptive text states: 'Azure Storage is a Microsoft-managed service providing cloud storage that is highly available, secure, durable, scalable, and redundant. Azure Storage includes Azure Blobs (objects), Azure Data Lake Storage Gen2, Azure Files, Azure Queues, and Azure Tables. The cost of your storage account depends on the usage and the options you choose below.' A 'Learn more' link is provided. The 'PROJECT DETAILS' section contains fields for 'Subscription' (set to 'Pay-As-You-Go') and 'Resource group' (set to 'wwtravelclub-rg', with a 'Create new' link). The 'INSTANCE DETAILS' section contains fields for 'Storage account name' (set to 'wwtravelclubmail') and 'Location' (set to 'Central US'). At the bottom, there are buttons for 'Review + create' (highlighted in blue), 'Previous', and 'Next : Advanced >'.

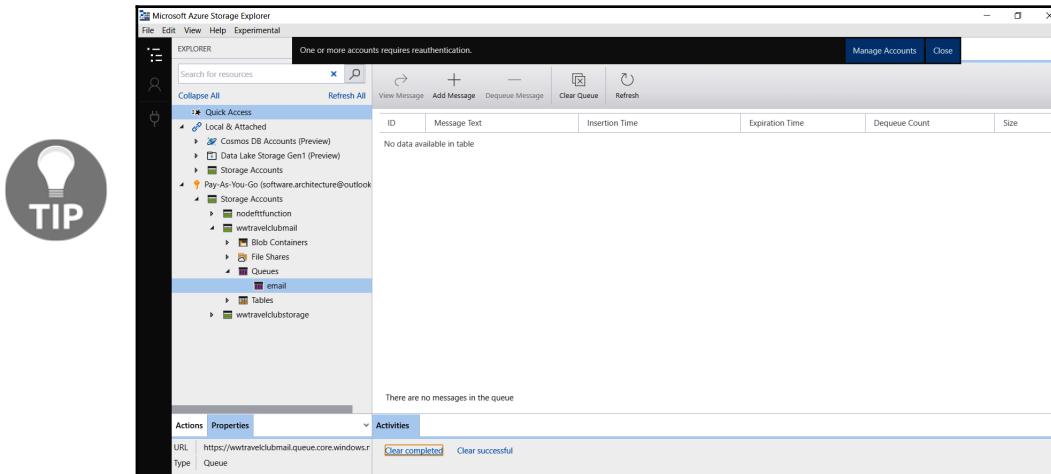
- Once you have the storage account in place, you will be able to set up a queue.
You just need to provide the queue's name:

The screenshot shows the 'Queues' blade for a storage account named 'wwtravelclubmail'. On the left, there is a sidebar with various navigation links: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Storage Explorer (preview), Settings, Access keys, and Geo-replication. The main area displays a modal dialog titled 'Add queue'. It contains a search bar labeled 'Search (Ctrl+ /)' and a form with a 'Queue name' field containing the value 'email'. Below the form are 'OK' and 'Cancel' buttons. The URL in the browser's address bar is 'Home > Microsoft.StorageAccount-20190225201427 - Overview > wwtravelclubmail - Queues'.

- The created queue will give you an overview of Azure Portal. There, you will find your queue's URL and use the Storage Explorer:

The screenshot shows the same 'Queues' blade after the queue 'email' has been created. The sidebar and overall layout remain the same. The main area now lists the single queue 'email' under the 'QUEUE' column and provides its 'URL' as 'https://wwtravelclubmail.queue.core.windows.net/email'. The URL in the browser's address bar is identical to the previous screenshot.

Note that you will also be able to connect to this storage using Microsoft Azure Storage Explorer (<https://azure.microsoft.com/en-us/features/storage-explorer/>):



- Now, you can start your functional programming to inform the queue that an email is waiting to be sent. Here, we need to use an HTTP trigger. Note that the function is a static class that runs asynchronously. The following code is gathering the request data coming from the HTTP trigger and is inserting the data into a queue that will be treated later:

```
public static class SendEmail
{
    [FunctionName(nameof(SendEmail))]
    public static async Task<HttpResponseMessage> RunAsync(
        [HttpTrigger(AuthorizationLevel.Function, "post")]
        HttpRequestMessage req, ILogger log)
    {
        var requestData = await req.Content.ReadAsStringAsync();
        var YOUR_CONNECTION_STRING =
            "YOUR_AZURE_STORAGE_ACCOUNT_CONNECTION_STRING_HERE";
        var storageAccount =
            CloudStorageAccount.Parse(YOUR_CONNECTION_STRING);
        var queueClient = storageAccount.CreateCloudQueueClient();
        var messageQueue = queueClient.GetQueueReference("email");
        var message = new CloudQueueMessage(requestData);
        await messageQueue.AddMessageAsync(message);
        log.LogInformation("HTTP trigger from SendEmail function processed a request.");
    }
}
```

```
        return req.CreateResponse(HttpStatusCode.OK, new { success = true }, JsonMediaTypeFormatter.DefaultMediaType);
    }
}
```

5. You can use a tool such as Postman to test your function by running the Azure Functions Emulator:

The screenshot shows the Postman interface with a POST request to `http://localhost:7071/api/SendEmail`. The request body is a JSON object:

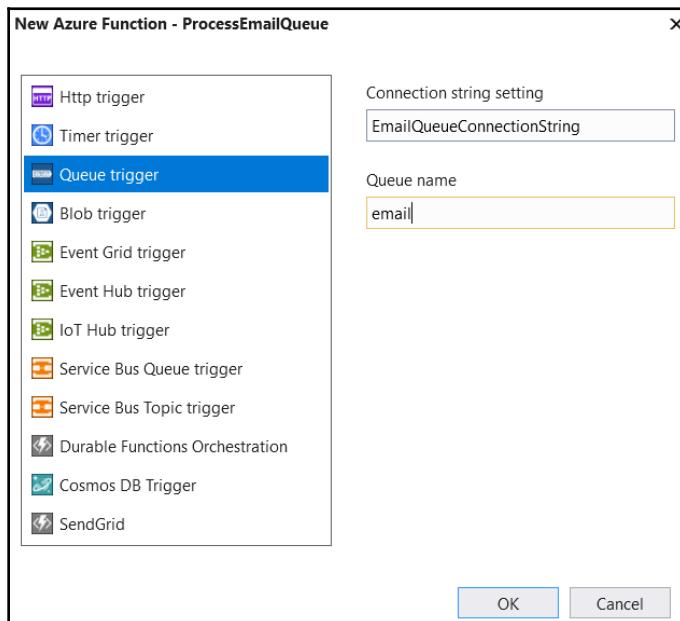
```
1 {  
2   "to" : "software.architecture@outlook.com",  
3   "subject" : "Simple e-mail test",  
4   "body" : "Hey, how are you doing?"  
5 }
```

The response details show a Status of 200 OK, Time of 3713 ms, and Size of 164 B. The response body is:

```
1 {  
2   "success": true  
3 }
```

6. The result will appear in Microsoft Azure Storage Explorer and Azure Portal. In Azure Portal, you can manage each message and dequeue each of them or even clear the queue storage:

- After this, you can create a second function. This one will be triggered by data entering your queue. It is worth mentioning that, for Azure Functions v2, you will need to add the `Microsoft.Azure.WebJobs.Extensions.Storage` library as a NuGet reference:



8. Once you've set the connection string inside `local.settings.json`, you will be able to run both functions and test them with Postman. The difference is that, with the second function running, if you set a breakpoint at the start of it, you will check whether the message has been sent:

```
using System;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;

namespace FunctionAppWithTravel
{
    public static class ProcessEmailQueue
    {
        [FunctionName("ProcessEmailQueue")]
        public static void Run([QueueTrigger("email", Connection = "EmailQueueConnectionString")]string myQueueItem, ILogger log)
        {
            log.LogInformation($"C# Queue trigger function processed: {myQueueItem}");
        }
    }
}
```

9. From this point, the way to send emails will depend on the mail options you have. You may decide to use a proxy or may connect directly to your email server.

There are several advantages to creating an email service this way:

- Once your service has been coded and tested, you can use it to send emails from any of your applications. This means that your code can always be reused.
- Apps that use this service will not be stopped from sending emails due to the asynchronous advantages of posting in an HTTP service.
- You don't need to poll the queue to check whether data is ready for processing.

Finally, the queue process runs concurrently, which delivers a better experience in most cases. It is possible to turn it off by setting some properties in `host.json`. All of the options for this can be found in the *Further reading* section, at the end of this chapter.

Summary

In this chapter, we looked at some of the advantages of developing functionality with Serverless Azure Functions. You can use it as a guideline for checking the different types of triggers that are available in Azure Functions and for planning how to monitor them. We also saw how to program and maintain Azure functions. Finally, we looked at an example of an architecture where you connect multiple functions to avoid pooling data and to enable concurrent processing.

In the next chapter, we will analyze the concept of design patterns, learn why they are so useful, and learn about some of their common patterns.

Questions

1. What are Azure Functions?
2. What are the programming options for Azure Functions?
3. What are the plans that can be used with Azure Functions?
4. How can you deploy Azure Functions with Visual Studio?
5. What triggers can you use to develop Azure Functions?
6. What is the difference between Azure Functions v1 and v2?
7. How does Application Insights help us maintain and monitor Azure Functions?

Further reading

If you want to learn more when it comes to creating Azure Functions, check out the following links:

- **Azure Functions scale and hosting:** <https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale>
- **Azure Functions - Essentials [Video]** by Praveen Kumar Sreeram: <https://www.packtpub.com/virtualization-and-cloud/azure-functions-essentials-video>
- **Introducing Azure Functions 2.0:** <https://azure.microsoft.com/en-us/blog/introducing-azure-functions-2-0/>
- **An overview of Azure Event Grid:** <https://azure.microsoft.com/en-us/resources/videos/an-overview-of-azure-event-grid/>
- **Timer trigger for Azure Functions:** <https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-timer>
- **Application insights section from the book, *Azure for Architects* by Ritesh Modi:** https://subscription.packtpub.com/book/virtualization_and_cloud/9781788397391/12/ch12lvl1sec95/application-insights
- **Monitoring Azure Functions using Application Insights section from the book, *Azure Serverless Computing Cookbook* by Praveen Kumar Sreeram:** https://subscription.packtpub.com/book/virtualization_and_cloud/9781788390828/6/06lvl1sec34/monitoring-azure-functions-using-application-insights
- **Get started with Azure Queue storage using .NET:** <https://docs.microsoft.com/en-us/azure/storage/queues/storage-dotnet-how-to-use-queues>
- **Azure Functions triggers and bindings concepts:** <https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings>
- **Azure Queue storage bindings for Azure Functions:** <https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-queue>

3

Section 3: Applying Design Principles for Software Delivered in the 21st Century

In this section, you will learn about the main patterns, best practices, and frameworks used in modern enterprise architectures. All examples are on C# 8 running on .NET Core 3.0.

In Chapter 9, *Design Patterns and .NET Core Implementation*, you will learn about the .NET Core implementation of well-known and general patterns and best practices, while Chapter 11, *Implementing Code Reusability in C# 8*, describes techniques and best practices that enhance code reusability.

Chapter 10, *Understanding the Different Domains in a Software Solution*, describes the modern domain-driven design software production methodology, which will enable you to get the most out of cloud- and microservice-based architectures and tackle complex applications that require several knowledge domains. There, you will learn both about analysis techniques and architectures and tools involved in projects based on domain-driven design.

Chapter 12, *Applying Service-Oriented Architectures with .NET Core*, and Chapter 13, *Presenting ASP.NET Core MVC*, describe web architectures that form the backbone of modern enterprise applications. Both chapters are based on ASP.NET Core 3.0, which is the web framework that comes with .Net Core 3.0. Chapter 12, *Applying Service-Oriented Architectures with .NET Core*, describes web architectures where the enterprise system can be accessed by external client applications through endpoints exposed on the web or on a private network. Chapter 12, *Applying Service-Oriented Architectures with .NET Core*, is dedicated to ASP.NET Core MVC web applications, which don't need specific clients because they use browsers as clients. More specifically, they interact with the user through the HTML they send to a standard browser.

This section includes the following chapters:

- Chapter 9, *Design Patterns and .NET Core Implementation*
- Chapter 10, *Understanding the Different Domains in a Software Solution*
- Chapter 11, *Implementing Code Reusability in C# 8*
- Chapter 12, *Applying Service-Oriented Architectures with .NET Core*
- Chapter 13, *Presenting ASP.NET Core MVC*

9

Design Patterns and .NET Core Implementation

Design patterns can be defined as ready to use architectural solutions for common problems you encounter during software development. They are essential for understanding the .NET Core architecture and useful for solving ordinary problems that we face when designing any piece of software. In this chapter, we will look at the implementation of some design patterns. It is worth mentioning that this book doesn't explain all the known patterns we can use. The focus here is to explain the importance of studying and applying them.

In this chapter, we will cover the following topics:

- Understanding design patterns and their purpose
- Understanding the available design patterns in .NET Core

By the end of this chapter, you will have learned about some of the use cases from WWTravelClub that you can implement with design patterns.

Technical requirements

You will require the following to complete this chapter:

- Visual Studio 2017 or 2019 free Community Edition or better with all the database tools installed.
- A free Azure account. The *Creating an Azure account* subsection of Chapter 1, *Understanding the Importance of Software Architecture*, explains how to create one.

You can find the sample code for this chapter at <https://github.com/PacktPublishing/Hands-On-Software-Architecture-with-CSharp-8/tree/master/ch09>.

Understanding design patterns and their purpose

Being able to decide on the design of a system is challenging, and the responsibility associated with this task is enormous. As a software architect, we must always keep in mind that features such as great reusability, good performance, and good maintainability are key. This is where design patterns help and accelerate the design process.

As we mentioned previously, design patterns are solutions that have already been discussed and defined so that they can solve common software architectural problems. This approach grew in popularity after the release of the book *Design Patterns – Elements of Reusable Object-Oriented Software*, where the **Gang of Four (GoF)** divided these patterns into three types:

- Creational
- Structural
- Behavioral

A little bit later, Uncle Bob introduced the SOLID principles to the developers community, giving us the opportunity to efficaciously organize functions and data structures of each system into classes. SOLID principles indicate how these classes should be connected. It is worth mentioning that, compared to the design patterns presented by GoF, SOLID principles don't deliver code recipes. Instead, they give you the basic principles to follow when you design your solutions.

As technologies and software problems change, more patterns are conceived. The advance of cloud computing has brought a bunch of them, all of which can be found at <https://docs.microsoft.com/azure/architecture/patterns/>.

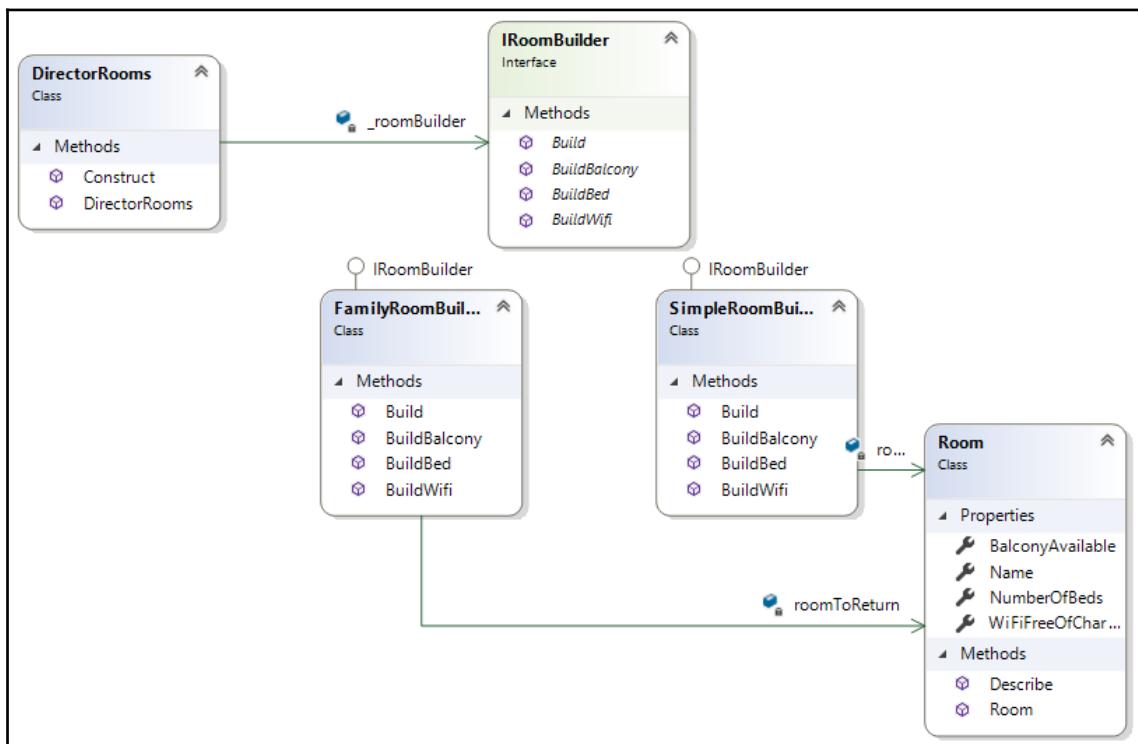
The reason why you should always consider them is quite simple – as a software architect, you cannot spend time reinventing the wheel. However, there is another great reason for using them: you will find many patterns implemented in .NET Core.

In the next few subsections, we will cover some of the most well-known patterns. However, the idea of this chapter is to let you know that they exist and need to be studied so that you can accelerate and simplify your project. Moreover, each pattern will be presented with a C# code snippet so that you can easily implement them in your projects.

Builder pattern

There are cases where you will have a complex object with different behaviors due to its configuration. Instead of setting this object up while using it, you may want to decouple its configuration from its usage, using a customized configuration already built. This way, you have different representations of the instances you are building. This is where you should use the Builder pattern.

The following class diagram shows the pattern that has been implemented for a scenario from this book's use case. The idea behind this design choice is to simplify the way rooms from WWTravelClub are described:



As shown in the following code, the code for this is implemented in a way where the configurations of the instances aren't set in the main program. Instead, you just build the objects using the `Construct()` method. This example is simulating the creation of different room styles (a single room and a family room) in the WWTravelClub:

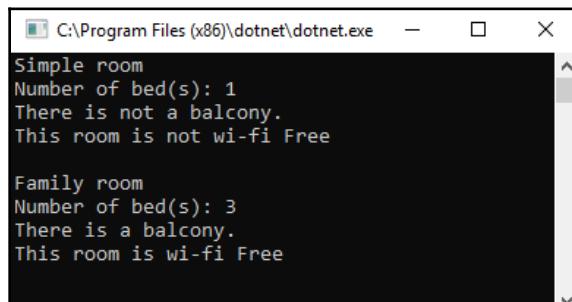
```
using DesignPatternsSample.BuilderSample;
using System;
```

```
namespace DesignPatternsSample
{
    class Program
    {
        static void Main()
        {
            #region Builder Sample
            Console.WriteLine("Builder Sample");

            var directorRoom = new DirectorRooms(new SimpleRoomBuilder());
            var simpleRoom = directorRoom.Construct();
            simpleRoom.Describe();
            directorRoom = new DirectorRooms(new FamilyRoomBuilder());
            var familyRoom = directorRoom.Construct();
            familyRoom.Describe();
            #endregion

            Console.ReadKey();
        }
    }
}
```

The result of this implementation is quite simple but clarifies the reason why you need to implement a pattern:

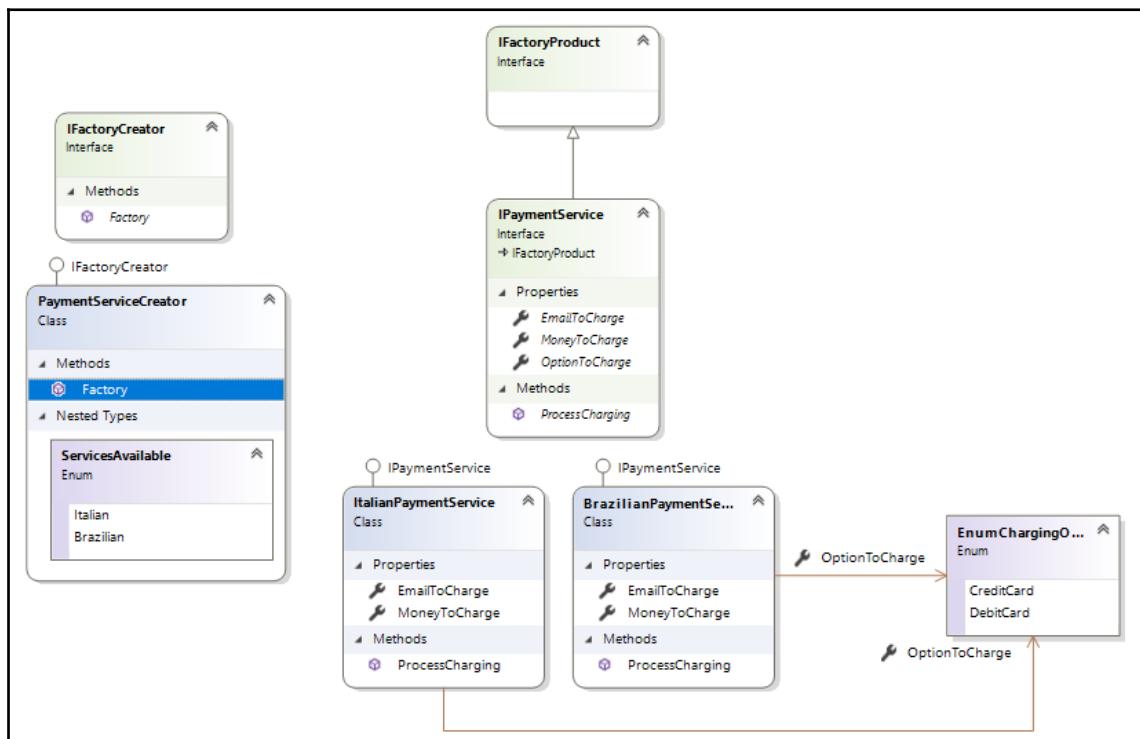


As soon as you have the implementation, evolving this code becomes simpler and easier. For example, if you need to build a different style of room, you just have to create the concrete builder for that and you will be able to use it. Fortunately, if you need to increase the configuration settings for the product, all the concrete classes you used previously will be defined in the Builder interface and stored there so that you can update them with ease.

Factory pattern

The Factory pattern is really useful in situations where you have multiple objects from the same abstraction and you don't know which need to be created by the time you start coding. This means you will have to create the instance according to a certain configuration or according to where the software is living at the moment.

For instance, let's check out the WWTravelClub sample. Here, there's a User Story that describes that this application will have customers from all over the world paying for their trips. However, in the real world, there are different payment services available for each country. The process of paying is similar for each country, but this system will have more than one payment service available. A good way to simplify this payment implementation is by using the Factory pattern. The following diagram shows the basic idea of its architectural implementation:



Notice that, since you have an interface that describes what the Payment Service for the application is, you can use the factory to change the concrete class according to the services that are available:

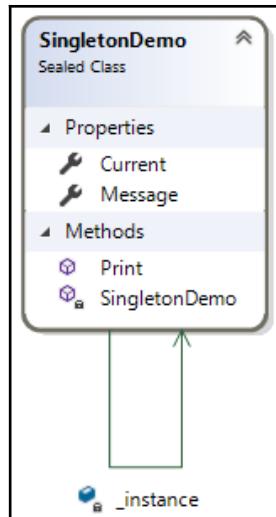
```
static void Main()
{
    #region Factory Sample
    var psCreator = new PaymentServiceCreator();
    var brazilianPaymentService = (IPaymentService)psCreator.Factory
        (PaymentServiceCreator.ServicesAvailable.Brazilian);
    brazilianPaymentService.EmailToCharge = "gabriel@sample.com";
    brazilianPaymentService.MoneyToCharge = 178.90f;
    brazilianPaymentService.OptionToCharge =
        FactorySample.Enums.EnumChargingOptions.CreditCard;
    brazilianPaymentService.ProcessCharging();
    var italianPaymentService = (IPaymentService)psCreator.Factory
        (PaymentServiceCreator.ServicesAvailable.Italian);
    italianPaymentService.EmailToCharge = "francesco@sample.com";
    italianPaymentService.MoneyToCharge = 188.70f;
    italianPaymentService.OptionToCharge =
        FactorySample.Enums.EnumChargingOptions.DebitCard;
    italianPaymentService.ProcessCharging();
    #endregion
    Console.ReadKey();
}
```

Once again, the service's usage has been simplified due to the implemented pattern. If you were to use this code in a real-world application, you would change the instance's behavior by defining the service you need in the Factory.

Singleton pattern

When you implement a Singleton in your application, you will have a single instance of the object implemented in the entire solution. This can be considered as one of the most used patterns in every application. The reason is simple – there are many use cases where you need some classes to have just one instance. Singletons solve this by providing a better solution than a global variable does.

In the Singleton pattern, the class is responsible for creating and delivering a single object that will be used by the application. In other words, the Singleton class creates a single instance:



To do so, the object that's created is `static` and is delivered in a `static` property or method. The following code implements the Singleton pattern, which has a `Message` property and a `Print()` method:

```
public sealed class SingletonDemo
{
    #region This is the Singleton definition
    private static SingletonDemo _instance;
    public static SingletonDemo Current
    {
        get
        {
            if (_instance == null)
                _instance = new SingletonDemo();
            return _instance;
        }
    }
    #endregion

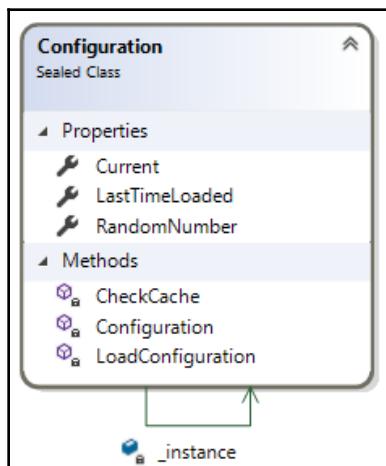
    public string Message { get; set; }

    public void Print()
    {
        Console.WriteLine(Message);
    }
}
```

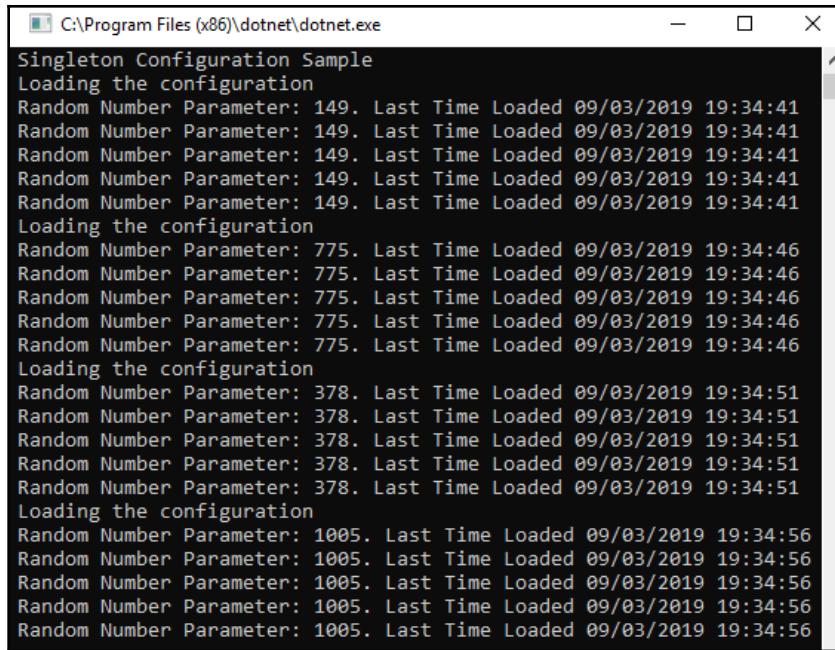
Its usage is really simple – you just need to call the static property every time you need to use the Singleton object:

```
SingletonDemo.Current.Message = "This text will be printed by the
singleton.";
SingletonDemo.Current.Print();
```

One of the places where you may use this pattern is when you need to deliver the app configuration in a way that can be easily accessed from anywhere in the solution. For instance, let's say you have some configuration parameters that are stored in a table that your app needs to query at several decision points. Instead of querying the configuration table directly, you can create a Singleton class to help you:



Moreover, you will need to implement a cache in this Singleton, thus improving the performance of the system, since you will be able to decide whether the system will check each configuration in the database every time it needs it or if the cache will be used. The following screenshot shows the implementation of the cache where the configuration is loaded every 5 seconds. The parameter that is being read in this case is just a random number:



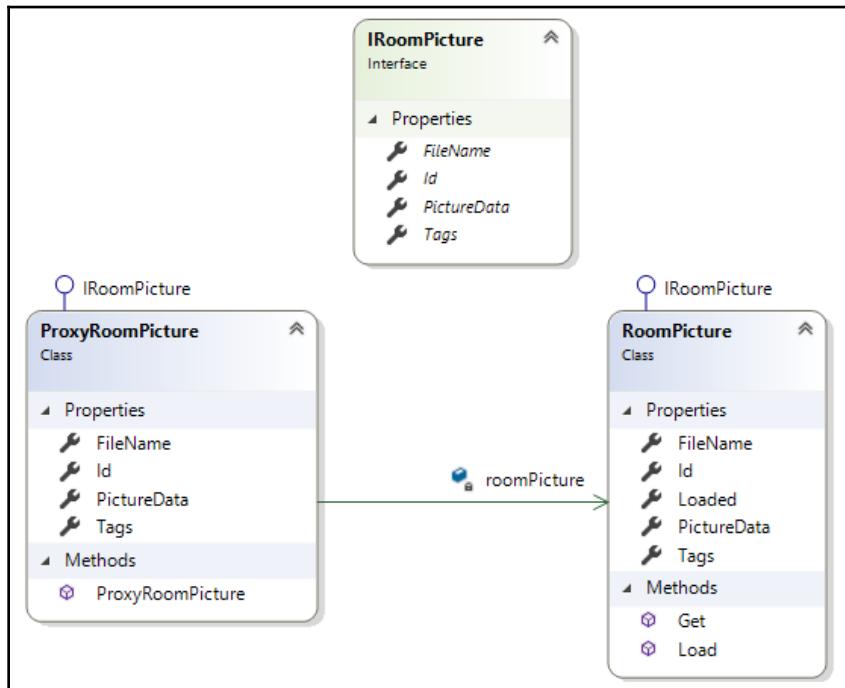
```
C:\Program Files (x86)\dotnet\dotnet.exe
Singleton Configuration Sample
Loading the configuration
Random Number Parameter: 149. Last Time Loaded 09/03/2019 19:34:41
Random Number Parameter: 149. Last Time Loaded 09/03/2019 19:34:41
Random Number Parameter: 149. Last Time Loaded 09/03/2019 19:34:41
Random Number Parameter: 149. Last Time Loaded 09/03/2019 19:34:41
Random Number Parameter: 149. Last Time Loaded 09/03/2019 19:34:41
Loading the configuration
Random Number Parameter: 775. Last Time Loaded 09/03/2019 19:34:46
Random Number Parameter: 775. Last Time Loaded 09/03/2019 19:34:46
Random Number Parameter: 775. Last Time Loaded 09/03/2019 19:34:46
Random Number Parameter: 775. Last Time Loaded 09/03/2019 19:34:46
Random Number Parameter: 775. Last Time Loaded 09/03/2019 19:34:46
Loading the configuration
Random Number Parameter: 378. Last Time Loaded 09/03/2019 19:34:51
Random Number Parameter: 378. Last Time Loaded 09/03/2019 19:34:51
Random Number Parameter: 378. Last Time Loaded 09/03/2019 19:34:51
Random Number Parameter: 378. Last Time Loaded 09/03/2019 19:34:51
Random Number Parameter: 378. Last Time Loaded 09/03/2019 19:34:51
Loading the configuration
Random Number Parameter: 1005. Last Time Loaded 09/03/2019 19:34:56
Random Number Parameter: 1005. Last Time Loaded 09/03/2019 19:34:56
Random Number Parameter: 1005. Last Time Loaded 09/03/2019 19:34:56
Random Number Parameter: 1005. Last Time Loaded 09/03/2019 19:34:56
Random Number Parameter: 1005. Last Time Loaded 09/03/2019 19:34:56
```

This is great for the application's performance. Besides, using parameters in several places in your code is simpler, since you don't have to create configuration instances everywhere in the code.

Proxy pattern

The Proxy pattern is used when you need to provide an object that controls access to another object. One of the biggest reasons why you should do this is related to the cost of creating the object that is being controlled. For instance, if the controlled object takes too long to be created or consumes too much memory, a proxy can be used to guarantee that the huge part of the object will only be created when it's required.

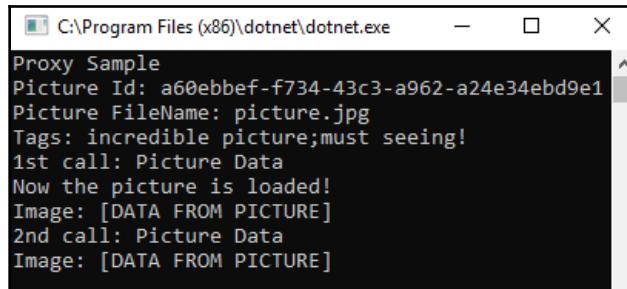
The following class diagram shows the class diagram of a Proxy pattern's implementation for loading pictures from the Room, but only when requested:



The client of this proxy will request its creation. Here, the proxy will only gather basic information (`Id`, `FileName`, and `Tags`) from the real object and won't query `PictureData`. When `PictureData` is requested, the proxy will load it:

```
static void Main()
{
    Console.WriteLine("Proxy Sample");
    var roomPicture = new ProxyRoomPicture();
    Console.WriteLine($"Picture Id: {roomPicture.Id}");
    Console.WriteLine($"Picture FileName: {roomPicture.FileName}");
    Console.WriteLine($"Tags: {string.Join(", ", roomPicture.Tags)}");
    Console.WriteLine("1st call: Picture Data");
    Console.WriteLine($"Image: {roomPicture.PictureData}");
    Console.WriteLine("2nd call: Picture Data");
    Console.WriteLine($"Image: {roomPicture.PictureData}");
}
```

If `PictureData` is requested again, since image data is already in place, the proxy will guarantee that image reloading will not be repeated. The following screenshot shows the result of running the preceding code:



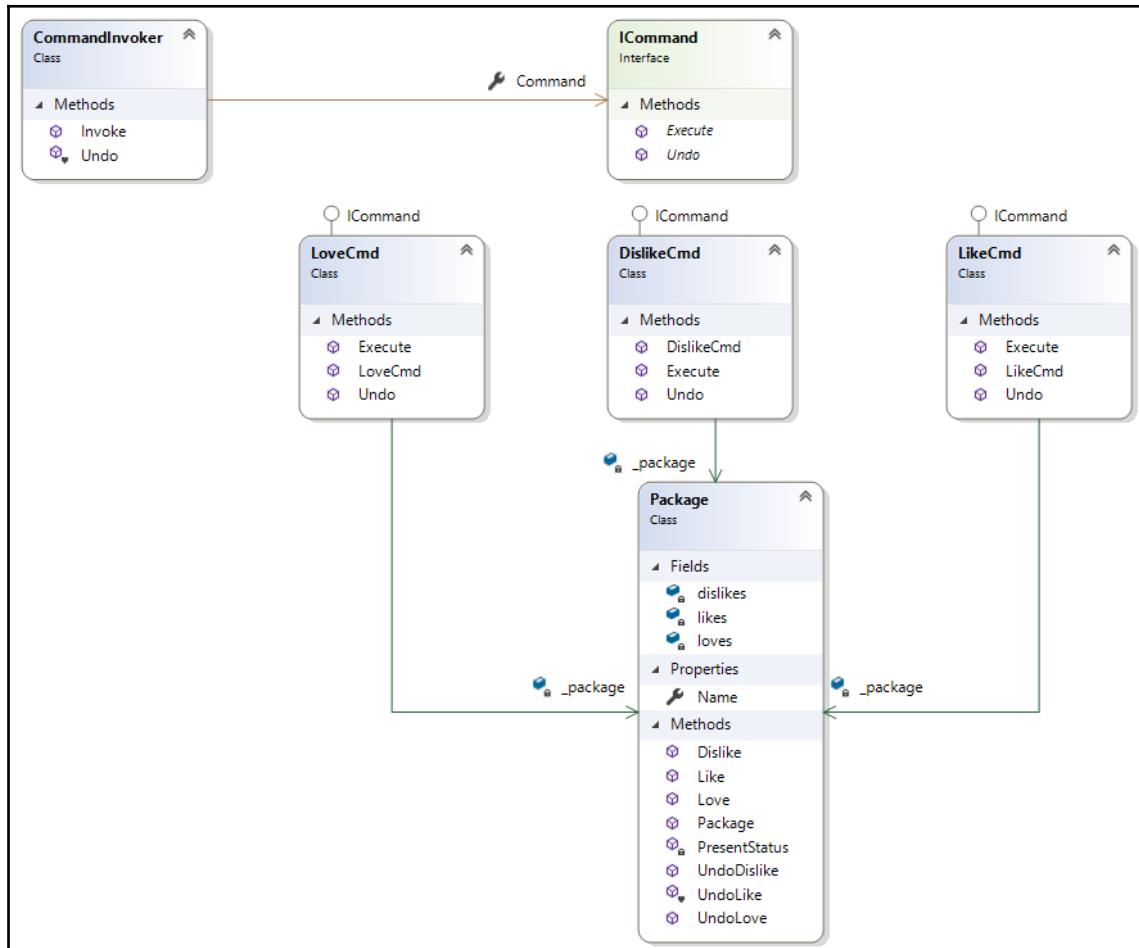
```
C:\Program Files (x86)\dotnet\dotnet.exe
Proxy Sample
Picture Id: a60ebbef-f734-43c3-a962-a24e34ebd9e1
Picture FileName: picture.jpg
Tags: incredible picture;must seeing!
1st call: Picture Data
Now the picture is loaded!
Image: [DATA FROM PICTURE]
2nd call: Picture Data
Image: [DATA FROM PICTURE]
```

This technique can be referred to as another well-known pattern: **lazy loading**. In fact, the Proxy pattern is a way of implementing lazy loading. For instance, in Entity Framework Core 2.1, as discussed in Chapter 6, *Interacting with Data in C# - Entity Framework Core*, you can turn on lazy loading using proxies. You can find out more about this at <https://docs.microsoft.com/en-us/ef/core/querying/related-data#lazy-loading>.

Command pattern

There are many cases where you need to execute a *command* that will affect the behavior of an object. The Command pattern can help you with this by encapsulating this kind of request in an object. The pattern also describes how to handle undo/redo support for the request.

For instance, let's imagine that, on the WWTravelClub website, the users have the ability to evaluate the packages by specifying whether they like, dislike, or even love them. The following class diagram is an example of what can be implemented to create this rating system with the Command pattern:

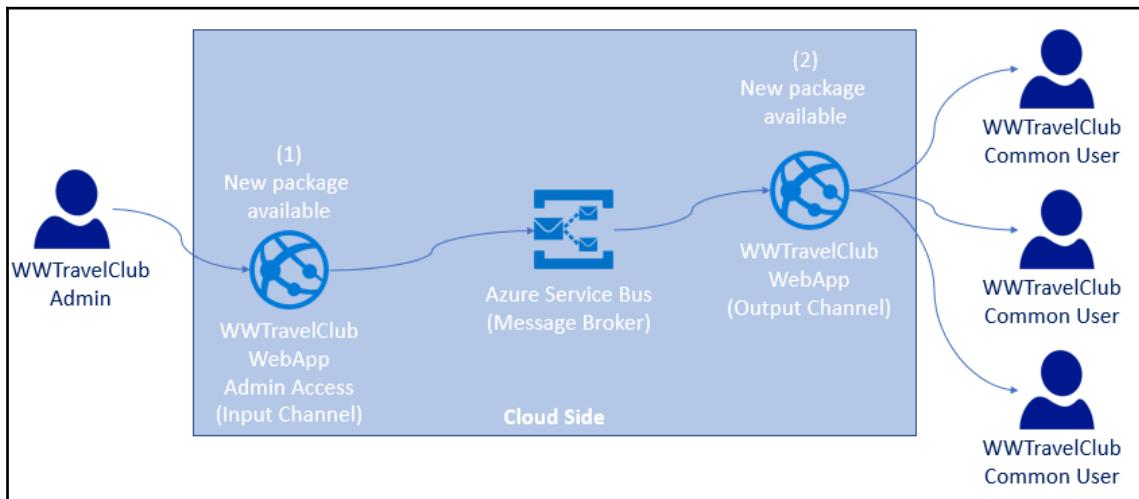


Notice the way this pattern works – if you need a different command, such as **Hate**, you don't need to change the code and classes that use the command. The Undo method can be added in a similar way to the Redo method. The full code sample for this is available in this book's GitHub repository.

Publisher/Subscriber pattern

Providing information from an object to a group of other objects is common in all applications. The Publisher/Subscriber pattern is almost mandatory when there's a large volume of components (subscribers) that will receive a message containing the information that was sent by the object (publisher).

The concept here is quite simple to understand and is shown in the following diagram:



When you have an indefinite number of different possible subscribers, it is essential to decouple the component that broadcasts information from the components that consume it. The Publisher/Subscriber pattern does this for us.

Implementing this pattern is complex, since distributing environments isn't a trivial task. Therefore, it is recommended that you consider already existing technologies for implementing the Message Broker that connects the Input Channel to the Output Channels, instead of building it from scratch. Azure Service Bus is a reliable implementation of this pattern, so all you need to do is connect to it.

RabbitMQ, which we mentioned in Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*, is another service that can be used to implement a Message Broker, but it is a lower-level implementation of the pattern and requires several related tasks, such as retries, in case errors have to be coded manually.

Dependency Injection pattern

The Dependency Injection pattern is considered a good way to implement the Dependency Inversion principle. Besides, it forces all the other SOLID principles to be followed by the implementation. As we discussed at the beginning of this chapter, a way to keep the software's structure strong and reliable is by following the SOLID design principles presented by Uncle Bob. These can be defined as follows:

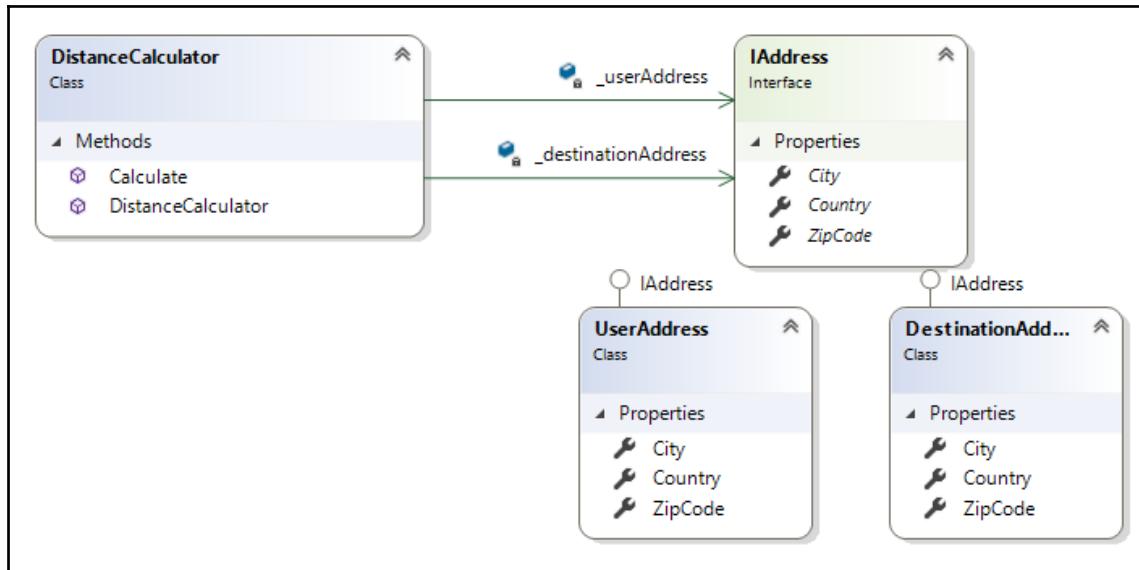
- **Single Responsibility:** A module or function should be responsible for a single purpose.
- **Open-Closed:** A software artifact should be open for extension but closed for modification.
- **Liskov Substitution:** The behavior of a program needs to remain unchanged when you substitute one of its components for another component that's been defined by a supertype of the primer object.
- **Interface Segregation:** Creating huge interfaces will cause dependencies to occur while you're building concrete objects, but these are harmful to the system architecture.
- **Dependency Inversion:** The most flexible systems are the ones where object dependencies only refer to abstractions.

This concept is quite simple. Instead of creating instances of the objects that the component depends on, you just need to define their dependencies, declare their interfaces, and enable the reception of the objects by *injection*.

There are three ways to perform dependency injection:

- Use the constructor of the class to receive the objects.
- Tag some class properties to receive the objects.
- Define an interface with a method to inject all the necessary components.

The following diagram shows the implementation of the Dependency Injection pattern:



Apart from this, dependency injection can be used with an **Inversion of Control (IoC)** container. This container enables the automatic injection of dependencies whenever they are asked for. There are several IoC container frameworks available on the market, but with .NET Core, there is no need to use third-party software since it contains a set of libraries to solve this in the `Microsoft.Extensions.DependencyInjection` namespace.

This IoC container is responsible for creating and disposing of the objects that are requested. The implementation of dependency injection is based on Constructor types. There are three options for the injected component's lifetime:

- **Transient**: The objects are created each time they are requested.
- **Scoped**: The objects are created for each **scope** defined in the application. In a Web App, a *scope* is identified with a web request.
- **Singleton**: Each object has the same application lifetime, so a single object is reused to serve all the requests for a given type.

The way you are going to use these options depends on the business rules of the project you are developing. You need to be careful in deciding the correct one, since the behavior of the application will change according to the type of object you are injecting.

Understanding the available design patterns in .NET Core

As we discovered in the previous sections, C# allows us to implement any of the aforementioned patterns. .NET Core provides many implementations in its SDK that follow all the patterns we've discussed, such as Entity Framework Core Lazy Loading. Another good example that's been available since .NET Core 2.1 is .NET Generic Host.

In Chapter 13, *Presenting ASP.NET Core MVC*, we will detail the hosting that's available for Web Apps in .NET Core. This web host helps us since the app's startup and lifetime management is set up alongside it. The idea of .NET Generic Host is to enable this pattern for applications that don't need HTTP implementation. With this Generic Host, any .NET Core program can have a Startup class where we can configure the Dependency Injection Engine. This can be really useful for creating multi-service apps.

You can find out more at .NET Generic Host at <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host>, which contains some sample code. The code provided in this book's GitHub repository is simpler, but it focuses on the creation of a console app that can run a service for monitoring. The great thing about this is the way the console app is set up to run, where the builder configures the services that will be provided by the application, and the way logging will be managed. This is shown in the following code:

```
public static void Main()
{
    var host = new HostBuilder()
        .ConfigureServices((hostContext, services) =>
    {
        services.AddHostedService<HostedService>();
        services.AddHostedService<MonitoringService>();
    })
        .ConfigureLogging((hostContext, configLogging) =>
    {
        configLogging.AddConsole();
    })
        .Build();
    host.Run();
    Console.WriteLine("Host has terminated. Press any key to finish the
App.");
    Console.ReadKey();
}
```

The preceding code gives us an idea of how .NET Core uses design patterns. Using the Builder pattern, .NET Generic Host allows you to set the classes that will be injected as services. Apart from this, the Builder pattern helps you configure some other features, such as the way logs will be shown/stored. This configuration allows the services to inject `ILogger<out TCategoryName>` objects into any instance.

Summary

In this chapter, we understood why design patterns help with the maintainability and reusability of the parts of the system you are building. We also looked at some typical use cases and code snippets that you can use in your projects. Finally, we presented .NET Generic Host, which is a good example of how .NET uses design patterns to enable code reusability and enforce best practices. All this content will help you while architecting a new software or even maintaining an existing one, since design patterns are solutions already known for some real life problems in software development.

In the next chapter, we will cover the domain-driven design approach. We will also learn how to use SOLID design principles so that we can map different domains to our software solutions.

Questions

1. What are design patterns?
2. What's the difference between design patterns and design principles?
3. When is it a good idea to implement the Builder pattern?
4. When is it a good idea to implement the Factory pattern?
5. When is it a good idea to implement the Singleton pattern?
6. When is it a good idea to implement the Proxy pattern?
7. When is it a good idea to implement the Command pattern?
8. When is it a good idea to implement the Publisher/Subscriber pattern?
9. When is it a good idea to implement the Dependency Injection pattern?

Further reading

The following are some books and websites where you can find out more regarding what was covered in this chapter:

- *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Martin, Robert C. Pearson Education, 2018.
- *Design Patterns: Elements of Reusable Object-Oriented Software*, Eric Gamma et al. Addison-Wesley, 1994.
- *Design Principles and Design Patterns*, Robert C. Martin, 2000.

If you need to get more info about design patterns and architectural principles, please check these links:

- <https://www.packtpub.com/application-development/design-patterns-using-c-and-net-core-video>
- <https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/architectural-principles>

If you want to understand better the idea of generic hosts, follow this link:

- <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host>

There is a very good explanation about service bus messaging in this link:

- <https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-dotnet-how-to-use-topics-subscriptions>

You can learn more about dependency injection checking these links:

- <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection>
- <https://www.martinfowler.com/articles/injection.html>

10

Understanding the Different Domains in Software Solutions

This chapter is dedicated to a modern software development technique called **domain-driven design (DDD)**, which was first proposed by Eric Evans. While DDD has existed for more than 15 years, it reached a great success in the last few years because of its ability to cope with two important problems:

- Modeling complex systems where no single expert has in-depth knowledge of the whole domain. This knowledge is split among several people.
- Facing big projects with several development teams. There are many reasons why a project is split among several teams, with the most common being the team's size and all of its members having different skills and/or different locations. In fact, experience has proven that teams of more than 6-8 people are not efficacious and clearly different skills and locations prevent a tight interaction from occurring. Team splitting prevents tight interaction from happening for all the people involved in the project.

In turn, the importance of the two aforementioned problems grew more in the last few years for the following reasons:

- Software systems always took up a lot of space inside every organization and became more and more complex and geographically distributed.
- At the same time, the need for frequent updates increased so that these complex software systems could be adapted, as per the needs of a quickly changing market.
- The preceding problems led to the conception of more sophisticated CI/CD cycles and the adoption of complex distributed architectures that may leverage reliability, high throughput, quick updates, and the capability to evolve legacy subsystems gradually. Yes – we are speaking of the microservices and container-based architectures we analyzed in *Chapter 5, Applying a Microservice Architecture to Your Enterprise Application*.

In this scenario, it's common to implement complex software systems with associated fast CI/CD cycles that always require more people to evolve and maintain them. In turn, this created the requirement of developing technologies that were adequate for high-complexity domains and for the cooperation of several loosely coupled development teams.

In this chapter, we will analyze the basic principles, advantages, and common patterns related to DDD, as well as how to use them in our solutions. More specifically, we will cover the following topics:

- What are software domains?
- Understanding domain-driven design
- Using SOLID principles to map your domains
- Use case – understanding the domains of the use case

Let's get started.

Technical requirements

This chapter requires Visual Studio 2017 or 2019 free Community Edition or better with all the database tools installed.

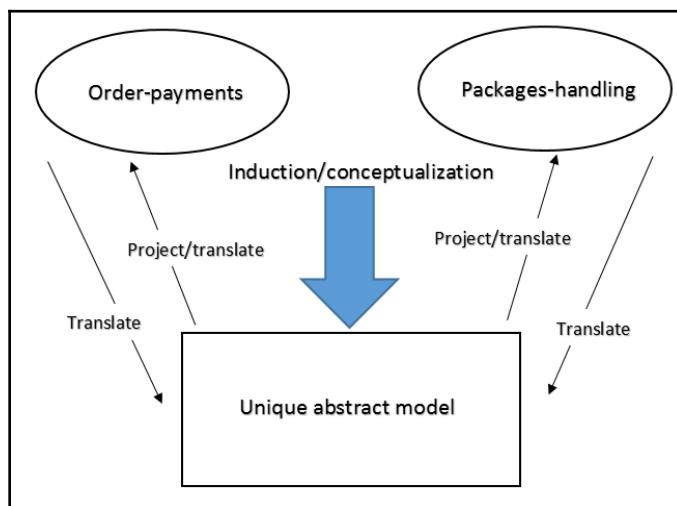
All the code snippets in this chapter can be found in the GitHub repository associated with this book, <https://github.com/PacktPublishing/Hands-On-Software-Architecture-with-CSharp-8>.

What are software domains?

As we discussed in *Chapter 2, Functional and Nonfunctional Requirements*, and *Chapter 3, Documenting the Requirements with Azure DevOps*, the transfer of knowledge from domain experts to the development team plays a fundamental role in software design. Developers try to communicate with experts and describe their solutions in a language that domain experts and stakeholders can understand. However, often, the same word has a different meaning in various parts of an organization, and what appear to be the same conceptual entities have completely different shapes in different contexts.

For instance, in our WWTravelClub use case, the order-payment and packages-handling subsystems use completely different models for customers. Order-payment characterizes a customer by their payment methods and currency, bank accounts, and credit cards, while package-handling is more concerned with the locations and packages that have been visited and/or purchased in the past, the user's preferences, and their geographical location. Moreover, while order-payment refers to various concepts with a language that we may roughly define as a *bank language*, packages-handling uses a language that is typical of travel agencies/operators.

The classical way to cope with these discrepancies is to use a unique abstract entity called **customer**, which projects into two different views – the order-payment view and the packages-handling view. Each projection operation takes some operations and some properties from the **customer** abstract entity and changes their names. Since domain experts only give us the projected views, our main task as system designers is to create a conceptual model that can explain all the views. The following diagram shows how different views are handled:



The main advantage of the classic approach is that we have a unique and coherent representation of the data of the domain. If this conceptual model is built successfully, all the operations will have a formal definition and purpose and the whole abstraction will be a rationalization of the way the whole organization *should* work, possibly highlighting and correcting errors and simplifying some procedures.

However, what are the downsides of this approach? First of all, the way work is organized may have an excessive impact on the preexisting organization that may prevent it from operating correctly for a certain amount of time since the constraint of using a unique coherent model doesn't offer enough options to mitigate this impact. Errors must be removed, duplication must be removed, and everything must be perfectly coherent so that there is a minimum irreducible impact that we can't mitigate since the only way to mitigate it would be to renounce the overall coherence.

This minimum impact can be acceptable in a small organization, when the software is destined for a small part of the overall organization, or when the software automatizes a small enough percentage of the data flow. However, as the software becomes the backbone of a whole geographically distributed organization, sharp changes become more unacceptable and unfeasible. Moreover, as the complexity of the software system grows, several other issues appear, as follows:

- Arriving at a uniquely coherent view of data becomes more difficult since we can't face complexity by breaking these tasks into smaller loosely coupled tasks.
- As complexity grows, there is a need for frequent system changes, but it is quite difficult to update and maintain a unique global model. Moreover, bugs/errors that are introduced by changes in small subparts of the system may propagate to the whole organization through the uniquely shared model.
- System modeling must be split among several teams and only loosely coupled tasks can be faced with separate teams.
- The need to move to a microservice-based architecture makes the bottleneck of a unique database more unacceptable.
- As the system grows, we need to communicate with more domain experts, each speaking a different language and each with a different view of that data model. Thus, we need to translate our unique model's properties and operations to/from more languages to be able to communicate with them.
- As the system grows, it becomes more inefficient to deal with records with several hundreds/thousands of fields. Such inefficiencies originate in database engines that inefficiently handle big records with several fields (memory fragmentation, problems with too many related indices, and so on). However, the main inefficiencies take place in ORMs and business layers that are forced to handle these big records in their update operations. In fact, while query operations usually require just a few fields that have been retrieved from the storage engine, updates and business processing involve the whole entity.

- As the traffic in the data storage subsystem grows, we need read and update/write parallelism in all the data operations. As we discussed in Chapter 7, *How to Choose Your Data Storage in the Cloud*, while read parallelism is easily achieved with data replication, write parallelism requires sharding, and it is difficult to shard a uniquely monolithic and tightly connected data model.

These issues are the reason for DDD's success in the last few years since they were characterized by more complex software systems that became the backbone of entire organizations. DDD's basic principles will be discussed in detail in the next section.

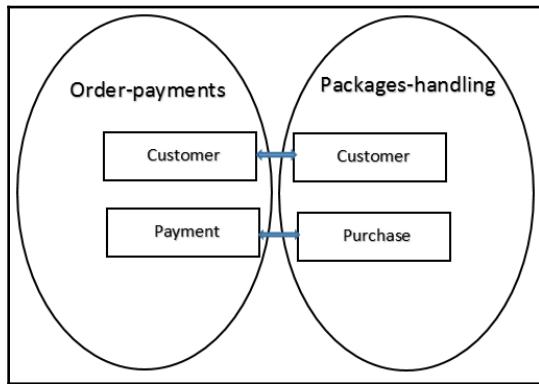
Understanding domain-driven design

Domain-driven design is about the construction of a unique domain model and keeps all the views as separate models. Thus, the whole application domain is split into smaller domains, each with a separate model. These separate domains are called **Bounded Contexts**. Each domain is characterized by the language spoken by the experts and used to name all the domain concepts and operations. Thus, each domain defines a common language used by both the expert and the development team called **Ubiquitous Language**. Translations are not needed anymore, and if the development team uses interfaces as a base for its code, the domain expert is able to understand and validate them since all the operations and properties are expressed in the same language that's used by the expert.

Here, we're getting rid of a cumbersome unique abstract model, but now we have several separated models that we need to relate somehow. DDD proposes that it will handle all of these separated models, that is, all the Bounded Contexts, as follows:

- We need to add Bounded Context boundaries whenever the meaning of the language terms change. For instance, in the WWTravelClub use case, order-payment and packages-handling belong to different Bounded Contexts because they give a different meaning to the word **customer**.
- We need to explicitly represent relations among bounded contexts. Different development teams may work on different bounded contexts, but each team must have a clear picture of the relationship between the Bounded Context it is working on and all the other models. For this reason, such relationships are represented in a unique document that's shared with every team.
- We need to keep all the bounded contexts aligned with continuous integration. Meetings are organized and simplified system prototypes are built in order to verify that all the Bounded Contexts are evolving coherently, that is, that all the Bounded Contexts can be integrated into the desired application behavior.

The following diagram shows how the WWTravelClub example that we discussed in the previous section changes with the adoption of DDD:



There is a relationship between the customer entities of both Bounded Contexts, while the Purchase entity of the packages-handling Bounded Context is related to the payments. Identifying entities that map to each other in the various Bounded Contexts is the first step of formally defining the interfaces that represent all the possible communications among the contexts.

For instance, from the preceding diagram, we know that payments are done after purchases, and so we can deduce that the payment-orders Bounded Context must have an operation to create a payment for a specific customer. In this domain, new customers are created if they don't already exist. The payment creation operation is triggered immediately after purchase. Since several more operations are triggered after an item is purchased, we can implement all the communication related to a purchase event with the Publisher/Subscriber pattern we explained in Chapter 9, *Design Patterns and .NET Core Implementation*. These are known as **domain events** in DDD. Using events to implement communications between Bounded Contexts is very common since it helps keep Bounded Contexts loosely coupled.

Once an instance of either an event or an operation that's been defined in the Bounded Context's interface crosses the context boundary, it is immediately translated into the Ubiquitous Language of the receiving context. It is important that this translation is performed before the input data starts interacting with the other domain entities to avoid the Ubiquitous Language of the other domain becoming contaminated by extra-context terms.



Each Bounded Context implementation must contain a Data Model Layer completely expressed in terms of the Bounded Context Ubiquitous Language (class and interface names and property and method names), with no contamination from other Bounded Contexts Ubiquitous Languages, and without contamination from programming technical stuff. This is necessary to ensure good communication with domain experts and to ensure that domain rules are translated correctly into code so that they can be easily validated by domain experts.

When there is a strong mismatch between the communication language and the target Ubiquitous Language, an anti-corruption layer is added to the receiving Bounded Context boundary. The only purpose of this anti-corruption layer is to perform a language translation.

The document that contains a representation of all the Bounded Contexts, along with the Bounded Context's mutual relationships and interface definitions, is called a **Context Mapping**. The relationships among contexts contain organizational constraints that specify the kind of cooperation that's required among the team that works on the different Bounded Contexts. Such relationships don't constrain the Bounded Context interfaces but do affect the way they may evolve during the software CI/CD cycle. They represent patterns of team cooperation. The most common patterns are as follows:

- **Partner:** This is the original pattern suggested by Eric Evans. The idea is that the two teams have a mutual dependency on each other for delivery. In other terms, they decide together and, if needed, change the Bounded Context's mutual communication specifications during the software CI/CD cycle.
- **Customer/supplier development teams:** In this case, a team acts as a customer and the other acts as a supplier. Both teams define the interface of the customer side of the Bounded Context and some automated acceptance tests to validate it. After that, the supplier can work independently. This pattern works when the customer's Bounded Context is the only active part that invokes the interface methods that are exposed by the other Bounded Context. This is adequate for the interaction between the order-payments and the packages-handling contexts, where order-payments acts as a supplier since its functions are subordinate to the needs of packages-handling. When this pattern can be applied, it decouples the two Bounded Contexts completely.

- **Conformist:** This is similar to the customer/supplier, but in this case, the customer side accepts an interface that's been imposed by the supplier side with no negotiation stage. This pattern offers no advantages to the other patterns, but sometimes we are forced into the situation depicted by the pattern since either the supplier's Bounded Context is implemented in a preexisting product that can't be configured/modified too much or because it is a legacy subsystem that we don't want to modify.

It is worth pointing out that the separation in Bounded Contexts is only efficacious if the resulting Bounded Contexts are loosely coupled; otherwise, the reduction of complexity that's obtained by breaking a whole system into subparts would be overcome by the complexity of the coordination and communication processes. However, if Bounded Contexts are defined with the *language criterion*, that is, Bounded Context boundaries are added whenever the *Ubiquitous Language* changes, this should actually be the case. In fact, different languages may arise just as a result of a loose interaction between organization subparts since the more subparts there are for an organization to interact with, the more they end up using a common language.

Moreover, all human organizations can grow by evolving into loosely coupled subparts for the same reason complex software systems may be implemented just for the cooperation of loosely coupled submodules: this is the only way humans are able to cope with complexity. From this, we can conclude that complex organizations/artificial systems can always be decomposed into loosely coupled subparts. We just need to understand *how*.

Beyond the basic principles we've mentioned so far, DDD furnishes a few basic primitives to describe each Bounded Context, as well as some implementation patterns. While Bounded Context primitives are an integral part of DDD, these patterns are just useful heuristics we can use in our implementation, so their usage in some or in all Bounded Contexts is not obligatory once we opt for DDD adoption.

In the next section, we will describe primitives and patterns.

Entities and value objects

DDD entities represent domain objects that have a well-defined identity, as well as all the operations that are defined on them. They don't differ too much from the entities of other, more classical approaches. Also, DDD entities are the starting point of the storage layer design. The main difference is that DDD stresses their object-oriented nature more, while other approaches use them mainly as *records* whose properties can be written/updated without too many constraints. DDD, on the other hand, forces strong SOLID principles on them to ensure that only certain information is encapsulated inside of them and that only certain information is accessible from outside of them, which operations are allowed on them, and which business-level validation criteria apply to them.

In other words, DDD entities are richer than the entities of record-based approaches. In other approaches, operations that manipulate entities are defined outside of them in classes that represent business and/or domain operations. In DDD, these operations are moved to the entity definitions as their class methods. The reason for this is that they offer better modularity and keep related chunks of software in the same place so that they can be maintained and tested easily.

For the same reason, business validation rules are moved inside of DDD entities. DDD entity validation rules are business-level rules, so they must not be confused with database integrity rules or with user-input validation rules. They contribute to the way entities represent domain objects by encoding the constraints the represented objects must obey. In .NET Core, business validation can be carried out with one of the following techniques listed:

- Calling the validation methods in all the class methods that modify the entity
- Hooking the validation methods to all the property setters
- Decorating the class and/or its properties with a custom validation attribute and then invoking the `TryValidateObject` static method of the `System.ComponentModel.DataAnnotations.Validator` class on the entity each time it is modified

Once detected, validation errors must be handled somehow, that is, the current operation must be aborted and the error must be reported to an appropriate error handler. The simplest way to handle validation errors is by throwing an exception. This way, both purposes are easily achieved and we can choose where to intercept and handle them. Unfortunately, as we discussed in the *Performance issues that need to be considered while programming in C#* section of Chapter 2, *Functional and Nonfunctional Requirements*, exceptions imply big performance penalties, so, often, different options are considered. Handling errors in the normal flow of control would break modularity by spreading the code that's needed to handle the error all over the stack of methods that caused the error, with a never-ending set of conditions all over that code. Therefore, more sophisticated options are needed.

A good alternative to exceptions is to notify the error handler of any errors that are defined in the Dependency Injection engine. Being scoped, the same service instance is returned while each request is being processed so that the handler that controls the execution of the whole call stack can inspect possible errors when the flow of control returns to it and can handle them appropriately. Unfortunately, this sophisticated technique can't abort the operation's execution immediately or return it to the controlling handler. Therefore, the developer is forced to add adequate control conditions to prevent the continuation of the operation. This is why exceptions are recommended for this scenario, notwithstanding their performance issues.



Business-level validation must not be confused with input validation, which will be discussed in more detail in Chapter 13, *Presenting ASP.NET Core MVC*, since the two types of validation have different and complementary purposes. While business-level validation rules encode domain rules, input validation enforces the format of every single input (string length, email and URL correct formats, and so on), ensures that all the necessary input has been provided, enforces the execution of the chosen user-machine interaction protocols, and provides fast and immediate feedback that drives the user to interact with the system.

Since DDD entities must have a well-defined identity, they must have properties that act as primary keys. It is common to override the `Object.Equal` method of all the DDD entities in such a way that two objects are considered equal whenever they have the same primary keys. This is easily achieved by letting all the entities inherit from an abstract `Entity` class, as shown in the following code:

```
public abstract class Entity<K> : IEntity<K>
{
    where K : IEqualityComparer<K>

    public virtual K Id { get; protected set; }
    public bool IsTransient()
```

```
{  
    return Object.Equals(Id, default(K));  
}  
public override bool Equals(object obj)  
{  
    if (obj == null || !(obj is Entity<K>))  
        return false;  
    if (Object.ReferenceEquals(this, obj))  
        return true;  
    if (this.GetType() != obj.GetType())  
        return false;  
    Entity<K> item = (Entity<K>)obj;  
    if (item.IsTransient() || this.IsTransient())  
        return false;  
    else  
        return Object.Equals(item.Id, Id);  
}  
int? _requestedHashCode;  
public override int GetHashCode()  
{  
    if (!IsTransient())  
    {  
        if (!_requestedHashCode.HasValue)  
            _requestedHashCode = this.Id.GetHashCode() ^ 31;  
        return _requestedHashCode.Value;  
    }  
    else  
        return base.GetHashCode();  
}  
public static bool operator ==(Entity<K> left, Entity<K> right)  
{  
    if (Object.Equals(left, null))  
        return (Object.Equals(right, null));  
    else  
        return left.Equals(right);  
}  
public static bool operator !=(Entity<K> left, Entity<K> right)  
{  
    return !(left == right);  
}
```

It is worth pointing out that, once we've redefined the `Object.Equal` method in the `Entity` class, we can also override with the `==` and `!=` operators.

The `IsTransient` predicate returns `true` whenever the entity has been recently created and hasn't been recorded in the permanent storage, so its primary key is still undefined.



In .NET, it is good practice that, whenever you override the `Object.Equal` method of a class, you also override its `Object.GetHashCode` method so that class instances can be efficiently stored in data structures such as dictionaries and sets. That's why the `Entity` class overrides it.

It is also worth implementing an `IEntity<K>` interface that defines all the properties/methods of `Entity<K>`. This interface is useful whenever we need to hide data classes behind interfaces.

Value objects, on the other hand, represent complex types that can't be encoded with numbers or strings. Therefore, they have no identity and no principal keys. They have no operations defined on them and are immutable; that is, once they've been created, all their fields can be read but cannot be modified. For this reason, they are usually encoded with classes whose properties have protected/private setters. Two value objects are considered equal when all their independent properties are equal (some properties are not independent since they just show data that's been encoded by other properties in a different way, as is the case for the ticks of `DateTime` and its representation of the date and time fields).

Typical value objects include costs represented as a number and a currency symbol, locations represented as longitude and latitude, addresses, contact information, and so on. When the interface of the storage engine is Entity Framework, which we analyzed in Chapter 6, *Interacting with Data in C# - Entity Framework Core*, and Chapter 7, *How to Choose Your Data Storage in the Cloud*, value objects are connected with the entity that uses them through the `OwnsMany` and `OwnsOne` relationships. In fact, such relationships also accept classes with no principal keys defined on them.

When the storage engine is a NoSQL database, value objects are stored inside the record of the entities that use them. On the other hand, in the case of relational databases, they can either be implemented with separated tables whose principal keys are handled automatically by Entity Framework and are hidden from the developer (no property is declared as a principal key) or, in the case of `OwnsOne`, they are flattened and added to the table associated with the entity that uses them.

Using SOLID principles to map your domains

In the following subsections, we will describe some of the patterns that are commonly used with DDD. Some of them can be adopted in all projects, while others can only be used for certain Bounded Contexts. The general idea is that the business layer is split into two layers:

- Application layer
- Domain layer

Here, the domain layer is based on the Ubiquitous Language and manipulates DDD entities and value objects. DDD entities and value objects are defined in this domain layer. The whole business layer communicates with the data layer that's implemented with Entity Framework through interfaces that are defined in the domain layer but are implemented in the data layer. Data that's passed/returned by these interface methods are known as DDD entities (the representation of queries and their results). The domain layer has no direct reference to the library that implements the data layer, but the connection between domain layer interfaces and their data layer implementations is performed in the dependency injection engine of the application layer. From this, we can understand the following:

- The data layer has a reference to the domain layer since it must implement its interfaces and must create DDD entities and value objects that are defined in the domain layer.
- The application layer has references to the domain and data layers, but references to the data layer types only appear in the dependency engine, where they are associated with the respective interfaces that were defined in the domain layer.

Thus, the domain layer contains the representation of the domain objects, the methods to use on them, validation constraints, and its relationship with various entities. To increase modularity and decoupling, communication among entities is usually encoded with events, that is, with a publisher/subscriber pattern. This means entity updates can trigger events that have been hooked to business operations.

This layered architecture allows us to change the whole data layer without affecting the domain layer, which only depends on the domain specifications and language and doesn't depend on the technical details of how the data is handled.

The application layer contains the definitions of all the operations that may potentially affect several entities and the definitions of all the queries that are needed by the applications. Both business operations and queries use the interfaces defined in the domain layer to interact with the data layer. However, while business operations manipulate and exchange entities with these interfaces, queries send query specifications and receive generic DTOs from them. Business operations are invoked either by other layers (typically the presentation layer) or by communication with the application layer. Business operations may also be hooked to events that are triggered when entities are modified by other operations.

Thus, the application layer operates on the interfaces defined in the domain layer instead of interacting directly with their data layer implementations, which means that the application layer is decoupled from the data layer. More specifically, data layer objects are only mentioned in the dependency injection engine definitions. All the other application layer components refer to the interfaces that are defined in the domain layers, and the dependency injection engine injects the appropriate implementations.

The application layer communicates with other application components through one or more of the following patterns:

- It exposes business operations and queries on a communication endpoint, such as an HTTP Web API (see Chapter 12, *Applying Service-Oriented Architectures with .NET Core*). In this case, the presentation layer may connect to this endpoint or to other endpoints that, in turn, take information from this and other endpoints. Application components that collect information from several endpoints and expose them in a unique endpoint are called gateways. They may be either custom or general-purpose, such as Ocelot.
- It is referenced as a library by an application that directly implements the presentation layer, such as an ASP.NET Core MVC Web application.
- It doesn't expose all the information through endpoints and communicates some of the data it processes/creates to other application components that, in turn, expose endpoints.

Before we describe these patterns, we need to understand the concept of aggregates.

Aggregates

So far, we have talked about entities as the *units* that are processed by a DDD-based business layer. However, several entities can be manipulated and made into single entities. An example of this is a purchase order and all of its items. In fact, it makes absolutely no sense to process a single order item independently of the order it belongs to. This happens because order-items are actually subparts of an order, not independent entities.

There is no transaction that may affect a single order-item without it affecting the order that the item is in. Imagine that two different people in the same company are trying to increase the total quantity of cement but one increases the quantity of type-1 cement (item 1) while the other increases the quantity of type-2 cement (item 2). If each item is processed as an independent entity, both quantities will be increased, which could cause an incoherent purchase order since the total quantity of cement would be increased twice.

On the other hand, if the whole order, along with all its order-items, is loaded and saved with every single transaction by both people, one of the two will overwrite the changes of the other one, so whoever makes the final change will have their requirements set. In a web application, it isn't possible to lock the purchase order for the whole time the user sees and modifies it, so an optimistic concurrency policy is used. For instance, it is enough to add a version number to each purchase order and to do the following:

1. Read the order without opening any transaction.
2. Before saving the modified order, we open a transaction and perform a second read.
3. If the version number of the newly retrieved order differs from the one of the order that was modified by the user, the operation is aborted because someone else modified the order that was shown to the user immediately after the first read. In this case, the user is informed of the problem and the newly retrieved order is shown to the user once more.
4. If the version number is unchanged, we increase the version number, proceed with the save, and commit the transaction.

A purchase order, along with all its subparts (its order-items), is called an **aggregate**, while the order entity is called the root of the aggregate. Aggregates always have roots since they are hierarchies of entities connected by *subparts* relations.

Since each aggregate represents a single complex entity, all the operations on it must be exposed by a unique interface. Therefore, the aggregate root usually represents the whole aggregate, and all the operations on the aggregate are defined as methods of the root entity.

When the *aggregate* pattern is used, the units of information that are transferred between the business layer and the data layer are called *aggregates*, queries, and query results. Thus, aggregates replace single entities.

What about the WWTravelClub location and packages entities we looked at in Chapter 6, *Interacting with Data in C# - Entity Framework Core*, and Chapter 7, *How to Choose Your Data Storage in the Cloud*? Are packages part of the unique aggregates that are rooted in their associated locations? No! In fact, locations are rarely updated and changes that are made to a package have no influence on its location and on the other packages associated with the same location.

The repository and Unit of Work patterns

The repository pattern is how we implement the interface between the domain data layer. Interfaces that are implemented by repositories are defined in the domain layer, while their implementations are defined in the data layer. The peculiarity of this way of implementing the interface with the data layer is its entity-centric nature, meaning that there should be a different repository for each root aggregate. Each repository contains all the save/creation operations that were performed on the associated aggregate, as well as all the query operations that were performed on the entities that compose the aggregate.

Since there are also transactions that can span several aggregates, usually, the repository pattern is applied with the *Unit of Work* pattern. The Unit of Work pattern states that each data layer interface (in our case, each repository) contains a reference to a *Unit of Work* interface that represents the identity of the current transaction. This means that several repositories with the same *Unit of Work* reference belong to the same transaction.

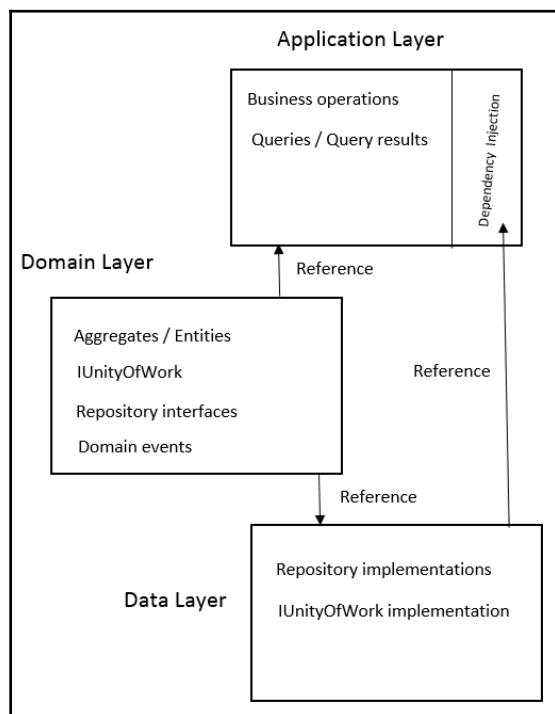
Both patterns can be implemented by defining some seed interfaces:

```
public interface IUnitOfWork
{
    Task<bool> SaveEntitiesAsync();
    Task StartAsync();
    Task CommitAsync();
    Task RollbackAsync();
}

public interface IRepository<T>: IRepository
{
    IUnitOfWork UnitOfWork { get; }
}
```

All the repository interfaces inherit from `IRepository<T>` and bind `T` to the aggregate root they are associated with, while *Unity of Work* simply implements `IUnitOfWork`. When using Entity Framework, `IUnitOfWork` is usually implemented with `DbContext`, which means that `SaveEntitiesAsync()` can perform other operations and then call the `DbContext SaveChangeAsync` method so that all the pending changes are saved with a single transaction. If a wider transaction that starts when some data is retrieved from the storage engine is needed, it must be started and committed/aborted by the application layer handler that takes care of the whole operation. `IRepository<T>` inherits from an empty `IRepository` interface to help automatic repository discovery. The GitHub repository associated with this book contains a `RepositoryExtensions` class whose `AddAllRepositories IServiceCollection` extension method automatically discovers all the repository implementations contained in an assembly and adds them to the dependency injection engine.

The following is a diagram of the data layer/domain layer/data layer architecture based on the repository and Unity of Work patterns:



The main advantage of avoiding direct references to repository implementations is that the various modules can be tested easily if we mock these interfaces.

DDD entities and Entity Framework Core

DDD requires entities to be defined in a way that is different from the way we defined entities in Chapter 6, *Interacting with Data in C# - Entity Framework Core*. In fact, Entity Framework entities are record-like lists of public properties with almost no methods, while DDD entities should have methods that encode domain logic, more sophisticated validation logic, and read-only properties. While further validation logic and methods can be added without breaking Entity Framework's operations, adding read-only properties that must not be mapped to database properties can create problems that must be handled adequately. Preventing properties from being mapped to the database is quite easy – all we need to do is decorate them with the `NotMapped` attribute.

The issues that read-only properties have are a little bit more complex and can be solved in three fundamental ways:

- Define the DDD entities as different classes and copy data to/from them when entities are returned/passed to repository methods. This is the easiest solution but it requires that you write some code so that you can convert the entities between the two formats. DDD entities are defined in the domain layer, while the EF entities continue being defined in the data layer.
- Let Entity Framework Core map fields to class private fields so that you can decide on how to expose them to properties by writing custom getters and/or setters. This can be done in the configuration code of the entity, as follows:

```
modelBuilder.Entity<MyEntity>()
    .Property("myPrivatefield");
```

The main disadvantage of this approach is that the field is provided as a string, which prevents any compile-time checks and also prevents automatic refactoring, thereby creating possible sources of bugs and maintainability issues. Moreover, we can't use data annotations to configure the property since the whole configuration must be performed with the fluent interface of the `OnModelCreating` method. In this case, entity definitions must be moved from the domain layer as prescribed by DDD.

- Hide the Entity Framework class with all its public properties behind an interface that, when needed, only exposes property getters. The interface is defined in the domain layer, while the entity continues being defined in the data layer.
In this case, the repository must expose a `Create` method that returns an implementation of the interface; otherwise, the higher layers won't be able to create new instances that can be added to the storage engine since interfaces can't be created with `new`.

For instance, suppose that we would like to define a DDD interface called `IDestination` for the `Destination` class defined in the *Defining DB Entities* subsection of Chapter 6, *Interacting with Data in C# – Entity Framework Core*, and suppose we would like to expose the `Id`, `Name`, and `Country` properties as read-only since once a destination is created they can't be modified anymore. Here, it is enough to let `Destination` implement `IDestination` and to define `Id`, `Name`, and `Country` as read-only in `IDestination`:

```
public interface IDestination
{
    int Id { get; }
    string Name { get; }
    string Country { get; }
    string Description { get; set; }
    ...
}
```

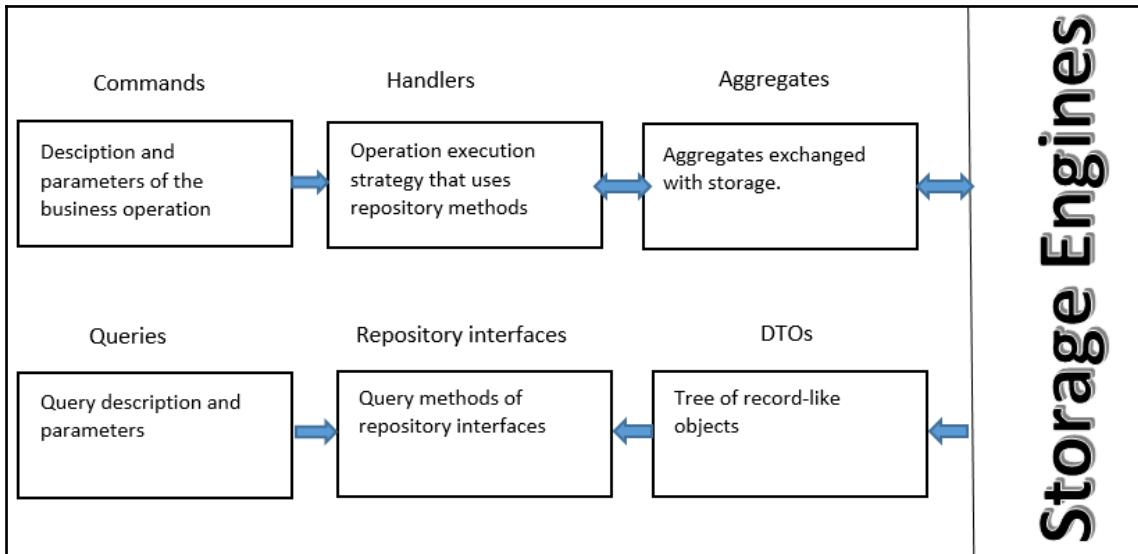
Now that we've discussed the basic patterns of DDD and how to adapt the Entity Framework for the needs of DDD, we can discuss more advanced DDD patterns. In the next subsection, we will introduce the CQRS pattern.

Command Query Responsibility Segregation (CQRS) pattern

In its more general form, the usage of this pattern is quite easy: use different structures to store and query data. Here, the requirements regarding how to store and update data differ from the requirements of queries. In the case of DDD, the unit of storage is the *aggregate*, so additions, deletions, and updation involve aggregates, while queries usually involve more or less complicated transformations of properties that have been taken from several aggregates.

Moreover, usually, we don't perform business operations on query results – we just use them to compute other data (averages, sums, and so on). Therefore, while updates require entities with full object-oriented semantics (methods, validation rules, encapsulated information, and so on), query results just need sets of property/values pairs, so DTOs with only public properties and no methods work well.

In its more common form, the pattern can be depicted as follows:



The main takeaway from this is that the extraction of query results don't need to pass through the construction of entities and aggregates, but the fields shown in the query must be extracted from the storage engine and projected into ad hoc DTOs. If queries are implemented with LINQ, we need to use the `Select` clause to project the necessary properties into DTOs:

```
ctx.MyTable.Where(...).Select(new MyDto{...}).ToList();
```

However, in more complex situations, the CQRS may be implemented in a stronger form. Namely, we can use different Bounded Contexts to store preprocessed query results. This approach is common when queries involve data stored in different Bounded Contexts that's handled by different distributed microservices.

In fact, the other option would be an aggregator microservice that queries all the necessary microservices in order to assemble each query result. However, recursive calls to other microservices to build an answer may result in unacceptable response times. Moreover, factoring out some preprocessing ensures better usage of the available resources. This pattern is implemented by sending changes caused by Bounded Context updates to all the microservices that need them for computing their preprocessed query results.

The usage of this stronger form of the CQRS pattern transforms usual local database transactions into complex time-consuming distributed transactions since a failure in a single query preprocessor microservice should invalidate the whole transaction. As we explained in *Chapter 5, Applying a Microservice Architecture to Your Enterprise Application*, implementing distributed transactions is usually unacceptable for performance reasons, so the common solution is to renounce to immediate overall coherent database and to accept that the overall database will eventually be coherent after each update. Transient failures can be solved with the retry policies that we analyzed in *Chapter 5, Applying a Microservice Architecture to Your Enterprise Application*, while permanent failures are handled by performing corrective actions on the already committed local transactions instead of pretending to implement an overall globally distributed transaction.

As we discussed in *Chapter 5, Applying a Microservice Architecture to Your Enterprise Application*, communication between microservices is often implemented with the publisher/subscriber pattern to improve microservice separation.

At this point, you may be asking the following question:

"Why do we need to keep the original data once we have all the preprocessed query results? We will never use them to answer queries!"

Some of the answers to this are as follows:

- They are the source of truth that we may need to recover from failures.
- We need them to compute new preprocessed results when we add new queries.
- We need them to process new updates. In fact, processing updates usually requires that some of the data is retrieved from the database, possibly shown to the user, and then modified. For instance, to modify an item in an existing purchase order, we need the whole order so that we can show it to the user and compute the changes so that we can forward it to other microservices. Moreover, whenever we modify or add data to the storage engine, we must verify the coherence of the overall database (unique key constraints, foreign key constraints, and so on).

In the next section, we will describe a common pattern that's used for handling operations that span several aggregates or several Bounded Contexts.

Command handlers and domain events

To keep aggregates separated, usually, interactions with other aggregates and other Bounded Contexts is done through events. It is good practice to store all the events when they are computed during each aggregate process instead of processing them immediately in order to avoid them interfering with the ongoing computation. This is easily achieved by adding the following code to the abstract Entity class defined in the *Entities and value objects* subsection of this chapter, as follows:

```
public List<IEventNotification> DomainEvents { get; private set; }
public void AddDomainEvent(IEventNotification evt)
{
    DomainEvents = DomainEvents ?? new List<IEventNotification>();
    DomainEvents.Add(evt);
}
public void RemoveDomainEvent(IEventNotification evt)
{
    DomainEvents?.Remove(evt);
}
```

Here, `IEventNotification` is an empty interface that's used to mark classes as events.

Event processing is usually performed immediately before changes are stored in the storage engine. Accordingly, a good place to perform event processing is in the `SaveEntitiesAsync()` method of each `IUnitOfWork` implementation (see the *The repository and Unit of Work patterns* subsection).

Subscriptions to an event, `T`, can be provided as an implementation of the `IEventHandler<T>` interface:

```
public interface IEventHandler<T>
    where T: IEventNotification
{
    Task HandleAsync(T ev);
}
```

Analogously, business operations can be described by the `command` object, which contains all the input data of the operation, while the code that implements the actual operation can be provided through the implementation of an `ICommandHandler<T>` interface:

```
public interface ICommandHandler<T>
    where T: ICommand
{
    Task HandleAsync(T command);
}
```

Here, `ICommand` is an empty interface that's used to mark classes as commands. `ICommandHandler<T>` and `IEventHandler<T>` are examples of the command pattern we described in Chapter 9, *Design Patterns and .NET Core Implementation*.

Each `ICommandHandler<T>` can be registered in the dependency injection engine so that classes that need to execute a command, `T`, can use `ICommandHandler<T>` in their constructor. This way, we decouple the abstract definition of a command (command class) from the way it is executed.

The same construction can't be applied to events, `T`, and their `IEventHandler<T>` because when an event is triggered, we need to retrieve several `IEventHandler<T>` and not just one. We need to do this since each event may have several subscriptions. However, a few lines of code can easily solve this difficulty. First, we need to define a class that hosts all the handlers for a given event type:

```
public class EventTrigger<T>
    where T: IEventNotification
{
    private IEnumerable<IEventHandler<T>> handlers;
    public EventTrigger(IEnumerable<IEventHandler<T>> handlers)
    {
        this.handlers = handlers;
    }
    public async Task Trigger(T ev)
    {
        foreach (var handler in handlers)
            await handler.HandleAsync(ev);
    }
}
```

The idea is that each class that needs to trigger event `T` requires an `EventTrigger<T>` and then passes the event to be triggered to its `Trigger` method, which, in turn, invokes all the handlers.

Then, we need to register `EventTrigger<T>` in the dependency injection engine. A good idea is to define the dependency injection extensions that we can invoke to declare each event, as follows:

```
service.AddEventHandler<MyEventType, MyHandlerType>()
```

This `AddEventHandler` extension must automatically produce a DI definition for `EventTrigger<T>` and must process all the handlers that are declared with `AddEventHandler` for each type, `T`.

The following extension class does this for us:

```
public static class EventDIExtensions
{
    private static IDictionary<Type, List<Type>> eventDictionary =
        new Dictionary<Type, List<Type>>();
    public static IServiceCollection AddEventHandler<T, H>
        (this IServiceCollection service)
        where T : IEventNotification
        where H : class, IEventHandler<T>
    {
        service.AddScoped<H>();
        List<Type> list = null;
        eventDictionary.TryGetValue(typeof(T), out list);
        if(list == null)
        {
            list = new List<Type>();
            eventDictionary.Add(typeof(T), list);
            service.AddScoped<EventTrigger<T>>(p =>
            {
                var handlers = new List<IEventHandler<T>>();
                foreach(var type in eventDictionary[typeof(T)])
                {
                    handlers.Add(p.GetService(type) as IEventHandler<T>);
                }
                return new EventTrigger<T>(handlers);
            });
        }
        list.Add(typeof(H));
        return service;
    }
    ...
}
```

The `H` types of all the handlers associated with each event, `T`, are recorded in a list contained in an entry of a dictionary indexed by the `T` type of the event. Then, each `H` is recorded in the dependency injection engine.

The first time an entry for an event, `T`, is added, the corresponding dictionary entry is created (a `List<Type>`) and the corresponding `EventTrigger<T>` is added to the dependency injection engine. The `EventTrigger<T>` instance is created by a function that's passed to `AddSingleton<EventTrigger<T>>`, which uses the dictionary entry for `T` to get all the handler types. Then, all the handler types are used to retrieve the instances for the dependency injection engine with `p.GetService(type)`. We can use this operation since all the handler types were registered in the dependency injection engine. Finally, the list of all the handlers is used to create the required instance of `EventTrigger<T>`.

When the program starts up, all the `ICommandHandler<T>` and `IEventHandler<T>` implementations can be retrieved with reflection and registered automatically. To help with automatic discovery, they inherit from `ICommandHandler` and `IEventHandler`, which are both empty interfaces. The `EventDIExtensions` class, which is available in this book's GitHub repository, contains methods for the automatic discovery and registration of command handlers and event handlers. The GitHub repository also contains an `IEventMediator` interface and its `EventMediator` interface, whose `TriggerEvents(IEnumerable<IEventNotification> events)` method retrieves all the handlers associated with the events it receives in its argument from the dependency injection engine and executes them. It is enough to have `IEventMediator` injected into a class so that it can trigger events. `EventDIExtensions` also contains an extension method that discovers all the queries that implement the empty `IQuery` interface and adds them to the dependency injection engine.

A more sophisticated implementation is given by the `MediatR` NuGet package. The previous subsection is dedicated to an extreme implementation of the CQRS pattern.

Event sourcing

Event sourcing is an extreme implementation of the stronger form of CQRS. It is useful when the original Bounded Context isn't used at all to retrieve information and just as a *source of truth* is used for new queries and for recovering from failures. In this case, instead of updating data, we simply add events that describe the operation that was performed: *deleted record Id 15*, changed the name to *John in Id 21*, and so on. These events are immediately sent to all the dependent Bounded Contexts, and in the case of failures and/or the addition of new queries, all we have to do is to reprocess some of them.

While all of the techniques we've described up until now can be used in every type of project if minor modifications are made, event sourcing requires a deep analysis to be performed before it can be adopted since, in several cases, it may create bigger problems than the ones it can solve. To get an idea of the problems it may cause when it's misused, imagine that we apply it to purchase orders that have been modified and validated by several users before being approved. Since purchase orders need to be retrieved before they're updated/validated, the purchase order's Bounded Context isn't used just as a *source of truth*, so event sourcing should not be applied to it. If this isn't the case, then we can apply event sourcing to it, in which case our code would be forced to rebuild the whole order from the recorded events each time the order is updated.

An example of its usage is the revenue logging system we described at the end of Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*. Single revenues are recorded with event sourcing and then sent to the microservice we described in Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*, which, in turn, uses them to preprocess future queries, that is, to compute daily revenues.

In the next section, we will learn how DDD can be applied to this book's WWTravelClub use case.

Use case – understanding the domains of the use case

From the requirements listed in the *Case study – WWTravelClub* section of Chapter 1, *Understanding the Importance of Software Architecture*, and for the analysis in the *Use case – where do I store data?* section of Chapter 7, *How to Choose Your Data Storage in the Cloud*, we know that the WWTravelClub system is composed of the following parts:

- Information about the available destinations and packages. We implemented the first prototype of this subsystem's data layer in Chapter 7, *How to Choose Your Data Storage in the Cloud*.
- Reservation/purchase orders subsystem.
- Communication with the experts/reviews subsystem.
- Payment subsystem. We briefly analyzed the features of this subsystem and its relationship with the reservation purchase subsystem at the beginning of the *Domain-driven design* section of this chapter.
- User accounts subsystem.
- Statistics reporting subsystem.

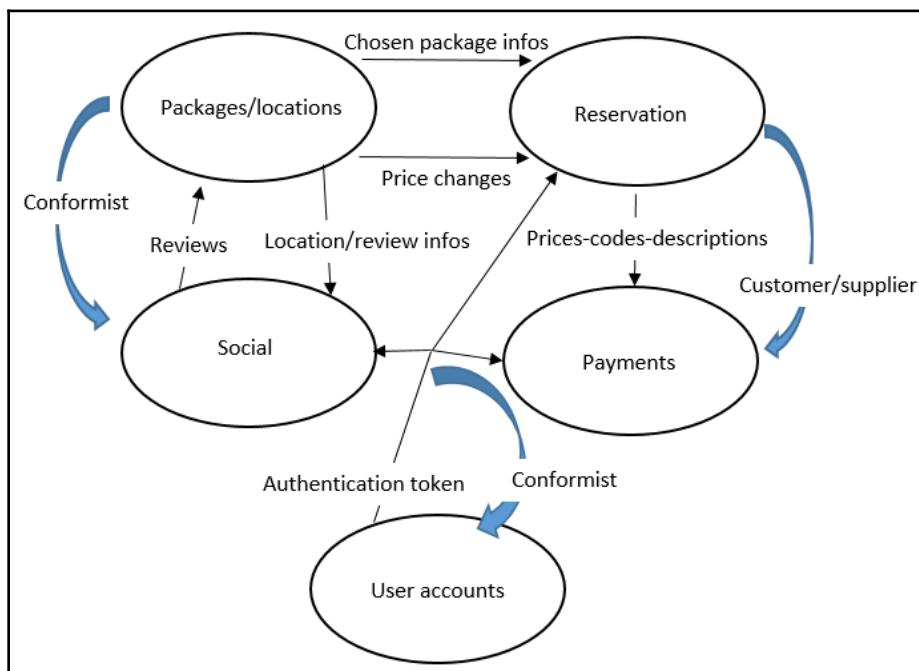
Do the preceding subsystems represent different *Bounded Contexts*? Can some subsystems be split into different Bounded Contexts? The answers to these questions are given by *the languages* that are spoken in each subsystem:

- The language that's spoken in subsystem 1 is the language of *travel agencies*. There is no concept of a customer; just of locations, packages, and their features.
- The language that's spoken in subsystem 2 is common to all service purchases, such as the available resources, reservations, and purchase orders. This is a separate Bounded Context.
- The language that's spoken in subsystem 3 has a lot in common with subsystem 1's language. However, there are also typical *social media* concepts, such as rating, chats, post sharing, media sharing, and so on. This subsystem can be split into two parts: a social media subsystem that has a new bounded context and an *available information* subsystem that is part of the Bounded Context of subsystem 1.
- As we pointed out in the *Domain-driven design* section, in subsystem 4, we speak the language of *banking*. This subsystem communicates with the reservation/purchase subsystem and executes tasks that are needed to carry out a purchase. From these observations, we can see that it is a different Bounded Context and has a customer/supplier relationship with the purchase/reservation system.
- Subsystem 5 is definitely a separate Bounded Context (as in almost all web applications). It has a relationship with all the Bounded Contexts that either have a concept of a user or a concept of a customer because the concept of user accounts always maps to these concepts. But how? Simple—the currently logged-in user is assumed to be the social media user of the social media Bounded Context, the customer of the reservation/purchase Bounded Context, and the payer of the payment Bounded Context.
- The query-only subsystem, that is, 6, speaks the language of analytics and statistics and differs a lot from the languages that are spoken in the other subsystems. However, it has a connection with almost all the Bounded Contexts since it takes all its input from them. The preceding constraints force us to adopt CQRS in its strong form, thereby considering it a query-only separated Bounded Context. We implemented a part of it in Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*, by using a microservice that conforms to a strong form of CQRS.

In conclusion, each of the listed subsystems defines a different Bounded Context, but part of the *communication with the experts/reviews subsystem* must be included in the *Information about available destinations and packages* Bounded Context.

As the analysis continues and a prototype is implemented, some Bounded Contexts may split and some others may be added, but it is fundamental to immediately start modeling the system and to immediately start analyzing the relations among the Bounded Contexts with the partial information we have since this will drive further investigations and will help us define the communication protocols and Ubiquitous Languages that are needed so that we can interact with the domain experts.

The following is a basic first sketch of the domain map:



For simplicity, we've omitted the **Statistics reporting** Bounded Context. Here, we're assuming that the **User accounts** and **Social** Bounded Contexts have a *conformist* relationship with all the other Bounded Contexts that communicate with them because they are implemented with already existing software, so all the other components must adapt to them.

As we mentioned previously, the relationship between **Reservation** and **Payments** is *customer/supplier* because *Payments* furnishes services that are used to execute the tasks of *Reservation*. All the other relationships are classed as *Partners*. The various concepts of customer/user that most Bounded Contexts have are coordinated by the **User accounts** authorization token, which indirectly takes care of mapping these concepts between all the Bounded Contexts.

The *Packages/location* subsystem not only communicates the packages information that's needed for carrying out a reservation/purchase – it also takes care of informing pending purchase orders of possible price changes. Finally, we can see that social interactions are started from an existing review or location, thereby creating communication with the *Package/locations* Bounded Context.

Summary

In this chapter, we analyzed the main reasons for the adoption of domain-driven design and why and how it faces the needs of the market. Here, we described how to identify domains and how to coordinate the teams that work on different domains of the same application with domain maps. Then, we analyzed the way DDD represents data with entities, value objects and aggregates, furnishing advice, and code snippets so that we could implement them in practice.

We also covered some typical patterns that are used with DDD, that is, the repository and Unit of Work patterns, domain event patterns, CQRS, and event sourcing. Then, we learned how to implement them in practice. We also showed you how to implement domain events and the command pattern with decoupled handling so that we can add furnishing code snippets to real-world projects.

Finally, we used the principles of DDD in practice to define domains and to create the first sketch of a domain map for this book's WWTravelClub use case.

In the next chapter, you will learn how to maximize code reuse in your projects.

Questions

1. What furnishes the main hints so that we can discover domain boundaries?
2. What is the main tool that's used for coordinating the development of a separate Bounded Context?
3. Is it true that each entry that composes an aggregate communicates with the remainder of the system with its own methods?
4. Why is there a single aggregate root?
5. How many repositories can manage an aggregate?
6. How does a repository interact with the application layer?
7. Why is the Unit of Work pattern needed?
8. What are the reasons for the light form of CQRS? What about the reasons for its strongest form?
9. What is the main tool that allows us to couple commands/domain events with their handlers?
10. Is it true that event sourcing can be used to implement any Bounded Context?

Further reading

- More resources on domain-driven design can be found here: <https://domainlanguage.com/ddd/>
- A detailed discussion of CQRS design principles can be found here: <http://udidahan.com/2009/12/09/clarified-cqrs/>
- More information on MediatR can be found on MediatR's GitHub repository: <https://github.com/jbogard/MediatR>
- A good description of event sourcing, along with an example of it, can be seen in the following blog post by Martin Fowler: <https://martinfowler.com/eaaDev/EventSourcing.html>

11

Implementing Code Reusability in C# 8

Code reusability is one of the most important topics in software architecture. This chapter aims to discuss ways to enable code reuse and understand how .NET Standard goes in this direction to solve the problem of managing and maintaining a reusable library.

The following topics will be covered in this chapter:

- Understanding the principles of code reuse
- The advantages of working with .NET Standard
- Creating reusable libraries

Technical requirements

This chapter requires the following things:

- You need Visual Studio 2017 or the 2019 free community edition or better with all the database tools installed.
- A free Azure account: The *Creating an Azure Account* section in Chapter 1, *Understanding the Importance of Software Architecture*, explains how to create one.
- An Azure DevOps account: The *What is Azure DevOps?* section in Chapter 3, *Documenting Requirements with Azure DevOps*, explains how to create one.

You will find the sample code of this chapter at <https://github.com/PacktPublishing/Hands-On-Software-Architecture-with-CSharp-8/tree/master/ch11>.

Understanding the principles of code reusability

There is a single reason that you can always use to justify code reuse—you cannot spend your valuable time recreating the wheel if it is already running well in other scenarios. That is why most engineering domains are based on reusability principles. Think about the light switch you have in your house.

Can you imagine the number of applications that can be made with the same interface components? The fundamentals of code reuse are the same. Again, it is a matter of planning a good solution so part of it can be reused later.

In software engineering, code reuse is one of the techniques that can bring to software project a bunch of advantages, such as the following:

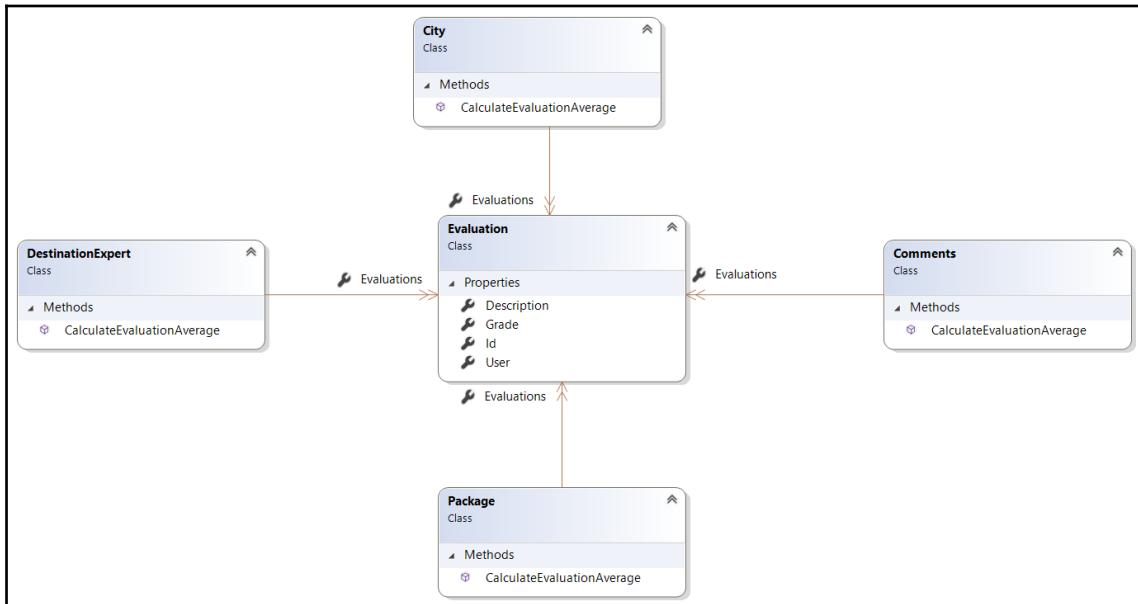
- There's confidence in the software, considering that the reused piece of code was already tested in another application
- There's better usage of software architects since they can be dedicated to solving this kind of problem
- There's the possibility of bringing to the project a pattern that's already accepted by the market
- Development speed goes up due to the already implemented components
- Maintenance is easier

These aspects indicate that code reuse shall be done whenever it is possible. It is your responsibility, as a software architect, to ensure the preceding advantages and, more than that, to incentivize your team to enable reuse in the software they are creating.

What is not code reuse?

The first thing you have to understand is that code reuse does not mean copying and pasting code from one class to another. Even if this code was written by another team or project, this does not indicate you are properly working with reusability principles. Let's imagine a scenario that we will find in this book's use case, the WWTravelClub evaluation.

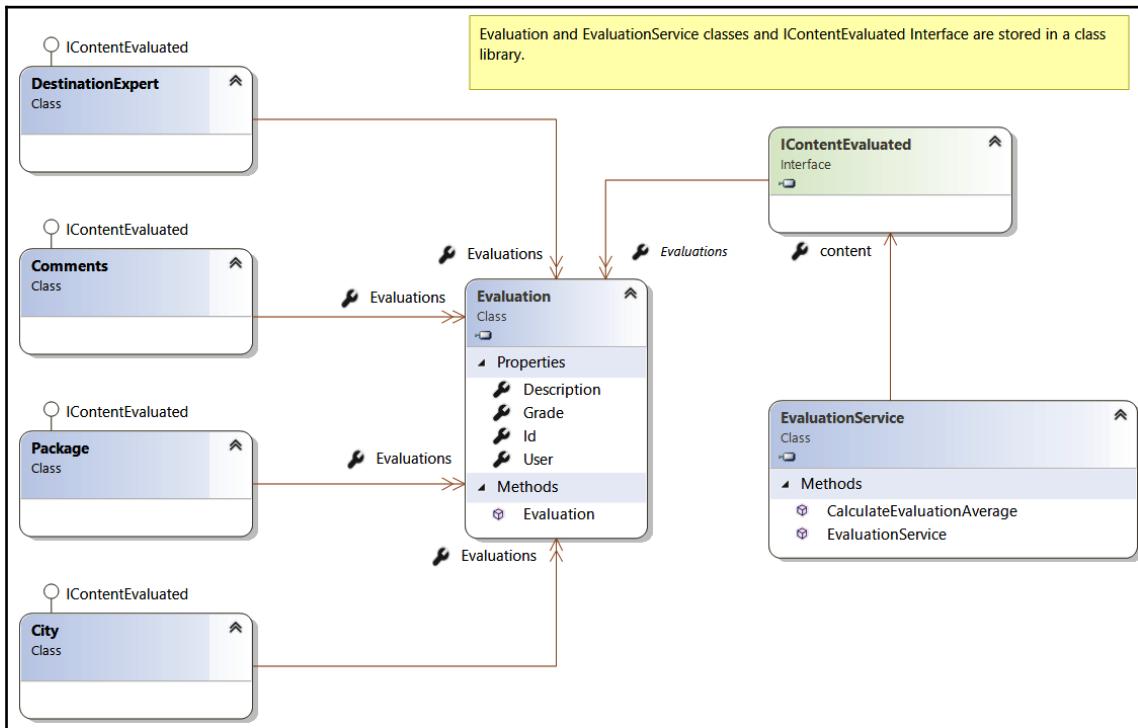
In the project scenario, you may want to evaluate different kinds of subjects, such as the **Package**, **Destination Expert**, **City**, **Comments**, and so on. The process for getting the evaluation average is the same, no matter which subject you are referring to. Due to this, you may want to *enable* reuse by copying and pasting the code for each evaluation. The (bad) result will be something like this:



In the preceding diagram, the process for calculating the evaluation average is decentralized, which means that the same code will be duplicated in different classes. This will cause a lot of trouble, especially if the same approach starts happening in other applications. For instance, if there is a new specification about how you have to calculate the average or if you just get a bug in the calculation formula, you will have to fix it in all instances of code.

What is code reuse?

The solution to the problem mentioned in the last section is quite simple. You have to analyze your code and select the parts of it that it would be a good idea to decouple from your application. The greatest reason why you should decouple it is related to how you are sure that this code can be reused in other parts of the application or even in another application:



The centralization of the code brings to you, as a software architect, a different responsibility for it. You will have to keep in mind that a bug or an incompatibility of this code can cause damage to many parts of the application or different applications. On the other hand, once you have this code tested and running, you will be able to propagate its usage with no worries. Besides, if you need to evolve the average calculation process, you will have to change the code in a single class.

It is worth mentioning that the more you use the same code, the cheaper this development will become. Cost needs to be mentioned because, in general, the conception of reusable software costs more in the beginning.

Inserting reusability into your development cycle

If you understood that reusability will take you to another level of code implementation, you should have been thinking about how to make this technique available in your development cycle. As a matter of fact, creating and maintaining a component library is not very easy, due to the responsibility you will have and the lack of good tools to support the search for existent components.

On the other hand, there are some things that you may consider implementing in your software development process every time you initiate a new development:

- **Use** already implemented components from your user library, selecting features in the software requirements specification that need them.
- **Identify** features in the software requirements specification that are candidates to be designed as library components.
- **Modify** the specification considering that these features will be developed using reusable components.
- **Design** the reusable components and be sure that they have the appropriate interfaces to be used in many projects.
- **Build** the project architecture with the new component library version.
- **Document** the component library version so every developer and team knows about it.

The *use-identify-modify-design-build* process is a technique that you may consider implement every time you need to enable software reuse. As soon as you have the components you need to write for this library, you will need to decide on the technology that will provide these components.

During the history of software development, there were many approaches for doing this; some of them are discussed in *Chapter 5, Applying a Microservice Architecture to Your Enterprise Application*, in the *Microservices as the evolution of the concept of module* section.

Using .NET Standard for code reuse

.NET has evolved a lot since its first version. This evolution is not only related to the number of commands and performance issues, but the supported platforms too. As discussed in Chapter 1, *Understanding the Importance of Software Architecture*, you can run C# .NET in billions of devices, even if they are running Linux, Android, macOS, or iOS. For this reason, .NET Standard was first announced together with .NET Core 1.0, but the breaking changes happened with .NET Standard 2.0, when .NET Framework 4.6, .NET Core, and Xamarin were compatible with it.

The key point is that .NET Standard is not only a kind of Visual Studio project. More than that, it is a formal specification available to all .NET implementations. As you can see in the following table, it covers everything from the .NET Framework to Unity (<https://github.com/dotnet/standard/tree/master/docs/versions>):

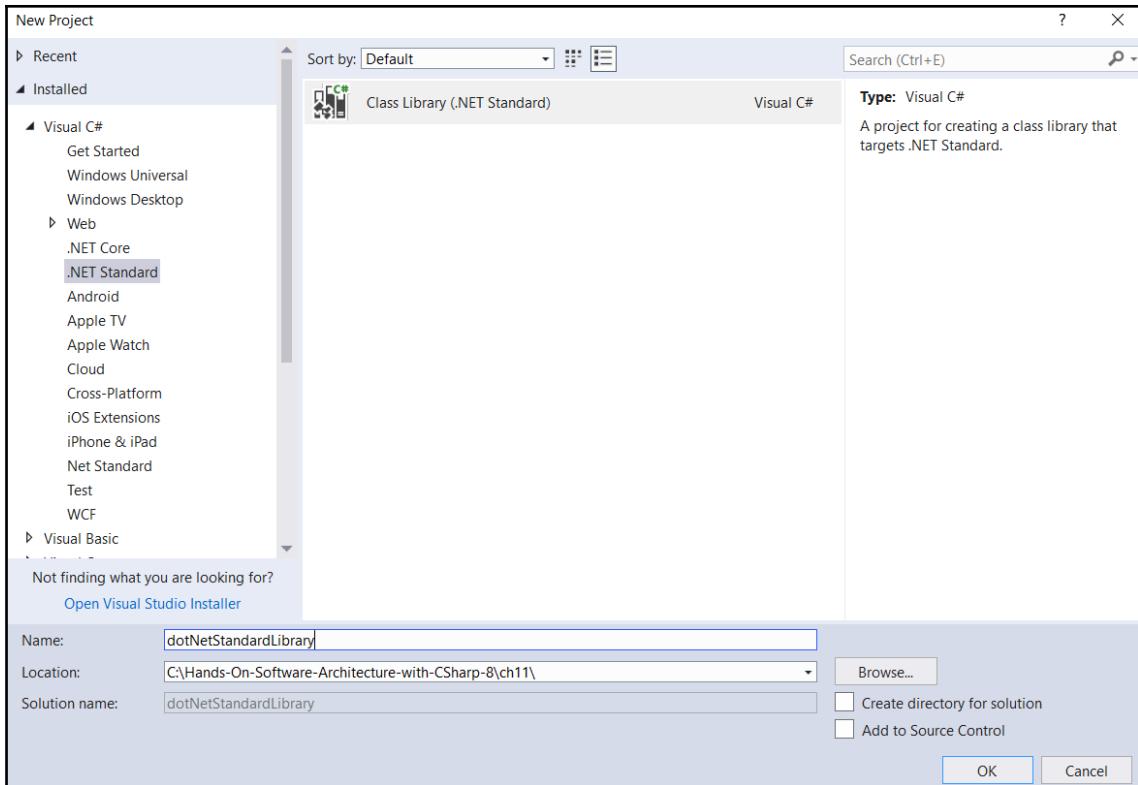
.NET Standard	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1 ¹	4.6.1 ¹	4.6.1 ¹	N/A ²
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4	6.2
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14	12.12
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8	5.12
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0	9.3
Unity	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	TBD
Universal Windows Platform	8.0	8.0	8.1	10.0	10.0	10.0.16299	10.0.16299	10.0.16299	TBD

The preceding table indicates that if you build a class library that's compatible with this standard, you will be able to reuse it in any of the platforms presented. Think about how fast your development process can become if you plan to do so in all your projects.

Obviously, some components are not included in .NET Standard, but its evolution is continuous. It is worth mentioning that Microsoft's official documentation indicates that *the higher the version, the more APIs are available to you*.

Creating a .NET Standard library

It is quite simple to create a class library compatible with .NET Standard. Basically, you need to choose the following project when creating the library:



Once you have concluded this part, you will notice that the only difference between a common class library and the one you created is the target framework defined in the project file:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>
</Project>
```

As soon as your project is loaded, you can start coding the classes that you intend to reuse. The advantage of building reusable classes using this approach is that you will be able to reuse the written code in all of the project types we checked before. On the other hand, you will find out that some APIs that are available in .NET Framework do not exist in this type of project. You can follow the future of the standard at <https://github.com/dotnet/standard/tree/master/docs/planning/netstandard-2.1>.

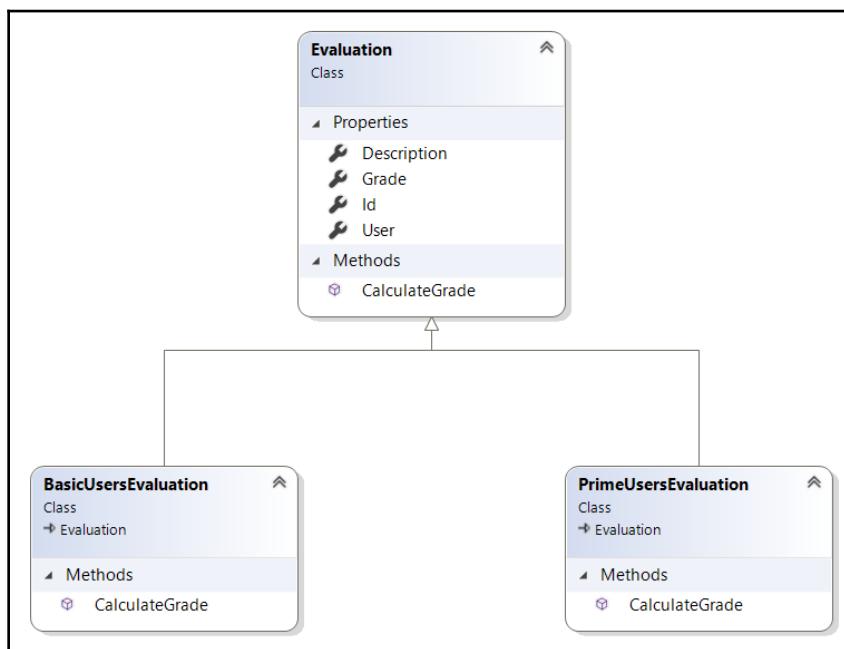
How does C# deal with code reuse?

There are many approaches where C# helps us deal with code reuse. The ability to build libraries, as we checked in the last topic, is one of them. The most important one is the fact that the language is object-oriented. Besides, it is worth mentioning the facilities that generics brought to C# language. This topic will discuss the last two mentioned.

Object-oriented analysis

The object-oriented analysis approach gives us the ability to reuse code in different ways, from the facility of inheritance to the changeability of polymorphism. Complete adoption of object-oriented programming, will let you implement abstraction and encapsulation too.

The following screenshot presents using the object-oriented approach to make reuse easier. As you can see, there are different ways to calculate the grades of an evaluation, considering you can be a basic or a prime user of the system:



There are two aspects to be analyzed as code reuse in this design. The first is that there's no need to declare the properties in each child class since inheritance is doing it for you.

The second is the opportunity we have to use polymorphism, enabling different behaviors for the same method:

```
public class PrimeUsersEvaluation : Evaluation
{
    /// <summary>
    /// The business rule implemented here indicates that grades that
    /// came from prime users have 20% of increase
    /// </summary>
    /// <returns>the final grade from a prime user</returns>
    public override double CalculateGrade()
    {
        return Grade * 1.2;
    }
}
```

You can check in the preceding code the usage of the polymorphism principle, where the calculation of evaluation for prime users will increase by 20%. Now, take a look at how easy it is to call different objects inherited by the same class. Since the collection content implements the same interface, `IContentEvaluated`, it can have basic and prime users too:

```
public class EvaluationService
{
    public IContentEvaluated content { get; set; }
    /// <summary>
    /// No matter the Evaluation, the calculation will always get
    /// values from the method CalculateGrade
    /// </summary>
    /// <returns>The average of the grade from Evaluations</returns>
    public double CalculateEvaluationAverage()
    {
        var count = 0;
        double evaluationGrade = 0;
        foreach (var evaluation in content.Evaluations)
        {
            evaluationGrade += evaluation.CalculateGrade();
            count++;
        }
        return evaluationGrade/count;
    }
}
```

Object-oriented adoption can be considered mandatory when using C#. However, more specific usage will need study and practice. You, as a software architect, shall always incentivize your team to study object-oriented analysis. The more they have good abstraction abilities, the easier code reuse will become.

Generics

Generics were introduced in C# in version 2.0, and it is definitely considered an approach that increases code reuse. It also maximizes type safety and performance.

The basic principle of generics is that you can define in an interface, class, method, property, event, delegate, or even a placeholder that will be replaced with a specific type at a later time when one of the preceding entities will be used. The opportunity you have with this feature is incredible since you can use the same code to run different versions of the type, generically.

The following code is a modification of `EvaluationService`, which was presented in the last section. The idea here is to enable the generalization of the service, giving the opportunity to define the goal of evaluation since its creation:

```
public class EvaluationService<T> where T: IContentEvaluated
```

This declaration indicates that any class that implements the `IContentEvaluated` interface can be used for this service. Besides, the service will be responsible for creating the evaluated content.

The following code implements exactly the evaluated content created since the construction of the service. This code uses `System.Reflection` and the generic definition from the class:

```
public EvaluationService()
{
    var name = GetTypeOfEvaluation();
    content = (T)Assembly.GetExecutingAssembly().CreateInstance(name);
}
```

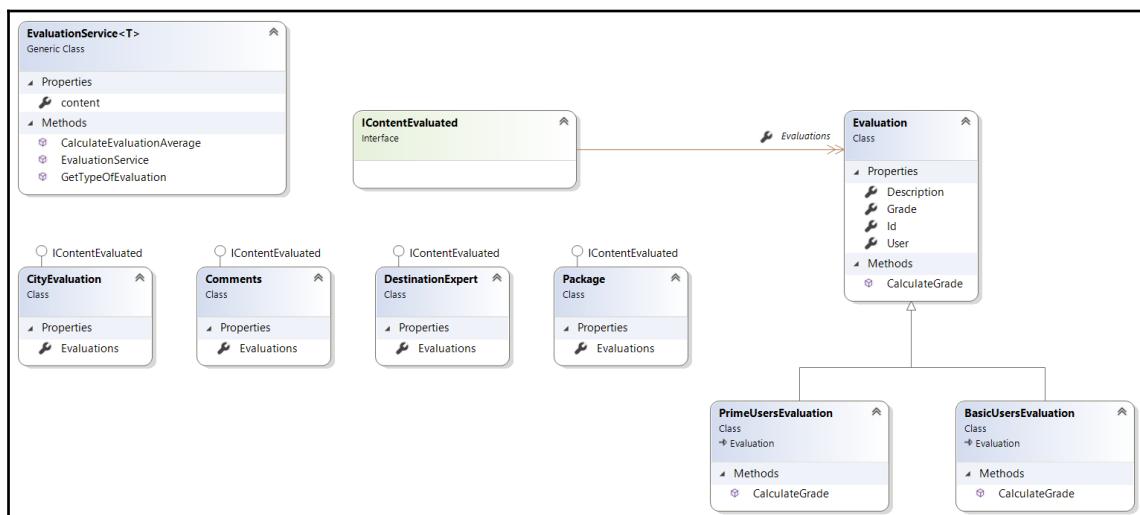
It is worth mentioning that this code will work because all of the classes are in the same assembly. The result of this modification can be checked in the instance creation of the service:

```
var service = new EvaluationService<CityEvaluation>();
```

The good news is that now you have a generic service that will automatically instantiate the list object with the evaluations of the content you need. It worth mentioning that generics obviously will need more time dedicated to the first project construction. However, after the design is done, you will have a good, fast, and easy-to-maintain code. This is what we call reuse!

Use case – reusing code as a fast track to deliver good and safe software

The final design of the solution for evaluating content for WWTravelClub can be checked as follows. This approach consists of the usage of many topics that were discussed in this chapter. First, all of the code is placed in a .NET Standard class library. This means that you can add this code to different types of solutions, such as .NET Core web apps and Xamarin apps for the Android and iOS platforms:



This design makes use of object-oriented principles such as inheritance, so you do not need to write properties and methods more than once that can be used in many classes; and polymorphism, so that you can change the behavior of the code without changing the name of the method.

To finish, the design abstracts the idea of the content by introducing generics as a tool that can facilitate the manipulation of similar classes, such as the ones we have in WWTravelClub to evaluate contents regarding cities, comments, destination experts, and travel packages.

The big difference between a team that incentivizes code reuse and one that does not is the velocity of delivering good software to end users. Of course, beginning this approach is not easy, but rest assured that you will get good results after some time working with it.

Summary

This chapter aimed to help you understand the advantages of code reuse. It also gave you an idea about what is not properly reused code. This chapter also presented approaches for reusing code.

Considering that technology without process does not take you anywhere, a process was presented to enable code reuse. This process is related to using already finished components from your library; identifying features in the software requirements specification that are candidates to be designed as library components; modifying the specification considering these features; designing the reusable components; and building the project architecture with the new component library version.

To finish, this chapter presented .NET Standard libraries as an approach to reuse code for different C# platforms, reinforced the principles of object-oriented programming as a way to reuse code, and presented generics as a sophisticated implementation to simplify the treatment of objects with the same characteristics. In the next chapter, we will be seeing how to apply **service-oriented architecture (SOA)** with .NET Core.

It is worth mentioning that SOA is considered a way to implement code reuse in sophisticated environments.

Questions

1. Can copy-and-paste be considered code reuse? What are the impacts of this approach?
2. How can you make use of code reuse without copying and pasting code?
3. Is there a process that can help code reuse?
4. What is the difference between .NET Standard and .NET Core?
5. What are the advantages of creating a .NET Standard library?
6. How does object-oriented analysis help with code reuse?
7. How do generics help with code reuse?

Further reading

These are some books and websites where you will find more information about this chapter:

- *Clean Architecture: A Craftsman's Guide to Software Structure and Design* by Martin, Robert C. Pearson Education, 2018.
- *Design Patterns: Elements of Reusable Object-Oriented Software* by Eric Gamma [et al.] Addison-Wesley, 1994.
- *Design Principles and Design Patterns* by Robert C. Martin, 2000.
- <https://devblogs.microsoft.com/dotnet/introducing-net-standard/>
- <https://www.packtpub.com/application-development/net-standard-20-cookbook>
- <https://github.com/dotnet/standard/blob/master/docs/versions.md>
- <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/generics/>

12

Applying Service-Oriented Architectures with .NET Core

The term **Service-Oriented Architecture (SOA)** refers to a modular architecture where interaction between system components is achieved through communication. SOA allows applications from different organizations to exchange data and transactions automatically and allows organizations to offer services on the internet.

Moreover, as we discussed in the *Microservices as the evolution of the concept of modules* section of Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*, communication-based interaction solves binary compatibility and version mismatch problems that inevitably appear in complex systems made up of modules that share the same address space. Moreover, with SOA, you don't need to deploy different copies of the same component in the various systems/subsystem that use it – each component only needs to be deployed in just one place. This can be a single server, a cluster located in a single data center, or a geographically distributed cluster. Here, each version of your component is deployed just once, and the server/cluster logic automatically creates all the necessary replicas, thus simplifying the overall **Continuous Integration / Continuous Delivery (CI/CD)** cycle.

As long as a newer version conforms to the communication interface that's declared to the clients, no incompatibilities can occur. On the other hand, with DLLs/packages, when the same interface is maintained, incompatibilities may arise because of possible version mismatches in terms of the dependencies of other DLLs/packages that the library module might have in common with its clients.

Organizing clusters/networks of cooperating services was discussed in Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*. In this chapter, we will mainly focus on the communication interface offered by each service. More specifically, we will discuss the following topics:

- Understanding the principles of the SOA approach
- How does .NET Core deal with SOA?
- Use case – exposing WWTravelClub packages

By the end of this chapter, you will know how to publicly expose data from the WWTravelClub book use case through an ASP.NET Core service.

Technical requirements

This chapter requires Visual Studio 2017 or 2019 free Community Edition or better with all the database tools installed.

All the concepts in this chapter will be clarified with practical examples based on this book's WWTravelClub book use case. You will find the code for this chapter at <https://github.com/PacktPublishing/Hands-On-Software-Architecture-with-CSharp-8>.

Understanding the principles of the SOA approach

Like classes in an object-oriented architecture, services are implementations of interfaces that, in turn, come from system functional specifications. Therefore, the first step in a *service* design is the definition of its abstract interface. During this stage, you define all the service operations as interface methods that operate on the types of your favorite language (C#, Java, C++, JavaScript, and so on) and decide which operations to implement with synchronous communication and which ones to implement with asynchronous communication.

The interfaces that are defined in this initial stage won't necessarily be used in the actual service implementation, and are just useful design tools. Once we've decided on the architecture of the services, these interfaces are usually redefined so that we can adapt them to the peculiarity of the architecture.

It is worth pointing out that SOA messages must keep the same kind of semantics as method calls/answers; that is, the reaction to a message must not depend on any previously received messages. Here, the messages must be independent of each other, and the service *must not remember* any previously received messages.

For instance, if the purpose of messages is to create a new database entry, this semantic must not change with the context of other messages, and the way the database entry is created must depend on the content of the current message and not on other previously received messages. As a consequence, a client can't create sessions and can't log in to a service, perform some operations, and then log out. An authentication token must be repeated in each message.

The reasons for this constraint are modularity, testability, and maintainability. In fact, a session-based service would be very hard to test and modify due to the interactions that are *hidden* in the session data.

Once you've decided on the interface that's going to be implemented by a service, you must decide which communication stack/ SOA architecture to adopt. The communication stack must be part of some official or de facto standard to ensure your service's interoperability. Interoperability is the main constraint prescribed by SOA: services must offer a communication interface that does not depend on the specific communication library used, on the implementation language, or on the deployment platform.

Once you've decided on the communication stack/architecture, you need to adapt your previous interfaces to the architecture's peculiarities (see the *REST web services* subsection of this chapter for more details). Then, you must translate these interfaces into the chosen communication language. This means that you have to map all the programming language types into types that are available in the chosen communication language.

The actual translation of data is usually performed automatically by the SOA libraries that are used by your development environment. However, some configuration might be needed and, in any case, we must be aware of how our programming language types are transformed before each communication. For instance, some numeric types might be transformed into types with less precision or with different ranges of values.

The interoperability constraint can be interpreted in a lighter form in the case of microservices that aren't accessible outside of their clusters, since they need to communicate with other microservices that belong to the same cluster. In this case, this means that the communication stack might be platform-specific so that it can increase performance, but it must be a de facto standard to avoid compatibility problems with other microservices that might be added to the cluster as the application evolves.

We've spoken of the *communication stack* and not of the *communication protocol* because SOA communication standards usually define the format of the message's content and provide different possibilities for the specific protocol that's used to embed those messages. For instance, the SOAP protocol just defines an XML-based format for the various kind of messages, but SOAP messages can be conveyed by various protocols. Usually, the most common protocol that's used for SOAP is HTTP, but you may decide to jump to the HTTP level and send SOAP messages directly over TCP/IP for better performance.

The choice of communication stack you should adopt depends on several factors:

- **Compatibility constraints:** If your service must be publicly available on the internet to business clients, then you must conform to the most common choices, which means using either SOAP over HTTP or JSON REST services. The most common choices are different if your clients aren't business clients but **Internet of Things (IoT)** clients. Also, within IoT, the protocols that are used in different application areas can be different. For instance, marine vehicle status data isn't typically exchanged with *Signal K*.
- **Development/deployment platform:** Not all communication stacks are available on all development frameworks and on all deployment platforms. For instance, .NET remoting, which we used in the code example at the end of Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*, is specific to .NET and Azure Service Fabric. Luckily, all the most common communication stacks that are used in public business services, such as SOAP and JSON-based REST communication, are available in all the main development/deployment platforms.
- **Performance:** If your system is not exposed to the outside world and is a private part of your microservice cluster, performance considerations have a higher priority. That's why, in the Service Fabric example at the end of Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*, we used .NET remoting as an internal communication stack. It is worth pointing out that, with *private* services, you need to be concerned about interoperability and refrain from using custom communication stacks. .NET remoting is not an official standard, but it is acceptable because it is a kind of de facto standard for internal communication within Azure Service Fabric.
- **Availability of tools and knowledge in your team:** Having knowledge and knowing about the availability of tools in your team/organization has an important weight when it comes to choosing between acceptable communication stacks. However, this kind of constraint always has less priority than compatibility constraints since it makes no sense to conceive a system that is easy to implement for your team but that almost nobody can use.

- **Flexibility versus available features:** Some communication solutions, while less complete, offer a higher degree of flexibility, while other solutions, while being more complete, offer less flexibility. The need for flexibility started a movement from SOAP-based services to the more flexible REST services in the last few years. This point will be discussed in more detail when we describe SOAP and REST services in the remainder of this section.
- **Service description:** When services must be exposed on the internet, client applications need a publicly available description of the service specifications in order to design their communication clients. Some communication stacks include languages and conventions to describe service specifications. Formal service specifications that are exposed this way can be processed so that they automatically create communication clients. SOAP goes further and allows service discoverability by means of a public XML-based directory containing information about the tasks each web service can carry out.

Once you've chosen the communication stack you wish to use, you must use the tools that are available in your development environment to implement the service in a way that conforms to the chosen communication stack. Sometimes, communication stack compliance is automatically ensured by the development tools, but sometimes, it may require some development effort. For instance, in the .NET world, the compliance of SOAP services is automatically ensured by development tools if you use WCF, while the compliance of REST services falls under the developer's responsibility.

Some of the fundamental features of SOA solutions are as follows:

- **Authentication:** Allows the client to authenticate to access service operations.
- **Authorization:** Handles the client's permissions.
- **Security:** This is how communication is kept safe, that is, how to prevent unauthorized systems from reading and/or modifying the content of the communication. Typically, encryption prevents both unauthorized modifications and reading, while electronic signature algorithms prevent just modifications.
- **Exceptions:** Returns exceptions to the client.
- **Message reliability:** Ensures that messages reliably reach their destination in case of possible infrastructure faults.

Though sometimes desirable, the following features aren't always necessary:

- **Distributed transactions:** The capability to handle distributed transactions, thus undoing all the changes you've made whenever the distributed transactions fail or are aborted

- **Support for the Publisher/Subscriber pattern:** If and how events and notifications are supported
- **Addressing:** If and how references to other services and or service/methods are supported
- **Routing:** If and how messages can be routed through a network of services

The remainder of this section is dedicated to describing SOAP and REST services since they are the de facto standard for business services that are exposed outside of their clusters/servers. For performance reasons, microservices use other protocols such as .NET Remoting and AMQP for inter-cluster communication. The usage of .NET Remoting was discussed in Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*, while links on AMQP are given in the *Further reading* section.

SOAP web services

The **Simple Object Access Protocol (SOAP)** allows both one-way messages and answer/response messages. Communication can be both synchronous and asynchronous, but, if the underlying protocol is synchronous, such as in the case of HTTP, the sender receives an acknowledgment saying that the message was received (but not necessarily processed). When asynchronous communication is used, the sender must listen for incoming communications. Often, asynchronous communication is implemented with the subscriber/publisher pattern that we described in Chapter 9, *Design Patterns and .NET Core Implementation*.

Messages are represented as XML documents called **envelopes**. Each envelope contains a header, a body, and a fault element. The body is where the actual content of the message is placed. The fault element contains possible errors, so it's the way exceptions are exchanged when communication occurs. Finally, the header contains any auxiliary information that enriches the protocol but does not contain domain data. For example, the header may contain an authentication token, and/or a signature if the message is signed.

The underlying protocol that's used to send the XML envelopes is usually HTTP, but the SOAP specification allows any protocol, so we can use TCP/IP or SMTP directly. As a matter of fact, the more diffused underlying protocol is HTTP, so, if you don't have a good reason to choose another protocol, you should use HTTP in order to maximize the service's interoperability.

SOAP specifications contain the basics of message exchange, while other auxiliary features are described in separate specification documents called `WS-*` and are usually handled by adding extra information in the SOAP header. `WS-*` specifications handle all the fundamental and desirable features of SOA we listed previously. For instance, `WS-Security` takes care of security, including authentication, authorization, and encryption/signatures; `WS-Eventing` and `WS-Notification` are two alternative ways of implementing the publisher/subscriber pattern; `WS-ReliableMessaging` is concerned with the reliable delivery of messages in case of possible faults, and `WS-Transaction` is concerned with distributed transactions.

The preceding `WS-*` specifications are in no way exhaustive but are the more relevant and supported features. In fact, actual implementations in various environments (such as Java and .NET) furnish the more relevant `WS-*` services, but no implementation supports all the `WS-*` specifications.

All the XML documents/document parts involved in the SOAP protocol are formally defined in XSD documents, which are special XML documents whose content provides a description of XML structures. Also, all your custom data structures (classes and interfaces in an object-oriented language) must be translated into XSD if they are going to be part of a SOAP envelope.

Each XSD specification has an associated namespace that identifies the specification and a physical location where it can be found. Both the namespace and the physical location are URIs. The location URI doesn't need to be publicly accessible if the web service is accessible just from within an intranet.

The whole definition of a service is an XSD specification that may contain references to other namespaces, that is, to other XSD documents. In a few words, all the messages of a SOAP communication must be defined in an XSD specification. Then, a server and a client can communicate if they refer to the same XSD specifications. This means, for instance, that you need to create a new XSD specification each time you add another field to a message. After that, you need to update all the XSD files that reference the old message definition to the new message definition by creating a new version of them. In turn, these modifications require the creation of other versions for other XSD files, and so on. Therefore, simple modifications that maintain compatibility with the previous behavior (clients could simply ignore the field that was added) may cause an exponential chain of version changes.

In the last few years, the difficulty in handling modifications, along with the complexity of handling the configuration of all the `ws-*` specifications and performance problems, caused a gradual move toward the simpler REST services that we will describe in the upcoming sections. This move started with services that were called from JavaScript due to the difficulty of implementing complete SOAP clients that were able to run efficiently in a web browser. Moreover, the complex SOAP machinery was oversized for the simple needs of the typical clients running in a browser and may have caused a complete waste of development time.

Around 2018, services aimed at non-JavaScript clients started a massive move toward REST services, and nowadays the preferred choice is REST services, with SOAP being used either for compatibility with legacy systems or when features that aren't supported by REST services are needed. A typical application area that continues to prefer to SOAP system is that of payment/banking systems because these systems need transactional support that is offered by the `WS-Transaction` SOAP specification. There is no equivalent in the REST services world.

REST web services

REST services were initially conceived to avoid the complex machinery of SOAP in simple cases such as calls to a service from the JavaScript code of a web page. Then, they gradually became the preferred choice for complex systems. REST services use HTTP to exchange data in JSON or, less commonly, in XML format. In a few words, they replace the SOAP body with the HTTP body, the SOAP header with the HTTP header, and the HTTP response code replaces the fault element and furnishes further auxiliary information on the operation that was performed.

The main reason for the success of REST services is that HTTP already offers most of SOAP's features natively, which means we can avoid building a SOAP level on top of HTTP. Moreover, the whole HTTP machinery is simpler than SOAP: simpler to program, simpler to configure, and simpler to implement efficiently.

Moreover, REST services impose fewer constraints on the clients. In particular, type compatibility between servers and clients conforms to the more flexible JavaScript type compatibility model because JSON is a subset of JavaScript. Moreover, when XML is used in place of JSON, it maintains the same JavaScript type compatibility rules. No XML namespaces need to be specified.

When using JSON and XML, if the server adds some more fields to the response while keeping the same semantic of all the other fields compatible with the previous client, they can simply ignore the new fields. Accordingly, changes that are made to a REST service definition only need to be propagated to previous clients in case of breaking changes that cause an actual incompatible behavior in the server.

Moreover, it is likely that changes are self-limited and do not result in an exponential chain of changes because type compatibility does not require the reference to a specific type to be defined in a unique shared place and simply requires that the shape of types is compatible.

Let's clarify the REST service's type compatibility rules with an example. Let's imagine that several services use a `Person` object that contains `Name`, `Surname`, and `Address` string fields:

```
{  
    Name: string,  
    Surname: string,  
    Address: string  
}
```

Type compatibility is ensured if the service and client refer to different copies of the preceding definition. It is also acceptable for the client to use a definition with fewer fields, since it can simply ignore all the other fields:

```
{  
    Name: string,  
    Surname: string,  
}
```

Now, let's say that a service, `S1`, that handles a `Persons` database, replaces the `Address` string with a complex object:

```
{  
    Name: string,  
    Surname: string,  
    Address:  
    {  
        Country: string,  
        Town: string  
        Location: string  
    }  
}
```

Now, let's say that a service, S2, takes `Persons` from S1 and adds it to the responses it returns on some of its methods. After the breaking change of S1, it can adapt its communication client that calls S1 to the new format. Then, it can convert the new `Person` format into the older one before using `Persons` in its responses. This way, S2 avoids propagating the breaking change of S1.

In general, basing type compatibility on the object shape (tree of nested properties), instead of a reference to the same formal type definition, increases flexibility and modifiability. The price we pay for this increased flexibility is that type compatibility can't be computed automatically by comparing the formal definition of server and client interfaces. In fact, in absence of a univocal specification, each time a new version of the service is released, the developer must verify that the semantics of all the fields that the client and server have in common remain unchanged from the previous version. The basic idea behind REST services is to give up the severity checks and complex protocols for greater flexibility and simplicity, while SOAP does exactly the opposite.

The REST services manifesto states that REST uses native HTTP features to implement all the required service features. So, for instance, authentication will be performed directly with the `HTTP Authorization` field, encryption will be achieved with `HTTPS`, exceptions will be handled with an `HTTP` error status code, and routing and reliable messaging will be handled by the machinery the `HTTP` protocol relies on. Addressing is achieved by using URLs to refer to services, their methods, and other resources.

There is no native support for asynchronous communication since `HTTP` is a synchronous protocol. There's also no native support for the Publisher/Subscriber pattern, but two services can interact with the Publisher/Subscriber pattern by each exposing an endpoint to the other. More specifically, the first service exposes a subscription endpoint, while the second one exposes an endpoint where it receives its notifications, which are authorized through a common secret that's exchanged during the subscription. This pattern is quite common. GitHub also allows us to send our REST services to repository events.

REST services offer no easy options when it comes to implementing distributed transactions, which is why payment/banking systems still prefer SOAP. Luckily, most application areas don't need the strong form of consistency that's ensured by distributed transactions. For them, lighter forms of consistency, such as *eventual consistency*, are enough and are preferred for performance reasons. Please refer to Chapter 7, *How to Choose Your Data Storage in the Cloud*, for a discussion on the various types of consistencies.

The REST manifesto not only prescribes the usage of the predefined solutions that are already available in HTTP but also the usage of a WEB-like semantic. More specifically, all the service operations must be conceived as CRUD operations on resources that are identified by URLs (the same resource may be identified by several URLs). In fact, REST is an acronym for **Representational State Transfer**, meaning that each URL is the representation of some sort of object. Each kind of service request needs to adopt the appropriate HTTP verb, as follows:

- **GET** (Read operation): The URL represents the resource that is returned by the read operation. Thus, GET operations mimic pointer dereferencing. In the case of a successful operation, a 200 (ok) status code is returned.
- **POST** (Creation operation): The JSON/XML object that's contained in the request body is added as a new resource to the object represented by the operation URL. If the new resource is successfully created immediately, a 201 (created) status code is returned, along with a response object that depends on the operation. The response object should contain the most specific URL that identifies the created resource. If creation is deferred to a later time, a 202 (accepted) status code is returned.
- **PUT**: The JSON/XML object contained in the request body replaces the object referenced by the request URL. In the case of successful operation, a 200 (ok) status code is returned. This operation is idempotent, meaning that repeating the same request twice causes the same modification.
- **PATCH**: The JSON/XML object contained in the request body contains instructions on how to modify the object referenced by the request URL. This operation is not idempotent since the modification may be an increment of a numeric field. In the case of successful operation, a 200 (ok) status code is returned.
- **DELETE**: The resource referenced by the request URL is removed. In the case of successful operation, a 200 (ok) status code is returned.

If the resource has been moved from the request URL to another URL, a redirect code is returned:

- 301 (moved permanently), plus the new URL where we can find the resource
- 307 (moved temporarily), plus the new URL where we can find the resource

If the operation fails, a status code that depends on the kind of failure is returned. Some examples of failures codes are as follows:

- 400 (bad request): The request that was sent to the server is ill-formed.
- 404 (not found): When the request URL doesn't refer to any known object.
- 405 (method not allowed): When the request verb is not supported by the resource referenced by the URL.
- 401 (unauthorized): The operation requires authentication, but the client has not furnished any valid authorization header.
- 403 (forbidden): The client is correctly authenticated but has no right to perform the operation.

The preceding list of status codes is not exhaustive. References to an exhaustive list will be provided in the *Further reading* section.

It is fundamental to point out that POST/PUT/PATCH/DELETE operations may have – and usually have – side effects on other resources. Otherwise, it would be impossible to code operations that act simultaneously on several resources.

In other words, the HTTP verb must conform with the operation that's performed on the resource and referenced by the request URL, but the operation might affect other resources. The same operation might be performed with a different HTTP verb on one of the other involved resources. It is the developer's responsibility to choose which way to perform the same operation in order to implement it in the service interface.

Thanks to the side effects of HTTP verbs, REST services are able to encode all of these operations as CRUD operations on resources represented by URLs.

Often, moving an existing service to REST requires us to split the various inputs between the request URL and the request body. More specifically, we extract the input fields that univocally define one of the objects involved in the method's execution and use them to create a URL that univocally identifies that object. Then, we decide on which HTTP verb to use based on the operation that's performed on the selected object. Finally, we place the remainder of the input in the request body.

If our services were designed with an object-oriented architecture focused on the business domain objects (such as DDD, as described in [Chapter 10, Understanding the Different Domains in Software Solutions](#)), the REST translation of all the service methods should be quite immediate, since services should already be organized around domain resources. Otherwise, moving to REST might require some service interface redefinitions.

The adoption of full REST semantics has the advantage that services can be extended with or without small modifications being made to the preexisting operation definitions. In fact, extensions should mainly manifest as additional properties of some objects and as additional resources URLs with some associated operations. Therefore, preexisting clients can simply ignore them.

Now, let's learn how methods can be expressed in the REST language with a simple example of an intra-bank money transfer. A bank account can be represented by an URL, as follows:

```
https://mybank.com/bankaccounts/{bank account number}
```

A transfer might be represented as a PATCH request whose body contains an object with properties representing the amount of money, time of transfer, description, and the account receiving the money. The operation modifies the account mentioned in the URL, but also the receiving account as a *side effect*. If the account has not enough money, a 403 (Forbidden) status code is returned, along with an object with all the error details (an error description, the available funds, and so on).

However, since all the bank operations are recorded in the account statement, the creation and addition of a new transfer object for a *bank account operations* collection associated with the bank account is a better way to represent the transfer. In this case, the URL might be something like the following:

```
https://mybank.com/bankaccounts/{bank account number}/operations
```

Here, the HTTP verb is POST since we are creating a new object. The body content is exactly the same and a 403 status code is returned in case there's a lack of funds.

Both representations of the transfer cause exactly the same changes in the database. Moreover, once the inputs are extracted from the different URLs and from the possibly different request bodies, the subsequent processing is exactly the same. In both cases, we have exactly the same inputs and the same processing – it's just the exterior appearance of the two requests that's different.

However, the introduction of the virtual *operations* collection allows us to extend the service with several more operations collection-specific methods. It is worth pointing out that the operations collection doesn't need to be connected with a database table or with any physical object: it lives in the world of URLs and creates a convenient way for us to model the transfer.

The increased usage of REST services leads to a description of REST service interfaces to be created, similar to the ones developed for SOAP. This standard is called **OpenAPI**. We will talk about this in the following subsection.

The OpenAPI standard

OpenAPI is a standard that's used for describing the REST API. It is currently version 3. The whole service is described by a JSON endpoint, that is, an endpoint that describes the service with a JSON object. This JSON object has a general section that applies to the whole service and contains the general features of the services, such as its version and description, as well as shared definitions.

Then, each service endpoint has a specific section that describes the endpoint URL or URL format (in case some inputs are included in the URL), all its inputs, all the possible output types and status codes, and all the authorization protocols. Each endpoint-specific section can reference the definitions contained in the general section.

A description of the OpenAPI syntax is out of the scope of this book, but references are provided in the *Further reading* section. Various development frameworks automatically generate OpenAPI documentation by processing the REST API code and further information is provided by the developer, so your team doesn't need to have in-depth knowledge of OpenAPI syntax.

The *How does .NET Core deal with SOA?* section explains how we can generate automatically OpenAPI documentation in ASP.NET Core REST API projects, while the use case at the end of this chapter provides a practical example of its usage.

We will end this subsection by talking about how to handle authentication and authorization in REST services.

REST services authorization and authentication

Since REST services are sessionless, when authentication is required, the client must send an authentication token in every single request. That token is usually placed in the HTTP authorization header, but this depends on the type of authentication protocol you're using. The simplest way to authenticate is through the explicit transmission of a shared secret. This can be done with the following code:

```
Authorization: Api-Key <string known by both server and client>
```

The shared secret is called an API key. Since, at the time of writing, there is no standard on how to send it, API keys can also be sent in other headers, as shown in the following code:

```
X-API-Key: <string known by both server and client>
```

Needless to say, API key-based authentication needs HTTPS to stop shared secrets from being stolen. API keys are very simple to use, but they do not convey information about user authorizations, so they can be adopted when the operations allowed by the client are quite standard and there are no complex authorization patterns. Moreover, when exchanged in requests, API keys are susceptible to being attacked on the server or client side.

Safer techniques use shared secrets that are valid for a long period of time, just by the user logging in. Then, the login returns a short-life token that is used as a shared secret in all the subsequent requests. When the short-life secret is going to expire, it can be renewed with a call to a renew endpoint.

The whole login logic is completely decoupled from the short-life token-based authorization logic. The login is usually based on login endpoints that receive long-term credentials and returns short-life tokens. Login credentials are either usual username-password pairs that are passed as input to the login method or other kinds of authorization tokens that are converted into short-life tokens that are served by the login endpoint. Login can also be achieved with various authentication protocols based on X.509 certificates.

The most widespread short-life token type is the so-called bearer token. Each bearer token encodes information about how long it lasts and a list of assertions, called claims, that can be used for authorization purposes. Bearer tokens are returned by either login operations or renewal operations. Their characteristic feature is that they are not tied to the client that receives them or to any other specific client.

No matter how a client gets a bearer token, this is all a client needs to be granted all the rights implied by its claims. It is enough to transfer a bearer token to another client to empower that client with all rights implied by all the bearer token claims, since no proof of identity is required by bearer token-based authorization.

Therefore, once a client gets a bearer token, it can delegate some operations to third parties by transferring its bearer token to them. Typically, when a bearer token must be used for delegation, during the login phase, the client specifies the claims to include in order to restrict what operations can be authorized by the token.

Compared to API key authentication, bearer token-based authentication is disciplined by standards. In particular, they must use the following Authorization header:

```
Authorization: Bearer <bearer token string>
```

Bearer tokens can be implemented in several ways. REST services typically use JWT tokens that are strung with a Base64URL encoding of JSON objects. More specifically, JWT creation starts with a JSON header, as well as a JSON payload. The JSON header specifies the kind of token and how it is signed, while the payload consists of a JSON object that contains all the claims as property/value pairs. The following is an example header:

```
{  
  "alg": "RS256",  
  "typ": "JWT"  
}
```

The following is an example payload:

```
{  
  "iss": "issuercdomain.com"  
  "sub": "example",  
  "aud": ["S1", "S2"],  
  "roles": [  
    "ADMIN",  
    "USER"  
  ],  
  "exp": 1512975450,  
  "iat": 1512968250230  
}
```

Then, the header and payload are BASE64URL-encoded and the corresponding string is concatenated, as follows:

```
<header base64 string>.<payload base64 string>
```

The preceding string is then signed with the algorithm specified in the header, which, in our example, is RSA +SHA256, and the signature string is concatenated with the original string as follows:

```
<header base64 string>.<payload base64 string>.<signature string>
```

The preceding code is the final bearer token string. A symmetric signature can be used instead of RSA, but, in this case, both the JWT issuer and all the services using it for authorization must share a common secret, while, with RSA, the private key of the JWT issuer doesn't need to be shared with anyone, since the signature can be verified with just the issuer public key.

Some payload properties are standard, such as the following:

- iss: Issuer of the JWT.
- aud: The audience, that is, the services and/or operations that can use the token for authorization. If a service doesn't see its identifier within this list, it should reject the token.
- sub: A string that identifies the *principal* (that is, the user) to which the JWT was issued.
- iat, exp, and nbf: These are for the time the JWT was issued, its expiration time, and, if set, the time after which the token is valid, respectively. All the times are expressed as a number of seconds from the 1st of January 1970 midnight UTC. Here, all the days are considered as having exactly 86,400 seconds in them.

Other claims may be defined as public if we represent them with a unique URI; otherwise, they are considered private to the issuer and to the services known to the issuer.

How does .NET Core deal with SOA?

.Net Core has excellent support for REST services through ASP.NET Core. In terms of SOAP services, classic .NET handles them with WCF technology. In WCF, service specifications are defined through .NET interfaces and the actual service code is supplied in classes that implement those interfaces.

Endpoints, underlying protocols (HTTP and TCP/IP), and any other features are defined in a configuration file. In turn, the configuration file can be edited with an easy to use configuration tool. Therefore, the developer is responsible for providing just the service behavior as a standard .NET class and for configuring all the service features in a declarative way. This way, the service configuration is completely decoupled from the actual service behavior and each service can be reconfigured so that it can be adapted to a different environment without the need to modify its code.

WCF technology has not been ported to .NET Core and there are no plans to perform a complete port of it. Instead, Microsoft is investing in gRPC, Google's open source technology.

The main reasons behind the decision to abandon WCF in .NET core are as follows:

- As we've already discussed, SOAP technology has been overtaken by REST technology in most application areas.
- WCF technology is strictly tied to Windows, so it would be very expensive to reimplement all its features from scratch in .NET Core. Since support for classic .NET will continue, users that need WCF can still rely on classic .NET.
- As a general strategy, with .NET Core, Microsoft prefers investing in open source technologies that can be shared with other competitors. That's why, instead of investing in WCF, Microsoft provided a gRPC implementation starting from .NET Core 3.0.

While .NET Core doesn't support SOAP technology, it does support SOAP clients. More specifically, it is quite easy to create a SOAP service proxy for an existing SOAP service in Visual Studio, starting from the 2017 version (please refer to Chapter 9, *Design Patterns and .NET Core Implementation*, for a discussion of what a proxy is and of the proxy pattern). In the case of services, a proxy is a class that implements the service interface and whose methods perform their job by calling the analogous methods of the remote service.

To create a service proxy, right-click on the *connected services* node in Visual Studio, go to **Solution Explorers**, and then select **Add connected service**. Then, in the form that appears, select **Microsoft WCF Service Reference Provider**. Here, you can specify the URL of the service (where the WSDL service description is contained), the namespace where you wish to add the proxy class, and much more. At the end of the wizard, Visual Studio automatically adds all the necessary NuGet packages and scaffolds the proxy class. This is enough to create an instance of this class and to call its methods so that we can interact with the remote SOAP service.

There are also third parties, such as NuGet packages that provide limited support for SOAP services, but at the moment, they aren't very useful, since such limited support does not include features that aren't available in REST services.

Starting from .NET Core SDK, Visual Studio 2019 supports the gRPC project template, which scaffolds both a gRPC server and a gRPC client. At the time of writing, gRPC is not a standard and just a Google open source project. However, if both Microsoft and Google continue investing in it, it might become a de facto standard. gRPC implements a remote procedure call pattern that offers both synchronous and asynchronous calls.

It is configured in a way that is similar to WCF and to .NET remoting, as we described at the end of Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*. That is, services are defined through interfaces and their code is provided in classes that implement those interfaces, while clients interact with those services through proxies that implement the same service interfaces.

gRPC is a good option for internal communications within a microservices cluster, especially if the cluster is not fully based on Service Fabric technology and can't rely on .NET remoting. Since there are gRPC libraries for all the main languages and development frameworks, it can be used in Kubernetes-based clusters, as well as in Service Fabric clusters that host Docker images that have been implemented in other frameworks.

gRPC is more efficient than the REST services protocol due to its more compact representation of data and it being easier to use, since everything to do with the protocol is taken care of by the development framework. However, at the time of writing, none of its features rely on well-established standards, so it can't be used for publicly exposed endpoints – it can only be used for intra-cluster communication. For this reason, we will not describe gRPC in detail, but the *Further reading* section of this chapter contains references to both gRPC in general and to its .NET Core implementation.

Using gRPC is super easy since Visual Studio's gRPC project template scaffolds everything so that the gRPC service and its clients are working. The developer just needs to define the application-specific C# service interface and a class that implements it.

The remainder of the section is dedicated to .NET Core support for REST services from both the server and client-side.

A short introduction to ASP.NET Core

ASP.NET Core applications are .NET Core applications based on the *Host* concept we described in the *Using generic hosts* subsection of Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*. The `program.cs` file of each ASP.NET application creates a Host, builds it, and runs it with the following code:

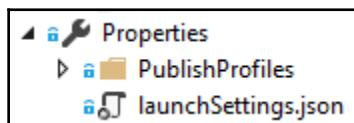
```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
```

```
.ConfigureWebHostDefaults(webBuilder =>
{
    webBuilder.UseStartup<Startup>();
});
```

CreatesDefaultBuilder sets up a standard Host, while ConfigureWebHostDefaults configures it so that it can handle an HTTP pipeline. More specifically, it does the following:

- It sets the ContentRootPath property of the IHostingEnvironment interface for the current directory.
- It loads the configuration information from appsettings.json and appsettings.[EnvironmentName].json. Once loaded, the configuration information contained in the JSON object properties can be mapped to .NET Object properties with the ASP.NET Core options framework. More specifically, appsettings.json and appsettings.[EnvironmentName].json are merged and the appsettings.[EnvironmentName] file's environment-specific information overrides the corresponding appsettings.json settings. EnvironmentName is taken from the ASPNETCORE_ENVIRONMENT environment variable. In turn, ASPNETCORE_ENVIRONMENT is defined in the Properties\launchSettings.json file when the application is run in Visual Studio. The following screenshot shows where you can find launchSettings.json in Visual Studio Solution Explorer:



In launchSettings.json, you can define several environments that can be selected with the dropdown next to Visual Studio's run button . By default, the **IIS Express** setting sets ASPNETCORE_ENVIRONMENT to Development. The following is a typical launchSettings.json file:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:2575",
```

```
        "sslPort": 44393
    },
},
"profiles": {
    "IIS Express": {
        "commandName": "IISExpress",
        "launchBrowser": true,
        "environmentVariables": {
            "ASPNETCORE_ENVIRONMENT": "Development"
        }
    },
    ...
    ...
}
}
```

The value to use for `ASPNETCORE_ENVIRONMENT` when the application is published can be added to the published XML file after it has been created by Visual Studio. This value

is `<EnvironmentName>Staging</EnvironmentName>`. It can be also specified in your Visual Studio ASP.NET Core project file (`.csproj`):

```
<PropertyGroup>
<EnvironmentName>Staging</EnvironmentName></PropertyGroup>.
```

- It configures Host logging so that it can write to the console and debug output. This setting can be changed with further configuration.
- It sets up/connects a web server to the ASP.NET Core pipeline.

When the application runs in Linux, the ASP.NET Core pipeline connects to the .NET Core Kestrel web server. Since Kestrel is a minimal web server, you are responsible for reverse proxying requests to it from a complete web server, such as Apache or Nginx, that adds features that Kestrel doesn't have. When the application runs in Windows, by default, `ConfigureWebHostDefaults` connects the ASP.NET Core pipeline directly to **Internet Information Services (IIS)**. However, you can also use Kestrel in Windows and you can reverse proxy IIS requests to Kestrel by changing the `AspNetCoreHostingModel` setting of your Visual Studio project file like so:

```
<PropertyGroup>
    ...
    <AspNetCoreHostingModel>OutOfProcess</AspNetCoreHostingModel>
</PropertyGroup>
```

`UseStartup<Startup>()` lets Host services (see the *Using generic hosts* subsection in Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*) and the definition of the ASP.NET Core pipeline be taken from the methods of the project's `Startup.cs` class. More specifically, services are defined in its `ConfigureServices(IServiceCollection services)` method, while the ASP.NET Core pipeline is defined in the `Configure` method. The following code shows the standard `Configure` method scaffolded with an API REST project:

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

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Each module in the pipeline is defined by an `app.Use<something>` method, which often accepts some options. Each module processes the requests and then either forwards the modified request to the next module in the pipeline or returns an HTTP response. When an HTTP response is returned, it is processed by all the previous modules in reverse order.

Modules are inserted in the pipeline in the order they are defined by the `app.Use<something>` method calls. The preceding code adds an error page if `ASPNETCORE_ENVIRONMENT` is `Development`; otherwise, `UseHsts` negotiates a security protocol with the client. Finally, `UseEndpoints` adds the MVC controllers that create the actual HTTP response. A complete description of the ASP.NET Core pipeline will be given in the *Understanding the presentation layers of web applications* section of Chapter 13, *Presenting ASP.NET Core MVC*.

In the next subsection, we will explain how the MVC framework lets you implement REST services.

Implementing REST services with ASP.NET Core

In the MVC framework, HTTP requests are processed by classes called Controllers. Each request is mapped to the call of a Controller public method. The selected controller and controller methods depend on the shape of the request path, and they are defined by routing rules, that, for the REST API, are usually provided through attributes associated with both the Controller class and its methods.

Controller methods that process HTTP requests are called action methods. When the controller and action methods are selected, the MVC framework creates a controller instance to serve the request. All the parameters of the controller constructors are resolved with dependency injection with types defined in the `ConfigureServices` method of the `Startup.cs` class.



Please refer to the *Using generic hosts* subsection of [Chapter 5, Applying a Microservice Architecture to Your Enterprise Application](#), for a description of how to use dependency injection with .NET Core Hosts, and to the *Dependency injection pattern* subsection of [Chapter 10, Understanding the Different Domains in Software Solutions](#), for a general discussion of dependency injection.

The following is a typical REST API controller and its controller method definitions:

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
    // GET api/values/5
    [HttpGet("{id}")]
    public ActionResult<string> Get(int id)
    {
        ...
    }
}
```

The `[ApiController]` attribute declares that the controller is a REST API controller. `[Route("api/[controller]")]` declares that the controller must be selected on paths that start with `api/<controller name>`. The controller name is the name of the controller class without the `Controller` postfix. Thus, in this case, we have `api/values`.

`[HttpGet("{id}")]` declares that the method must be invoked on GET requests of the `api/values/<id>` type, where `id` must be a number that's passed as an argument to the method invocation. This can be done with `Get(int id)`. There's also an `Http<verb>` attribute for each HTTP verb: `HttpPost` and `HttpPatch`.

We may also have another method defined like so:

```
[HttpGet]  
public ... Get()
```

This method is invoked on GET requests of the `api/values` type, that is, on GET requests without `id` after the controller name.

Several action methods can have the same name, but only one should be compatible with each request path; otherwise, an exception is thrown. In other words, routing rules and `Http<verb>` attributes must univocally define which controller and which of its action methods to select for each request.

By default, parameters are passed to the action methods of API controllers according to the following rules:

- Simple types (`integers`, `floats`, and `DatesTimes`) are taken from the request path if routing rules specify them as parameters, as in the case of the previous example's `[HttpGet("{id}")]` attribute. If they are not found in the routing rules, the MVC framework looks for query string parameters with the same name. Thus, for instance, if we replace `[HttpGet("{id}")]` with `[HttpGet]`, the MVC framework will look for something like `api/values?id=<an integer>`.
- Complex types are extracted from the request body by formatters. The right formatter is chosen according to the value of the request's `Content-Type` header. If no `Content-Type` header is specified, the JSON formatter is taken. The JSON formatter tries to parse the request body as a JSON object and then tries to transform this JSON object into an instance of the .NET Core complex type. If either the JSON extraction or the subsequent conversion fails, an exception is thrown. By default, just the JSON input formatter is supported, but you can also add an XML formatter that can be used when `Content-Type` specifies XML content. It is enough to add
the `Microsoft.AspNetCore.Mvc.Formatters.Xml` NuGet package and
replace `services.AddMvc()`
with `services.AddMvc().AddXmlSerializerFormatters()` in
the `ConfigureServices` method of `Startup.cs`.

You can customize the source that's used to fill an action method parameter by prefixing the parameter with an adequate attribute. The following code shows some examples of this:

```
...MyActionMethod(...[FromHeader] string myHeader....)
// x is taken from a request header named myHeader

...MyActionMethod(...[FromServices] MyType x....)
// x is filled with an instance of MyType through dependency injection
```

The return type of an `Action` method must be an `IAsyncResult` interface or a type that implements that interface. In turn, `IAsyncResult` has just the following method:

```
public Task ExecuteResultAsync (ActionResultContext context)
```

This method is called by the MVC framework at the right time to create the actual response and response headers. The `ActionContext` object, when passed to the method, contains the whole context of the HTTP request, which includes a request object with all the necessary information about the original HTTP requests (headers, body, and cookies), as well as a response object that collects all the pieces of the response that is being built.

You don't have to create an implementation of `IAsyncResult` manually, since `ControllerBase` already has methods to create `IAsyncResult` implementations so that all the necessary HTTP responses are generated. Some of these methods are as follows:

- `OK`: This returns a 200 status code, as well as an optional result object. It is used either as `return OK()` or as `return OK(myResult)`.
- `BadRequest`: This returns a 400 status code, as well as an optional request object.
- `Created(string uri, object o)`: This returns a 201 status code, as well as a result object and the URI of the created resource.
- `Accepted`: This returns a 202 status result, as well as an optional result object and resource URI.
- `Unauthorized`: This returns a 401 status result, as well as an optional result object.
- `Forbidden`: This returns a 403 status result, as well as an optional list of failed permissions.
- `StatusCode(int statusCode, object o = null)`: This returns a custom status code, as well as an optional result object.

An action method can return a result object directly with `return myObject`. This is equivalent to returning `OK(myObject)`.

When all the result paths return a result object of the same type, say, `MyType`, the action method can be declared as returning `ActionResult<MyType>` to get a better type check.

By default, result objects are serialized in JSON in the response body. However, if an XML formatter has been added to the MVC framework processing pipeline, as shown previously, the way the result is serialized depends on the `Accept` header of the HTTP request. More specifically, if the client explicitly requires an XML format with the `Accept` header, the object will be serialized in XML; otherwise, it will be serialized in JSON.

Complex objects that are passed as input to action methods can be validated with validation attributes, as follows:

```
public class MyType
{
    [Required]
    public string Name{get; set;}
    ...
    [MaxLength(64)]
    public string Description{get; set;}
}
```



If the controller has been decorated with the `[ApiController]` attribute and if validation fails, the MVC framework automatically creates a `BadRequest` response containing a dictionary with all the validation errors detected, without executing the action method. Therefore, you don't need to add further code to handle validation errors.

Action methods can also be declared as `async` methods, as follows:

```
public async Task<IActionResult> MyMethod(.....)
{
    await MyBusinessObject.MyBusinessMethod();
    ...
}

public async Task<ActionResult<MyType>> MyMethod(.....)
{
    ...
}
```

Practical examples of controllers/action methods will be shown in the use case section of this chapter. In the next subsection, we will explain how to handle authorization and authentication with JWT tokens.

ASP.NET Core service authorization

When using a JWT token, authorizations are based on the claims contained in the JWT token. All the token claims in any action method can be accessed through the `User.Claims` controller property. Since `User.Claims` is an `IEnumerable<Claim>`, it can be processed with `Linq` to verify complex conditions on claims. If authorization is based on *role* claims, you can simply use the `User.IsInRole` function, as shown in the following code:

```
If(User.IsInRole("Administrators") || User.IsInRole("SuperUsers"))
{
    ...
}
else return Forbid();
```

However, permissions are not usually checked from within action methods and are automatically checked by the MVC framework, according to authorization attributes that decorate either the whole controller or a single action method. If an action method or the whole controller is decorated with `[Authorize]`, then access to the action method is possible only if the request has a valid authentication token, which means we don't have to perform a check on the token claims. It is also possible to check whether the token contains a set of roles using the following code:

```
[Authorize(Roles = "Administrators,SuperUsers")]
```

More complex conditions on claims require that authorization policies are defined in the `ConfigureServices` method of `Startup.cs`, as shown in the following code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    ...
    services.AddAuthorization(options =>
    {
        options.AddPolicy("Father", policy =>
            policy.RequireAssertion(context =>
                context.User
                    .HasClaim(c =>c.Type == "Married") &&
                context.User
                    .HasClaim(c => c.Type == "Hasson")));
    });
}
```

After that, you can decorate the action methods or controllers with `[Authorize(Policy = "Father")]`.

Before using JWT-based authorization, you must configure it in `Startup.cs`. First of all, you must add the middleware that processes authentication tokens in the ASP.NET Core processing pipeline defined in the `Configure` method, as shown here:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    ...
    app.UseAuthentication(); //authentication middleware
    app.UseMvc();
}
```

Then, you must configure the authentication services in the `ConfigureServices` section. Here, you define the authentication options that will be injected through dependency injection into the authentication middleware:

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options => {
        options.TokenValidationParameters =
            new TokenValidationParameters
            {
                ValidateIssuer = true,
                ValidateAudience = true,
                ValidateLifetime = true,
                ValidateIssuerSigningKey = true,

                ValidIssuer = "My.Issuer",
                ValidAudience = "This.Website.Audience",
                IssuerSigningKey =
                    new
                    SymmetricSecurityKey(Encoding.ASCII.GetBytes("MySecret"))
            };
    });
});
```

The preceding code provides a name to the authentication scheme, that is, a default name. Then, it specifies JWT authentication options. Usually, we require that the authentication middleware verifies that the JWT token is not expired (`ValidateLifetime = true`), that it has the right issuer and audience (see the *REST services authorization and authentication* section of this chapter), and that its signature is valid.

The preceding example uses a symmetric signing key generated from a string. This means that the same key is used to sign and to verify the signature. This is an acceptable choice if JWT tokens are created by the same website that uses them, but it is not an acceptable choice if there is a unique JWT issuer that controls access to several Web API sites.

Here, we should use an asymmetric key (typically, a `RsaSecurityKey`), so JWT verification requires just the knowledge of the public key associated with the actual private signing key. Identity Server 4 can be used to quickly create a website that works as an authentication server. It emits a JWT token with the usual username/password credentials or converts other authentication tokens. If you use an authentication server such as Identity Server 4, you don't need to specify the `IssuerSigningKey` option, since the authorization middleware is able to retrieve the required public key from the authorization server automatically. It is enough to provide the authentication server URL, as shown here:

```
.AddJwtBearer(options => {
    options.Authority = "https://www.MyAuthorizationserver.com";
    options.TokenValidationParameters = ...
    ...
})
```

On the other hand, if you decide to emit JWT in your Web API's site, you can define a `Login` action method that accepts an object with a username and password, and that, while relying on database information, builds the JWT token with code similar to the following:

```
var claims = new List<Claim>
{
    new Claim(...),
    new Claim(...),
    ...
};
var token = new JwtSecurityToken(
    issuer: "MyIssuer",
    audience: ...,
    claims: claims,
    expires: DateTime.UtcNow.AddMinutes(expiryInMinutes),
    signingCredentials:
        new
        SymmetricSecurityKey(Encoding.ASCII.GetBytes("MySecret")));
    return OK(new JwtSecurityTokenHandler().WriteToken(token));
```

Here, `JwtSecurityTokenHandler().WriteToken(token)` generates the actual token string from the token properties contained in the `JwtSecurityToken` instance.

In the next subsection, we will learn how to empower our Web API with an OpenAPI documentation point so that proxy classes for communicating with our services can be generated automatically.

ASP.NET Core support for OpenAPI

Most of the information that's needed to fill in an OpenAPI JSON document can be extracted from Web API controllers through reflection, that is, input types and sources (path, request body, and header) and endpoint paths (these can be extracted from routing rules). Returned output types and status codes, in general, can't be easily computed since they can be generated dynamically. Therefore, the MVC framework provides the `ProducesResponseType` attribute so that we can declare a possible return type – a status code pair. It is enough to decorate each action method with as many `ProducesResponseType` attributes as there are possible types, that is, possible status code pairs, as shown in the following code:

```
[HttpGet("{id}")]
[ProducesResponseType(typeof(MyReturnType), StatusCodes.Status200OK)]
[ProducesResponseType(typeof(MyErrorReturnType),
    StatusCodes.Status404NotFound)]
public IActionResult GetById(int id)...
```

If no object is returned along a path, we can just declare the status code, as follows:

```
[ProducesResponseType(StatusCodes.Status403Forbidden)]
```

We can also specify just the status code when all the paths return the same type and when that type is specified in the action method return type as `ActionResult<CommonReturnType>`.

Once all the action methods have been documented, in order to generate any actual documentation for the JSON endpoints, we must install the `Swashbuckle.AspNetCore` NuGet package and place some code in the `Startup.cs` file. More specifically, we must add some middleware in the `Configure` method, as shown here:

```
app.UseSwagger(); //open api middleware
app.UseAuthentication();
app.UseMvc();
```

Then, we must add some configuration options in the `ConfigureServices` method, as follows:

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("MyServiceName", new Info
    {
        Version = "v1",
        Title = "ToDo API",
        Description = "My service description",
```

```
        TermsOfService = "My terms of service",
        Contact = new Contact
        {
            Name = "My Contact Name",
            Email = string.Empty,
            Url = "https://MyContatcUrl.com"
        },
        License = new License
        {
            Name = "My License name",
            Url = "https://MyLicensecUrl.com"
        }
    );
});
```

The first argument of the `SwaggerDoc` method is the documentation endpoint name. By default, the documentation endpoint is accessible through the `<webroot>/swagger/<endpoint name>/swagger.json` path, but this can be changed in several ways. The rest of the information contained in the `Info` class is self-explanatory.

We can add several `SwaggerDoc` calls to define several documentation endpoints. However, by default, all the documentation endpoints will contain the same documentation, which includes a description of all the REST services included in the project. This default can be changed by calling the `c.DocInclusionPredicate(Func<string, ApiDescription> predicate)` method from within `services.AddSwaggerGen(c => {...})`.

`DocInclusionPredicate` must be passed a function that receives a JSON document name and an action method description and must return `true` if the documentation of the action must be included in that JSON document.

To declare that your REST APIs need a JWT token, you must add the following code within `services.AddSwaggerGen(c => {...})`:

```
var security = new Dictionary<string, IEnumerable<string>>
{
    {"Bearer", new string[] { }},
};

c.AddSecurityDefinition("Bearer", new ApiKeyScheme
{
    Description = "JWT Authorization header using the Bearer scheme.",
    Example: "\"Authorization: Bearer {token}\"",
    Name = "Authorization",
    In = "header",
```

```
Type = "apiKey"  
});  
c.AddSecurityRequirement(security);
```

You can enrich the JSON documentation endpoint with information that's been extracted from triple-slash comments, which are usually added to generate automatic code documentation. The following code shows some examples of this. The following snippet shows how we can add a method description and parameter information:

```
//adds a description to the REST method  
  
/// <summary>  
/// Deletes a specific TodoItem.  
/// </summary>  
/// <param name="id"></param>  
[HttpDelete("{id}")]  
public IActionResult Delete(long id)
```

The following snippet shows how we can add an example of usage:

```
//adds an example of usage  
  
/// <summary>  
/// Creates an item.  
/// </summary>  
/// <remarks>  
/// Sample request:  
///  
/// POST /MyItem  
/// {  
///   "id": 1,  
///   "name": "Item1"  
/// }  
///  
/// </remarks>
```

The following snippet shows how we can add parameter descriptions and return type descriptions for each HTTP status code:

```
//Add input parameters and return object descriptions  
  
/// <param name="item">item to be created</param>  
/// <returns>A newly created TodoItem</returns>  
/// <response code="201">Returns the newly created item</response>  
/// <response code="400">If the item is null</response>
```

To enable extraction from triple-slash comments, we must enable code documentation creation by adding the following code in our project file (.csproj):

```
<PropertyGroup>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  <NoWarn>$ (NoWarn) ; 1591</NoWarn>
</PropertyGroup>
```

Then, we must enable code documentation processing from within `services.AddSwaggerGen(c => { ... })` by adding the following code:

```
var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
var xmlPath = Path.Combine(ApplicationContext.BaseDirectory, xmlFile);
c.IncludeXmlComments(xmlPath);
```

Once our documentation endpoints are ready, we can add some more middleware that's contained in the same `Swashbuckle.AspNetCore` NuGet package to generate a friendly user interface that we can test our REST API on:

```
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/<documentation name>/swagger.json", " 
        <api name that appears in dropdown>");
});
```

If you have several documentation endpoints, you need to add a `SwaggerEndpoint` call for each of them. We will use this interface to test the REST API defined in this chapter's use case.

Once you have a working JSON documentation endpoint, you can automatically generate the C# or TypeScript code of a proxy class with one of the following methods:

- The `NSwagStudio` Windows program, which is available at <https://github.com/RicoSuter/NSwag/wiki/NSwagStudio>.
- The `NSwag.CodeGeneration.CSharp` or `NSwag.CodeGeneration.TypeScript` NuGet packages if you want to customize code generation.
- The `NSwag.MSBuild` NuGet package if you want to tie code generation to Visual Studio build operations. The documentation for this can be found at <https://github.com/RicoSuter/NSwag/wiki/MSBuild>.

In the next subsection, you will learn how to invoke a REST API from another REST API or from a .NET Core client.

.Net Core HTTP clients

The `HttpClient` class in the `System.Net.Http` namespace is a .NET standard 2.0 built-in HTTP client class. While it could be used directly whenever we need to interact with a REST service, there are some problems in creating and releasing `HttpClient` instances repeatedly, as follows:

- Their creation is expensive.
- When an `HttpClient` is released, for instance, in a `using` statement, the underlying connection isn't closed immediately but at the first garbage collection session, which is a repeated creation. Release operations quickly exhaust the maximum number of connections the operating system can handle.

Therefore, either a single `HttpClient` instance is reused, such as a singleton, or `HttpClient` instances are somehow pooled. Starting from the 2.1 version of .NET Core, an `HttpClientFactory` class was introduced to pool HTTP clients. More specifically, whenever a new `HttpClient` instance is required for an `HttpClientFactory` object, a new `HttpClient` is created. However, the underlying `HttpClientMessageHandler` instances, which are expansive to create, are pooled until their maximum lifetime expires.

`HttpClientMessageHandler` instances must have a finite duration since they cache DNS resolution information that may change over time. The default lifetime of `HttpClientMessageHandler` is 2 minutes, but it can be redefined by the developer.

Using `HttpClientFactory` allows us to automatically pipeline all the HTTP operations with other operations. For instance, we can add a Polly retry strategy to handle all the failures of all our HTTP operations automatically. For an introduction to Polly, please refer to the *Resilient task execution* subsection of Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*.

The simplest way to exploit the advantages offered by the `HttpClientFactory` class is to add the `Microsoft.Extensions.Http` NuGet package and then to follow these steps:

1. Define a proxy class, say, `MyProxy`, to interact with the desired REST service.
2. Let `MyProxy` accept an `HttpClient` instance in its constructor.
3. Use the `HttpClient` that was injected into the constructor to implement all the necessary operations.

4. Declare your proxy in the services configuration method of your Host which, in the case of an ASP.NET Core application, is the `ConfigureServices` method of the `Startup.cs` class, while, in the case of a client application, this is the `ConfigureServices` method of the `HostBuilder` instance. In the simplest case, the declaration is something similar

to `services.AddHttpClient<MyProxy>()`. This will automatically add `MyProxy` to the services that are available for dependency injection, so you can easily inject it, for instance, in your controller's constructors. Moreover, each time an instance of `MyProxy` is created, an `HttpClient` is returned by an `HttpClientFactory` and is automatically injected into its constructor.

In the constructors of the classes that need to interact with a REST service, we may also need an interface instead of a specific proxy implementation with a declaration of the type:

```
services.AddHttpClient<IMyProxy, MyProxy>()
```

A Polly resilient strategy (see the *Resilient task execution* subsection of Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*) can be applied to all the HTTP calls issued by our proxy class, as shown here:

```
var myRetryPolicy = Policy.Handle<HttpRequestException>()
    ...//policy definition
    ...
services.AddHttpClient<IMyProxy, MyProxy>()
    .AddPolicyHandler(myRetryPolicy);
```

Finally, we can preconfigure some of the properties of all the `HttpClient` instances that are passed to our proxy, as shown here:

```
services.AddHttpClient<IMyProxy, MyProxy>(clientFactory =>
{
    clientFactory.DefaultRequestHeaders.Add("Accept", "application/json");
    clientFactory.BaseAddress = new Uri("https://www.myService.com/");
})
    .AddPolicyHandler(myRetryPolicy);
```

This way, each client that's passed to the proxy is preconfigured so that they require a JSON response and have to work with a specific service. Once the base address has been defined, each HTTP request needs to specify the relative path of the service method to call.

The following code shows how to perform a POST to a service. Here, we're stating that the `HttpClient` that was injected into the proxy constructor has been stored in the `webClient` private field:

```
//Add a bearer token to authenticate the call
webClient.DefaultRequestHeaders.Add("Authorization", "Bearer " + token);
...
//Call service method with a POST verb and get response
var response = await
    webClient.PostAsJsonAsync<MyPostModel>("my/method/relative/path",
        new MyPostModel
    {
        //fill model here
        ...
    });
//extract response status code
var status = response.StatusCode;
...
//extract body content from response
string stringResult = await response.Content.ReadAsStringAsync();
```

If you use Polly, you don't need to intercept and handle communication errors since this job is performed by Polly. First, you need to verify the status code to decide what to do next. Then, you can parse the JSON string contained in the response body to get a .NET instance of a type, that, in general, depends on the status code. The code to perform the parsing is based on the `Newtonsoft.Json` NuGet package's `JsonConvert` class and is as follows:

```
var result=JsonConvert.DeserializeObject<MyResultClass>(stringResult);
```

Performing a GET request is similar but, instead of calling `PostAsJsonAsync`, you need to call `GetAsync`, as shown here:

```
var response = await webClient.GetAsync("my/getmethod/relative/path");
```

The use of other HTTP verbs is completely analogous.

Use case – exposing WWTravelClub packages

In this section, we will implement an ASP.NET REST service that lists all the packages that are available for a given vacation's start and end dates. For didactic purposes, we won't structure the application according to the best practices described in Chapter 10, *Understanding the Different Domains in Software Solutions*; instead, we will simply generate the results with a LINQ query that will be directly placed in the controller action method. A well-structured ASP.NET Core application will be presented in Chapter 13, *Presenting ASP.NET Core MVC*, which is dedicated to the MVC framework.

Let's make a copy of the WWTravelClubDB solution folder and rename the new folder WWTravelClubREST. The WWTravelClubDB project was built step by step in the various sections of Chapter 6, *Interacting with Data in C# - Entity Framework Core*. Let's open the new solution and add a new ASP.NET Core API project to it named WWTravelClubREST (the same name as the new solution folder). For simplicity, select no authentication. Right-click on the newly created project and select **Set as StartUp project** to make it the default project that's launched when the solution is run.

Finally, we need to add a reference to the WWTravelClubDB project.

ASP.NET Core projects store configuration constants in the `appsettings.json` file. Let's open this file and add the database connection string for the database we created in the WWTravelClubDB project to it, as shown here:

```
{  
    "ConnectionStrings": {  
        "DefaultConnection": "Server=  
            (localdb)\\mssqllocaldb;Database=wwtravelclub;  
            Trusted_Connection=True;MultipleActiveResultSets=true"  
    },  
    ...  
    ...  
}
```

Now, we must add the WWTravelClubDB entity framework database context to the `ConfigureServices` method in `Startup.cs`, as shown here:

```
services.AddDbContext<WWTravelClubDB.MainDBContext>(options =>  
    options.UseSqlServer(  
        Configuration.GetConnectionString("DefaultConnection"),  
        b => b.MigrationsAssembly("WWTravelClubDB")));
```

The option object settings that are passed to `AddDbContext` specify the usage of SQL server with a connection string that is extracted from the `ConnectionStrings` section of the `appsettings.json` configuration file with the `Configuration.GetConnectionString("DefaultConnection")` method. The `b => b.MigrationsAssembly("WWTravelClubDB")` lambda function declares the name of the assembly that contains the database migrations (see Chapter 6, *Interacting with Data in C# - Entity Framework Core*) which, in our case, is the DLL that was generated by the `WWTravelClubDB` project. For the preceding code to compile, you should add `using Microsoft.EntityFrameworkCore;`.

Since we want to enrich our REST service with OpenAPI documentation, let's add a reference to the `Swashbuckle.AspNetCore` NuGet package. For .NET 3.0, you must select at least version 5.0 RC-4, so, if you don't see the 5.0 version among the search results, please enable the **Include prerelease** checkbox. Now, we can add the following very basic configuration to the `ConfigureServices` method:

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("WWWTravelClub", new OpenAPIInfo
    {
        Version = "WWWTravelClub 1.0.0",
        Title = "WWWTravelClub",
        Description = "WWWTravelClub Api",
        TermsOfService = null
    });
});
```

Then, we can add the middleware for the OpenAPI endpoint and for adding a user interface for our API documentation, as shown here:

```
app.UseSwagger();
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint(
        "/swagger/WWWTravelClub/swagger.json",
        "WWWTravelClub Api");
});

app.UseEndpoints(endpoints => //preexisting code//)
{
    endpoints.MapControllers();
});
```

Now, we are ready to encode our service. Let's delete `ValueController`, which is automatically scaffolded by Visual Studio. Then, right-click on the `Controller` folder and select **Add | Controller**. Now, choose an empty API controller called `PackagesController`. First, let's modify the code, as follows:

```
[Route("api/packages")]
[ApiController]
public class PackagesController : ControllerBase
{
    [HttpGet("bydate/{start}/{stop}")]
    [ProducesResponseType(typeof(IEnumerable<PackagesListDTO>), 200)]
    [ProducesResponseType(400)]
    [ProducesResponseType(500)]
    public async Task<IActionResult> GetPackagesByDate(
        [FromServices] WWTravelClubDB.MainDBContext ctx,
        DateTime start, DateTime stop)
    {
    }
}
```

The `Route` attribute declares that the basic path for our service will be `api/packages`. The unique action method that we implement is `GetPackagesByDate`, which is invoked on `HttpGet` requests on paths of the `bydate/{start}/{stop}` type, where `start` and `stop` are the `DateTime` parameters that are passed as input to `GetPackagesByDate`. The `ProducesResponseType` attributes declare the following:

- When a request is successful, a 200 code is returned, and the body contains an `IEnumerable` of the `PackagesListDTO` (which we will soon define) type containing the required package information.
- When the request is ill-formed, a 400 code is returned. We don't specify the type returned since Bad Requests are automatically handled by the MVC framework through the `ApiController` attribute.
- In the case of unexpected errors, a 500 code is returned with an empty body.

Now, let's define the `PackagesListDTO` class in a new `DTOs` folder:

```
namespace WWTravelClubREST.DTOs
{
    public class PackagesListDTO
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
        public int DurationInDays { get; set; }
        public DateTime? StartValidityDate { get; set; }
    }
}
```

```
        public DateTime? EndValidityDate { get; set; }
        public string DestinationName { get; set; }
        public int DestinationId { get; set; }
    }
}
```

Finally, let's add the following `using` clauses to our controller code so that we can easily refer to our DTO and to Entity Framework LINQ methods:

```
using Microsoft.EntityFrameworkCore;
using WWTravelClubREST.DTOs;
```

Now, we are ready to fill the body of the `GetPackagesByDate` method with the following code:

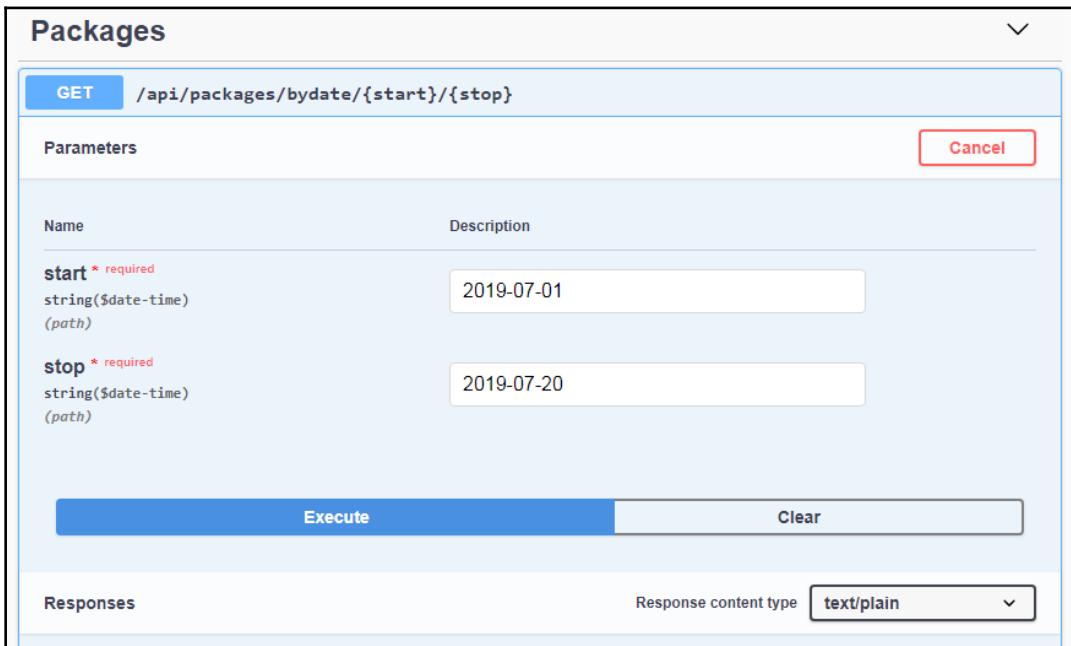
```
try
{
    var res = await ctx.Packages
        .Where(m => start >= m.StartValidityDate
        && stop <= m.EndValidityDate)
        .Select(m => new PackagesListDTO
    {
        StartValidityDate = m.StartValidityDate,
        EndValidityDate = m.EndValidityDate,
        Name = m.Name,
        DurationInDays = m.DurationInDays,
        Id = m.Id,
        Price = m.Price,
        DestinationName = m.MyDestination.Name,
        DestinationId = m.DestinationId
    })
    .ToListAsync();
    return Ok(res);
}
catch
{
    return StatusCode(500);
}
```

The LINQ query is similar to the one contained in the WWTravelClubDBTest project we tested in Chapter 6, *Interacting with Data in C# - Entity Framework Core*. Once the result has been computed, it is returned with an `OK` call. The method's code handles internal server errors by catching exceptions and returning a 500 status code, since Bad Requests are automatically handled before the controller method is called by the `ApiController` attribute.

Let's run the solution. When the browser opens, it's unable to receive any result from our ASP.NET Core website. Let's modify the browser URL so that it's `https://localhost:<previous port>/swagger`. The user interface of the OpenAPI documentation will look as follows:

The screenshot shows the Swagger UI interface for the 'WWWTravelClub Api'. At the top, there is a green header bar with the 'swagger' logo, a dropdown menu labeled 'Select a spec' set to 'WWWTravelClub Api', and a dropdown arrow icon. Below the header, the title 'WWWTravelClub' is displayed in large blue text, followed by 'WWWTravelClub 1.0.0' in a smaller box. A sub-header '/swagger/WWWTravelClub/swagger.json' is shown. The main content area has a heading 'Packages' with a dropdown arrow. Underneath, a 'GET' button is highlighted in blue, followed by the endpoint '/api/packages/bydate/{start}/{stop}'. Below this, a 'Models' section is expanded, showing two items: 'PackagesListDTO > {...}' and 'ProblemDetails > {...}'.

PackagesListDTO is the model we defined to list the packages, while ProblemDetails is the model that's used to report errors in the case of Bad Requests. By clicking the GET button, we can get more details about our GET method and we can also test it, as shown in the following screenshot:



Pay attention when it comes to inserting dates that are covered by packages in the database; otherwise, an empty list will be returned. The ones shown in the preceding screenshot should work.

Dates must be entered in a correct JSON format; otherwise, a 400 Bad Request error is returned, like the one shown in the following code:

```
{  
    "errors": {  
        "start": [  
            "The value '2019' is not valid."  
        ]  
    },  
    "title": "One or more validation errors occurred.",  
    "status": 400,  
    "traceId": "80000008-0000-f900-b63f-84710c7967bb"  
}
```

If you insert the correct input parameters, the Swagger UI returns the packages that satisfy the query in JSON format.

That's all! You have implemented your first API with OpenAPI documentation!

Summary

In this chapter, we introduced SOA, its design principles, and its constraints. Among them, it is worth remembering interoperability.

Then, we focused on well-established standards for business applications that achieve the interoperability that's needed for publicly exposed services. Therefore, SOAP and REST services were discussed in detail, along with the transition from SOAP services to REST services, which has taken place in most application areas in the last few years. Then, REST services principles, authentication/authorization, and its documentation were described in greater detail.

Finally, we looked at the tools that are available in .NET Core that we can use to implement and interact with services. We looked at a variety of frameworks for intra-cluster communication, such as .NET remoting and gRPC, and tools for SOAP and REST-based public services.

Here, we mainly focused on REST services. Their ASP.NET Core implementations were described in detail, along with the techniques we can use in order to authenticate/authorize them and their documentation. We also focused on how to implement efficient .NET Core proxies so that we can interact with REST services.

In the next chapter, we will learn how to use .NET Core 3.0 while building an application on ASP .NET Core MVC.

Questions

1. Can services use cookie-based sessions?
2. Is it good practice to implement a service with a custom communication protocol? Why or why not?
3. Can a POST request to a REST service cause a delete?
4. How many dot-separated parts are contained in a JWT bearer token?
5. By default, where are the complex type parameters of a REST service's action methods taken from?

6. How is a controller declared as a REST service?
7. What are the main documentation attributes of ASP.NET Core services?
8. How are ASP.NET Core REST service routing rules declared?
9. How should a proxy be declared so that we can take advantage of .NET Core's `HttpClientFactory` class features?

Further reading

This chapter mainly focused on the more commonly used REST service. If you are interested in SOAP services, a good place to start is the Wikipedia page regarding SOAP specifications: https://en.wikipedia.org/wiki/List_of_web_service_specifications. On the other hand, if you are interested in the Microsoft .NET WCF technology for implementing SOAP services, you can refer to WCF's official documentation here: <https://docs.microsoft.com/en-us/dotnet/framework/wcf/>.

This chapter mentioned the AMQP protocol as an option for intra-cluster communication without describing it. Detailed information on this protocol is available on AMQP's official site: <https://www.amqp.org/>.

More information on gRPC is available on Google gRPC's official site: <https://grpc.io/>. More information on the Visual Studio gRPC project template can be found here: <https://docs.microsoft.com/en-US/aspnet/core/grpc/?view=aspnetcore-3.0>.

More details on ASP.NET Core services are available in the official documentation: <https://docs.microsoft.com/en-US/aspnet/core/web-api/?view=aspnetcore-3.0>. More information on .NET Core's HTTP client is available here: <https://docs.microsoft.com/en-US/aspnet/core/fundamentals/http-requests?view=aspnetcore-3.0>.

More information on JWT token authentication is available here: <https://jwt.io/>. If you would like to generate JWT tokens with Identity Server 4, you may refer to its official documentation page: <http://docs.identityserver.io/en/latest/>.

More information on OpenAPI is available at <https://swagger.io/docs/specification/about/>, while more information on Swashbuckle can be found on its GitHub repository page: <https://github.com/domaindrivendev/Swashbuckle>.

13

Presenting ASP.NET Core MVC

In this chapter, you will learn how to implement an application presentation layer. More specifically, you will learn how to implement a web application based on ASP.NET Core MVC.

ASP.NET Core is a .NET framework for implementing web applications. ASP.NET Core has been partially described in previous chapters, so this chapter focuses mainly on ASP.NET Core MVC. More specifically, the contribution of this chapter is as follows:

- Understanding the presentation layers of web applications
- Understanding the ASP.NET Core MVC structure
- What is new in .NET Core 3.0 for ASP.NET Core?
- Understanding the connection between ASP.NET Core MVC and design principles
- Use case – implementing a web app in ASP.NET Core MVC

We will review and give further details on the structure of the ASP.NET Core framework that, in part, was discussed in *Chapter 12, Applying Service-Oriented Architectures with .NET Core*, and *Chapter 4, Deciding the Best Cloud-Based Solution*. Here, the main focus is on how to implement web-based presentation layers based on the so-called **Model View Controller (MVC)** architectural pattern.

We will also analyze all of the new features available in the last ASP.NET Core 3.0 version and the architectural patterns included in the ASP.NET Core MVC framework and/or used in typical ASP.NET Core MVC projects. Some of these patterns were discussed in *Chapter 9, Design Patterns and .NET Core Implementation*, and *Chapter 10, Understanding the Different Domains in Software Solutions*, whereas some others, such as the MVC pattern itself, are new.

You will learn how to implement an ASP.NET Core MVC application, and how to organize the whole Visual Studio solution with the practical example at the end of this chapter. This example describes a complete ASP.NET Core MVC application for editing the packages of the WWTravelClub book use case.

Technical requirements

This chapter requires Visual Studio 2017 or the 2019 free Community Edition or better with all database tools installed.

All concepts are clarified with practical examples based on the WWTravelClub book use case. The code for this chapter is available at <https://github.com/PacktPublishing/Hands-On-Software-Architecture-with-CSharp-8>.

Understanding the presentation layers of web applications

This chapter discusses an architecture for the implementation of presentation layers of web-based applications based on the ASP.NET Core framework. Presentation layers of web applications are based on three techniques:

- Mobile or desktop native applications that exchange data with servers through REST or SOAP services: We have not discussed them since they are strictly tied to the client device and its operating system, therefore, analyzing them, which would require a dedicated book, is completely beyond the scope of this book.
- **Single Page Applications (SPA)**: These are HTML-based applications whose dynamic HTML is created on the client either in JavaScript or with the help of WebAssembly (a kind of cross-browser assembly that can be used as a high-performance alternative to JavaScript). Like native applications, SPAs exchange data with the server through REST or SOAP services, but they have the advantage of being independent of the device and its operating system since they run in a browser. SPA frameworks are complex subjects that require dedicated books, so they cannot be described in this book. Some related links are listed in the *Further reading* section.
- HTML pages created by the server whose content depends on the data to be shown to the user: The ASP.NET Core MVC framework, which will be discussed in this chapter, is a framework for creating such dynamic HTML pages.

The remainder of this chapter focuses on how to create HTML pages on the server side and, more specifically, on ASP.NET Core MVC, which is introduced in the next section.

Understanding the ASP.NET Core MVC structure

ASP.NET Core is based on the concept of the Generic Host explained in the *Using Generic Hosts* subsection of Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*. The basic architecture of ASP.NET Core is outlined in the *A short introduction to ASP.NET Core* subsection of Chapter 12, *Applying Service-Oriented Architectures with .NET Core*.

It is worth reminding readers that the host configuration is delegated to the `Startup` class defined in the `Startup.cs` file by calling the `.UseStartup<Startup>()` method of the `IWebHostBuilder` interface. `ConfigureServices(IServiceCollection services)` of the `Startup` class defines all services that can be injected in object constructors through DI. DI is described in detail in the *Using Generic Hosts* subsection of Chapter 5, *Applying Microservice Architecture to Your Enterprise Application*.

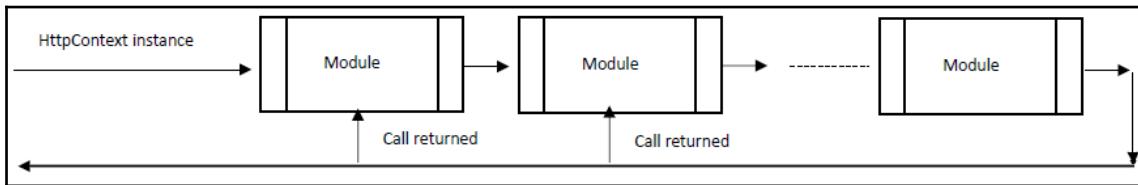
The `Configure(IApplicationBuilder app, IHostingEnvironment env)` `startup` method, instead, defines the so-called ASP.NET Core pipeline that was briefly described in *A short introduction to ASP.NET Core* subsection of Chapter 12, *Applying Service-Oriented Architectures with .NET Core*, and that will be described in more detail in the next subsection.

How ASP.NET Core pipeline works

ASP.NET Core furnishes a set of configurable modules you may assemble according to your needs. Each module takes care of functionality that you may need or not. Examples of functionalities are authorization, authentication, static file processing, protocol negotiation, CORS handling, and so on.

You can put together all of the modules you need by inserting them into a common processing framework called the **ASP.NET Core pipeline**.

More specifically, ASP.NET Core requests are processed by pushing a context object through a pipeline of ASP.NET Core modules, as shown in the following diagram:



The object that is inserted in the pipeline is an `HttpContext` instance that contains the whole data of the incoming request. More specifically the `Request` property of `HttpContext` contains an `HttpRequest` object whose properties represent the incoming request in a structured way. There are properties for headers, cookies, request path, parameters, form fields, and the request body.

The various modules can contribute to the construction of the final response by writing in an `HttpResponse` object contained in the `Response` property of the `HttpContext` instance. The `HttpResponse` class is similar to the `HttpRequest` class, but its properties refer to the response being built.

Some modules can build an intermediate data structure that is then used by other modules in the pipeline. In general, such intermediary data can be stored in custom entries of `IDictionary<object, object>` contained in the `Items` property of the `HttpContext` object. However, there is a predefined property, `User`, which contains information on the currently logged user. The logged-in user is not computed automatically but must be computed by an authentication module. The *ASP.NET Core services authorization* subsection of Chapter 12, *Applying Service-Oriented Architectures with .NET Core*, explains how to add the standard module that performs JWT token-based authentication to the ASP.NET Core pipeline.

`HttpContext` has also a `Connection` property that contains information on the underlying connection established with the client and a `WebSockets` property that contains information on possible WebSocket-based connections established with the clients.

`HttpContext` also has a `Features` property that contains `IDictionary<Type, object>`, which specifies the features supported by the web server that hosts the web application and by the modules of the pipeline. Features can be set with the `.Set<Type>(Type o)` method and can be retrieved with the `.Get<Type>()` method.

Web server features are automatically added by the framework, when all other features are added by pipeline modules while they process `HttpContext`. Features are not specific for the incoming request but depend just on the application-hosting environment, and on the modules added to the ASP.NET Core pipeline.

`HttpContext` gives access also to the dependency injection engine through its `RequestServices` property. You can get an instance of a type managed by the dependency engine by calling the `.RequestService(Type t)` method.



The `HttpContext` instance that is created for processing a web request is not available only to modules, but also to the application code through DI. It is enough to insert an `IHttpContextAccessor` parameter in the constructor of a class that is automatically dependency injected, such as a controller (see later on in this section), and then access its `HttpContext` property.

A module is any class with the following structure:

```
public class CoreMiddleware
{
    private readonly RequestDelegate _next;
    public CoreMiddleware(RequestDelegate next, ILoggerFactory
loggerFactory)
    {
        ...
        _next = next;
        ...
    }

    public async Task Invoke(HttpContext context)
    {
        /*
            Insert here the module specific code that processes the
            HttpContext instance

        */
        await _next.Invoke(context);
        /*
            Insert here other module specific code that processes the
            HttpContext instance
        */
    }
}
```

In general, each module processes the `HttpContext` instance passed by the previous module in the pipeline, then calls `await _next.Invoke(context)` to invoke the modules in the remainder of the pipeline. When all other modules finish their processing and the response for the client has been prepared, each module can perform further post-processing of the response in the code that follows the `_next.Invoke(context)` call.

Modules are registered in the ASP.NET Core pipeline by calling the `UseMiddleware<T>` method in the `Startup.cs` file's `Configure` method, as shown here:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
IServiceProvider serviceProvider)
{
    ...
    app.UseMiddleware<MyCustomModule>
    ...
}
```

Modules are inserted in the pipeline in the same order when `UseMiddleware` is called. Since each functionality added to an application might require several modules and might require operations other than adding modules, you usually define an `IApplicationBuilder` extension such as `UseMyFunctionality`, as shown in the following code:

```
public static class MyMiddlewareExtensions
{
    public static IApplicationBuilder UseMyFunctionality(this
IApplicationBuilder builder,...)
    {
        //other code
        ...
        builder.UseMiddleware<MyModule1>();
        builder.UseMiddleware<MyModule2>();
        ...
        //Other code
        ...
        return builder;
    }
}
```

After that, the whole functionality can be added to the application by calling `app.UseMyFunctionality(...)`. For instance, the ASP.NET Core MVC functionality is added to the ASP.NET Core pipeline by calling `app.UseEndpoints(...)`.

Often, functionalities added with each `app.Use...` require that some .NET types are added to the application DI engine. In these cases, we also define an `IServiceCollection` extension named `AddMyFunctionality` that must be called in the `Startup.cs` file's `ConfigureServices(IServiceCollection services)` method. For instance, ASP.NET Core MVC requires a call like the following:

```
services.AddControllersWithViews(o =>
{
    //set here MVC options by modifying the o option parameter
})
```

If you don't need to change the default MVC options, you can simply call `services.AddControllersWithViews()`.

The next subsection describes another important feature of the ASP.NET Core framework, namely, how to handle application configuration data.

Loading configuration data and using it with the options framework

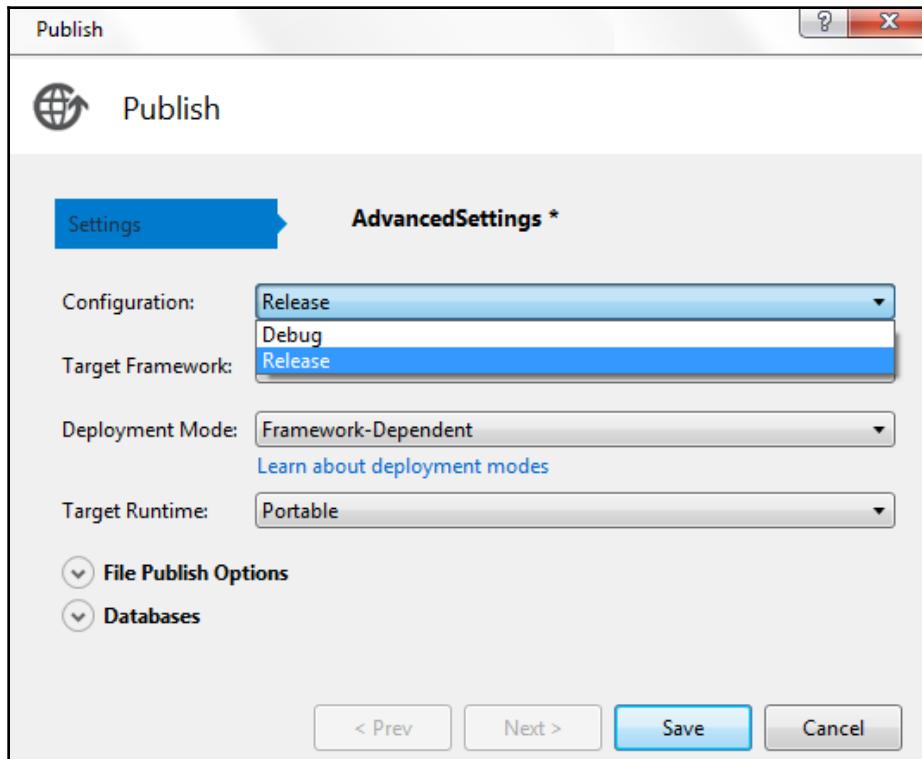
When an ASP.NET Core application starts, it reads configuration information (such as a database connection string) from the `appsettings.json` and `appsettings.[EnvironmentName].json` files, where `EnvironmentName` is a string value that depends on where the application is deployed. Typical values for `EnvironmentName` are as follows:

- Production is used for production deployment.
- Development is used during development.
- Staging is used when the application is tested in staging.

The two JSON trees extracted from the `appsettings.json` and `appsettings.[EnvironmentName].json` files are merged into a unique tree where values contained in `[EnvironmentName].json` override the values contained in the corresponding paths of `appsettings.json`. This way, the application can be run with different configurations in different deployment environments. In particular, you may use a different database connection string, and hence, a different database instance in each different environment.

The [EnvironmentName] string is taken from the ASPNETCORE_ENVIRONMENT operating system environment variable. In turn, ASPNETCORE_ENVIRONMENT can be automatically set during the application's deployment with Visual Studio in two ways:

- During Visual Studio deployment, Visual Studio publish wizard creates an XML publish profile. If the publish wizard allows you to choose the ASPNETCORE_ENVIRONMENT from a drop-down list, you are done:



Otherwise, you may proceed as follows:

1. Once you fill in the information in the wizard, save the publish profile without publishing.

2. Then, edit the profile with a text editor and add an XML property such as, <EnvironmentName>Staging</EnvironmentName>. Since all already defined publish profiles can be selected during the application publication, you may define a different publish profile for each of your environments, and then, you may select the one you need during each publication.
- The value to set ASPNETCORE_ENVIRONMENT to during deployment can also be specified in the Visual Studio ASP.NET Core project file (.csproj) of your application by adding the following code:

```
<PropertyGroup>
    <EnvironmentName>Staging</EnvironmentName>
</PropertyGroup>
```

During development in Visual Studio, the value to give to ASPNETCORE_ENVIRONMENT when the application is run can be specified in the

Properties\launchSettings.json file of the ASP.NET Core project. The launchSettings.json file contains several named groups of settings. These settings configure how to launch the web application when it is run from Visual Studio. You may choose to apply all settings of a group by selecting the group name with the drop-down list next to Visual Studio's run button:



Your selection in this drop-down list will be shown in the run button, the default selection being **IIS Express**.

The following code shows a typical launchSettings.json file in which you can either add a new group of settings or change the settings of the existing default groups:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:2575",
      "sslPort": 44393
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

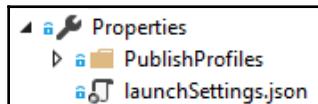
```
        "launchBrowser": true,
        "environmentVariables": {
            "ASPNETCORE_ENVIRONMENT": "Development"
        },
        ...
        ...
    }
}
```

The named groups of settings are under the `profiles` properties. There, you may choose where to host the application (`IISExpress`), to launch the browser, and the values of some environment variables.

The current environment loaded from the `ASPNETCORE_ENVIRONMENT` operating system environment variable can be tested through the `IHostingEnvironment` interface during the ASP.NET Core pipeline definition since an `IHostingEnvironment` instance is passed as a parameter to the `Startup.cs` file's `Configure` method. `IHostingEnvironment` is also available to the remainder of the user code through DI.

`IHostingEnvironment.IsEnvironment(string environmentName)` checks whether the current value of `ASPNETCORE_ENVIRONMENT` is `environmentName`. There are also specific shortcuts for testing development (`.IsDevelopment()`), production (`.IsProduction()`), and staging (`.IsStaging()`). `IHostingEnvironment` also contains the current root directory of the ASP.NET Core application (`.WebRootPath`) and the directory reserved for static files (`.ContentRootPath`) that are served as they are by the web server (CSS, JavaScript, images, and so on).

Both `launchSettings.json` and all publish profiles can be accessed as children of the **Properties** node in Visual Studio Explorer, as shown in the following screenshot:



Once `appsettings.json` and `appsettings.[EnvironmentName].json` are loaded, the configuration tree resulting from their merge can be mapped to the properties of .NET objects. For example, let's suppose we have an `Email` section of the `appsettings` files that contains all of the information needed to connect to an email server, as shown here:

```
{  
    "ConnectionStrings": {  
        "DefaultConnection": "...."  
    },  
    "Logging": {  
        "LogLevel": {  
            "Default": "Warning"  
        }  
    },  
    "Email": {  
        "FromName": "MyName",  
        "FromAddress": "info@MyDomain.com",  
        "LocalDomain": "smtps.MyDomain.com",  
        "MailServerAddress": "smtps.MyDomain.com",  
        "MailServerPort": "465",  
        "UserId": "info@MyDomain.com",  
        "UserPassword": "mypassword"  
    }  
}
```

Then, the whole `Email` section can be mapped to an instance of the following class:

```
public class EmailConfig  
{  
    public String FromName { get; set; }  
    public String FromAddress { get; set; }  
    public String LocalDomain { get; set; }  
  
    public String MailServerAddress { get; set; }  
    public String MailServerPort { get; set; }  
  
    public String UserId { get; set; }  
    public String UserPassword { get; set; }  
}
```

The code that performs the mapping must be inserted in the `ConfigureServices` method in the `Startup.cs` file since the `EmailConfig` instance will be available through DI. The code we need is shown here:

```
public Startup(IConfiguration configuration)  
{  
    Configuration = configuration;  
}
```

```
....  
public void ConfigureServices(IServiceCollection services)  
{  
    ...  
    services.Configure<EmailConfig>(Configuration.GetSection("Email"));  
    ..
```

After the preceding settings, classes that need `EmailConfig` data must declare an `IOptions<EmailConfig>` options parameter that will be provided by the DI engine. An `EmailConfig` instance is contained in `options.Value`.

The next subsection describes the basic ASP.NET Core pipeline modules needed by an ASP.NET Core MVC application.

Defining the ASP.NET Core MVC pipeline

If you create a new ASP.NET Core MVC project in Visual Studio, a standard pipeline is created in the `Startup.cs` file's `Configure` method. There, if needed, you may add further modules or change the configuration of the existing modules.

The initial code of the `Configure` method handles errors and performs basic HTTPS configuration:

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

If there are errors, if the application is in a development environment, the module installed by `UseDeveloperExceptionPage` adds a detailed error report to the response, while the module installed by `UseDatabaseErrorPage` processes and adds details of Entity Framework database errors, if any, to the response. These modules are valuable debugging tools.

If an error occurs when the application is not in development mode, `UseExceptionHandler` restores the request processing from the path it receives as an argument, that is, from `/Home/Error`. In other words, it simulates a new request with the `/Home/Error` path. This request is pushed into the standard MVC processing until it reaches the endpoint associated with the `/Home/Error` path, where the developer is expected to place the custom code that handles the error.

When the application is not in development, `UseHsts` adds the `Strict-Transport-Security` header to the response that informs the browser that the application must be accessed only with HTTPS. After this declaration, compliant browsers should automatically convert any HTTP request to the application into an HTTPS request for the time specified in the `Strict-Transport-Security` header. As a default, `UseHsts` specifies 30 days as the time in the header, but you may specify a different time and other header parameters by adding an `options` object in the `ConfigureServices` method of `Startup.cs`:

```
services.AddHsts(options => {
    ...
    options.MaxAge = TimeSpan.FromDays(60);
    ...
});
```

`UseHttpsRedirection` causes an automatic redirection to an HTTPS URL when an HTTP URL is received, in a way to force a secure connection. Once the first HTTPS secure connection is established, the `Strict-Transport-Security` header prevents future redirections that might be used to perform man-in-the-middle attacks.

The following code shows the remainder of the default pipeline:

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

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

...
```

`UseStaticFiles` makes accessible from the web all files contained in the `wwwroot` folder of the project (typically CSS, JavaScript, images, and font files) through their actual path.

`UseCookiePolicy` ensures that cookies are processed by the ASP.NET Core pipeline only if the user has given consent for cookie usage. Consent to cookie usage is given through a consent cookie, that is, cookie processing is enabled only if this consent cookie is found among the request cookies. This cookie must be created by JavaScript when the user clicks a consent button. The whole string that contains both the consent cookie names and its contents can be retrieved from the `HttpContext.Features`, as shown in the following snippet:

```
var consentFeature = context.Features.Get<ITrackingConsentFeature>();  
var showBanner = !consentFeature?.CanTrack ?? false;  
var cookieString = consentFeature?.CreateConsentCookie();
```

`CanTrack` is `true` only if consent is required and has not been given yet. When the consent cookie is detected, `CanTrack` is set to `false`. This way, `showBanner` is `true` only if consent is required and has not been given yet. Therefore, it tells us whether to ask the user for consent or not.

Options of the consent module are contained in a `CookiePolicyOptions` instance that must be configured manually with the options framework. The following code snippet shows the default configuration code scaffolded by Visual Studio that configures `CookiePolicyOptions` in the code instead of using the configuration file:

```
services.Configure<CookiePolicyOptions>(options =>  
{  
    options.CheckConsentNeeded = context => true;  
});
```

`UseAuthentication` enables authentication schemes. As a default, it only enables cookie-based authentication, that is, an authentication scheme where the authentication token is placed in a cookie. The authentication cookie is created during user login.

Cookies authorization options (such as the cookie name) and other authentication schemes can be enabled by configuring an options object in the `ConfigureServices` method, as shown here:

```
services.AddAuthentication(o =>  
{  
    o.DefaultScheme =  
        CookieAuthenticationDefaults.AuthenticationScheme;  
})  
.AddCookie(o =>  
{
```

```
    o.Cookie.Name = "my_cookie";
})
.AddJwtBearer(o =>
{
    ...
});
```

The preceding code specifies a custom authentication cookie name and adds JWT-based authentication for the REST service contained in the application. Both `AddCookie` and `AddJwtBearer` have overloads that accept the name of the authentication scheme before the action where you can define the authentication scheme options. Since the authentication scheme name is necessary to refer to a specific authentication scheme, when it is not specified, a default name is used:

- The standard name contained in `CookieAuthenticationDefaults.AuthenticationScheme` for cookie authentication
- The standard name contained in `JwtBearerDefaults.AuthenticationScheme` for JWT authentication

The name passed in `o.DefaultScheme` selects the authentication scheme used for filling the `User` property of `HttpContext`.



For more information about JWT authentication, please refer to the *ASP.NET Core services authorization* subsection of Chapter 12, *Applying Service-Oriented Architectures with .NET Core*.

`UseAuthorization` enables authorization based on the `Authorize` attribute. Options can be configured with an `AddAuthorization` method placed in the `ConfigureServices` method. These options allow the definition of policies for claims-based authorization.



For more information on authorization, please refer to the *ASP.NET Core services authorization* subsection of Chapter 12, *Applying Service-Oriented Architectures with .NET Core*.

`UseRouting` and `UseEndpoints` handle the so-called ASP.NET Core endpoints. An endpoint is an abstraction of a handler that serves specific classes of URLs. URLs are transformed into an `Endpoint` instance by using patterns. When a pattern matches a URL, an `Endpoint` instance is created and filled with both the pattern name and data extracted from the URL as a consequence of matching URL parts with named parts of the pattern, as shown in the following code snippet:

```
Request path: /UnitedStates/NewYork  
Pattern: Name="location", match="/{Country}/{Town}"  
  
Endpoint: DisplayName="Location", Country="UnitedStates", Town="NewYork"
```

`UseRouting` adds a module that processes the request path to get the request `Endpoint` instance and adds it to the `HttpContext.Features` dictionary under the `IEndpointFeature` type. The actual `Endpoint` instance is contained in the `Endpoint` property of `IEndpointFeature`.

Each pattern also contains the handler that should process all requests that match the pattern. This handler is passed to `Endpoint` when the `Endpoint` is created.

`UseEndpoints` instead adds the middleware that invokes the handler associated with the request endpoint. It is placed at the end of the pipeline since the execution of the handler is expected to produce the final response.

As the following code snippet shows, patterns are processed in the `UseRouting` middleware but they are specified in the `UseEndpoints` method. This splitting is not necessary but is done for coherence with the previous ASP.NET Core versions that contained no method analogous to `UseRouting`, but a unique call at the end of the pipeline. In the new version, patterns are still defined in `UseEndpoints`, which is placed at the end of the pipeline, but `UseEndpoints` just creates a data structure containing all patterns, when the application starts. Then, this data structure is processed by the `UseRouting` middleware, as shown in the following code:

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

`.MapControllerRoute` defines the patterns associated with the MVC engine that will be described in the next subsection. There are other methods that define other types of patterns. A call such as `.MapHub<MyHub>("/chat")` maps paths to hubs that handle WebSockets, whereas `.MapHealthChecks("/health")` maps paths to ASP.NET Core components that return application health data. You can also directly map a pattern to a custom handler with `.MapGet`, which intercepts GET requests, and `.MapPost`, which intercepts POST requests. The following is an example of `MapGet`:

```
MapGet("hello/{country}", context =>
    context.Response.WriteAsync(
        $"Selected country is {context.GetRouteValue("country")});
```

Patterns are processed in the order in which they are defined until a matching pattern is found. Since the authentication/authorization middleware is placed after the routing middleware, it can process the `Endpoint` request to verify whether the current user has the required authorizations to execute the `Endpoint` handler. Otherwise, a 401 (Unauthorized) or 403 (Forbidden) response is immediately returned. Only requests that survive authentication and authorization have their handlers executed by the `UseEndpoints` middleware.

With the ASP.NET Core RESTful API described in Chapter 12, *Applying Service-Oriented Architectures with .NET Core*, ASP.NET Core MVC also uses attributes placed on controllers or on controller methods to specify authorization rules. However, an instance of `AuthorizeAttribute` can be also added to a pattern to apply its authorization constraints to all URLs matching that pattern, as shown in the following example:

```
endpoints
    .MapHealthChecks("/healthz")
    .RequireAuthorization(new AuthorizeAttribute() { Roles = "admin", });
```

The preceding code makes the health check path available only to administrative users.

Defining controllers and ViewModels

The various `.MapControllerRoute` calls in `UseEndpoints` associate URL patterns to controllers and to methods of these controllers, where controllers are classes that inherit from the `Microsoft.AspNetCore.Mvc.Controller` class. Controllers are discovered by inspecting all of the application's `.dll` files and are added to the DI engine. This job is performed by the call to `AddControllersWithViews` in the `ConfigureServices` method of the `startup.cs` file.

The pipeline module added by `UseEndpoints` takes the controller name from the `controller` pattern variable, and the name of the controller method to invoke from the `action` pattern variable. Since, by convention, all controller names are expected to end with the `Controller` suffix, the actual controller type name is obtained from the name found in the `controller` variable by adding this suffix. Hence, for instance, if the name found in `controller` is "Home", then the `UseEndpoints` module tries to get an instance of the `HomeController` type from the DI engine. All of the controller public methods can be selected by the routing rules. Use of a controller public method can be prevented by decorating it with the `[NonAction]` attribute. All controller methods available to the routing rules are called action methods.

MVC controllers work like the API controllers that we described in the *Implementing REST services with ASP.NET Core* subsection of Chapter 12, *Applying Service-Oriented Architectures with .NET Core*. The only difference is that API controllers are expected to produce JSON or XML, while MVC controllers are expected to produce HTML. For this reason, while API controllers inherit from the `ControllerBase` class, MVC controllers inherit from the `Controller` class, which, in turn, inherits from the `ControllerBase` class and adds its methods that are useful for HTML generation, such as invoking views, which are described in the next subsection, and creating a redirect response.

MVC controllers may use also a routing technique similar to one of the API controllers, that is, routing based on controllers and controller method attributes. This behavior is enabled by calling the `.MapDefaultControllerRoute()` method in `UseEndpoints`. If this call is placed before all `MapControllerRoute` calls, then controller routes have priority on `MapControllerRoute` patterns; otherwise, the converse is true.

All attributes we have seen for API controllers can be also used with MVC controllers and action methods (`HttpGet`, `HttpPost`, ... `Authorize`, and so on). Developers can write their own custom attributes by inheriting from the `ActionFilter` class or other derived classes. I will not give details on this right now, but these details can be found in the official documentation referred to in the *Further reading* section.

When the `UseEndpoints` module invokes a controller, all of its constructor parameters are filled by the DI engine since the controller instance itself is returned by the DI engine, and since DI automatically fills constructor parameters with DI in a recursive fashion.

Action method parameters, instead, are taken from the following sources:

- Request headers
- Variables in the pattern matched by the current request
- Query string parameters

- Form parameters (in the case of POST requests)
- **Dependency injection (DI)**

While the parameters filled with DI are matched by type, all other parameters are matched by *name* ignoring the letter casing. That is, the action method parameter name must match the header, query-string, form, or pattern variable. When the parameter is a complex type, a match is searched for each property, using the property name for the match. In the case of nested complex types, a match is searched for each path and the name associated with the path is obtained by chaining all property names in the path and separating them with dots. For instance, the name associated with a path composed by `Property1`, `Property2`, ..., `Propertyn`, is `Property1.Property2.Property3...Propertyn`. The name obtained this way must match a header name, pattern variable name, query string parameter name, and so on.

By default, simple type parameters are matched with pattern variables and query string variables, while complex types parameters are matched with form parameters. However, the preceding defaults can be changed by prefixing the parameters with attributes as detailed here:

- `[FromForm]` forces a match with form parameters.
- `[FromHeader]` forces a match with a request header.
- `[FromRoute]` forces a match with pattern variables.
- `[FromQuery]` forces a match with a query string variable.
- `[FromServices]` forces the use of DI.

During the match, the string extracted from the selected source is converted into the type of the action method parameter using the current thread culture. If either a conversion fails or no match is found for a not nullable action method parameter, then the whole action method invocation process fails, and a 404 response is automatically returned. For instance, in the following example, the `id` parameter is matched with query string parameters or pattern variables since it is a simple type, while `myclass` properties and nested properties are matched with form parameters since `MyClass` is a complex type. Finally, `myservice` is taken from DI since it is prefixed with the `[FromServices]` attribute:

```
public class HomeController : Controller
{
    public IActionResult MyMethod(
        int id,
        MyClass myclass,
        [FromServices] MyService myservice)
    {
        ...
    }
}
```

If no match is found for the `id` parameter, a 404 response is automatically returned since integers are not nullable. If, instead, no `MyService` instance is found in the DI container, an exception is thrown because in this case the failure doesn't depend on a wrong request but a design error.

MVC controllers return an `IActionResult` interface or a `Task<IActionResult>` result if they are declared as `async`. `IActionResult` has a unique `ExecuteResultAsync(ActionContext)` method that, when invoked by the framework, produces the actual response.

For each different `IActionResult`, MVC controllers have methods that return them. The most commonly used `IActionResult` is `ViewResult`, which is returned by a `View` method:

```
public IActionResult MyMethod(...)  
{  
    ...  
    return View("myviewName", MyViewModel)  
}
```

`ViewResult` is a very common way for a controller to create an HTML response. More specifically, the controller interacts with business/data layers to produce an abstraction of the data that will be shown in the HTML page. This abstraction is an object called a **ViewModel**. The `ViewModel` is passed as a second argument to the `View` method, while the first argument is the name of an HTML template called `View` that is instantiated with the data contained in the `ViewModel`.

Summing up, the MVC controllers' processing sequence is as follows:

1. Controllers perform some processing to create the `ViewModel`, which is an abstraction of the data to show on the HTML page.
2. Then, controllers create `ViewResult` by passing a `View` name and `ViewModel` to the `View` method.
3. The MVC framework invokes `ViewResult` and causes the template contained in the `View` to be instantiated with the data contained in the `ViewModel`.
4. The result of the template instantiation is written in the response with adequate headers.

This way, the controller performs the conceptual job of HTML generation by building a `ViewModel`, while the `View`, that is, the template, takes care of all the graphical details.

Views are described in greater detail in the next subsection, while the Model (ViewModel) View Controller pattern is discussed in more detail in the *Connection between ASP.NET Core MVC and design principles* section of this chapter. Finally, a practical example is given in the *Use case – web app in ASP.NET Core MVC* section of this chapter.

Another common `IActionResult` is `RedirectResult`, which creates a redirect response, hence forcing the browser to move to a specific URL. Redirects are often used after the user has successfully submitted a form that completes a previous operation. In this case, it is common to redirect the user to a page where they can select another operation.

The simplest way to return `RedirectResult` is by bypassing a URL to the `Redirect` method. This is the advised way to perform a redirect to a URL that is outside the web application. When the URL is within the web application, instead, it is advisable to use the `RedirectToAction` method, which accepts the controller name, the action method name, and the desired parameters for the target action method. The framework uses this data to compute a URL that causes the desired action method to be invoked with the provided parameters. This way, if the routing rules are changed during the application development or maintenance, the new URL is automatically updated by the framework with no need to modify all occurrences of the old URL in the code. The following code shows how to call `RedirectToAction`:

```
return RedirectToAction("MyActionName", "MyControllerName",
    new {par1Name=par1Value,..parNName=parNValue});
```

Another useful `IActionResult` is `ContentResult`, which can be created by calling the `Content` method. `ContentResult` allows you to write any string to the response and to specify its MIME type, as shown in the following example:

```
return Content("this is plain text", "text/plain");
```

Finally, the `File` method returns `FileResult`, which writes binary data in the response. There are several overloads of this method that allow the specification of a byte array, a stream, or the path of a file, plus the MIME type of the binary data.

Now, let's move to describe how actual HTML is generated in Views.

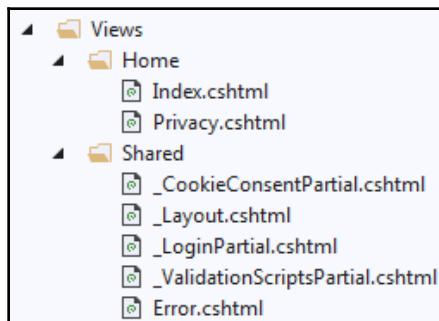
Understanding Razor Views

ASP.NET Core MVC uses a language called Razor to define the HTML templates contained in the Views. Razor views are files that are compiled into .NET classes either at their first usage, when the application is built, or when the application is published. As a default, both pre-compilation on each build and on publish are enabled, but this behavior can be changed by adding the following code to the web application project file:

```
<PropertyGroup>
  <TargetFramework>netcoreapp3.0</TargetFramework>
  <!-- add code below -->
  <RazorCompileOnBuild>false</RazorCompileOnBuild>
  <RazorCompileOnPublish>false</RazorCompileOnPublish>
  <!-- end of code to add -->
  ...
</PropertyGroup>
```

Views can also be precompiled into views libraries by choosing a Razor view library project in the windows that appear after you have chosen an ASP.NET Core project.

Also, after the compilation, views remain associated with their paths, which become their full names. Each controller has an associated folder under the **Views** folder with the same name as the controller, which is expected to contain all the views used by that controller. The following screenshot shows the folder associated with the `HomeController` and its Views:



The preceding screenshot also shows the **Shared** folder, which is expected to contain all the views used by several controllers. The controller refers to views in the `View` method through their paths without the `.cshtml` extension. If the path starts with `/`, the path is interpreted as relative to the application root. Otherwise, as a first attempt, the path is interpreted as relative to the folder associated with the controller and, if no view is found there, the view is searched in the **Shared** folder.

Hence, for instance, the `Privacy.cshtml` View file in the preceding screenshot can be referred to from within `HomeController` as `View("Privacy", MyViewModel)`. If the name of the View is the same as the name of the action method, we can simply write `View(MyViewModel)`.

Razor views are a mix of HTML code with C# code, plus some Razor-specific statements. They all begin with a header that contains the type of ViewModel that the View is expected to receive:

```
@model MyViewModel
```

Each view may contain also some `using` statements whose effect is the same as the `using` statements of standard code files:

```
@model MyViewModel  
@using MyApplication.Models
```

`@using` statements declared in the special `_ViewImports.cshtml` file, that is, in the root of the Views folder, are automatically applied to all views.

Each view can also require instances of types from the DI engine in its header with the syntax shown here:

```
@model MyViewModel  
@using MyApplication.Models  
@inject IViewLocalizer Localizer
```

The preceding code requires an instance of the `IViewLocalizer` interface and places it in the `Localizer` variable. The remainder of the View is a mix of C# code, HTML, and Razor control flow statements. Each area of a view can be either in HTML mode or C# mode. The code in a View area that is in HTML mode is interpreted as HTML, while the code in a View area that is in C# mode is interpreted as C#.

The topic that follows explains Razor flow of control statements.

Learning Razor flow of control statements

If you want to write some C# code in an HTML area, you can create a C# area with the `@{...}` flow of control Razor statement, as shown here:

```
@{  
    //place C# code here  
    var myVar = 5;  
    ...  
    <div>
```

```
<!-- here you are in HTML mode again -->
...
</div>
//after the HTML block you are still in C# mode
var x = "my string";
}
```

The preceding example shows that it is enough to write an HTML tag to create an HTML area inside of the C# area and so on recursively. As soon as the HTML tag closes, you are again in C# mode.

C# code produces no HTML, while HTML code is added to the response in the same order it appears. You can add text computed with C# code while in HTML mode by prefixing any C# expression with @. If the expression is complex, composed of a chain of properties and method calls, it must be enclosed by parentheses. The following code shows some examples:

```
<span>Current date is: </span>
<span>@DateTime.Today.ToString("d")</span>
...
<p>
    User name is: @(myName+ " "+mySurname)
</p>
...
<input type="submit" value="@myUserMessage" />
```

Types are converted into strings using the current culture settings (see the *Connection between ASP.NET Core MVC and design principles* section for details on how to set the culture of each request). Moreover, strings are automatically HTML encoded to avoid the < and > symbols that might interfere with the view HTML. HTML encoding can be prevented with the @HTML.Raw function, as shown here:

```
@HTML.Raw(myDynamicHtml)
```

In an HTML area, alternative HTML can be selected with the @if Razor statement:

```
@if (myUser.IsRegistered)
{
    //this is a C# code area
    var x=5;
    ...
    <p>
        <!-- This is an HTML area -->
    </p>
    //this is a C# code area again
}
else if(callType == CallType.WebApi)
```

```
{  
    ...  
}  
else  
{  
    ...  
}
```

An HTML template can be instantiated several times with the `for`, `foreach`, and `while` Razor statements, as shown in the following examples:

```
@for(int i=0; i< 10; i++)  
{  
  
}  
  
@foreach(var x in myIEnumerable)  
{  
  
}  
  
@while(true)  
{  
}
```

Please do not confuse the statements described so far with the usual C# `if`, `for`, `foreach`, and `while` statements, since they are Razor-specific statements whose syntax is similar to their standard C# counterparts.

Razor views can contain comments that do not generate any code. Any text included within `@*...*@` is considered a comment and is removed when the page is compiled. The next topic describes properties that are available in all Views.

Understanding Razor View properties

Some standard variables are predefined in each view. The most important variable is `Model`, which contains the `ViewModel` passed to the view. For instance, if we pass a `Person` model to a view, then `@Model.Name` displays the name of the `Person` passed to the view.

The `ViewData` variable contains `IDictionary<string, object>`, which is shared with the controller that invoked the view. That is, all controllers also have a `ViewData` property containing `IDictionary<string, object>`, and every entry that is set in the controller is available also in the `ViewData` variable of the invoked view. `ViewData` is an alternative to the `ViewModel` for a controller to pass information to its invoked view.

The `User` variable contains the currently logged user, that is, the same instance contained in the current request's `HttpContext.User` property. The `Url` variable contains an instance of the `IUrlHelper` interface whose methods are utilities for computing URLs of application pages. For instance, `Url.Action("action", "controller", new {par1=valueOfPar1, ...})` computes the URL that causes the action method `action` of the `controller` to be invoked with all parameters specified in the anonymous object passed as its parameters.

The `Context` variable contains the whole request `HttpContext`. The `ViewContext` variable contains data about the context of the view invocation, included metadata about the action method that invoked the view.

The next topic describes how Razor enhances HTML tag syntax.

Using Razor tag helpers

In ASP.NET Core MVC, the developer can define the so-called tag helpers that either enhance existing HTML tags with new tag attributes or define new tags. While Razor views are compiled, any tag is matched against existing tag helpers. When a match is found, the source tag is replaced with HTML created by the tag helpers. Several tag helpers may be defined for the same tag. They are all executed in an order that can be configured with a `priority` attribute associated with each tag helper.

All tag helpers defined for the same tag may cooperate during the processing of each tag instance because they are passed as a shared data structure where each of them may apply a contribution. Usually, the final tag helper that is invoked processes this shared data structure to produce the output HTML.

Tag helpers are classes that inherit from the `TagHelper` class. This topic doesn't discuss how to create new tag helpers but introduces the main predefined tag helpers that come with ASP.NET Core MVC. A complete guide on how to define tag helpers is available in the official documentation that is referenced in the *Further reading* section.

To use a tag helper, you must declare the .dll file containing it with a declaration like in the following:

```
@addTagHelper *, Dll.Complete.Name
```

If you would like to use just one of the tag helpers defined in the .dll file, you must replace * with the tag name.

The preceding declaration can be placed either in each view that uses the tag helpers defined in the library or, once and for all, in the _ViewImports.cshtml file in the root of the Views folder. As a default, _ViewImports.cshtml adds all predefined ASP.NET Core MVC tag helpers with the following declaration:

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

The anchor tag is enhanced with attributes that automatically compute the URL and that invoke a specific action method with given parameters, as shown here:

```
<a asp-controller="{controller name}"  
asp-action="{action method name}"  
asp-route-{action method parameter1}="value1"  
...  
asp-route-{action method parameterN}="valuen">  
    put anchor text here  
</a>
```

A similar syntax is added to the form tags:

```
<form asp-controller="{controller name}"  
asp-action="{action method name}"  
asp-route-{action method parameter1}="value1"  
...  
asp-route-{action method parameterN}="valuen"  
...  
>  
...
```

The script tag is enhanced with attributes that allow falling back to a different source if the download fails. Typical usage is to download scripts from some cloud service to optimize the browser cache and to fall back to a local copy of the script in case of failure. The following code uses the fallback technique to download the bootstrap JavaScript file:

```
<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+8zzc5TX0GRTLCCmivcbNJWm2rs5C8PRhcEn3czEjhAO9o">
</script>
```

asp-fallback-test contains a JavaScript test that verifies whether the download succeeded. In the preceding example, the test verifies whether a JavaScript object has been created.

The environment tag can be used to select different HTML for different environments (development, staging, and production). Its typical usage is selecting the debug version of JavaScript files during development, as shown in this example:

```
<environment include="Development">
    {*development version of JavaScript files*}
</environment>
<environment exclude="Development">
    {*development version of JavaScript files *}
</environment>
```

There is also a cache tag, which caches its content in memory to optimize rendering speed:

```
<cache>
    {* heavy to compute content to cache *}
</cache>
```

As a default, content is cached for 20 minutes, but the tag has attributes to define when the cache expires, such as expires-on="`{datetime}`", expires-after="`{timespan}`", and expires-sliding="`{timespan}`". Here, the difference between expires-sliding and expires-after is that, in the second attribute, the expiration time count is reset each time the content is requested. The vary-by attribute causes the creation of a different cache entry for each different value passed to vary-by. There are also attributes such as vary-by-header—which creates a different entry for each different value assumed by the request header specified in the attribute, vary-by-cookie, and so on.

All input tags, that is, `textarea`, `input`, and `select`, have an `asp-for` attribute that accepts a properties path rooted in the view `ViewModel` as their value. For instance, if the view has a `Person` `ViewModel`, we may have something like this:

```
<input type="text" asp-for="Address.Town"/>
```

The first effect of the preceding code is to assign the value of the `Town` nested property to the `value` attribute of the `input` tag. In general, if the value is not a string, it is converted into a string using the current request culture.

However, it also sets the name of the input field to `Address.Town` and the ID of the input field to `Address_Town` since dots are not allowed in tag IDs.

A prefix can be added to these standard names by specifying it in `ViewData.TemplateInfo.HtmlFieldPrefix`. For instance, if the previous property is set to `MyPerson`, the name becomes `MyPerson.Address.Town`.

If the form is submitted to an action method that has the same `Person` class as one of its parameters, the name `Address.Town` given to the `input` field will cause the `Town` property of this parameter to be filled with the `input` field. In general, the string contained in the `input` field is converted into the type of the property they are matched with using the current request culture. Summing up, names of `input` fields are created in such a way that a complete `Person` model can be recovered in the action method when the HTML page is posted.

The same `asp-for` attribute can be used in a `label` tag to cause the label to refer to the `input` field with the same `asp-for` value.

The following code is an example of an `input`/`label` pair:

```
<label asp-for="Address.Town"></label>
<input type="text" asp-for="Address.Town"/>
```

When no text is inserted in the label, the text shown in the label is taken from a `Display` attribute that decorates the property (`Town`, in the example), if any; otherwise, the name of the property is used.

If `span` or `div` contains a `data-valmsg-for="Address.Town"` error attribute, then validation messages concerning the `Address.Town` input will be inserted automatically inside that tag. The validation framework is described in the *Connection between ASP.NET Core MVC and design principles* section.

It is also possible to automatically create a validation error summary by adding the attribute that follows to `div` or `span`:

```
asp-validation-summary="ValidationSummary.{All, ModelOnly}"
```

If the attribute is set to `ValidationSummary.ModelOnly`, only messages that are not associated with specific input fields will be shown in the summary, and if the value is `ValidationSummary.All`, all error messages will be shown.

The `asp-items` attribute allows specifying the options of `select` through `IEnumerable<SelectListItem>`, where each `SelectListItem` contains both the text and value of each option. `SelectListItem` contains also an optional `Group` property you can use to organize into groups the options shown in `select`.

The next topic shows how to reuse view code.

Reusing view code

ASP.NET Core MVC includes several techniques for reusing view code. The most important is the layout page.

In each web application, several pages share the same structure, for instance, the same main menu or the same left or right bar. In ASP.NET Core, this common structure is factored out in views called layout pages/views.

Each view can specify the view to use as its layout page with the following code:

```
@{  
    Layout = "_MyLayout";  
}
```

If no layout page is specified, a default layout page, defined in the `_ViewStart.cshtml` file located in the `Views` folder, is used. The default content of `_ViewStart.cshtml` is as follows:

```
@{  
    Layout = "_Layout";  
}
```

Therefore, the default layout page in the files scaffolded by Visual Studio is `_Layout.cshtml`, which is contained in the `Shared` folder.

The layout page contains the HTML shared with all of its children pages, the HTML page headers, and the page references to CSS and JavaScript files. The HTML produced by each view is placed inside of its layout place, where the layout page calls the `@RenderBody()` method, as shown in the following example:

```
...
<main role="main" class="pb-3">
    ...
    @RenderBody()
    ...
</main>
...
```

`ViewState` of each `View` is copied into `ViewState` of its layout page, so `ViewState` can be used to pass information to the view layout page. Typically, it is used to pass the view title to the layout page that use it to compose the page's title header, as shown here:

```
@*In the view *@

{@{
    ViewData["Title"] = "Home Page";
}

@*In the layout view*@
<head>
    <meta charset="utf-8" />
    ...
    <title>@ViewData["Title"] - My web application</title>
    ...
}
```

While the main content produced by each view is placed in a single area of its layout page, each layout page can also define several sections placed in different areas where each view can place further secondary contents.

For instance, suppose a layout page defines a `Scripts` section, as shown here:

```
...
<script src="~/js/site.js" asp-append-version="true"></script>

@RenderSection("Scripts", required: false)
...
```

Then, the view can use the previously defined section to pass some view specific JavaScript references, as shown here:

```
....  
@section scripts{  
    <script src="~/js/pages/pageSpecificJavaScript.min.js"></script>  
}  
....
```

If an action method is expected to return HTML to an Ajax call, it must produce an HTML fragment instead of a whole HTML page. Therefore, in this case, no layout page must be used. This is achieved by calling the `PartialView` method instead of the `View` method in the controller action method. `PartialView` and `View` have exactly the same overloads and parameters.

Another way to reuse view code is to factor out a view fragment that's common to several views into another view that is called by all previous views. A view can call another view with the `partial` tag, as shown here:

```
<partial name="_viewname" for="ModelProperty.NestedProperty"/>
```

The preceding code invokes `_viewname` and passes it to the object contained in `ModelProperty.NestedProperty` as its `ViewModel`. When a view is invoked by the `partial` tag, no layout page is used since the called view is expected to return an HTML fragment.

The `ViewData.TemplateInfo.HtmlFieldPrefix` property of the called view is set to the `"ModelProperty.NestedProperty"` string. This way, possible input fields rendered in `_viewname.cshtml` will have the same name as if they were rendered directly by the calling view.

Instead of specifying the `ViewModel` of `_viewname` through a property of the caller view (`ViewModel`), you can also pass an object directly that is contained in a variable or returned by a C# expression by replacing `for` with `model`, as shown in this example:

```
<partial name="_viewname" model="new MyModel{...})" />
```

In this case, `ViewData.TemplateInfo.HtmlFieldPrefix` of the called view keeps its default value, that is, the empty string.

A view can also call something more complex than another view, that is, another controller method that, in turn, renders a view. Controllers that are designed to be invoked by views are called **view components**. The following code is an example of component invocation:

```
<vc:[view-component-name] par1="par1 value" par2="parameter2 value">
</vc:[view-component-name]>
```

Parameter names must match the ones used in the view component method. However, both component name and parameter names must be translated into kebab case, that is, all characters must be transformed into lowercase and all characters that in the original name were in uppercase and each word must be separated by a -. For instance, `MyParam` must be transformed into `my-param`.

Actually, view components are classes that derive from the `ViewComponent` class. When a component is invoked, the framework looks for either an `Invoke` method or an `InvokeAsync` method and passes it to the parameters defined in the component invocation. `InvokeAsync` must be used if the method is defined as `async`; otherwise, we must use `Invoke`.

The following code is an example of a view component definition:

```
public class MyTestViewComponent : ViewComponent
{
    public async Task<IViewComponentResult> InvokeAsync(
        int par1, bool par2)
    {
        var model= ....
        return View("ViewName", model);
    }
}
```

The previously defined component must be invoked with a call such as the following:

```
<vc:my-test par1="10" par2="true"></my-test>
```

If the component is invoked by a view of a controller called `MyController`, `ViewName` is searched in the following paths:

- `/Views/MyController/Components/MyTest/ViewName`
- `/Views/Shared/Components/MyTest/ViewName`

Now, let's look at the new features that came along with .NET Core 3.0.

What is new in .NET Core 3.0 for ASP.NET Core?

The main innovation introduced by ASP.NET 3.0 is that the routing engine was factored out of the MVC engine and is now available for other handlers. In previous versions, routes and routing were a part of the MVC handler added with `app.UseMvc(...)`; that now has been replaced by `app.UseRouting()` plus `UseEndpoints(...)`, which can route requests not only to controllers but also to other handlers.

Endpoints and their associated handlers are now defined in `UseEndpoints`, as shown here:

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

`MapControllerRoute` associates patterns with controllers, but we may use also something such as `endpoints.MapHub<ChatHub>("/chat")`, which associates a pattern with a hub that handles WebSocket connections. In the previous section, we have seen that patterns can be associated also with custom handlers using `MapPost` and `MapGet`.

An independent router also allows us to add authorizations not only to controllers but also to any handler, as shown here:

```
MapGet("hello/{country}", context =>
    context.Response.WriteAsync(
        $"Selected country is {context.GetRouteValue("country")}")
    .RequireAuthorization(new AuthorizeAttribute(){ Roles = "admin" });
```

In the 3.0 version, ASP.NET Core has an independent JSON formatter and doesn't depend on the third-party Newtonsoft JSON serializer any more. However, if you have more sophisticated needs, you have still the option to replace the minimal ASP.NET Core JSON formatter with Newtonsoft JSON serializer by installing the `Microsoft.AspNetCore.Mvc.NewtonsoftJson` NuGet package and configuring controllers, as shown here:

```
services.AddControllersWithViews()
    .AddNewtonsoftJson();
```

Here, `AddNewtonsoftJson` has also an overload that accepts configuration options for the Newtonsoft JSON serializer:

```
.AddNewtonsoftJson(options =>
    options.SerializerSettings.ContractResolver =
        new CamelCasePropertyNamesContractResolver());
```

In previous versions, you were forced to add both controllers and views to the DI engine. In version 3, we can still inject both controllers and views with `services.AddControllersWithViews` but you can also add controllers with `AddControllers` if you are going to implement REST endpoints only.

In previous versions, ASP.NET Core had a custom implementation of `IWebHostBuilder`, which was completely independent of `HostBuilder`, which is the standard implementation of `IHostBuilder` used to configure a generic host. The following code shows how the ASP.NET Core host was configured before version 3.0:

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

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

ASP.NET Core 3.0, instead, uses a type that inherits from `HostBuilder` and adds the methods of `IWebHostBuilder`, as shown here:

```
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>();
            });
}
```

As a first step, `Host.CreateDefaultBuilder` creates a standard `HostBuilder`, `ConfigureWebHostDefaults` copies data contained in this `HostBuilder` in a derived class that also implements `IWebHostBuilder` and lets the developer configure this `IWebHostBuilder` in the action that it receives as an argument. `IWebHostBuilder`, once configured, is returned as `IHostBuilder` so that all web-specific things remain hidden.

This way development paths of `HostBuilder` and `WebHostBuilder` have been merged, and new enhancements and extension methods of `IHostBuilder` are automatically available also for the ASP.NET Core host.

Understanding the connection between ASP.NET Core MVC and design principles

The whole ASP.NET Core framework is built on top of the design principles and patterns that we analyzed in Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*, Chapter 6, *Interacting with Data in C# - Entity Framework Core*, Chapter 9, *Design Patterns and .NET Core Implementation*, Chapter 10, *Understanding the Different Domains in Software Solutions*, and Chapter 11, *Implementing Code Reusability in C# 8*.

All functionalities are provided through DI so that each of them can be replaced without affecting the remainder of the code. However, providers needed by ASP.NET Core pipeline modules are grouped into option objects instead of being added individually to the DI engine to conform to the SOLID Single Responsibility Principle.

Moreover, configuration data, instead of being available from a unique dictionary created from a configuration file, is organized into option objects thanks to the options framework we described in the first section of this chapter. This is an application of the SOLID Interface Segregation Principle.

However, ASP.NET Core also applies other patterns that are specific instances of the general Separation of Concerns principle, which is a generalization of the Single Responsibility Principle. They are as follows:

- The middleware modules architecture (ASP.NET Core pipeline)
- Factoring out validation and globalization from the application code
- The MVC pattern itself

We will analyze all of these in the various subsections that follow.

Advantages of the ASP.NET Core pipeline

The ASP.NET Core pipeline architecture has two important advantages:

- All different operations performed on the initial request are factored out into different modules, according to the Single Responsibility Principle.
- The modules that perform these different operations don't need to call each other because each module is invoked once and for all by the ASP.NET Core framework. This way, the code for each module is not required to perform any action that is connected with responsibilities assigned to other modules.

This ensures maximum independence of functionalities and simpler code. For instance, once authorization and authentication modules are on, no other module needs to worry about authorization anymore. Each controller code can focus on application-specific business stuff.

Server-side and client-side validation

Validation logic has been completely factored out from the application code and has been confined in the definition of validation attributes. The developer needs to just specify the validation rule to apply to each model property by decorating the property with an adequate validation attribute.

Validation rules are checked automatically when action method parameters are instantiated. Both errors and paths in the model (where they occurred) are then recorded in a dictionary that is contained in the `ModelState` controller property. The developer has the responsibility to verify whether there are errors by checking `ModelState.IsValid`, in which case the developer must return the same `ViewModel` to the same view so that the user can correct all errors.

Error messages are automatically shown in the view with no action required to the developer. The developer is only required to do the following:

- Add `span` or `div` with `data-valmsg-for` attribute next to each input field that will be automatically filled with the possible error.
- Add `div` with an `asp-validation-summary` attribute that will be automatically filled with the validation error summary. See the *Tag helpers* topic for more details.

It is enough to add some JavaScript references by invoking the `_ValidationScriptsPartial.cshtml` view with the `partial` tag to enable the same validation rules also on the client-side, so that errors are shown to the user before the form is posted to the server. Some predefined validation attributes are contained in the `System.ComponentModel.DataAnnotations` and `Microsoft.AspNetCore.Mvc` namespaces and include the following attributes:

- The `Required` attribute requires the user to specify a value for the property that it decorates. An implicit `Required` attribute is automatically applied to all non-nullable properties such as all floats, integers, and decimals since they can't have a `null` value.
- The `Range` attribute constrains numeric quantities within a range.
- They also include attributes that constrain string lengths.

Custom error messages can be inserted directly in the attributes, or attributes can refer to the property of resource types containing them.

The developer can define its custom attributes by providing the validation code both in C# and in JavaScript for client-side validation.

Attribute-based validation can be replaced by other validation providers, such as fluent validation that defines validation rules for each type using a fluent interface. It is enough to change a provider in a collection contained in the MVC options object that can be configured through an action passed to the `services.AddControllersWithViews` method. MVC options are configured as shown here:

```
services.AddControllersWithViews(o => {
    ...
    // code that modifies o properties
});
```

The validation framework automatically checks whether numeric and date inputs are well formatted according to the selected culture.

ASP.NET Core globalization

In multicultural applications, pages must be served according to the language and culture preferences of each user. Typically, multicultural applications can serve their content in a few languages, and they can handle dates and numeric formats in several more languages. In fact, while the content in all supported languages must be produced manually, .NET Core has the native capability of formatting and parsing dates and numbers in all cultures.

For instance, a web application might support unique content for all English-based cultures (en), but all known English-based cultures for numbers and dates formats (en-US, en-GB, en-CA, and so on).

The culture used for numbers and dates in a .NET thread is contained in the `Thread.CurrentCulture` property. Hence, by setting this property to `new CultureInfo("en-CA")`, numbers and dates will be formatted/parsed according to the Canadian culture. `Thread.CurrentThread.CurrentUICulture`, instead, decides the culture of the resource files, that is, it selects a culture-specific version of each resource file or view. Accordingly, a multicultural application is required to set the two cultures associated to the request thread and to organize multilingual content into language dependent resource files and/or views.

According to the Separation of Concerns principle, the whole logic used to set the request culture according to the user preferences is factored out into a specific module of the ASP.NET Core pipeline. To configure this module, as a first step, we set the supported date/numbers cultures, as in the following example:

```
var supportedCultures = new[]
{
    new CultureInfo("en-AU"),
    new CultureInfo("en-GB"),
    new CultureInfo("en"),
    new CultureInfo("es-MX"),
    new CultureInfo("es"),
    new CultureInfo("fr-CA"),
    new CultureInfo("fr"),
    new CultureInfo("it-CH"),
    new CultureInfo("it")
};
```

Then, we set the languages supported for the content. Usually, a version of the language that is not specific for any country is selected to keep the number of translations small enough, as shown here:

```
var supportedUICultures = new[]
{
    new CultureInfo("en"),
    new CultureInfo("es"),
    new CultureInfo("fr"),
    new CultureInfo("it")
};
```

Then, we add the culture middleware to the pipeline, as shown here:

```
app.UseRequestLocalization(new RequestLocalizationOptions
{
    DefaultRequestCulture = new RequestCulture("en", "en"),
    // Formatting numbers, dates, etc.
    SupportedCultures = supportedCultures,
    // UI strings that we have localized.
    SupportedUICultures = supportedUICultures,
    FallBackToParentCultures = true,
    FallBackToParentUICultures = true
});
```

If the culture requested by the user is explicitly found among the ones listed in `supportedCultures` or `supportedUICultures`, it is used without modifications.

Otherwise, since `FallBackToParentCultures` and `FallBackToParentUICultures` are true, the parent culture is tried, that is, for instance, if the required `fr-FR` culture is not found among those listed, then the framework searches for its generic version, `fr`. If this attempt also fails, the framework uses the cultures specified in `DefaultRequestCulture`.

By default, the culture middleware searches the culture selected for the current user with three providers that are tried in the order shown here:

1. The middleware looks for the `culture` and `ui-culture` query string parameters.
2. If the previous step fails, the middleware looks for a cookie named `.AspNetCore.Culture`, the value of which is expected to be as in this example: `c=en-US|uic=en`.
3. If both previous steps fail, the middleware looks for the `Accept-Language` request header sent by the browser, which can be changed in the browser settings, and that is initially set to the operating system culture.

With the preceding strategy, the first time a user requests an application page, the browser culture is taken (the provider listed in step 3). Then, if the user clicks a language-change link with the right query string parameters, a new culture is selected by provider 1. Usually, after a language link is clicked, the server also generates a language cookie to remember the user's choice through provider 2.

The simplest way to provide content localization is to provide a different view for each language. Hence, if we would like to localize the `Home.cshtml` view for different languages, we must provide views named `Home.en.cshtml`, `Home.es.cshtml`, and so on. If no view specific for the `ui-culture` thread is found, the not localized `Home.cshtml` version of the view is chosen.

View localization must be enabled by calling the `AddViewLocalization` method, as shown here:

```
services.AddControllersWithViews()
    .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix)
```

Another option is to store simple strings or HTML fragments in resource files specific for all supported languages. The usage of resource files must be enabled by calling the `AddLocalization` method in the `configure services` section, as shown here:

```
services.AddLocalization(options =>
    options.ResourcesPath = "Resources");
```

`ResourcesPath` is the root folder where all resource files will be placed. If it is not specified, the empty string is assumed, and resource files will be placed in the web application root. Resource files for a specific view, say the `/Views/Home/Index.cshtml` view, must have a path like this:

```
<ResourcesPath>/Views/Home/Index.<culture name>.resx
```

Hence, if `ResourcesPath` is empty, resources must have the `/Views/Home/Index.<culture name>.resx` path, that is, they must be placed in the same folder as the view.

Once key-value pairs to all resource files associated with a view are added, localized HTML fragments can be added to the view as follows:

- Inject `IViewLocalizer` in the view with `@inject IViewLocalizer Localizer`.
- Where needed, replace the text in the View with accesses to the `Localizer` dictionary, such as `Localizer["myKey"]`, where "myKey" is a key used in the resource files.

The following code shows an example of the `IViewLocalizer` dictionary:

```
@{  
    ViewData["Title"] = Localizer["HomePageTitle"];  
}  
<h2>@ViewData["MyTitle"]</h2>
```

If localization fails because the key is not found in the resource file, the key itself is returned. Strings used in data annotation, such as validation attributes, are used as a key in resource files if data annotation localization is enabled, as shown here:

```
services.AddControllersWithViews()  
    .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix)  
    .AddDataAnnotationsLocalization();
```

Resource files for data annotations applied to a class whose full name is, say, `MyWebApplication.ViewModels.Account.RegisterViewModel`, must have the following path:

```
<ResourcesPath>/ViewModels/Account/RegisterViewModel.<culture name>.resx
```

It is worth to point out that the first segment of the namespace that corresponds to the `.dll` application name is replaced by `ResourcePath`. If `ResourcesPath` is empty and if you use the default namespaces created by Visual Studio, then resource files must be placed in the same folder of the classes they are associated with.

It is possible to localize strings and HTML fragments in controllers or wherever dependencies can be injected by associating each group of resource files with a type, such as `MyType`, and then injecting either `IHtmlLocalizer<MyType>` for HTML fragments or `IStringLocalizer<MyType>` for strings that need to be HTML encoded.

Their usage is identical to the usage of `IViewLocalizer`. The path of the resource files associated with `MyType` is computed as in the case of data annotations. If you would like to use a unique group of resource files for the whole application, a common choice is to use the `Startup` class as the reference type (`IStringLocalizer<Startup>` and `IHtmlLocalizer<Startup>`). Another common choice is the creation of various empty classes to use as reference types for various groups of resource files.

After having learned how to manage globalization in your ASP.NET Core projects, in the next subsection, we can move to the description of the more important pattern used by ASP.NET Core MVC to enforce the *Separation of Concerns*, the MVC pattern itself.

The MVC pattern

MVC is a pattern for the implementation of presentation layers of a web application. The basic idea is to apply a *Separation of Concerns* between the logic of the presentation layer and its graphics. Logic is taken care of by controllers, while graphics are factored out into views. Controllers and views communicate through the model, which is often called the ViewModel to distinguish it from the models of the business and data layers.

However, what is the logic of a presentation layer? In Chapter 1, *Understanding the Importance of Software Architecture*, we saw that software requirements can be documented with use cases that describe the interaction between the user and the system. Roughly, the logic of the presentation layer consists of the management of use cases, hence, roughly, use cases are mapped to controllers and every single operation of a use case is mapped to an action method of those controllers. Hence, controllers take care of managing the protocol of interaction with the user and rely on the business layer for any business processing involved during each operation.

Each action method receives data from the user, performs some business processing and, depending on the results of this processing, decides what to show to the user and encodes it in the ViewModel. Views receive ViewModels that describe what to show to the user and decide the graphics to use, that is, HTML to use.

What are the advantages of separating logic and graphics into two different components? The main advantages are listed here:

- Changes in graphics do not affect the remainder of the code, so you can experiment with various graphic options to optimize the interaction with the user without putting the reliability of the remainder of the code at risk.
- The application can be tested by instantiating controllers and passing the parameters, with no need to use testing tools that operate on the browser pages. In this way, tests are easier to implement. Moreover, they do not depend on the way graphics are implemented, so they do not need to be updated each time graphic changes.
- It is easier to split the job between developers that implements controllers and graphic designers that implements views. Often, graphical designers have difficulties with Razor, so they might just furnish an example HTML page that developers transform into Razor views that operate on the actual data.

Now, let's look at how to create a web app in ASP.NET Core MVC.

Use case – implementing a web app in ASP.NET Core MVC

In this section, as an example of the ASP.NET Core application, we will implement the administrative panel for managing destinations and packages of the WWTravelClub book use case. The application will be implemented with the **Domain-Driven Design (DDD)** approach described in chapter 10, *Understanding the Different Domains in Software Solutions*, therefore, a good understanding of that chapter is a fundamental prerequisite for reading this section. The subsections that follow describe, the overall application specifications and organization and then the various application parts.

Defining application specifications

The destinations and packages have been described in chapter 6, *Interacting with Data in C# - Entity Framework Core*. Here, we will use exactly the same data model, with the necessary modifications to adapt it to the DDD approach. The administrative panel must allow packages, a destinations listing, and CRUD operations on them. To simplify the application, the two listings will be quite simple: the application will show all destinations sorted according to their names and all packages sorted starting from the ones with a higher-end validity date.

Moreover, we suppose the following things:

- The application that shows destinations and packages to the user shares the same database used by the administrative panel. Since only the administrative panel application needs to modify data, there will be just one write copy of the database with several read-only replicas.
- Price modifications and package deletions are immediately used to update the user shopping carts. For this reason, the administrative application must send asynchronous communications about price changes and package removals. We will not implement the whole communication logic, but we will just add all such events to an event table, which should be used as input to a parallel thread in charge of sending these events to all relevant microservices.

Here, we will give the full code for just the package management, while most of the code for destination management is left as an exercise for the reader. The full code is available in the chapter 13 folder of the GitHub repository associated with this book. In the remainder of the section, we will describe the application overall organization and we will discuss some relevant samples of code.

Defining the application architecture

The application is organized along with the guidelines described in Chapter 10, *Understanding Different Domains in a Software Solution*, considering the DDD approach and using SOLID principles to map your domain sections. That is, the application is organized within three layers, each implemented as a different project:

- There's a data layer that contains repository implementation and classes describing database entities. It is a .NET Core library project. However, since it needs some HTTP stack interfaces and classes, we must add a reference not only to the .NET Core SDK but also to the ASP.NET Core SDK. This can be done as follows:
 1. Right-click on the project icon in the solution explorer and select **Edit project file**.
 2. In the edit window, replace `<Project Sdk="Microsoft.NET.Sdk">` with `<Project Sdk="Microsoft.NET.Sdk.web">` and save.
 3. Since, after this modification, Visual Studio automatically turns this library project into an application project, please right-click on the project again and select **Properties**. In the project property window, replace **Console Application** with **Class Library** in the **Output Type** drop-down list.
- There's also a domain layer that contains repository specifications, that is, interfaces that describe repository implementations and DDD aggregates. In our implementation, we decided to implement aggregates by hiding forbidden operations/properties of root data entities behind interfaces. Hence, for instance, the `Package` data layer class, which is an aggregate root, has a corresponding `IPackage` interface in the domain layer that hides all the property setters of the `Package` entity. The domain layer also contains the definitions of all the domain events, while the corresponding event handlers are defined in the application layer.
- Finally, there's the application layer, that is, the ASP.NET Core MVC application, where we define DDD queries, commands, command handlers, and event handlers. Controllers fill query objects and execute them to get ViewModels they can pass to views. They update storage by filling command objects and executing their associated command handlers. In turn, command handlers use `IRepository` interfaces and `IUnitOfWork` coming from the domain layer to manage and coordinate transactions.

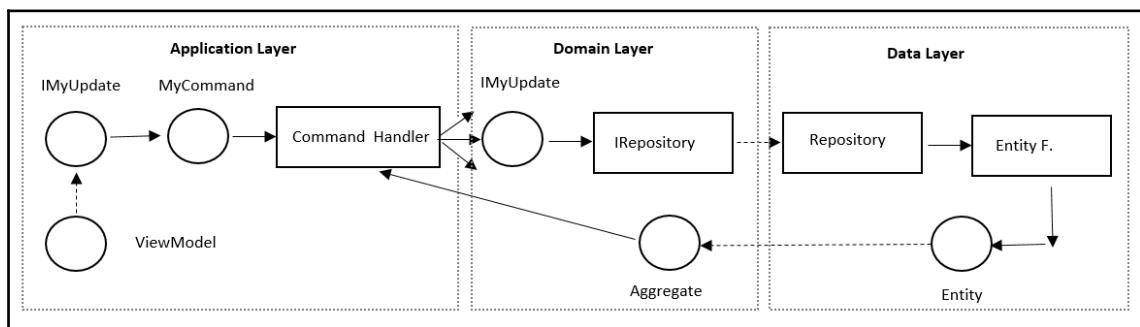
The application uses the Query Command Segregation pattern; therefore, it uses command objects to modify the storage and the query object to query it.

The query is simple to use and implement: controllers fill their parameters and then call their execution methods. In turn, query objects have direct LINQ implementations that project results directly on the ViewModels used by the controller Views with `Select` LINQ methods. You may also decide to hide the LINQ implementation behind the same repository classes used for the storage update operations.

However, since repositories don't know anything about ViewModels, which is presentation layer stuff, in this case, you are forced to use intermediate objects defined in the domain layer (DTOs), which must then be copied into ViewModels. This would make the definition of new queries and the modification of existing queries time-consuming operations since you are forced to modify several classes. In any case, it is good practice to hide query objects behind interfaces so their implementations can be replaced by fake implementations when you test controllers.

The chain of objects and calls involved in the execution of commands, instead, is more complex since it requires the construction and modification of aggregates and the definition of the interaction between several aggregates and between aggregates and other applications through domain events.

The following diagram is a sketch of how storage update operations are performed:



1. A controller's action method receives one or more ViewModels and performs validation.
2. One or more ViewModels containing changes to apply are hidden behind interfaces defined in the domain layer. They are used to fill the properties of a command object.

3. A command handler matching the previous command is retrieved via DI in the controller action method (through the `[FromServices]` parameter attribute we described in the *Controllers and ViewModels* subsection). Then, the handler is executed.
4. When creating the command handler discussed in *step 3*, the ASP.NET Core DI engine automatically injects all parameters declared in its constructor. In particular, it injects all `IRepository` implementations needed to perform all command handler transactions. The command handler performs its job by calling the methods of these `IRepository` implementations received in its constructor to build aggregates and to modify the built aggregates. Aggregates either represent already-existing entities or newly created ones. Handlers use the `IUnitOfWork` interface contained in each `IRepository`, and the concurrency exceptions returned by the data layer to organize their operations as transactions. It is worth pointing out that each aggregate has its own `IRepository`, and that the whole logic for updating each aggregate is defined in the aggregate itself and not in its associated `IRepository` to keep the code more modular.
5. Behind the scenes, in the data layer, `IRepository` implementations use Entity Framework to perform their job. Aggregates are implemented by root data entities hidden behind interfaces defined in the domain layer, while `IUnitOfWork` methods that handle transactions and that pass changes to the database are implemented with `DbContext` methods. In other words, `IUnitOfWork` is implemented with the application's `DbContext`.
6. Domain events are generated during each aggregate processing and added to the aggregates themselves by calling their `AddDomainEvent` methods. However, they are not triggered immediately. Usually, they are triggered at the end of all the aggregates' processing and before changes are passed to the database; however, this is not a general rule.
7. The application handles errors by throwing exceptions. A more efficient approach would be to define a request-scoped object in the dependency engine where each application subpart may add its errors as domain events. However, while this approach is more efficient, it increases the complexity of the code and the application development time.

The Visual Studio solution is composed of three projects:

- There's a project containing the domain layer called `PackagesManagementDomain`, which is a standard 2.0 library.
- There's a project containing the whole data layer called `PackagesManagementDB`, which is a .NET Core 3.0 library.

- Finally, there's an ASP.NET Core MVC 3.0 project called PackagesManagement that contains both application and presentation layers. When you define this project, select **no authentication**, otherwise the user database will be added directly to the ASP.NET Core MVC project instead of adding it to the database layer. We will add the user database manually in the data layer.

Let's start by creating the PackagesManagement ASP.NET Core MVC project so that the whole solution has the same name as the ASP.NET Core MVC project. Then, let's add the other two library projects to the same solution.

Finally, let the ASP.NET Core MVC project reference both projects, while PackagesManagementDB references PackagesManagementDomain. We suggest you define your own projects and then copy the code of this book's GitHub repository into them as you progress through reading this section.

The next subsection describes the code of the PackagesManagementDomain data layer project.

Defining the domain layer

Once the PackagesManagementDomain standard 2.0 library project is added to the solution, let's add a Tools folder to the project root. Then, let's place there all DomainLayer tools contained in the code associated with chapter 10. Since the code contained in this folder uses data annotations and defines DI extension methods, we must also add references to the System.ComponentModel.Annotations and Microsoft.Extensions.DependencyInjection NuGet packages.

Then, we need an Aggregates folder containing all the aggregate definitions (remember, we implemented aggregates as interfaces), namely, IDestination, IPackage, and IPackageEvent. Here, IPackageEvent is the aggregate associated with the table where we will place events to be propagated to other applications.

As an example, let's analyze IPackage:

```
public interface IPackage : IEntity<int>
{
    void FullUpdate(IPackageFullEditDTO o);
    string Name { get; set; }

    string Description { get; }
    decimal Price { get; set; }
```

```
    int DurationInDays { get; }
    DateTime? StartValidityDate { get; }
    DateTime? EndValidityDate { get; }
    int DestinationId { get; }
}
```

It contains the same properties of the `Package` entity, which we saw in Chapter 6, *Interacting with Data in C# - Entity Framework Core*. The only differences are the following:

- It inherits from `IEntity<int>`, which furnishes all basic functionalities of aggregates.
- It has no `Id` property since it is inherited from `IEntity<int>`.
- All properties are read-only, and it has an `Update` method since all aggregates can only be modified through update operations defined in the user domain (in our case, the `Update` method)

Now, let's also add a `DTOs` folder. Here, we place all interfaces used to pass updates to the aggregates. Such interfaces are implemented by the application layer ViewModels used to define such updates. In our case, it contains `IPackageFullEditDTO`, which we can use to update existing packages. If you would like to add the logic to manage destinations, you must define an analogous interface for the `IDestination` aggregate.

An `IRepository` folder contains all repository specifications, namely `IDestinationRepository`, `IPackageRepository`, and `IPackageEventRepository`. Here, `IPackageEventRepository` is the repository associated with the `IPackageEvent` aggregate. As an example, let's have a look at the `IPackageRepository` repository:

```
public interface IPackageRepository:
    IRepository<IPackage>
{
    Task<IPackage> Get(int id);
    IPackage New();
    Task<IPackage> Delete(int id);
}
```

Repositories always contain just a few methods since all business logic should be represented as aggregate methods, in our case, just the methods to create a new package, to retrieve an existing package, and to delete an existing package. The logic to modify an existing package is included in the `Update` method of `IPackage`.

Finally, as with all domain layer projects, `PackagesManagementDomain` contains an event folder with all domain event definitions. In our case, the folder is named `Events` and contains the package-deleted event and the price-changed event:

```
public class PackageDeleteEvent : IEventNotification
{
    public PackageDeleteEvent(int id, long oldVersion)
    {
        PackageId = id;
        OldVersion = oldVersion;
    }
    public int PackageId { get; private set; }
    public long OldVersion { get; private set; }
}

{
    public class PackagePriceChangedEvent : IEventNotification
    {
        public PackagePriceChangedEvent(int id, decimal price,
            long oldVersion, long newVersion)
        {
            PackageId = id;
            NewPrice = price;
            OldVersion = oldVersion;
            NewVersion = newVersion;
        }
        public int PackageId { get; private set; }
        public decimal NewPrice { get; private set; }
        public long OldVersion { get; private set; }
        public long NewVersion { get; private set; }
    }
}
```

When an aggregate sends all its changes to another application, it must have a version property. The application that receives the changes uses this version property to apply all changes in the right order. An explicit version number is necessary because changes are sent asynchronously, so the order they are received may differ from the order they were sent. For this purpose, events that are used to publish changes outside of the application have both `OldVersion` (the version before the change) and `NewVersion` (the version after the change) properties. Events associated with delete events have no `NewVersion`, since after being deleted, an entity can't store any versions.

The next subsection explains how all interfaces defined in the domain layer are implemented in the data layer.

Defining the data layer

The data layer project contains references to the `Microsoft.AspNetCore.Identity.EntityFrameworkCore` and `Microsoft.EntityFrameworkCore.SqlServer` NuGet packages, since we use Entity Framework Core with SQL server. It references `Microsoft.EntityFrameworkCore.Tools` and `Microsoft.EntityFrameworkCore.Design`, which is needed to generate database migrations, as explained in the *Entity Framework Core migrations* section of Chapter 6, *Interacting with Data in C# - Entity Framework Core*.

We have a `Models` folder that contains all database entities. They are similar to the ones in Chapter 6, *Interacting with Data in C# - Entity Framework Core*. The only differences are as follows:

- They inherit from `Entity<T>`, which contains all basic features of aggregates. Please notice that inheriting from `Entity<T>` is only needed for aggregate roots; all other entities must be defined as explained in Chapter 6, *Interacting with Data in C# - Entity Framework Core*. In our example, all entities are aggregate roots.
- They have no `Id` since it is inherited from `Entity<T>`.
- Some of them have an `EntityVersion` property that is decorated with the `[ConcurrencyCheck]` attribute. It contains the entity version that is needed for sending property all entity changes to other applications. The `ConcurrencyCheck` attribute is needed to prevent concurrency errors while updating the entity version without suffering the performance penalty implied by a transaction.

More specifically, when saving entity changes, if the value of a field marked with the `ConcurrencyCheck` attribute is different from the one that was read when the entity was loaded in memory, a concurrency exception is thrown to inform the calling method that someone else modified this value after the entity was read but before we attempted to save its changes. This way, the calling method can repeat the whole operation with the hope that, this time, no-one will write the same entity in the database during its execution.

It is worth analyzing the `Package` entity:

```
public class Package: Entity<int>, IPackage
{
    public void FullUpdate(IPackageFullEditDTO o)
    {
        if (IsTransient())
    }
```

```
        Id = o.Id;
        DestinationId = o.DestinationId;
    }
    else
    {
        if (o.Price != this.Price)
            this.AddDomainEvent(new PackagePriceChangedEvent(
                Id, o.Price, EntityVersion, EntityVersion+1));
    }
    Name = o.Name;
    Description = o.Description;
    Price = o.Price;
    DurationInDays = o.DurationInDays;
    StartValidityDate = o.StartValidityDate;
    EndValidityDate = o.EndValidityDate;
}
[MaxLength(128), Required]
public string Name { get; set; }
[MaxLength(128)]
public string Description { get; set; }
public decimal Price { get; set; }
public int DurationInDays { get; set; }
public DateTime? StartValidityDate { get; set; }
public DateTime? EndValidityDate { get; set; }
public Destination MyDestination { get; set; }
[ConcurrencyCheck]
public long EntityVersion{ get; set; }

public int DestinationId { get; set; }
}
```

The `FullUpdate` method is the only way to update the `IPackage` aggregate when the price changes add `PackagePriceChangedEvent` to the entity list of events.

The `MainDbContext.cs` file contains the data layer database context definition. It doesn't inherit from `DbContext` but from the following predefined context class:

```
IdentityDbContext<IdentityUser<int>, IdentityRole<int>, int>
```

This context defines the user's tables needed for the authentication. In our case, we opted for the `IdentityUser<T>` standard and the `IdentityRole<S>` respectively for users and roles and used integers for both the `T` and `S` Entity keys. However, we may also use classes that inherit from `IdentityUser` and `IdentityRole` and add then further properties.

In the `OnModelCreating` method, we must call `base.OnModelCreating(builder)` in order to apply the configuration defined in `IdentityDbContext`.

MainDbContext implements IUnitOfWork. The following code shows the implementation of all methods that start, rollback, and commit a transaction:

```
public async Task StartAsync()
{
    await Database.BeginTransactionAsync();
}

public async Task CommitAsync()
{
    Database.CommitTransaction();
}

public async Task RollbackAsync()
{
    Database.RollbackTransaction();
}
```

However, they are rarely used by command classes in a distributed environment since retrying the same operation until no concurrency exception is returned usually ensures better performance than transactions.

It is worth analyzing the implementation of the method that passes all changes applied to DBContext to the database:

```
public async Task<bool> SaveEntitiesAsync()
{
    try
    {
        return await SaveChangesAsync() > 0;
    }
    catch (DbUpdateConcurrencyException ex)
    {
        foreach (var entry in ex.Entries)
        {

            entry.State = EntityState.Detached;
        }
        throw ex;
    }
}
```

The preceding implementation just calls the SaveChangesAsync DBContext context method that saves all changes to the database, but then it intercepts all concurrency exceptions and detaches from the context all entities involved in the concurrency error. This way, next time a command retries the whole failed operation, their updated versions will be reloaded from the database.

The `Repositories` folder contains all repository implementations. It is worth analyzing the implementation of the `IPackageRepository.Delete` method:

```
public async Task<IPackage> Delete(int id)
{
    var model = await Get(id);
    if (model == null) return null;
    context.Packages.Remove(model as Package);
    model.AddDomainEvent(
        new PackageDeleteEvent(
            model.Id, (model as Package).EntityVersion));
    return model;
}
```

It reads the entity from the database and formally removes it from the `Packages` dataset. This will force the entity to be deleted in the database when changes are saved to the database. Moreover, it adds `PackageDeleteEvent` to the aggregate list of events.

The `Extensions` folder contains the `DBExtensions` static class that, in turn, defines two extension methods to be added to the application DI engine and the ASP.NET Core pipeline respectively. Once added to the pipeline, these two methods will connect the database layer to the application layer.

The `IServiceCollection` extension of `AddDbLayer` accepts (as its input parameters) the database connection string and the name of the `.dll` file that contains all migrations. Then, it does the following:

```
services.AddDbContext<MainDbContext>(options =>
    options.UseSqlServer(connectionString,
        b => b.MigrationsAssembly(migrationAssembly)));
```

That is, it adds the database context to the DI engine and defines its options, namely, that it uses SQL Server, the database connection string, and the name of the `.dll` that contains all migrations.

Then, it does the following:

```
services.AddIdentity<IdentityUser<int>, IdentityRole<int>>()
    .AddEntityFrameworkStores<MainDbContext>()
    .AddDefaultTokenProviders();
```

That is, it adds and configures all the types needed to handle database-based authentication. In particular, it adds the `UserManager` and `RoleManager` types, which the application layer can use to manage users and roles. `AddDefaultTokenProviders` adds the provider that creates the authentication tokens using data contained in the database when users log in.

Finally, it discovers and adds to the DI engine all repository implementations by calling the `AddAllRepositories` method that is defined in the DDD tools we added to the domain layer project.

The `UseDBLayer` extension method ensures migrations are applied to the database by calling `context.Database.Migrate()` and then populates the database with some initial objects. In our case, it uses `RoleManager` and `UserManager` to create an administrative role and an initial administrator. Then, it creates some sample destinations and packages.

To create migrations, we must add the aforementioned extension methods to the ASP.NET Core MVC `Startup.cs` file, as shown here:

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    services.AddRazorPages();
    services.AddDbLayer(
        Configuration.GetConnectionString("DefaultConnection"),
        "PackagesManagementDB");

    ...
}

public void Configure(IApplicationBuilder app,
    IWebHostEnvironment env, IServiceProvider serviceProvider)
{
    ...
    app.UseAuthentication();
    app.UseAuthorization();
    ...
    app.UseDBLayer(serviceProvider);
}
```

Please be sure that both the authorization and authentication modules have been added to the ASP.NET Core pipeline, otherwise, the authentication/authorization engine will not work.

Then, we must add the connection string to the `appsettings.json` file, as shown here:

```
{  
    "ConnectionStrings": {  
        "DefaultConnection":  
            "Server=(localdb)\\mssqllocaldb;Database=package-  
management;Trusted_Connection=True;MultipleActiveResultSets=true"  
    },  
    ...  
}
```

Finally, let's add `Microsoft.EntityFrameworkCore.Design` to the ASP.NET Core project.

We are forced to configure all database stuff in the startup project because migration tools use the startup project DI engine to create and apply migrations.

At this point, let's open Visual Studio Package Manager Console and select `PackageManagementDB` as the default project and then launch the following command:

```
Add-Migration Initial -Project PackagesManagementDB
```

The preceding command will scaffold the first migration. We may apply it to the database with the `Update-Database` command. Please note that if you copy the project from GitHub, you don't need to scaffold migrations since they have already been created, but you just need to update the database.

The next subsection describes the application layer.

Defining the application layer

As a first step, for simplicity, let's freeze the application culture to `en-US` by adding the code that follows to the ASP.NET Core pipeline:

```
app.UseAuthorization();  
  
// Code to add: configure the Localization middleware  
var ci = new CultureInfo("en-US");  
app.UseRequestLocalization(new RequestLocalizationOptions  
{  
    DefaultRequestCulture = new RequestCulture(ci),  
    SupportedCultures = new List<CultureInfo>  
    {  
        ci,  
    },  
},
```

```
= new List<CultureInfo>
{
    ci,
}
});
```

Then, let's create a `Tools` folder and place the `ApplicationLayer` code there, which you may find in the chapter 10 code of the GitHub repository associated with this book. With these tools in place, we can add the code that automatically discovers and adds all queries, command handlers, and event handlers to the DI engine, as shown here:

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    ...
    services.AddAllQueries(this.GetType().Assembly);
    services.AddAllCommandHandlers(this.GetType().Assembly);
    services.AddAllEventHandlers(this.GetType().Assembly);
}
```

Then, we must add a `Queries` folder to place all queries and their associated interfaces. As an example, let's have a look at the query that lists all packages:

```
public class PackagesListQuery : IPackagesListQuery
{
    MainDbContext ctx;
    public PackagesListQuery(MainDbContext ctx)
    {
        this.ctx = ctx;
    }
    public async Task<IEnumerable<PackageInfosViewModel>> GetAllPackages()
    {
        return await ctx.Packages.Select(m => new PackageInfosViewModel
        {
            StartValidityDate = m.StartValidityDate,
            EndValidityDate = m.EndValidityDate,
            Name = m.Name,
            DurationInDays = m.DurationInDays,
            Id = m.Id,
            Price = m.Price,
            DestinationName = m.MyDestination.Name,
            DestinationId = m.DestinationId
        })
        .OrderByDescending(m=> m.EndValidityDate)
        .ToListAsync();
    }
}
```

The query object is automatically injected in the application DB context. The `GetAllPackages` method uses LINQ to project all of the required information into `PackageInfosViewModel` and sorts all results in descending order on the `EndValidityDate` property.

`PackageInfosViewModel` is placed in the `Models` folder together with all other `ViewModels`. It is good practice to organize `ViewModels` in folders, by defining a different folder for each controller. It is worth analyzing the `ViewModel` used for editing packages:

```
public class PackageFullEditViewModel : IPackageFullEditDTO
{
    public PackageFullEditViewModel() { }

    public PackageFullEditViewModel(IPackage o)
    {
        Id = o.Id;
        DestinationId = o.DestinationId;
        Name = o.Name;
        Description = o.Description;
        Price = o.Price;
        DurationInDays = o.DurationInDays;
        StartValidityDate = o.StartValidityDate;
        EndValidityDate = o.EndValidityDate;
    }
    ...
}
```

It has a constructor that accepts an `IPackage` aggregate. This way, package data is copied into the `ViewModel` that is used to populate the edit view. It implements the `IPackageFullEditDTO` DTO interface defined in the domain layer. This way, it can be directly used to send `IPackage` updates to the domain layer.

All properties contain validation attributes that are automatically used by client-side and server-side validation engines. Each property contains a `Display` attribute that defines the label to give to the input field that will be used to edit the property. It is better to place the field labels in the `ViewModels` than placing them directly in the views since, this way, the same names are automatically used in all views that use the same `ViewModel`. The following code block lists all its properties:

```
public int Id { get; set; }
[StringLength(128, MinimumLength = 5), Required]
[Display(Name = "name")]
public string Name { get; set; }
[Display(Name = "package infos")]
[StringLength(128, MinimumLength = 10), Required]
public string Description { get; set; }
[Display(Name = "price")]
```

```
[Range(0, 100000)]
public decimal Price { get; set; }
[Display(Name = "duration in days")]
[Range(1, 90)]
public int DurationInDays { get; set; }
[Display(Name = "available from"), Required]
public DateTime? StartValidityDate { get; set; }
[Display(Name = "available to"), Required]
public DateTime? EndValidityDate { get; set; }
[Display(Name = "destination")]
public int DestinationId { get; set; }
```

The Commands folder contains all commands. As an example, let's have a look at the command used to modify packages:

```
public class UpdatePackageCommand: ICommand
{
    public UpdatePackageCommand(IPackageFullEditDTO updates)
    {
        Updates = updates;
    }
    public IPackageFullEditDTO Updates { get; private set; }
}
```

Its constructor must be invoked with an implementation of the `IPackageFullEditDTO` DTO interface, that, in our case, is the edit ViewModel we described before. Command handlers are placed in the Handlers folder. It is worth analyzing the command that updates packages:

```
IPackageRepository repo;
IEventMediator mediator;
public UpdatePackageCommandHandler(IPackageRepository repo, IEventMediator mediator)
{
    this.repo = repo;
    this.mediator = mediator;
}
```

Its constructor has automatically injected the `IPackageRepository` repository and an `IEventMediator` instance needed to triggers events handler. The following code also shows the implementation of the standard `HandleAsync` command handler method:

```
public async Task HandleAsync(UpdatePackageCommand command)
{
    bool done = false;
    IPackage model = null;
    while (!done)
```

```
        {
            try
            {
                model = await repo.Get(command.Updates.Id);
                if (model == null) return;
                model.FullUpdate(command.Updates);
                await mediator.TriggerEvents(model.DomainEvents);
                await repo.UnitOfWork.SaveEntitiesAsync();
                done = true;
            }
            catch (DbUpdateConcurrencyException)
            {

            }
        }
    }
}
```

Command operations are repeated until no concurrency exception is returned. `HandleAsync` uses the repository to get an instance of the entity to modify. If the entity is not found (it has been deleted), the commands stop its execution. Otherwise, all changes are passed to the retrieved aggregate. Immediately after the update, all events contained in the aggregate are triggered. In particular, if the price has changed, the event handler associated with the price change is executed. The concurrency check ensures that the package version is updated properly (by incrementing its previous version number by 1) and that the price changed event is passed the right version numbers.

Also, event handlers are placed in the `Handlers` folder. As an example, let's have a look at the price changed event handler:

```
public class PackagePriceChangedEventHandler :
    IEventHandler<PackagePriceChangedEventArgs>
{
    IPackageEventRepository repo;
    public PackagePriceChangedEventHandler(IPackageEventRepository repo)
    {
        this.repo = repo;
    }
    public async Task HandleAsync(PackagePriceChangedEventArgs ev)
    {
        repo.New(PackageEventType.CostChanged, ev.PackageId,
                 ev.OldVersion, ev.NewVersion, ev.NewPrice);
    }
}
```

The constructor has automatically injected the `IPackageEventRepository` repository that handles the database table with all events to send to other applications. The `HandleAsync` implementation simply calls the repository method that adds a new record to this table.

All records in the table handled by `IPackageEventRepository`, which can be retrieved and sent to all interested microservices by a parallel task defined in the DI engine with a call such as `services.AddHostedService<MyHostedService>();` as detailed in the *Using Generic Hosts* subsection of Chapter 5, *Applying Microservice Architecture to Your Enterprise Application*. However, this parallel task is not implemented in the GitHub code associated with this chapter.

The next subsection describes how controllers and views are designed.

Controllers and views

We need to add two more controllers to the one automatically scaffolded by Visual Studio, namely, `AccountController`, which takes care of user login/logout and registration, and `ManagePackageController` to handle all package-related operations. It is enough to right-click on the `Controllers` folder and then select **Add | Controller**. Then, choose the controller name and select the empty MVC controller to avoid that Visual Studio might scaffold code you don't need.

For simplicity, `AccountController` just has login and logout methods, so you can log in just with the initial administrator user. However, you can add further action methods that use the `UserManager` class to define, update, and delete users. The `UserManager` class can be provided through DI, as shown here:

```
private readonly UserManager<IdentityUser<int>> _userManager;
private readonly SignInManager<IdentityUser<int>> _signInManager;

public AccountController(
    UserManager<IdentityUser<int>> userManager,
    SignInManager<IdentityUser<int>> signInManager)
{
    _userManager = userManager;
    _signInManager = signInManager;
}
```

SignInManager takes care of login/logout operations. The Logout action method is quite simple and is shown here:

```
[HttpPost]
public async Task<IActionResult> Logout()
{
    await _signInManager.SignOutAsync();
    return RedirectToAction(nameof(HomeController.Index), "Home");
}
```

It just calls the `signInManager.SignOutAsync` method and then redirects the browser to the home page. To avoid it being called by clicking a link it is decorated with `HttpPost`, so it can only be invoked via a form submit.

Login instead requires two action methods. The first one is invoked via Get and shows the login form, where the user must place their username and password. It is shown here:

```
[HttpGet]
public async Task<IActionResult> Login(string returnUrl = null)
{
    ViewData["ReturnUrl"] = returnUrl;
    return View();
}
```

It receives `returnUrl` as its parameter when the browser is automatically redirected to the login page by the authorization module. This happens when an unlogged user tries to access a protected page. `returnUrl` is stored in the `ViewState` dictionary that is passed to the login view. The form in the login view passes it back, together with the username and password, to the controller when it is submitted, as shown in this code:

```
<form asp-route-returnurl="@ViewData["ReturnUrl"]" method="post">
...
</form>
```

The form post is intercepted by an action method with the same `Login` name but decorated with the `[HttpPost]` attribute, as shown here:

```
[ValidateAntiForgeryToken]           public async Task<IActionResult> Login(
    LoginViewModel model,
    string returnUrl = null)
{
    ...
}
```

The preceding method receives the `Login` model used by the login view together with the `returnUrl` query string parameter. The `ValidateAntiForgeryToken` attribute verifies a token (called an anti-forgery token) that MVC forms automatically add to a hidden field to prevent cross-site attacks.

As a first step, the action method logs the user out if they are already logged in:

```
if (User.Identity.IsAuthenticated)
{
    await _signInManager.SignOutAsync();
    return View(model);
}
```

Otherwise, it verifies whether there are validation errors, in which case, it shows the same view filled with the data of the `ViewModel` to let the user correct their errors:

```
if (ModelState.IsValid)
{
    ...
}
else
    // If we got this far, something failed, redisplay form
    return View(model);
```

If the model is valid, `_signInManager` is used to log in the user:

```
var result = await _signInManager.PasswordSignInAsync(
    model.UserName,
    model.Password, model.RememberMe,
    lockoutOnFailure: false);
```

If the result returned by the operation is successful, the action method redirects the browser to `returnUrl`, if not null, otherwise to the home page:

```
if (result.Succeeded)
{
    if (!string.IsNullOrEmpty(returnUrl))
        return LocalRedirect(returnUrl);
    else
        return RedirectToAction(nameof(HomeController.Index), "Home");
}
else
{
    ModelState.AddModelError(string.Empty,
        "wrong username or password");
    return View(model);
}
```

If, the login fails, it adds an error to `ModelState` and shows the same form to let the user try again.

`ManagePackagesController` contains an `Index` method that shows all packages in table format:

```
[HttpGet]
public async Task<IActionResult> Index(
    [FromServices] IPackagesListQuery query)
{
    var results = await query.GetAllPackages();
    var vm = new PackagesListViewModel { Items = results };
    return View(vm);
}
```

This action method is injected into the proper query object by DI, invokes it, and inserts the resulting `IEnumerable` in the `Items` property of a `PackagesListViewModel` instance. It is a good practice to include `IEnumerables` in ViewModels, so if necessary, other properties can be added without modifying the existing view code. Results are shown in a Bootstrap 4 table since Bootstrap 4 CSS is automatically scaffolded by Visual Studio.

The result is shown here:

Manage packages							
Delete	Edit	Destination	Name	Duration/days	Price	Available from	Available to
delete	edit	Florence	Winter in Florence	7	600.00	12/1/2019	2/1/2020
delete	edit	Florence	Summer in Florence	7	1000.00	6/1/2019	10/1/2019
New package							

The **New package** link (it is shaped like a **Bootstrap 4** button, but it is a link) invokes a controller `Create` action method, while the **delete** and **edit** links in each row invoke a `Delete` and `Edit` action method and pass them the ID of the package shown in the row. Here is the implementation of the two-row links:

```
@foreach(var package in Model.Items)
{
<tr>
<td>
    <a asp-controller="ManagePackages"
```

```
        asp-action="@nameof(ManagePackagesController.Delete)"  
        asp-route-id="@package.Id">  
        delete  
    </a>  
  </td>  
  <td>  
    <a asp-controller="ManagePackages"  
        asp-action="@nameof(ManagePackagesController.Edit)"  
        asp-route-id="@package.Id">  
        edit  
    </a>  
  </td>  
  ...  
  ...
```

It is worth describing the code of the `HttpGet` and `HttpPost` `Edit` action methods:

```
[HttpGet]  
public async Task<IActionResult> Edit(  
    int id,  
    [FromServices] IPackageRepository repo)  
{  
    if (id == 0) return RedirectToAction(  
        nameof(ManagePackagesController.Index));  
    var aggregate = await repo.Get(id);  
    if (aggregate == null) return RedirectToAction(  
        nameof(ManagePackagesController.Index));  
    var vm = new PackageFullEditViewModel(aggregate);  
    return View(vm);  
}
```

The `Edit` method of `HttpGet` uses `IPackageRepository` to retrieve the existing package. If the package is not found, that means it has been deleted by some other user, and the browser is redirected again to the list page to show the updated list of packages. Otherwise, the aggregate is passed to the `PackageFullEditViewModel` View Model that is rendered by the `Edit` view.

The view used to render the package must render `select` with all possible package destinations, so it needs an instance of the `IDestinationListQuery` query that was implemented to assist with the destination selection HTML logic. This query is injected directly in the view since it is a view responsibility to decide how to enable the user to select a destination. The code that injects the query and uses it is shown here:

```
@inject PackagesManagement.Queries.IDestinationListQuery destinationsQuery  
{@  
    ViewData["Title"] = "Edit/Create package";  
    var allDestinations =
```

```
        await destinationsQuery.AllDestinations();
    }
```

The action method that processes the post of the view form is given here:

```
[HttpPost]
public async Task<IActionResult> Edit(
    PackageFullEditViewModel vm,
    [FromServices] ICommandHandler<UpdatePackageCommand> command)
{
    if (ModelState.IsValid)
    {
        await command.HandleAsync(new UpdatePackageCommand(vm));
        return RedirectToAction(
            nameof(ManagePackagesController.Index));
    }
    else
        return View(vm);
}
```

If ModelState is valid, UpdatePackageCommand is created and its associated handler is invoked; otherwise, the View is displayed again to the user to enable them to correct all the errors.

The new links to the package list page and login page must be added to the main menu, which is in the _Layout view, as shown here:

```
<li class="nav-item">
    <a class="nav-link text-dark"
       asp-controller="ManagePackages"
       asp-action="Index">Manage packages</a>
</li>
@if (User.Identity.IsAuthenticated)
{
    <li class="nav-item">
        <a class="nav-link text-dark"
           href="javascript:document.getElementById('logoutForm').submit()">
            Logout
        </a>
    </li>
}
else
{
    <li class="nav-item">
        <a class="nav-link text-dark"
           asp-controller="Account" asp-action="Login">Login</a>
    </li>
}
```

`logoutForm` is an empty form whose only purpose is to send a post to the `Logout` action method. It has been added at the end of the body, as shown here:

```
@if (User.Identity.IsAuthenticated)
{
    <form asp-area="" asp-controller="Account"
          asp-action="Logout" method="post"
          id="logoutForm" ></form>
}
```

Now, the application is ready! You can run it, log in, and start to manage packages.

Summary

In this chapter, we analyzed the ASP.NET Core pipeline and various modules that compose an ASP.NET Core MVC application in detail, such as authentication/authorization, the options framework, and routing. Then, we described how controllers and Views map requests to response HTML. We also analyzed all the improvements introduced in version 3.0.

Finally, we analyzed all the design patterns implemented in the ASP.NET Core MVC framework, and, in particular, the importance of the Separation of Concerns principle and how ASP.NET Core MVC implements it with the ASP.NET Core pipeline and in its validation and globalization modules. Finally, we focused in more detail on the importance of Separation of Concerns between the presentation layer logic and graphics and how the MVC pattern ensures it.

The next chapter discusses best practices that will help you to program safe, simple, and maintainable software.

Questions

1. Can you list all the middleware modules scaffolded by Visual Studio in an ASP.NET Core project?
2. Does the ASP.NET Core pipeline module need to inherit from a base class or implement some interface?
3. Is it true that a tag must have just one tag helper defined for it or an exception is thrown?

4. Do you remember how to test in a controller if validation errors occurred?
5. What is the instruction in a layout view to include the output of the main view called?
6. How are secondary sections of the main view invoked in a layout view?
7. How does a controller invoke a view?
8. As a default, how many providers are installed in the globalization module?
9. Are ViewModels the only way for controllers to communicate with their invoked views?

Further reading

More details on the ASP.NET MVC framework are available in its official documentation at <https://docs.microsoft.com/en-US/aspnet/core/>. More details on Razor syntax can be found at <https://docs.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-3.0&tabs=visual-studio>.

Documentation on the creation custom tag helpers that were not discussed in this chapter can be found at <https://docs.microsoft.com/en-US/aspnet/core/mvc/views/tag-helpers/authoring>. Documentation on the creation of custom controller attributes can be found at <https://docs.microsoft.com/en-US/aspnet/core/mvc/controllers/filters>.

The definition of custom validation attributes is discussed in this article: <https://blogs.msdn.microsoft.com/mvpawardprogram/2017/01/03/asp-net-core-mvc/>.

For alternative approaches to the construction of presentation layers for web applications, the official documentation of Blazor is

at <https://dotnet.microsoft.com/apps/aspnet/web-apps/client>. A good introduction to all the techniques and tools needed to implement a modern JavaScript-based Single Page Application is found in this book: <https://www.packtpub.com/application-development/hands-typescript-c-and-net-core-developers>, which describes TypeScript, advanced JavaScript features, WebPack, and the Angular SPA framework.

4

Section 4: Programming Solutions for an Unavoidable Future Evolution

This section will focus on the necessity of delivering good code. The goal is to present techniques that will help you deliver good software that can be maintained and evolved continuously.

In Chapter 14, *Best Practices in Coding C# 8*, we will present some coding best practices to help developers program safe, simple, and maintainable software. The chapter also includes tips and tricks for coding in C#.

In Chapter 15, *Testing Your Code with Unit Test Cases and TDD*, we will present code testing techniques and principles, taking the view that a necessary element of software development is ensuring that an application is bug-free and that it satisfies all specifications. Besides that, it will present test-driven development, a software development methodology that gives unit tests a central role.

Then, in Chapter 16, *Using Tools to Write Better Code*, a bunch of the techniques and tools you need to achieve well-written code for your project will be presented. The idea here is that, although coding can be considered an art, writing understandable code is surely a philosophy, and there are tools that can help you with this.

This section includes the following chapters:

- Chapter 14, *Best Practices in Coding C# 8*
- Chapter 15, *Testing Your Code with Unit Test Cases and TDD*
- Chapter 16, *Using Tools to Write Better Code*

14

Best Practices in Coding C# 8

When you act as a software architect on a project, it is your responsibility to define and/or maintain a coding standard that will direct the team for programming according to the company's expectations. This chapter covers some of the best practices in coding that will help developers like you to program safe, simple, and maintainable software. It also includes tips and tricks for coding in C#.

The following topics will be covered in this chapter:

- How the complexity of your code can affect performance
- The importance of using a version control system
- Writing safe code in C#
- .NET core tips and tricks for coding
- Book use case – dos and don'ts in writing code

Technical requirements

This chapter requires the Visual Studio 2019 free community edition or better with all database tools installed.

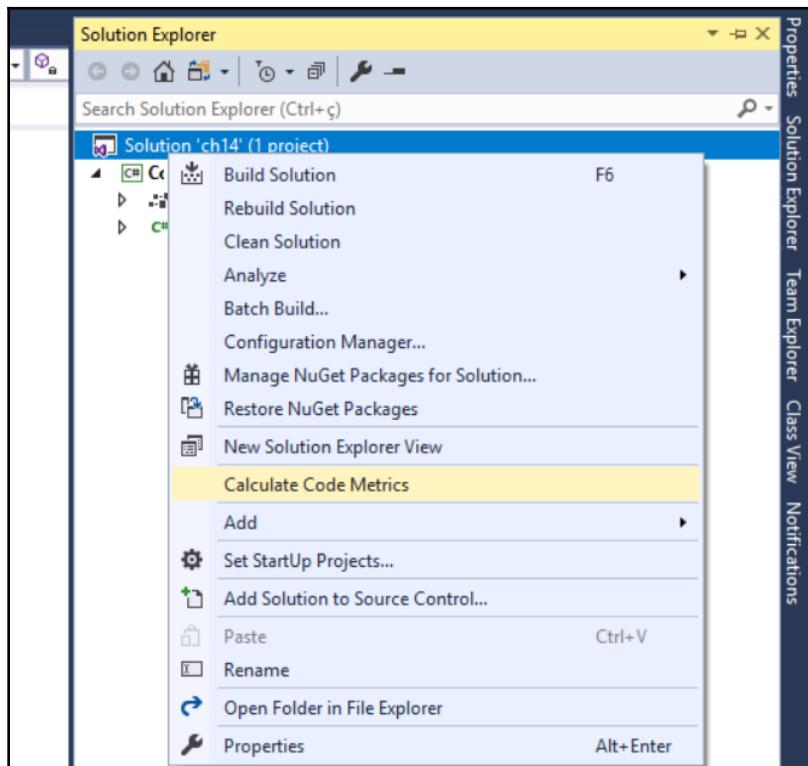
You will find the sample code of this chapter here: <https://github.com/PacktPublishing/Hands-On-Software-Architecture-with-CSharp-8/tree/master/ch14>.

The more complex your code is, the worse a programmer you are

For many people, a good programmer is one who writes complex code. However, the evolution of maturity in software development means there is a different way of thinking about it. The complexity does not mean a good job, it means poor code quality. Some incredible scientists and researchers confirm this theory and emphasize that professional code needs to be focused on time, of high quality, and within budget.

So, if you want to write good code, you need to keep the focus on how to do it, considering you are not the only one who will read it later. This is a good tip that changes the way you write code. This is how we will discuss each point of this chapter.

If your understanding of the importance of writing good code is aligned to the idea of simplicity and clarity while writing it, you should have to take a look at the Visual Studio tool **Code Metrics**:



The **Code Metrics** tool will deliver metrics that will give you insights about the quality of the software you are delivering. The metrics that the tool provides are listed here and can be found at this link <https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2019>:

- Maintainability index
- Cyclomatic complexity
- Depth of inheritance
- Class coupling
- Lines of code

The next subsections are focused on describing how they are useful in some real-life scenarios.

Maintainability index

This index indicates how easy it is to maintain the code—the easier the code, the higher the index (limited to 100). Easy maintenance is one of the key points to keep software in good health. It is obvious that any software will require changes in the future since change is inevitable. For this reason, consider refactoring your code if you have low levels of maintainability. Writing classes and methods dedicated to a single responsibility, avoiding duplicate code, and limiting the number of lines of code of each method are examples of how you can improve the maintainability index.

Cyclomatic complexity

The author of *Cyclomatic Complexity Metric* is Thomas J. McCabe. He defines the complexity of a software function according to the number of code paths available (graph nodes). The more paths you have, the more complex your function is. McCabe considers that each function must have a complexity score of less than 10. That means that, if the code has more complex methods, you have to refactor it, transforming parts of these codes into separate methods. There are some real scenarios where this behavior is easily detected:

- Loops inside loops
- Lots of consecutive `if-else`
- `switch` with code processing for each `case` inside the same method

For instance, look at the first version of this method for processing different responses of a credit card transaction. As you can check, the cyclomatic complexity is bigger than the number considered by McCabe as a basis. The reason why this happens is because of the number of `if-else` inside each case of the main switch:

```
static void Main()
{
    var billingMode = GetBillingMode();
    var messageResponse = ProcessCreditCardMethod();
    switch (messageResponse)
    {
        case "A":
            if (billingMode == "M1")
                Console.WriteLine($"Billing Mode {billingMode} for Message
                    Response {messageResponse}");
            else
                Console.WriteLine($"Billing Mode {billingMode} for Message
                    Response {messageResponse}");
            break;
        case "B":
            if (billingMode == "M2")
                Console.WriteLine($"Billing Mode {billingMode} for Message
                    Response {messageResponse}");
            else
                Console.WriteLine($"Billing Mode {billingMode} for Message
                    Response {messageResponse}");
            break;
        case "C":
            if (billingMode == "M3")
                Console.WriteLine($"Billing Mode {billingMode} for Message
                    Response {messageResponse}");
            else
                Console.WriteLine($"Billing Mode {billingMode} for Message
                    Response {messageResponse}");
            break;
        case "D":
            if (billingMode == "M4")
                Console.WriteLine($"Billing Mode {billingMode} for Message
                    Response {messageResponse}");
            else
                Console.WriteLine($"Billing Mode {billingMode} for Message
                    Response {messageResponse}");
            break;
        case "E":
            if (billingMode == "M5")
                Console.WriteLine($"Billing Mode {billingMode} for Message
                    Response {messageResponse}");
    }
}
```

```

        else
            Console.WriteLine($"Billing Mode {billingMode} for Message
                           Response {messageResponse}");
        break;
    case "F":
        if (billingMode == "M6")
            Console.WriteLine($"Billing Mode {billingMode} for Message
                           Response {messageResponse}");
        else
            Console.WriteLine($"Billing Mode {billingMode} for Message
                           Response {messageResponse}");
        break;
    case "G":
        if (billingMode == "M7")
            Console.WriteLine($"Billing Mode {billingMode} for Message
                           Response {messageResponse}");
        else
            Console.WriteLine($"Billing Mode {billingMode} for Message
                           Response {messageResponse}");
        break;
    case "S":
        if (billingMode == "M8")
            Console.WriteLine($"Billing Mode {billingMode} for Message
                           Response {messageResponse}");
        else
            Console.WriteLine($"Billing Mode {billingMode} for Message
                           Response {messageResponse}");
        break;
    default:
        Console.WriteLine("The result of processing is unknown");
        break;
    }
}
}

```

If you calculate the code metrics of this code, you will find a really bad result when it comes to cyclomatic complexity, as you can see in the following screenshot:

Code Metrics Results						
Hierarchy	Maintainability In...	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code	
CodeMetrics (Debug)	57	31	1	1	54	
{} CodeMetrics	57	31	1	1	54	
Program	57	31	1	1	54	
Main() : void	37	28		1	51	
GetBillingMode() : string	93	1		0	1	
ProcessCreditCardMethod() : strin	93	1		0	1	
Program()	100	1		0	1	

The code itself makes no sense, but the point here is to show you the number of improvements that can be made in order to write better code:

- The options from the `switch-case` could be written using `Enum`.
- Each case processing can be done in a specific method.
- `switch-case` can be substituted with `Dictionary<Enum, Method>`.

By refactoring this code with the preceding techniques, the result is a piece of code that is much easier to understand, as you can see in the following code snippet of its main method:

```
static void Main()
{
    var billingMode = GetBillingMode();
    var messageResponse = ProcessCreditCardMethod();
    Dictionary<CreditCardProcessingResult, CheckResultMethod>
    methodsForCheckingResult =
    GetMethodsForCheckingResult();
    if (methodsForCheckingResult.ContainsKey(messageResponse))
        methodsForCheckingResult[messageResponse](billingMode,
            messageResponse);
    else
        Console.WriteLine("The result of processing is unknown");
}
```

The full code can be found on the GitHub of this chapter and presents how lower-complexity code can be achieved. The following screenshot shows these results according to code metrics:

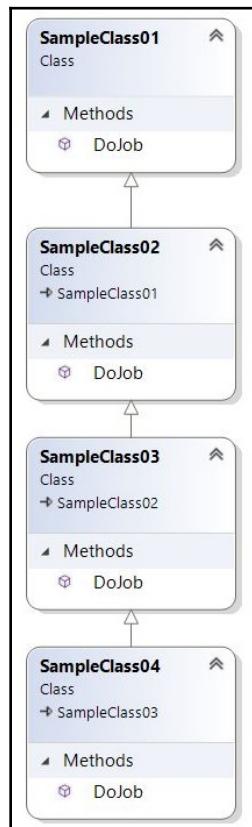
	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
CodeMetricsGoodCode (Debug)	91	24	1	5	146
{ } CodeMetricsGoodCode	90	22	1	5	116
Program	80	21	1	5	112
Main() : void	73	2		5	11
GetMethodsForCheckingResult() : Dictionary<CreditCardProces	62	1		3	15
CheckResultSucceed(BillingMode, CreditCardProcessingResult)	90	2		3	8
CheckResultG(BillingMode, CreditCardProcessingResult) : void	90	2		3	8
CheckResultF(BillingMode, CreditCardProcessingResult) : void	90	2		3	8
CheckResultE(BillingMode, CreditCardProcessingResult) : void	90	2		3	8
CheckResultD(BillingMode, CreditCardProcessingResult) : void	90	2		3	8
CheckResultC(BillingMode, CreditCardProcessingResult) : void	90	2		3	8
CheckResultB(BillingMode, CreditCardProcessingResult) : void	90	2		3	8
CheckResultA(BillingMode, CreditCardProcessingResult) : void	90	2		3	8
GetBillingMode() : BillingMode	100	1		1	9
ProcessCreditCardMethod() : CreditCardProcessingResult	100	1		1	9
Program.CheckResultMethod	100	1	1	2	1
{ } CodeMetricsGoodCode.Enums	93	2	1	0	30

As you can see in the preceding screenshot, there is a considerable reduction of complexity after refactoring. The key point here is that with the techniques applied, understanding of the code increased and the complexity decreased, proving the importance of cyclomatic complexity.

Depth of inheritance

This metric represents the number of classes connected to the one that is being analyzed. The more classes you have inherited, the worse your code is. This is similar to class coupling and indicates how difficult it is to change your code.

For instance, the following screenshot has four inherited classes:



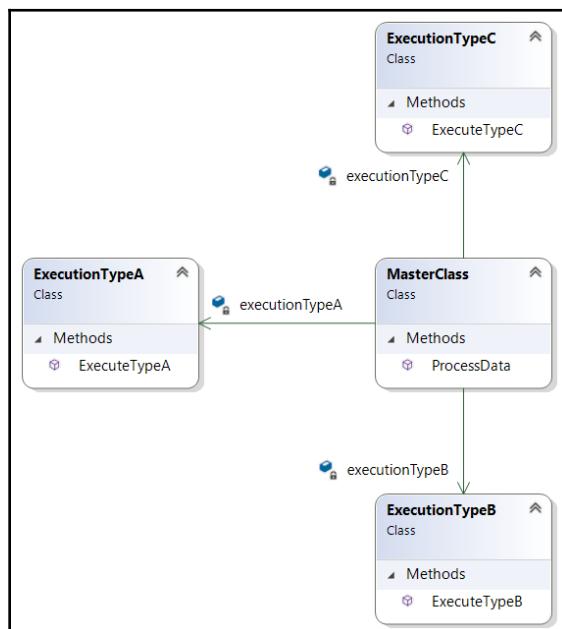
You can see in the following screenshot that the deeper class has the worse metric, considering there are three other classes that can change its behavior:

Hierarchy		Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
CodeMetricsBadCode (Debug)		95	36	4	13	250
{} CodeMetricsBadCode		90	32	1	13	194
{} CodeMetricsBadCode.SampleClasses		100	4	4	3	56
SampleClass01		100	1	1	0	10
SampleClass02		100	1	2	1	10
SampleClass03		100	1	3	1	10
SampleClass04		100	1	4	1	10

Inheritance is one of the basic object-oriented analysis principles. However, it can sometimes be bad for your code in that it can cause dependencies. So, if it makes sense to do so, instead of using inheritance, consider using aggregation.

Class coupling

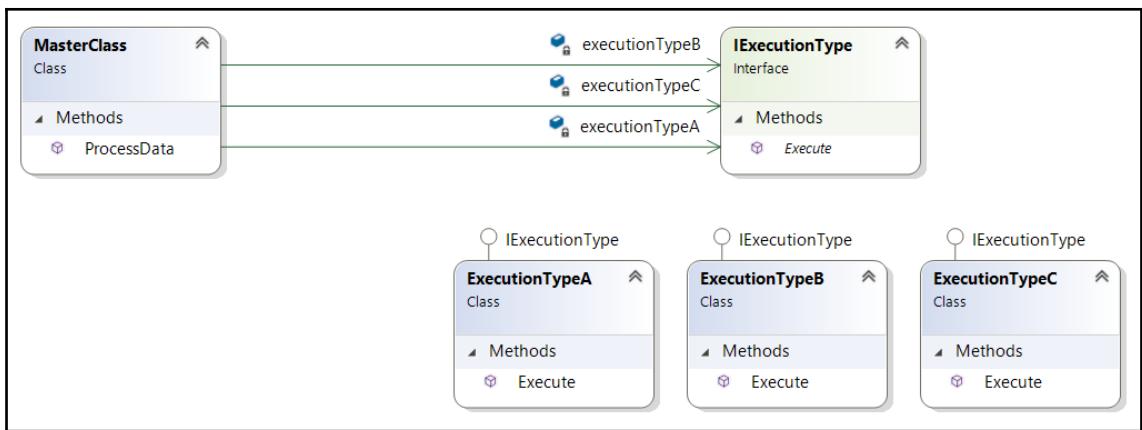
When you connect too many classes in a single class, obviously you will get coupling and this can cause bad maintenance of your code. For instance, see the following screenshot. It shows a design where aggregation has been done a lot. There is no sense to the code itself:



Once you have calculated the code metrics for the preceding design, you will see that the number of class coupling instances for the `ProcessData()` method, which calls `ExecuteTypeA()`, `ExecuteTypeB()`, and `ExecuteTypeC()`, equals three (3):

		Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
CodeMetricsBadCode (Debug)		96	40	4	16	302
{} CodeMetricsBadCode		90	32	1	13	194
{} CodeMetricsBadCode.CouplingSample		86	1	1	3	19
MasterClass		86	1	1	3	15
executionTypeA : ExecutionTypeA		93	0		1	1
executionTypeB : ExecutionTypeB		93	0		1	1
executionTypeC : ExecutionTypeC		93	0		1	1
ProcessData() : void		81	1		3	7

Some papers indicate that the maximum number of class coupling instances should be nine (9). With aggregation being a better practice than inheritance, the use of interfaces will solve class coupling problems. For instance, the same code with the following design will give you a better result:



Notice that using the interface in the design will allow you the possibility of increasing the number of execution types without increasing the **Class Coupling** of the solution:

Hierarchy		Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
CodeMetricsGoodCode (Debug)		94	29	1	9	206
{} CodeMetricsGoodCode		90	22	1	5	116
{} CodeMetricsGoodCode.CouplingSample		93	2	1	4	27
{} IExecutionType		100	1	0	0	4
MasterClass		86	1	1	4	15
executionTypeA : IExecutionType		93	0		2	1
executionTypeB : IExecutionType		93	0		2	1
executionTypeC : IExecutionType		93	0		2	1
ProcessData() : void		81	1	1	7	

As a software architect, you have to consider designing your solution to have more cohesion than coupling. The literature indicates that good software has low coupling and high cohesion. This is a basic principle that can guide you to a better architectural model.

Lines of code

This metric is useful to make you understand the size of the code you are dealing with. There is no way to connect lines of code and complexity since the number of lines is not indicative of that. On the other hand, the lines of code show the software size and software design. For instance, if you have too many lines of code in a single class (more than 1,000 lines of code—1 KLOC), it indicates that it is a bad design.

Using a version control system

You may find this topic in this book a bit obvious, but many people and companies still do not consider having a version control system as a basic tool for software development! The idea of writing about it is to force you to understand it. There is no architectural model or best practice that can save software development if you do not use a version control system.

In the last few years, we have been enjoying the advantages of online version control systems, such as GitHub, BitBucket, and Azure DevOps. The fact is, you have to have a tool like that in your software development life cycle and there is no reason to not have it anymore since most providers offer free versions for small groups. Even if you develop by yourself, these tools are useful for tracking your changes, managing your software versions, and guaranteeing the consistency and integrity of your code.

Dealing with version control systems in teams

The use of a version control system tool when you are alone is quite obvious. You want to keep your code safe. But this kind of system was definitely developed to solve team problems while writing code. For this reason, some features such as branching and merging were introduced to keep code integrity even in scenarios where the number of developers is quite big.

As a software architect, you will have to decide which branch strategy you will conduct in your team. Azure DevOps and GitHub suggest different ways to deliver that, and both of them are useful in some scenarios.

The information about how Azure DevOps team deals with it can be found here: <https://devblogs.microsoft.com/devops/release-flow-how-we-do-branching-on-the-vsts-team/>. GitHub describes its process here: <https://guides.github.com/introduction/flow/>. We have no idea about which is the one that best fits your needs, but we do want you to understand that you need to have a strategy for controlling your code.

Writing safe code in C#

C# can be considered a safe programming language by design. Unless you force it, there is no need for pointers, and memory release is, in most cases, managed by the garbage collector. Even so, some care should be taken so you can get better and safe results from your code. Let's have a look at them.

try-catch

Exceptions in coding are so frequent that you may have a way to manage them whenever they happen. try-catch statements are built to manage these exceptions and they are really important to keeping your code safe. There are a lot of cases where an application crashes and the reason for that is the lack of using try-catch. The following code shows an example of the lack of usage of the try-catch statement:

```
private static int CodeWithNoTryCatch(string textToConvert)
{
    return Convert.ToInt32(textToConvert);
}
```

On the other hand, bad `try-catch` usage can cause damage to your code too, especially because you will not see the correct behavior of that code and may misunderstand the results provided. The following code shows an example of an empty `try-catch` statement:

```
private static int CodeWithEmptyTryCatch(string textToConvert)
{
    try
    {
        return Convert.ToInt32(textToConvert);
    }
    catch
    {
        return 0;
    }
}
```

`try-catch` statements must always be connected to logging solutions, so that you can have a response from the system that will indicate the correct behavior and, at the same time, will not cause application crashes. The following code shows an ideal `try-catch` statement with logging management:

```
private static int CodeWithCorrectTryCatch(string textToConvert)
{
    try
    {
        return Convert.ToInt32(textToConvert);
    }
    catch (Exception err)
    {
        Logger.GenerateLog(err);
        return 0;
    }
}
```

As a software architect, you should conduct code inspections to fix this kind of behavior found in the code. Instability in a system is often connected to the lack of `try-catch` statements in the code.

try-finally and using

Memory leaks can be considered one of the worst behaviors of software. They cause instability, bad usage of computer resources, and undesired application crashes. C# tries to solve this with Garbage Collector, which automatically releases objects from memory as soon as it realizes the object can be freed.

Objects that interact with I/O are the ones that generally are not managed by Garbage Collector: filesystem, sockets, and so on. The following code is a sample of wrong usage of `FileStream` object, because it considers the Garbage Collector will release the memory used, but it will not:

```
private static void CodeWithIncorrectFileStreamManagement()
{
    FileStream file = new FileStream("C:\\file.txt", FileMode.CreateNew);
    byte[] data = GetFileData();
    file.Write(data, 0, data.Length);
}
```

Besides, it takes a while for Garbage Collector to interact with objects that need to be released and sometimes you may want to do it yourself. For both cases, the use of `try-finally` or `using` statements is the best practice:

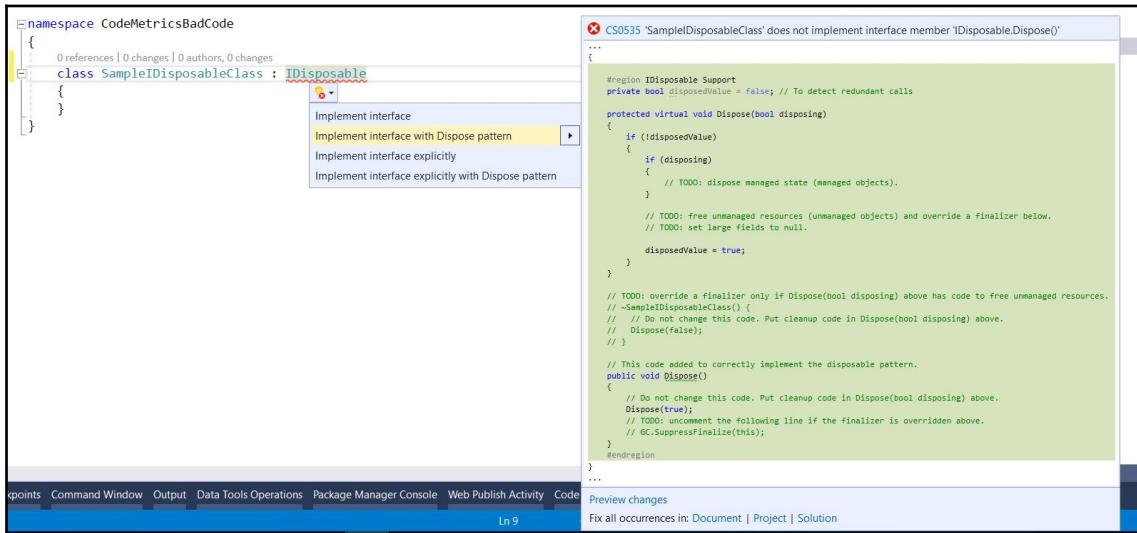
```
private static void CodeWithCorrectFileStreamManagementFirstOption()
{
    using (FileStream file = new FileStream("C:\\file.txt",
        FileMode.CreateNew))
    {
        byte[] data = GetFileData();
        file.Write(data, 0, data.Length);
    }
}

private static void CodeWithCorrectFileStreamManagementSecondOption()
{
    FileStream file = new FileStream("C:\\file.txt", FileMode.CreateNew);
    try
    {
        byte[] data = GetFileData();
        file.Write(data, 0, data.Length);
    }
    finally
    {
        file.Dispose();
    }
}
```

The preceding code shows exactly how to deal with objects that are not managed by Garbage Collector. You have both `try-finally` and `using` being implemented. As a software architect, you do need to pay attention to this kind of code. The lack of `try-finally` or `using` statements can cause huge damage to software behavior when it is running.

The IDisposable interface

The same way you will have trouble if you do not manage objects created inside a method with try-finally/using statements, objects created in a class that does not properly implement the `IDisposable` interface may cause memory leaks in your application. For this reason, when you have a class that deals with and creates objects, you should implement the disposable pattern to guarantee the release of all resources created by it:



The good news is that Visual Studio gives you the code snippet to implement this interface by just indicating it in your code and right-clicking on the **Quick Actions and refactorings** option, as you can see in the preceding screenshot. Once you have the code inserted, you need to follow the TODO instructions so that you have the correct pattern implemented.

.NET Core tips and tricks for coding

.NET Core implements some good features that help us to write better code. One of the most useful for having safer code is **dependency injection (DI)**, which was already discussed in Chapter 9, *Design Patterns and .NET Core Implementation*. There are some good reasons to consider this. The first one is because you will not need to worry about disposing of injected objects since you are not going to be the creator of them.

Besides, DI enables you to inject `ILogger`, a really useful tool for debugging exceptions that will need to be managed by try-catch statements in your code. Furthermore, programming in C# with .NET Core must follow the common good practices of any programming language. The following list shows some of them:

- **Classes, methods, and variables should have understandable names:** The name should explain everything the reader needs to know. There should be no need for an explanatory comment.
- **Methods cannot have high complexity levels:** Cyclomatic complexity should be checked so that methods do not have too many lines of code.
- **Duplicate code should be avoided:** There is no reason for having duplicate code in a high-level programming language like C#.
- **Objects should be checked before usage:** Since null objects can exist, the code must have null-type checking.
- **Constants and enumerators should be used:** A good way for avoiding magic numbers and text inside code is transforming this information into constants and enumerators, which generally are more understandable.
- **Unsafe code should be avoided:** Unless there is no other way to implement code, unsafe code should be avoided.
- **try-catch statements cannot be empty:** There is no reason for a `try-catch` statement without treatment in the `catch` area.
- **try-finally/using statements should always be used:** Even for objects where Garbage Collector will take care of the disposed-of object, consider disposing of objects that you were responsible for creating yourself.
- **At least public methods should be commented:** Considering that public methods are the ones used outside your library, they have to be explained for their correct external usage.
- **switch-case statements must have a default treatment:** Since the `switch-case` statement may receive an entrance variable unknown in some cases, the default treatment will guarantee that the code will not break in such a situation.

As a software architect, a good practice is to provide your developers with a code pattern that will be used by all programmers as a way to keep the style of the code consistent. You can use a code pattern as a checklist for coding inspection, which will enrich software code quality.

WWTravelClub – dos and don'ts in writing code

As a software architect, you have to define a code standard that matches the needs of the company you are working for.

In the sample project of this book (check out more about the WWTravelClub project in Chapter 1, *Understanding the Importance of Software Architecture*), this is no different. The way we decided to present the standard for it is describing a list of dos and don'ts that we followed while writing the samples we produced. It is worth mentioning that the list is a good way to start your standard and as a software architect, you should discuss this list with the developers you have in the team, so you can evolve it in a practical and good manner:

- DO write your code in English.
- DO follow C# coding standards with CamelCase.
- DO write classes, methods, and variables with understandable names.
- DO comment public classes, methods, and properties.
- DO use the `using` statement whenever possible.
- DO use `async` implementation whenever possible.
- DO ask for authorization before implementing unmanaged code.
- DO ask for authorization before implementing threads.
- DON'T write empty `try-catch` statements.
- DON'T write methods with more than a score of 10 of cyclomatic complexity.
- DON'T use `break` and `continue` inside `for/while/do-while/foreach` statements.
- DON'T use `goto` statements.

These dos and don'ts are simple to follow and, better than that, will yield great results for the code your team produces. In Chapter 16, *Using Tools to Write Better Code*, we will discuss the tools to help you to implement these rules.

Summary

We discussed during this chapter some important tips for writing safe code. This chapter introduced a tool for analyzing code metrics, so you can manage the complexity and maintainability of the software you are developing. To finish, we presented some good tips to guarantee your software will not crash due to memory leaks and exceptions. In real life, a software architect will always be asked to solve this kind of problem.

In the next chapter, we will learn about some unit testing techniques, the principles of unit testing, and a software process model that focuses on C# test projects.

Questions

1. Why do we need to care about maintainability?
2. What is cyclomatic complexity?
3. List the advantages of using a version control system.
4. What is the difference between `try-catch`, `try-finally`, and `try-catch-finally`?
5. What is Garbage Collector?
6. What is the importance of implementing the `IDisposable` interface?
7. What advantages do we get from .NET Core when it comes to coding?

Further reading

These are some books and websites where you will find more information about the topics of this chapter:

- *The Art of Designing Embedded Systems* by Jack G. Ganssle. Elsevier, 1999.
- *Refactoring: Improving the Design of Existing Code* by Martin Fowler. Addison Wesley, 1999.
- *A Complexity Measure* by Thomas J. McCabe. IEEE Trans. Software Eng. 2(4): 308-320, 1976 (<https://dblp.uni-trier.de/db/journals/tse/tse2.html>).
- <https://blogs.msdn.microsoft.com/zainab/2011/05/25/code-metrics-class-coupling/>

- <https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2019>
- <https://github.com/>
- <https://bitbucket.org/>
- <https://azure.microsoft.com/en-us/services/devops/>
- <https://guides.github.com/introduction/flow/>
- <https://blogs.msdn.microsoft.com/devops/2018/04/19/release-flow-how-we-do-branching-on-the-vsts-team/>
- <https://docs.microsoft.com/aspnet/core/fundamentals/logging/>

15

Testing Your Code with Unit Test Cases and TDD

When developing software, it is essential that you ensure that an application is bug-free and that it satisfies all specifications. This can be done by testing all the modules while they are being developed or when the overall application has been either completely or partially implemented.

Performing all the tests manually is not a feasible option since most of the tests must be executed each time the application is modified, and, as explained throughout this book, modern software is being continuously modified to adapt the applications to the needs of a fast-changing market. This chapter discusses all the types of tests needed to deliver reliable software, and how to organize and automate them.

More specifically, this chapter covers the following topics:

- Understanding automated tests and their usage
- Understanding the basics of **test-driven development (TDD)**
- Optimizing a software investment using TDD
- Defining C# test projects in Visual Studio

In this chapter, we'll see which types of tests are worth implementing, and what unit tests are. We'll see the different types of projects available and how to write unit tests in them. By the end of the chapter, the book use case will help us to execute our tests in Azure DevOps during the **Continuous Integration/Continuous Delivery (CI/CD)** cycle of our applications automatically.

Technical requirements

This chapter requires the 2019 free Community Edition with all database tools installed. It also requires a free Azure account; if you have not already created one, see the *Creating an Azure account* section in Chapter 1, *Understanding the Importance of Software Architecture*.

All concepts in this chapter are clarified with practical examples based on the WWTravelClub book use case. The code for this chapter is available at: <https://github.com/PacktPublishing/Hands-On-Software-Architecture-with-CSharp-8>.

Understanding automated tests

Delaying the application testing until immediately after most of its functionalities have been completely implemented must be avoided for the following reasons:

- If a class or module has been incorrectly designed or implemented, it might have already influenced the way other modules were implemented. Therefore, at this point, fixing the problem might have a very high cost.
- The possible combination of input that is needed to test all possible paths that execution can take grows exponentially with the number of modules or classes that are tested together. Thus, for instance, if the execution of a class method A can take three different paths, while the execution of another method B can take four paths, then testing A and B together would require 3×4 different inputs. In general, if we test several modules together, the total number of paths to test is the product of the number of paths to test in each module. If modules are tested separately, instead, the number of inputs required is just the sum of the paths needed to test each module.
- If a test of an aggregate made of N modules fails, then locating the origin of the bug among the N modules is usually a very time consuming activity.
- When N modules are tested together, we have to redefine all tests involving the N modules, even if just one of the N modules changes during the application's CI/CD cycle.

The preceding considerations show that it is more convenient to test each module method separately. Unluckily, a battery of tests that verifies all methods independently from their context is incomplete because some bugs may be caused by incorrect interactions between modules.

Therefore, tests are organized into two stages:

- **Unit tests:** These verify that all execution paths of each module behave properly. They are quite complete and usually cover all possible paths. This is feasible because there are not very many possible execution paths of each method or module compared to the possible execution paths of the whole application.
- **Integration tests:** These are executed once the software passes all its unit tests. Integration tests verify all modules interact properly to get the expected results. Integration tests do not need to be complete since unit tests will have already verified that all execution paths of each module work properly. They need to verify all patterns of interaction, that is, all the possible ways the various modules may cooperate.

Usually, each interaction pattern has more than one test associated with it: a typical activation of a pattern, and some extreme cases of activation. For instance, if a whole pattern of interaction receives an array as input, we will write a test for the typical size of the array, a test with a null array, a test for an empty array, and a test with a very big array. This way we verify that the way the single module was designed is compatible with the needs of the whole interaction pattern.

With the preceding strategy in place, if we modify a single module without changing its public interface, we need to change the unit tests for that module.

If, instead, the change involves the way some modules interact, then we also have to add new integration tests or to modify existing ones. However, usually, this is not a big problem since most of the tests are unit tests, so rewriting a large percentage of all integration tests does not require too big an effort. Moreover, if the application was designed according to the **Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion (SOLID)** principles, then the number of integration tests that must be changed after a single code modification should be small since the modification should affect just a few classes that interact directly with the modified method or class.

At this point, it should be clear that both unit tests and integration tests must be reused during the entire lifetime of the software. That is why it is worth automating them. Automation of unit and integration tests avoids possible errors of manual test execution and saves time. A whole battery of several thousand automated tests can verify software integrity after each small modification in a few minutes, thus enabling the frequent changes needed in the CI/CD cycles of modern software.



As new bugs are found, new tests are added to discover them so that they cannot reappear in future versions of the software. This way automated test always become more reliable and protect the software more from bugs added by new changes. Thus, the probability of adding new bugs (that are not immediately discovered) is greatly reduced.

The next section will give us the basics for organizing and designing automated unit and integration tests, as well as practical details on how to write a test in C# in the *C# Test Projects* section.

Writing automated (unit and integration) tests

Tests are not written from scratch; all software development platforms have tools that help us to both write tests and launch them (or some of them). Once the selected tests have been executed, all tools show a report and give the possibility to debug the code of all failed tests.

More specifically, all unit and integration test frameworks are made of three important parts:

- **Facilities for defining all tests:** They verify if the actual results correspond to expected results. Usually, a test is organized into test classes, where each test calls tests either a single application class or a single class method. Each test is split into three stages:
 1. **Test preparation:** The general environment needed by the test is prepared. This stage doesn't prepare the single input each method to test must be called with, but just the global environment, such as objects to inject in class constructors or simulations of database tables. Usually, the same preparation procedure is used in several tests, so test preparations are factored out into dedicated modules.
 2. **Test execution:** The methods to test are invoked with adequate input and all results of their executions are compared with expected results with constructs such as `Assert.Equal(x, y)`, `Assert.NotNull(x)`, and so on.
 3. **Tear-down:** The whole environment is cleaned up to avoid the execution of a test influencing other tests. This step is the converse of step 1.

- **Mock facilities:** While integration tests use all (or almost all) classes involved in a pattern of objects cooperation, in unit tests the use of other application classes is forbidden. Thus, if a class under test, say, A, uses a method of another application class, B, that is injected in its constructor in one of its methods, M, then in order to test M we must inject a fake implementation of B. It is worth pointing out that only classes that do some processing are not allowed to use another class during unit tests, while pure data classes can. Mock frameworks contain facilities to define fake implementations of interfaces and interface methods that return data that can be defined in tests. Typically, fake implementations are also able to report information on all fake method calls. Such fake implementations do not need the definition of actual class files but are done online in the test code by calling methods such as new Mock<IMyInterface>().
- **Execution and reporting tool:** This is a visual configuration-based tool that the developer may use to decide which tests to launch and when to launch them. Moreover, it also shows the final outcome of the tests as a report containing all successful tests, all failed tests, each test's execution time, and other information that depends on the specific tool and on how it was configured. Usually, execution and reporting tools that are executed in development IDEs such as Visual Studio also give you the possibility of launching a debug session on each failed test.

Since mock frameworks can only create fake implementations of interfaces but not of classes, we should inject interfaces or pure data classes (that don't need to be mocked) in class constructors and methods; otherwise, classes cannot be unit tested. Therefore, for each cooperating class that we want to inject into another class, we must define a corresponding interface.



Moreover, classes should use instances that are injected in their constructors or methods, and not class instances available in the public static fields of other classes; otherwise, the hidden interactions might be forgotten while writing tests, and this might complicate the *preparation* step of tests.

The next section describes other types of test used in software development.

Writing acceptance and performance tests

Acceptance tests define the contract between the project stakeholders and the development team. They are used to verify that the software developed actually behaves as agreed with them. Acceptance tests verify not only functional specifications but also constraints on the software usability and user interface. Since they also have the purpose of showing how the software appears and behaves on actual computer monitors and displays, they are never completely automatic but consist mainly of lists of recipes and verifications that must be followed by an operator.

Sometimes, automatic tests are developed to verify just the functional specifications, but such tests usually bypass the user interface and inject the test input directly in the logic that is immediately behind the user interface. For instance, in the case of an ASP.NET Core MVC application, the whole website is run in a complete environment that includes all the needed storage filled with test data; input is not provided to HTML pages but is injected directly in the ASP.NET Core controllers. Tests that bypass the user interface are called subcutaneous tests. ASP.NET Core supplies various tools to perform subcutaneous tests and also tools that automate the interaction with HTML pages.

Subcutaneous tests are usually preferred in the case of automated tests, while full tests are executed manually for the following reasons:

- No automatic test can verify how the user interface appears and how usable it is.
- Automating the actual interaction with the user interface is a very time-consuming task.
- User interfaces are changed frequently to improve their usability and to add new features, and also small changes in a single application screen, may force a complete rewrite of all tests that operate on that screen.

In a few words, user interface tests are very expansive and have low reusability, so it's rarely worth automating them. However, ASP.NET Core supplies the `Microsoft.AspNetCore.Mvc.Testing` NuGet package to run the whole website in a testing environment. Using it together with the `AngleSharp` NuGet package, which parses HTML pages into DOM trees, you can write automated full tests with an acceptable programming effort. The automated ASP.NET Core acceptance tests will be described in detail in Chapter 20, *Automation for Software Testing*.

Performance tests apply a fake load to an application to see if it is able to handle the typical production load, to discover its load limits, and to locate bottlenecks. The application is deployed in a staging environment that is a copy of the actual production environment in terms of hardware resources. Then, fake requests are created and applied to the system, and response times and other metrics are collected. Fake request batches should have the same composition as the actual production batches. They can be generated from the actual production request logs if they are available.

If response times are not satisfactory, other metrics are collected to discover possible bottlenecks (low memory, slow storages, or slow software modules). Once located, a software component that is responsible for the problem can be analyzed in the debugger to measure the execution time of the various method calls involved in a typical request.

Failures in the performance tests may lead either to a redefinition of the hardware needed by the application or to the optimization of some software modules, classes or methods.

Both Azure and Visual Studio offer tools to create fake loads and to report execution metrics. However, they have been declared obsolete and will be discontinued in quite a short time (about one year from writing this book), and so we will not describe them. As an alternative, there are both open source and third-party tools that can be used. Some of them are listed in the *Further reading* section.

The next section describes a software development methodology that gives a central role to tests.

Understanding test-driven development (TDD)

Test-driven development (TDD) is a software development methodology that gives a central and central role to unit tests. According to this methodology, unit tests are a formalization of the specifications of each class, so they must be written before the code of the class. Actually, a full test that covers all code paths univocally defines the code behavior, so it can be considered a specification for the code. It is not a formal specification that defines the code behavior through some formal language, but a specification based on behavior examples.

The ideal way to test software would be to write formal specifications of the whole software behavior and to verify with some completely automatic tools if the software that was actually produced conforms with them. In the past, some research effort was spent defining formal languages for describing code specifications, but expressing the behavior the developer has in mind with similar languages was a very difficult and error-prone task. Therefore, these attempts were quickly abandoned in favor of approaches based on examples. At that time, the main purpose was the automatic generation of code. Nowadays, automatic code generation has been substantially abandoned and survives in small application areas, such as the creation of device drivers. In these areas, the effort of formalizing the behavior in a formal language is worth the time saved in trying to test difficult-to-reproduce behaviors of parallel threads.

Unit tests were initially conceived as a way to encode example-based specifications in a completely independent way, as a part of a specific agile development methodology called **Extreme Programming**. However, nowadays, TDD is used independently of Extreme Programming and is included as an obligatory prescription in other agile methodologies.

While it is undoubtedly true that unit tests refined after finding hundreds of bugs act as reliable code specifications, it is not obvious that developers can easily design unit tests that can be immediately used as reliable specifications for the code to be written. In fact, generally, you need an infinite or at least an immense number of examples to univocally define a code's behavior if examples are chosen at random.

The behavior can be defined with an acceptable number of examples only after you have understood all possible execution paths. In fact, at this point, it is enough to select a typical example for each execution path. Therefore, writing a unit test for a method after that method has been completely coded is easy: it simply requires selecting a typical instance for each execution path of the already existing code. However, writing unit tests this way does not protect from errors in the design of the execution paths themselves. For instance, it doesn't prevent the typical error of forgetting to test a variable for the `null` value before invoking a member. That is why TDD suggests writing unit tests before the application code.

We may conclude that, while writing unit tests, the developer must forecast somehow all execution paths by looking for extreme cases and by possibly adding more examples than strictly needed. However, the developer can make mistakes while writing the application code, and he or she can also make mistakes in forecasting all possible execution paths while designing the unit tests.

We have found the main drawback of TDD: unit tests themselves may be wrong. That is, not only application code, but also its associated TDD unit tests may be incoherent with the behavior the developer has in mind. Therefore, in the beginning, unit tests can't be considered software specifications, but rather a possible wrong and incomplete description of the software behavior. Therefore, we have two descriptions of the behavior we have in mind, the application code itself and its TDD unit tests that were written before the application code.



What makes TDD work is that the probability of making exactly the same error while writing the tests and while writing the code is very low.

Therefore, whenever a test fails there is an error either in the tests or in the application code, and, conversely, if there is an error either in the application code or in the test, there is a very high probability a test will fail. That is, the usage of TDD ensures that most of the bugs are immediately found!

Writing a class method or a chunk of code with TDD is a loop composed of three stages:

- Red stage: In this stage, the developer designs new unit tests that must necessarily fail because at this time there is no code that implements the behavior they describe.
- Green stage: In this stage, the developer writes the minimum code or makes the minimum modifications to existing code that are necessary to pass all unit tests.
- Refactoring stage: Once the test is passed, code is refactored to ensure good code quality and the application of best practices and patterns. In particular, in this stage, some code can be factored out in other methods or in other classes. During this stage, we may also discover the need for other unit tests, because new execution paths or new extreme cases are discovered or created.

The loop stops as soon as all tests pass without writing new code or modifying the existing code.

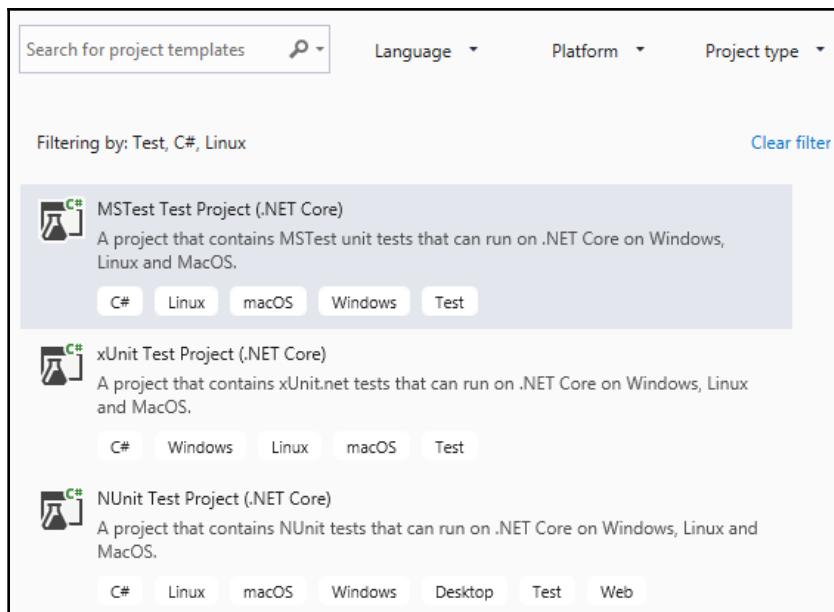
Sometimes, it is very difficult to design the initial unit tests because it is quite difficult to imagine how the code might work and the execution paths it might take. In this case, you can get a better understanding of the specific algorithm to use by writing an initial sketch of the application code. In this initial stage, we need to focus just on the main execution path, completely ignoring extreme cases and input verifications. Once we get a clear picture of the main ideas behind an algorithm that should work we can enter the standard three-stage TDD loop.

In the next section, we will list all test projects available in Visual Studio and describe xUnit in detail.

Defining C# test projects

Visual Studio contains project templates for three types of unit testing frameworks, namely, MSTest, xUnit, and NUnit. Once you start the new project wizard, in order to visualize the version of all of them that is adequate for .NET Core C# applications, set **Project type** as **Test**, **Language** as **C#**, and **Platform** as **Linux**, since .NET Core projects are the only ones that can be deployed on Linux.

The following screenshot shows the selection that should appear:



All the preceding projects automatically include the NuGet package for running all the tests in the Visual Studio test user interface (Visual Studio test runner). However, they do not include any facility for mocking interfaces, so you need to add the `Moq` NuGet package that contains a popular mocking framework.

All test projects must contain a reference to the project to be tested.



In the next section, we will describe xUnit, since it is probably the most popular of the three frameworks. However, all three frameworks are quite similar and differ mainly in the names of the methods and in the names of the attributes used to decorate various testing stuff.

Using the xUnit test framework

In xUnit, tests are methods decorated with either with the `[Fact]` or with the `[Theory]` attributes. Tests are automatically discovered by the test runner that lists all of them in the user interface so the user can run either all of them or just a selection of them.

A new instance of the test class is created before running each test, so the *test preparation* code contained in the class constructor is executed before each test of the class. If you also need *tear-down* code, the test class must implement the `IDisposable` interface so that the tear-down code can be included in the `IDisposable.Dispose` method.

The test code invokes the methods to be tested and then tests the results with methods of the `Assert` static class, such as `Assert.NotNull(x)`, `Assert.Equal(x, y)`, and `Assert.NotEmpty(IEnumerable x)`. There are also methods that verify if a call throws an exception of a specific type, for instance:

```
Assert.Throws<MyException>(() => /* test code */ ...).
```

When an assertion fails, an exception is thrown. A test fails if a not-intercepted exception is thrown either by the test code or by an assertion.

The following is an example of a method that defines a single test:

```
[Fact]
public void Test1()
{
    var myInstanceToTest = new ClassToTest();
    Assert.Equal(5, myInstanceToTest.MethodToTest(1));
}
```

The `[Fact]` attribute is used when a method defines just one test, while the `[Theory]` attribute is used when the same method defines several tests, each on a different tuple of data. Tuples of data can be specified in several ways and are injected in the test as method parameters.

The previous code can be modified to test MethodToTest on several input as follows:

```
[Theory]
[InlineData(1, 5)]
[InlineData(3, 10)]
[InlineData(5, 20)]
public void Test1(int testInput, int testOutput)
{
    var myInstanceToTest = new ClassToTest();
    Assert.Equal(testOutput,
        myInstanceToTest.MethodToTest(testInput));
}
```

Each `InlineData` attribute specifies a tuple to be injected in the method parameters. Since just simple constant data can be included as attribute arguments, xUnit gives you also the possibility to take all data tuples from a class that implements `IEnumerable`, as shown in the following example:

```
public class Test1Data: IEnumerable<object[]>
{
    public IEnumerator<object[]> GetEnumerator()
    {
        yield return new object[] { 1, 5 };
        yield return new object[] { 3, 10 };
        yield return new object[] { 5, 20 };
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

...
...
[Theory]
[ClassData(typeof(Test1Data))]
public void Test1(int testInput, int testOutput)
{
    var myInstanceToTest = new ClassToTest();
    Assert.Equal(testOutput,
        myInstanceToTest.MethodToTest(testInput));
}
```

The type of the class that provides the test data is specified with the `ClassData` attribute.

It is also possible to take data from a static method of a class that returns an `IEnumerable` with the `MemberData` attribute, as shown in the following example:

```
[Theory]
[MemberData(nameof(MyStaticClass.Data),
    MemberType= typeof(MyStaticClass))]
public void Test1(int testInput, int testOutput)
{
    ...
}
```

The `MemberData` attribute is passed the method name as the first parameter, and the class type in the `MemberType` named parameter. If the static method is part of the same test class the `MemberType` parameter can be omitted.

The next section shows how to deal with some advanced preparation and tear-down scenarios.

Advanced test preparation and tear-down scenarios

Sometimes the preparation code contains very time-consuming operations, such as opening a connection with a database, that don't need to be repeated before each test but can be executed once before all the tests contained in the same class. In xUnit, this kind of test preparation code can't be included in the test class constructor; since a different instance of the test class is created before every single test, it must be factored out in a separate class called a fixture class.

If we also need a corresponding tear-down code, the fixture class must implement `IDisposable`. In other test frameworks, such as NUnit, the test class instances are created just once instead, so they don't need the fixture code to be factored out in other classes. However, test frameworks that, like NUnit, do not create a new instance before each test may suffer from bugs because of unwanted interactions between test methods.

The following is an example of an xUnit fixture class that opens and closes a database connection:

```
public class DatabaseFixture : IDisposable
{
    public DatabaseFixture()
    {
        Db = new SqlConnection("MyConnectionString");
    }
}
```

```
public void Dispose()
{
    Db.Close()
}
public SqlConnection Db { get; private set; }
}
```

Since a fixture class instance is created just once before all tests associated with the fixture are executed and the same instance is disposed of immediately after the tests, then the database connection is created just once when the fixture class is created and is disposed of immediately after the tests when the fixture object is disposed of.

The fixture class is associated with each test class by letting the test class implement the empty `IClassFixture<T>` interface, as follows:

```
public class MyTestClass : IClassFixture<DatabaseFixture>
{
    DatabaseFixture fixture;

    public MyDatabaseTests(DatabaseFixture fixture)
    {
        this.fixture = fixture;
    }
    ...
    ...
}
```

A fixture class instance is automatically injected in the test class constructor in order to make all data computed in the fixture test preparation available for the tests. This way, for instance, in our previous example we can get the database connection instance so that all test methods of the class can use it.

If we want to execute some test preparation code on all tests contained in a collection of test classes instead of a single test class, we must associate the fixture class to an empty class that represents the collection of test classes, as follows:

```
[CollectionDefinition("My Database collection")]
public class DatabaseCollection : ICollectionFixture<DatabaseFixture>
{
    // this class is empty, since it is just a placeholder
}
```

The `CollectionDefinition` attribute declares the name of the collection, and the `IClassFixture<T>` interface has been replaced with `ICollectionFixture<T>`.

Then we declare that a test class belongs to the previously defined collection by applying it to the **Collection** attribute with the name of the collection, as follows:

```
[Collection("My Database collection")]
public class MyTestsClass
{
    DatabaseFixture fixture;

    public MyDatabaseTests(DatabaseFixture fixture)
    {
        this.fixture = fixture;
    }
    ...
}
```

The **Collection** attribute declares which collection to use, while the **DataBaseFixture** argument in the test class constructor provides an actual fixture class instance, so it can be used in all class tests.

The next section shows how to mock interfaces with the Moq framework.

Mocking interfaces with Moq

Mocking capabilities are not included in any of the test frameworks we listed in this section as they are not included in xUnit. Therefore, they must be provided by installing a specific NuGet package. The Moq framework available in the **Moq** NuGet package is the most popular mock framework available for .NET and .NET Core. It is quite easy to use and will be briefly described in this section.

Once we've installed the NuGet package, we need to add a `using Moq` statement in our test files. A mock implementation is easily defined, as follows:

```
var myMockDependency = new Mock<IMyInterface>();
```

The behavior of the mock dependency on specific input of the specific method can be defined with the `Setup/Return` method pair as follows:

```
myMockDependency.Setup(x=>x.MyMethod(5)).Returns(10);
```

After `Return`, we may place another `Setup/Return` pair that defines either the behavior of different input of the same method or the behavior of a different method. This way we can specify an indefinite number of input/output behaviors.

Instead of specific input values, we may also use wildcards that match a specific type as follows:

```
myMockDependency.Setup(x => x.MyMethod(It.IsAny<int>))
    .Returns(10);
```

Once configured the mock dependency we may extract the mocked instance from its `Object` property and use it as if it were an actual implementation, as follows:

```
var myMockedInstance = myMockDependency.Object;
...
myMockedInstance.MyMethod(10);
```

However, mocked methods are usually called by the code under test, so we just need to extract the mocked instance and use it as an input in our tests.

We may also mock properties and async methods as follows:

```
myMockDependency.Setup(x => x.MyProperty)
    .Returns(42);
...
myMockDependency.Setup(p => p.MyMethodAsync(1))
    .ReturnsAsync("aasas");
var res = await myMockDependency.Object
    .MyMethodAsync(1);
```

With async methods, `Returns` must be replaced by `ReturnsAsync`.

Each mocked instance records all calls to its methods and properties, so we may use this information in our tests. The following code shows an example:

```
myMockDependency.Verify(x => x.MyMethod(1), Times.AtLeast(2))
```

The preceding statement asserts `MyMethod` that has been invoked with the given arguments at least twice. There are also `Times.Never`, a `Times.Once` (that asserts the method was called just once), and more.

The Moq documentation summarized up to now should cover 99% of the needs that may arise in your tests, but Moq also offers more complex options. The *Further reading* section contains the link to the complete documentation.

The next section shows how to define in practice unit tests and how to run them both in Visual Studio and in Azure DevOps with the help of the book use case.

Use case – automating unit tests in DevOps Azure

In this section, we add some unit test projects to the example application we built in Chapter 13, *Presenting ASP.NET Core MVC*. If you don't have it, you can download it from the Chapter 13, *Presenting ASP.NET Core MVC*, section of the GitHub repository associated with the book. The Chapter 4, *Deciding The Best Cloud-Based Solution*, section of the GitHub repository contains the code we will add in this section and all the instructions to add it.

As a first step, let's make a new copy of the solution folder and name it PackagesManagementWithTests. Then open the solution and add it to xUnit .NET Core C# test project named PackagesManagementTest. Finally, add a reference to the ASP.NET Core project (PackagesManagement), since we will test it, and a reference to the last version of the Moq NuGet package, since we need mocking capabilities. At this point, we are ready to write our tests.

As an example, we will write unit tests for the `Edit` method decorated with `[HttpPost]` of the `ManagePackagesController` controller, which is shown as follows:

```
[HttpPost]
public async Task<IActionResult> Edit(
    PackageFullEditViewModel vm,
    [FromServices] ICommandHandler<UpdatePackageCommand> command)
{
    if (ModelState.IsValid)
    {
        await command.HandleAsync(new UpdatePackageCommand(vm));
        return RedirectToAction(
            nameof(ManagePackagesController.Index));
    }
    else
        return View(vm);
}
```

Before writing our test methods, let's rename the test class that was automatically included in the test project as `ManagePackagesControllerTests`.

The first test verifies that in case there are errors in `ModelState` the action method renders a view with the same model it received as an argument so that the user can correct all errors. Let's delete the existing test method and write an empty `DeletePostValidationFailedTest` method, as follows:

```
[Fact]
public async Task DeletePostValidationFailedTest()
{
}
```

The method must be `async` since the `Edit` method that we have to test is `async`. In this test, we don't need mocked objects since no injected object will be used. Thus, as a preparation for the test we just need to create a controller instance, and we must add an error to `ModelState` as follows:

```
var controller = new ManagePackagesController();
controller.ModelState
    .AddModelError("Name", "fake error");
```

Then we invoke the method, injecting `ViewModel` and a `null` command handler as its arguments since the command handler will not be used:

```
var vm = new PackageFullEditViewModel();
var result = await controller.Edit(vm, null);
```

In the verification stage, we verify that the result is `ViewResult` and that it contains the same model that was injected in the controller:

```
Assert.IsType<ViewResult>(result);
Assert.Equal(vm, (result as ViewResult).Model);
```

Now we also need a test to verify that in case there are no errors the command handler is called, and then the browser is redirected to the `Index` controller action method. We call the `DeletePostSuccessTest` method:

```
[Fact]
public async Task DeletePostSuccessTest()
{}
```

This time the preparation code must include the preparation of a command handler mock, as follows:

```
var controller = new ManagePackagesController();
var commandDependency =
    new Mock< ICommandHandler<UpdatePackageCommand>>();
commandDependency
    .Setup(m => m.HandleAsync(It.IsAny<UpdatePackageCommand>()))
    .Returns(Task.CompletedTask);
var vm = new PackageFullEditViewModel();
```

Since the handler `HandleAsync` method returns no `async` value, we can't use `ReturnsAsync`, but we have to return just a completed `Task` (`Task.Complete`) with the `Returns` method. The method to test is called with both `ViewModel` and the mocked handler:

```
var result = await controller.Edit(vm,
    commandDependency.Object);
```

In this case, the verification code is as follows:

```
commandDependency.Verify(m => m.HandleAsync(
    It.IsAny<UpdatePackageCommand>()),
    Times.Once);
Assert.IsType<RedirectToActionResult>(result);
var redirectResult = result as RedirectToActionResult;
Assert.Equal(nameof(ManagePackagesController.Index),
    redirectResult.ActionName);
Assert.Null(redirectResult.ControllerName);
```

As the first step, we verify that the command handler has actually been invoked once. A better verification should also include a check that it was invoked with a command that includes `ViewModel` passed to the action method. This can be done by extracting this information from `commandDependency.Invocations`. We will take it up as an exercise.

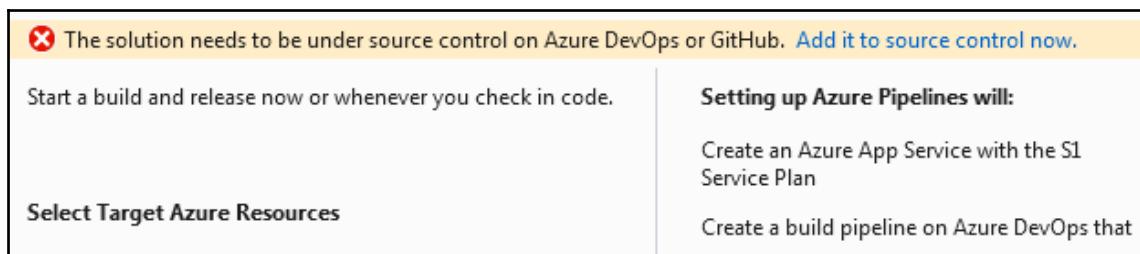
Then we verify that the action method returns `RedirectToActionResult` with the right action method name and with no controller name specified.

Once all tests are ready, if the test windows don't appear on the left bar of Visual Studio, we may simply select the **Run all tests** item from Visual Studio **Test** menu. Once the test window appears, further invocations can be launched from within this window.

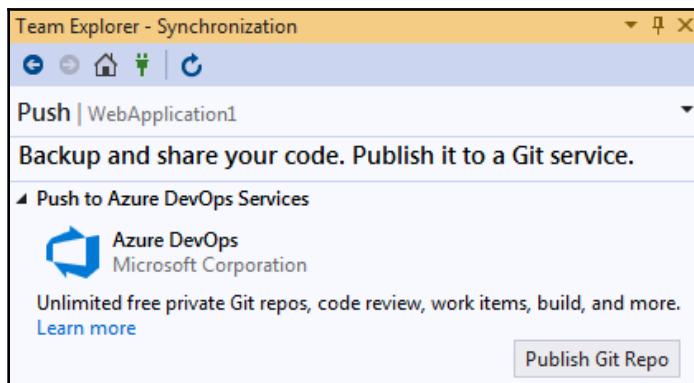
If a test fails, we can add a breakpoint to its code, so we can launch a debug session on it by right-clicking on it in the test window and then by selecting **Debug selected tests**.

The following steps show how to connect our solution with an Azure DevOps repository, and we will define an Azure DevOps pipeline that builds the project and launches its tests. In this way, every day after that all developers have pushed their changes we can launch the pipeline to verify that the repository code compiles and passes all the tests:

1. As a first step, we need a free DevOps subscription. If you don't already have one, please create one by clicking the **Start Free** button on this page: <https://azure.microsoft.com/en-us/services/devops/>. Here, let's define an organization but stop before creating a project, since we will create the project from within Visual Studio.
2. Ensure you are logged into Visual Studio with your Azure account (the same used in the creation of the DevOps account). At this point, you may create a DevOps repository for your solution by right-clicking on the solution and by selecting **Configure continuous delivery to Azure....** In the window that appears, an error message will inform you that you have no repository configured for your code:



3. Click the **Add to source control now** link. After that, the DevOps screen will appear in the Visual Studio **Team Explorer** tab:

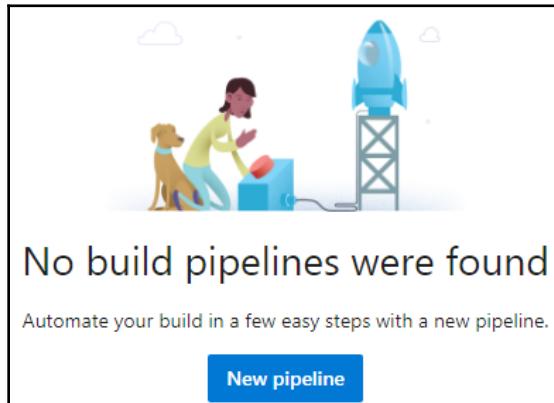


4. Once you click the **Publish Git Repo** button, you will be prompted to select your DevOps organization and a name for the repository. After you successfully publish your code to a DevOps repository, the DevOps screen should change as follows:

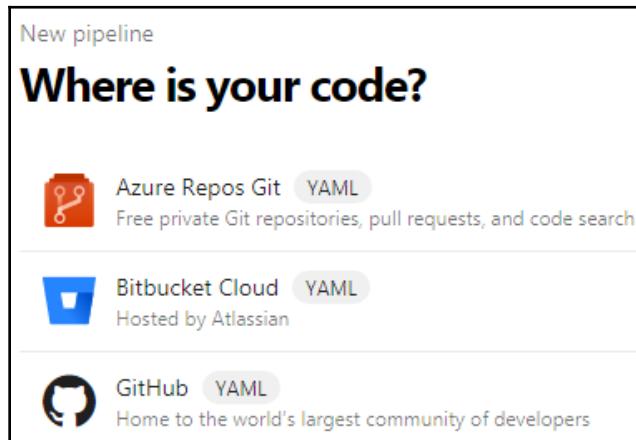


The DevOps screen shows a link to your online DevOps project. In future when you open your solution, if the link does not appear, please click the DevOps screen **Connect** button or the **Manage connections** link (whichever appears) to select and connect your project.

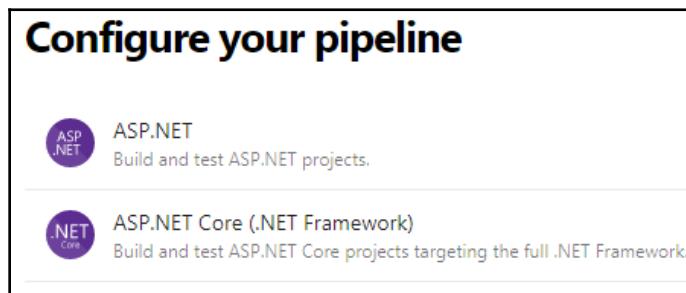
5. Click this link to go to the online project. Once there, if you click the **Repos** item, on the left-hand menu, you will see the repository you just published.
6. Now, click the **Pipelines** menu item to create a DevOps pipeline to build and test your project. In the window that appears, click the button to create a new pipeline:



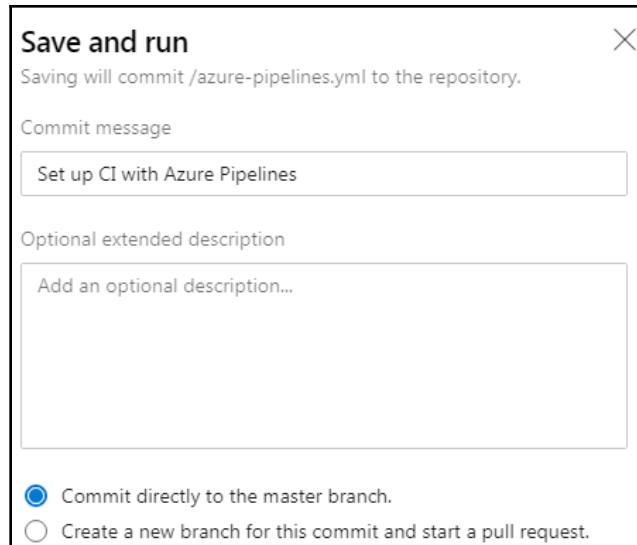
7. You will be prompted to select where your repository is:



8. Select **Azure Repos Git** and then your repository. Then you will be prompted about the kind of project:



9. Select **ASP.NET Core**. A pipeline for building and testing your project will be automatically created for you. Save it by committing the newly created `.yaml` file to your repository:

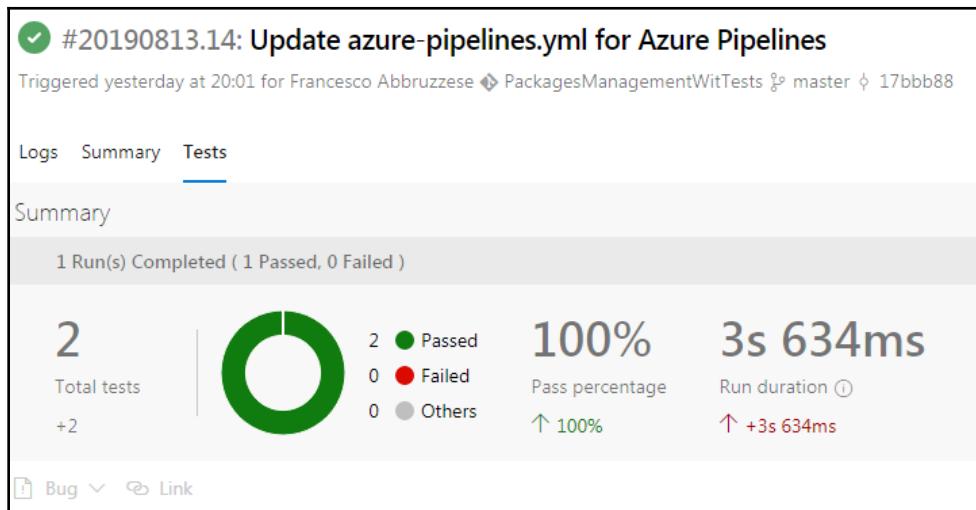


10. The pipeline can be run by selecting the **Queue** button, but since the standard pipeline scaffolded by DevOps has a trigger on the master branch of the repository, it is automatically launched each time changes to this branch are committed and each time the pipeline is modified. The pipeline can be modified by clicking the **Edit** button:

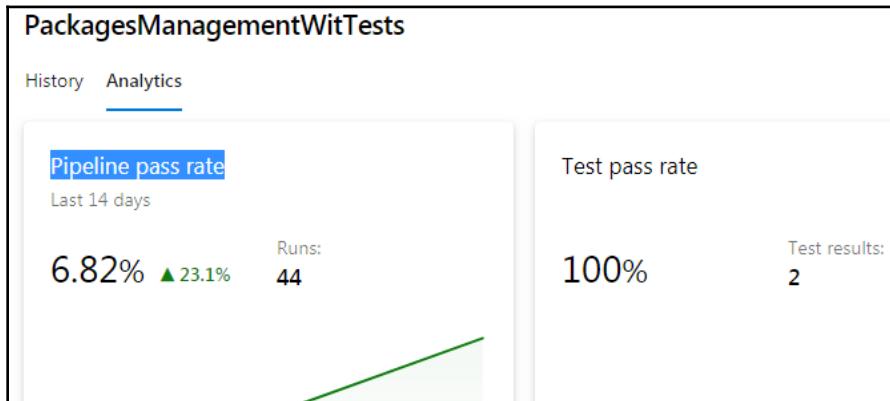
```
steps:  
  Settings  
    - task: NuGetToolInstaller@1  
  
  Settings  
    - task: NuGetCommand@2  
      inputs:  
        restoreSolution: '$(solution)'  
  
  Settings  
    - task: VSBuild@1  
      inputs:  
        solution: '$(solution)'  
        msbuildArgs: '/p:DeployOnBuild=true /p:WebPublishMethod=Package /p:PackageAsSingleFile'  
        platform: '$(buildPlatform)'  
        configuration: '$(buildConfiguration)'  
  
  Settings  
    - task: VSTest@2  
      inputs:  
        platform: '$(buildPlatform)'  
        configuration: '$(buildConfiguration)'
```

11. Once in edit mode, all pipeline steps can be edited by clicking the **Settings** link that appears above each of them. New pipeline steps can be added as follows:
 1. Write task : where the new step must be added and then accept one of the suggestions that appear while you are typing the task name.
 2. After you have written a valid task name a **Settings** link appears above the new step, click it.
 3. Insert the desired task parameters in the window that appears and save.
12. In order to have our test working, we need to specify the criteria to locate all assemblies that contain tests. In our case, since we have a unique .dll file containing the tests, it is enough to specify its name. Click the **Settings** link of the VSTest@2 test task, and replace the content that is automatically suggested for the **Test files** field with the following:

```
**\PackagesManagementTest.dll  
!**\*TestAdapter.dll  
!**\obj\**
```
13. Then click **Add** to modify the actual pipeline content. As soon as you confirm your changes in the **Save and run** dialog, the pipeline is launched, and if there are no errors, test results are computed. The results of tests launched during a specific build can be analyzed by selecting the specific build in the pipeline **History** tab and by clicking the **Tests** tab on the page that appears. In our case, we should see something like the following screenshot:



14. If you click the **Analytics** tab of the pipeline page, you will see analytics about all builds, including analytics about the test results:



15. Clicking the test area of the **Analytics** page gets us a detailed report about all pipeline test results.

Summary

In this chapter, we explained why it is worth automating software tests, and then we focused on the importance of unit tests. We also listed all types of tests and their main features, focusing mainly on unit tests. We analyzed the advantages of TDD, and how to use it in practice. With this knowledge, you should be able to produce software that is both reliable and easy to modify.

Finally, we analyzed all test tools available for .NET Core projects, focusing on the description of xUnit and Moq and showed how to use them in practice both in Visual Studio and in Azure DevOps with the help of the book use case.

The next chapter looks at how to test and measure the quality of the code.

Questions

1. Why is it worth automating unit tests?
2. What is the main reason TDD is able immediately to discover most bugs?
3. What is the difference between the [Theory] and [Fact] attributes of xUnit?
4. Which xUnit static class is used in test assertions?
5. Which methods allow the definition of the Moq mocked dependencies?
6. Is it possible to mock async methods with Moq? If yes, how?

Further reading

While the documentation on xUnit included in the chapter is quite complete, it doesn't include the few configuration options offered by xUnit. The full xUnit documentation is available at <https://xunit.net/>. Documentation for MSTest and NUnit can be found at <https://github.com/microsoft/testfx> and at <https://github.com/nunit/docs/wiki/NUnit-Documentation> respectively.

Moq full documentation is at <https://github.com/moq/moq4/wiki/Quickstart>.

Here are some links to performance test frameworks for web applications:

- <https://jmeter.apache.org/> (free and open source)
- <https://www.neotys.com/neoload/overview>
- <https://www.microfocus.com/en-us/products/loadrunner-load-testing/overview>
- <https://www.microfocus.com/en-us/products/silk-performer/overview>

16

Using Tools to Write Better Code

As we saw in Chapter 14, *Best Practices in Coding C# 8*, coding can be considered an art, but writing understandable code is surely more like philosophy. In the aforementioned chapter, we discussed practices that you, as a software architect, need for your developers. In this chapter, we will describe the techniques and tools for code analysis, so you have well-written code for your project.

The following topics will be covered in this chapter:

- Identifying well-written code
- Understanding the tools that can be used in the process to make things easier
- A book use case—implementing code inspection before publishing the application

By the end of the chapter, you will be able to define which tools you are going to incorporate into your software development life cycle to enable code analysis.

Technical requirements

This chapter requires Visual Studio 2017 or the 2019 free Community Edition or better. You will find the sample code for this chapter at <https://github.com/PacktPublishing/Hands-On-Software-Architecture-with-CSharp-8/tree/master/ch16>.

Identifying a well-written code

It is not easy to define whether a code is well-written or not. The best practices described in Chapter 14, *Best Practices in Coding C# 8*, can certainly guide you as a software architect to define a standard for your team. But even with a standard, mistakes will happen and you will probably find them only after the code is in production. The decision to refactor code in production just because it does not follow all the standards you define is not an easy one to take, especially if this code is working properly. Some people conclude that a well-written code is one that works well in production. However, this surely can cause damage to the software's life, since developers can be inspired by that not-standard code.

For this reason, you—as a software architect—need to find ways to anticipate the lack of application of the coding standard you defined. Luckily, nowadays, we have many options for tools that can help us with this task. They are considered the automation of static code analysis; this technique is seen as a great opportunity to improve the software developed and to help the developers.

The reason your developers will evolve with code analysis is that you start to disseminate knowledge between them during code inspections. The tools that we have now have the same purpose. Better than that, with Roslyn they do this task while you are writing the code. Roslyn is the compiler platform for .NET, and it enables you to develop some tools for analyzing code. These analyzers can check style, quality, design, and other issues.

For instance, look at the following code. It does not make any sense, but you can see that there are some mistakes:

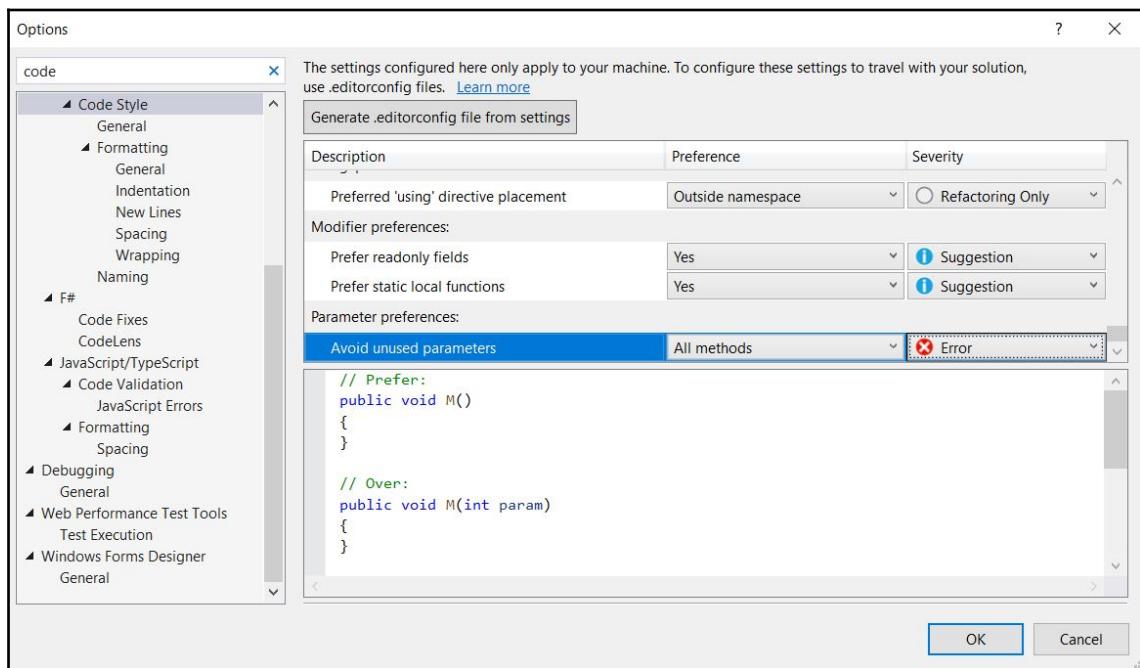
```
using System;
namespace SampleCodeChapter16
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int variableUnused = 10;
                int variable = 10;
                if (variable == 10)
                {
                    Console.WriteLine("variable equals 10");
                }
                else
                {
                    switch (variable)
                    {
                        case 0:
                            Console.WriteLine("variable equals 0");
                            break;
                    }
                }
            }
            catch
            {
            }
        }
    }
}
```

The idea of this code is to show you the power of some tools to improve the code you are delivering. Let's check each of them in the next topic, including how to set up them.

Understanding and applying tools that can evaluate C# code

The evolution of code analysis in Visual Studio is continuous. This means that Visual Studio 2019 certainly has more tools for this purpose than Visual Studio 2017, and so on.

One of the issues that you (as a software architect) need to deal with is the *coding style of the team*. This certainly results in a better understanding of the code. For instance, if you go to *Visual Studio Menu | Tools | Options*, you will find ways to set up how to deal with different code style patterns, and it even indicates a bad coding style as an error in the **Code Style** option, as follows:



For instance, the preceding screenshot suggests that **Avoid unused parameters** were considered an error. After this change, the result of the compilation of the same code presented at the beginning of the chapter was different, as you can see in the following screenshot:

The screenshot shows a code editor window with the following C# code:

```
3  namespace SampleCodeChapter16
4  {
5      class Program
6      {
7          static void Main(string[] args)
8      }
}

```

Below the code editor is the Error List window, which displays one error:

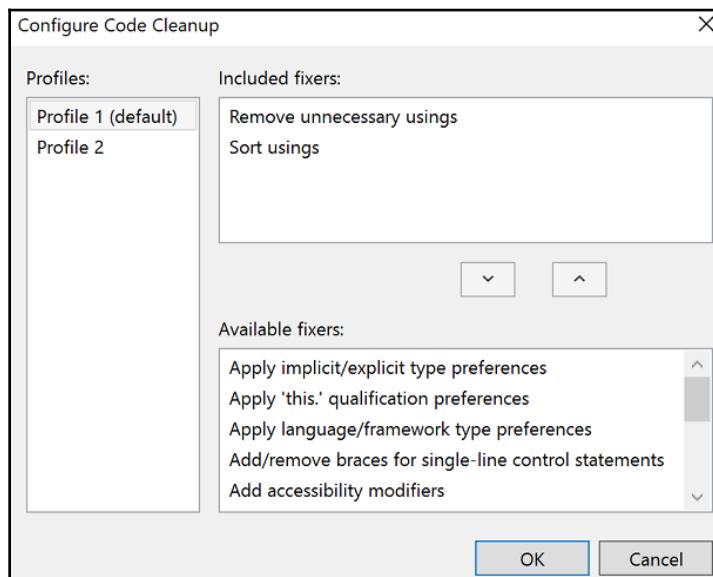
1 Error | 0 Warnings | 0 Messages | Build + IntelliSense

IDE0060 Remove unused parameter 'args'

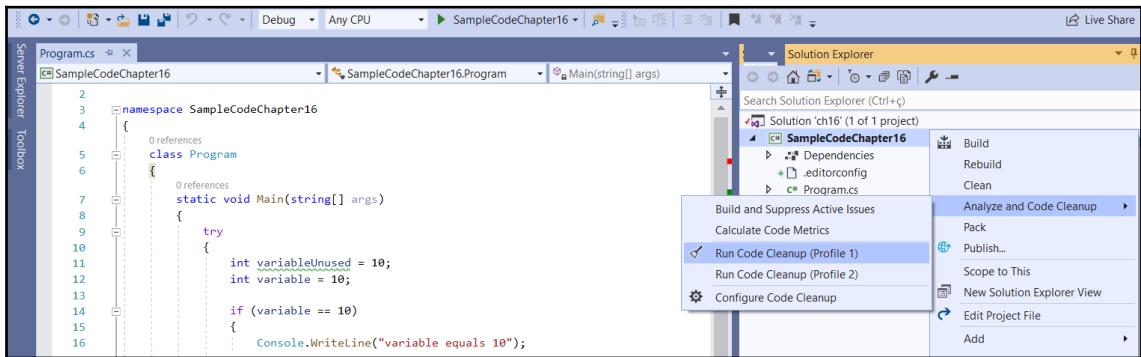
You can export your coding style configuration and attach it to your project so it will follow the rules you have defined.



Another good tool that Visual Studio 2019 provides is **Analyze and Code Cleanup**. In this tool, you are able to set up some code standards that can be cleaned from your code using the tool. For instance, in the following screenshot, it was set to remove unnecessary code:



The way you run the code cleanup is by selecting it in the **Solution Explorer** area, as you can see in the following screenshot. This process will run in all the code files you have:



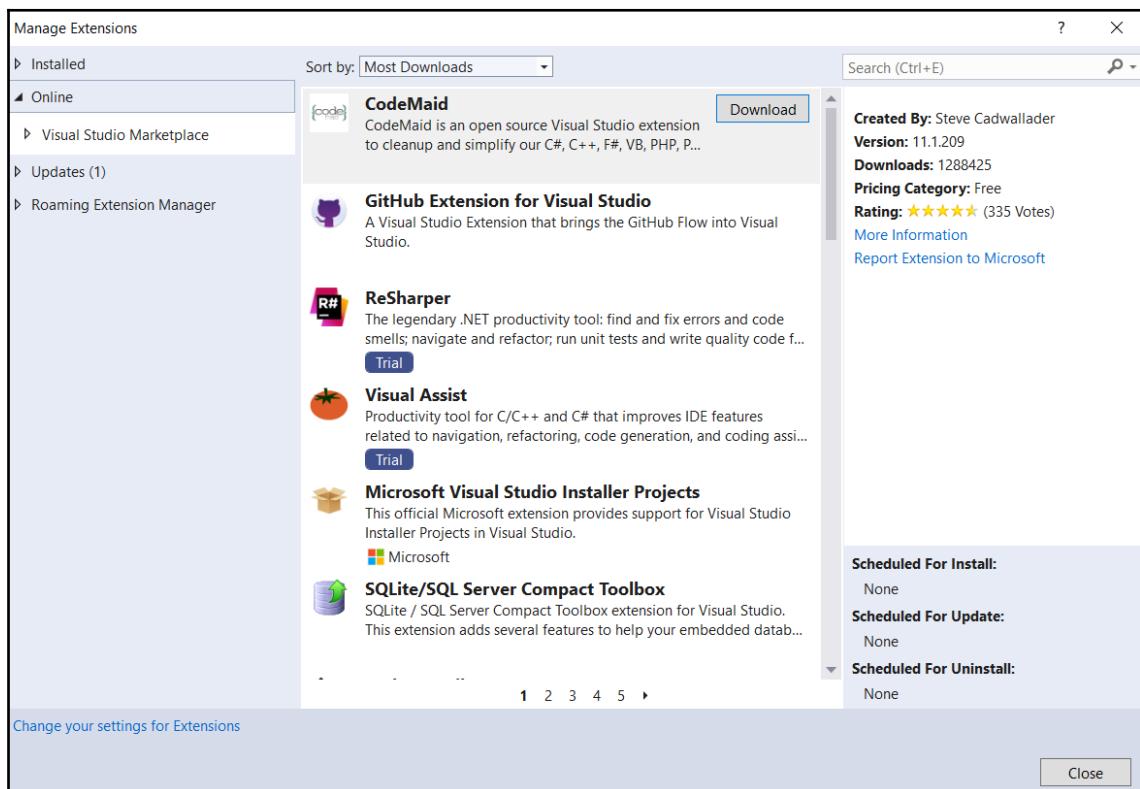
After solving the errors indicated by the Code Style and Code Cleanup tools, the sample code we are working on has some minimal simplifications, as follows:

```
using System;
namespace SampleCodeChapter16
{
    class Program
    {
        static void Main()
        {
            try
            {
                int variable = 10;
                if (variable == 10)
                {
                    Console.WriteLine("variable equals 10");
                }
                else
                {
                    switch (variable)
                    {
                        case 0:
                            Console.WriteLine("variable equals 0");
                            break;
                    }
                }
            }
            catch
            {
            }
        }
    }
}
```

```
    }  
}
```

It is worth mentioning that the preceding code has many improvements that need to be. Beyond that, Visual Studio enables additional tools for the IDE by installing extensions to it. These tools can help you to improve your code quality since some of them were built to do code analysis. This section will list some free options so that you can decide the one that best fits your needs. There are certainly other options and even paid ones. The idea here is not to indicate a specific tool but to give you an idea of their abilities.

To install these extensions, you will need to find the menu on Visual Studio. Here, you have a screenshot of the Extension Manager:





There are many other cool extensions that can improve the productivity and quality of your code and solution. Do a search for them in this manager.

After you select the extension that will be installed, you will need to restart Visual Studio. Most of them are easy to identify after installation since they modify the behavior of the IDE.

Applying extension tools to analyze code

Although the sample code delivered after the Code Style and Code Cleanup tools is better than the one we presented at the beginning of the chapter, it is clearly far from the best practices discussed in *Chapter 14, Best Practices in Coding C# 8*. In the next sections, you will be able to check the behavior of three extensions that can help you evolve this code: Microsoft Code Analysis 2019, SonarLint for Visual Studio 2019, and Code Cracker for Visual Studio 2017.

Using Microsoft Code Analysis 2019

This extension is provided by Microsoft DevLabs and is an upgrade for the FxCop rules that we used to automate in the past. Basically, it has more than 100 rules for detecting problems in the code as you type it.

For instance, just by enabling the extension and rebuilding the small sample we are using in this chapter, Code Analysis found a new issue to solve, as you can see in the following screenshot:

The screenshot shows a code editor with C# code and an 'Error List' window.

```
class Program
{
    0 references | gabriell, 139 days ago | 1 author, 1 change
    static void Main()
    {
        try
        {
            int variable = 10;

            if (variable == 10)
            {
                Console.WriteLine("variable equals 10");
            }
            else
            {
                switch (variable)
                {
                    case 0:
                        Console.WriteLine("variable equals 0");
                        break;
                }
            }
        }
        catch
        {
        }
    }
}
```

The 'Error List' window shows one message:

Code	Description
CA1031	Modify 'Main' to catch a more specific exception type, or rethrow the exception. A general exception such as System.Exception or System.SystemException is caught in a catch statement, or a general catch clause is used. General exceptions should not be caught.

It is worth mentioning that we discussed the usage of empty `try-catch` statements as an anti-pattern in Chapter 14, *Best Practices in Coding C# 8*. So, it would be really good for the health of the code if this kind of problem could be exposed like this.

Applying SonarLint for Visual Studio 2019

SonarLint is an open source initiative from the Sonar Source community to detect bugs and quality issues while you code. There is support for C#, VB .NET, C, C++, and JavaScript. The great thing about this extension is that it comes with explanations to resolve detected issues, and that is why we say developers learn how to code well while using these tools. Check out the following screenshot with the analysis made in the sample code:

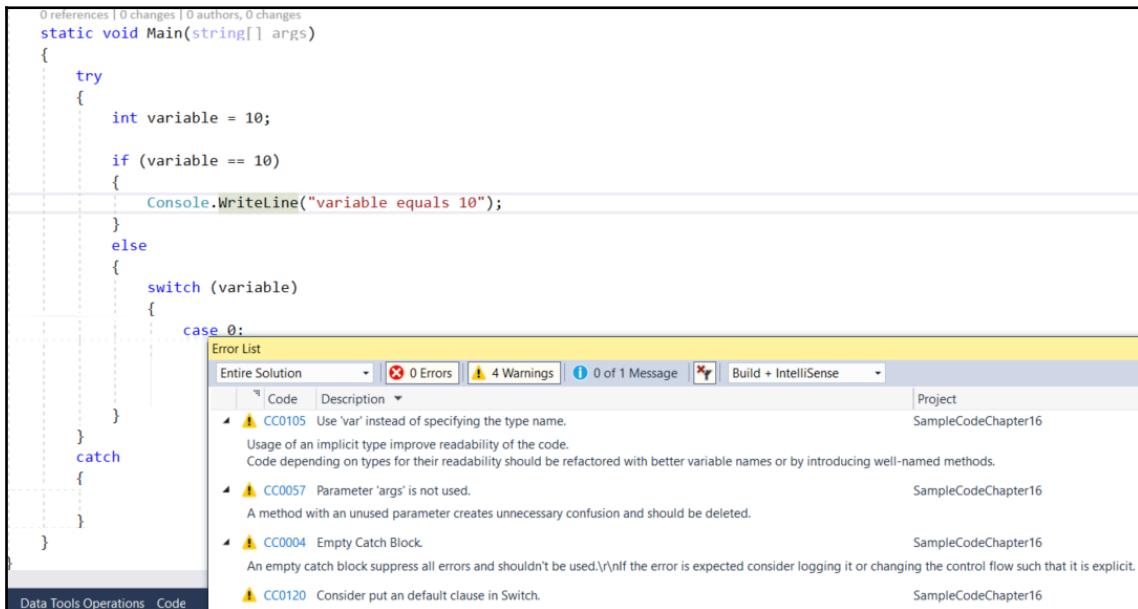
The screenshot shows a code editor window with a C# file named 'Program.cs'. The code contains a Main() method with a try-catch block. Inside the try block, there is an if statement that prints 'variable equals 10' to the console. The catch block is empty. An error list window is overlaid on the code editor, showing three warnings:

- S1118: Add a 'protected' constructor or the 'static' keyword to the class declaration. Explanation: Utility classes, which are collections of static members, are not meant to be instantiated.
- S2486: Handle the exception or explain in a comment why it can be ignored. Explanation: When exceptions occur, it is usually a bad idea to simply ignore them. Instead, it is better to handle them properly, or at least to log them.
- S108: Either remove or fill this block of code. Explanation: Most of the time a block of code is empty when a piece of code is really missing. So such empty block must be either filled or removed.

We can check that this extension was able to point other mistakes and, as they promise, there is an explanation for each warning. This is really useful not only for detecting problems but for training developers in good coding practices.

Getting Code Cracker for Visual Studio 2017 as a helper to write better code

Code Cracker is another tool with the same idea of analyzing code using Roslyn. It is an initiative of some Microsoft MVPs and it is open source, too. According to Visual Studio Marketplace, more people are using this tool than any other:



It is worth mentioning that there are new rules detected in this tool that were not found in the other tools. The reason why there are differences between the code analysis from one extension to others is that probably the rules programmed were not the same.

Checking the final code after analysis

After the analysis of the three extensions, we have finally solved all the issues presented. We can check the final code, as follows:

```
using System;
namespace SampleCodeChapter16
{
    static class Program
    {
        static void Main()
```

```
{  
    try  
    {  
        int variable = 10;  
        if (variable == 10)  
        {  
            Console.WriteLine("variable equals 10");  
        }  
        else  
        {  
            switch (variable)  
            {  
                case 0:  
                    Console.WriteLine("variable equals 0");  
                    break;  
                default:  
                    Console.WriteLine("Unknown behavior");  
                    break;  
            }  
        }  
    }  
    catch (Exception err)  
    {  
        Console.WriteLine(err);  
    }  
}  
}
```

As you can see, the preceding code is not only easier to understand, it is safer and is enabled to consider different paths of programming since the default for the `switch-case` was programmed. This pattern was discussed in *Chapter 14, Best Practices in Coding C# 8*, too, which concludes that best practices can be easily followed using one (or all) of the extensions mentioned in this chapter.

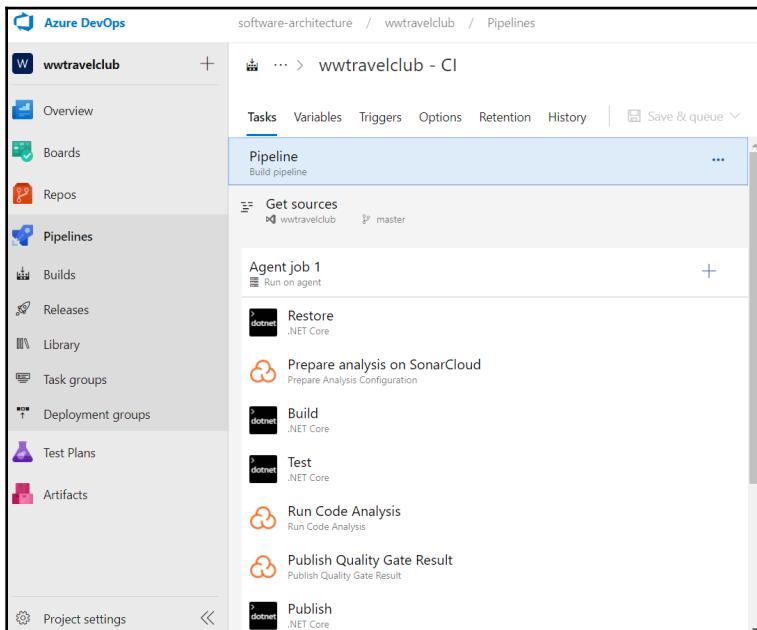
Use case – evaluating the C# code before publishing the application

In Chapter 3, *Documenting Requirements with Azure DevOps*, we created the WWTravelClub repository in the platform. As we saw there, Azure DevOps enables continuous integration, and this can be really useful. In this section, we will discuss more reasons why the DevOps concept and the Azure DevOps platform are so useful.

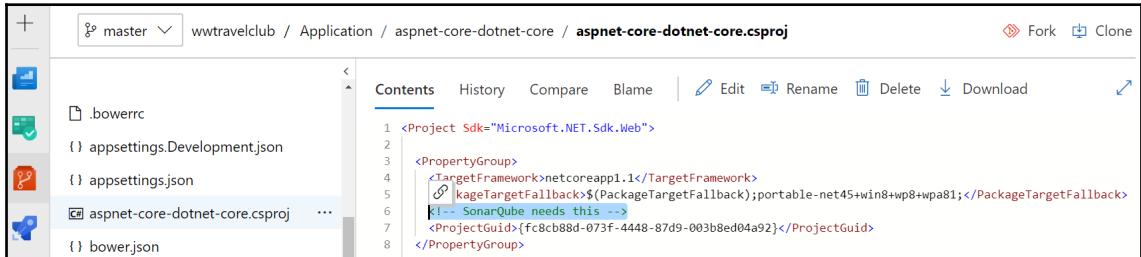
For now, the only thing we would like to introduce is the possibility of analyzing code after it is committed by the developers but has not yet been published. Nowadays, in a SaaS world for application life cycle tools, this is only possible thanks to some of the SaaS code analysis platforms that we have. The use case will use Sonar Cloud.

Sonar Cloud is free for open source code and can analyze code stored in GitHub, Bitbucket, and Azure DevOps. The registration needs a user for these platforms. As soon as you log in, you can follow the steps described in the following article to create the connection between your Azure DevOps and Sonar Cloud: <https://sonarcloud.io/documentation/analysis/scan/sonarscanner-for-azure-devops/>.

By setting up the connection between your project in Azure DevOps and Sonar Cloud, you will have a build pipeline like the one that follows:



It is worth mentioning that C# projects do not have a GUID number, and this is required by Sonar Cloud. You can easily generate one using this link (<https://www.guidgenerator.com/>), and it will need to be placed as in the following screenshot:

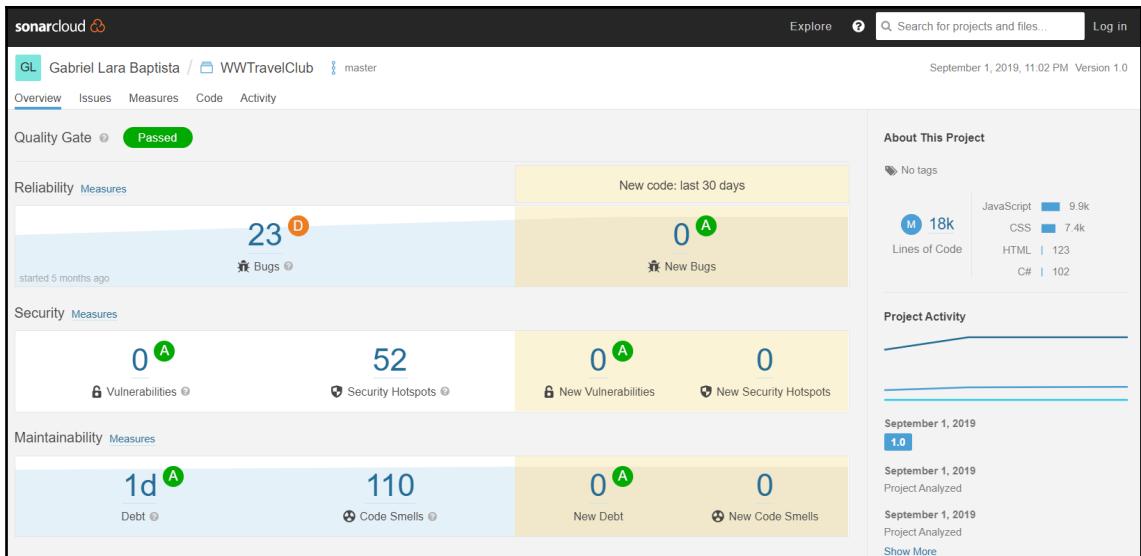


```

<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
    <PackageTargetFallback>$(PackageTargetFallback);portable-net45+win8+wpa81;</PackageTargetFallback>
    <SonarQube.needs_this -->
    <ProjectGuid>{fc8cb88d-073f-4448-87d9-003b8ed04a92}</ProjectGuid>
  </PropertyGroup>

```

As soon as you finish the build, the result of code analysis will be presented in Sonar Cloud, as you can see in the next screenshot. If you want to navigate down to this project, you can visit: <https://sonarcloud.io/dashboard?id=WWWTravelClub>:



Also, by this time, the code analyzed is not in the release yet. So this can be really useful for getting the next step of quality before releasing your system. You can use this approach as a reference for automating code analysis during committal.

Summary

This chapter presented tools that can be used to apply the best practices of coding described in Chapter 14, *Best Practices in Coding C# 8*. We looked at the Roslyn compiler, which enables code analysis at the same time the developer is coding, and we looked at the use case—evaluating the C# code before publishing the application, which implements code analysis during the Azure DevOps building process using Sonar Cloud.

As soon as you apply to your projects everything you have learned in this chapter, the code analysis will give you the opportunity to improve the quality of the code you are delivering to your customer. This is a very important role of a software architect.

In the next chapter, we will be deploying your application with Azure DevOps.

Questions

1. How can software be described as well-written code?
2. What is Roslyn?
3. What is code analysis?
4. What is the importance of code analysis?
5. How does Roslyn help in code analysis?
6. What are Visual Studio extensions?
7. What are the extension tools presented for code analysis?

Further reading

These are some websites where you will find more information on the topics in this chapter:

- <https://marketplace.visualstudio.com/items?itemName=VisualStudioPlatformTeam.MicrosoftCodeAnalysis2019>
- <https://marketplace.visualstudio.com/items?itemName=SonarSource.SonarLintforVisualStudio2019>
- <https://marketplace.visualstudio.com/items?itemName=GiovanniBassi-MVP.CodeCrackerforC>
- <https://github.com/dotnet/roslyn-analyzers>
- <https://docs.microsoft.com/en-us/visualstudio/ide/code-styles-and-code-cleanup>
- <https://sonarcloud.io/documentation/analysis/scan/sonarscanner-for-azure-devops/>
- <https://www.guidgenerator.com/>

5

Section 5: Delivering Software Continuously and at a High Quality Level

This last section of the book will guide you on how to deliver software using DevOps principles for **Continuous Integration (CI)** and **Continuous Deployment (CD)**. The greatest players in the software services world have transformed the concept of software delivery in the last few years. Therefore, this section will discuss the new and revolutionary ways of thinking that have emerged.

In Chapter 17, *Deploying Your Application with Azure DevOps*, we will discuss the new philosophy of software delivery and how Azure DevOps can help you embrace this new approach, and master build and deploy pipelines principles.

In Chapter 18, *Understanding DevOps Principles*, we will cover the main concepts of DevOps, the process that everybody is learning about and putting into practice these days. Besides that, the chapter will present tools for developing and delivering your software with DevOps.

Then, in Chapter 19, *Challenges of Applying CI Scenarios in DevOps*, the focus will be directed toward the challenges related to continuously integrating and deploying a solution. The idea is to make sure you understand the risks and follow best practices when using CI.

Chapter 20, *Automation for Software Testing*, is focused on software testing automation. Since testing is an unavoidable (and repetitive) process, its automation is extremely important. This chapter will teach you how to write automated functional test cases for a project.

This section includes the following chapters:

- Chapter 17, *Deploying Your Application with Azure DevOps*
- Chapter 18, *Understanding DevOps Principles*
- Chapter 19, *Challenges of Applying CI Scenarios in DevOps*
- Chapter 20, *Automation for Software Testing*

17

Deploying Your Application with Azure DevOps

This chapter focuses on so-called **service design thinking**, that is, keeping in mind the software you are designing as a service offered to an organization/part of an organization. The main takeaway of this approach is that the highest priority is the value your software gives to the target organization. Moreover, you are not offering just working code and an agreement to fix bugs, but a solution for all of the needs that your software was conceived for. In other words, your job includes everything it needs to satisfy those needs, such as monitoring users' satisfaction and adapting the software when the user needs change.

Finally, it is easier to monitor the software to reveal issues and new needs and to modify it to adapt it quickly to ever-changing needs.

Service design thinking is strictly tied to the **Software as a Service (SaaS)** model, which we discussed in Chapter 4, *Deciding the Best Cloud-Based Solution*. In fact, the simplest way to offer solutions based on web services is to offer the usage of web services as a service instead of selling the software that implements them.

More specifically, this chapter covers the following topics:

- Understanding SaaS
- Preparing a solution for a service scenario
- Use case – deploying our package-management application with Azure Pipelines

By the end of this chapter, you will be able to design software according to service design thinking principles and use Azure Pipelines to deploy your application.

Technical requirements

This chapter requires Visual Studio 2017 or 2019 free Community Edition or better with all database tools installed. It requires a free Azure account. If you have not already created one, the *Creating an Azure account* subsection of Chapter 1, *Understanding the Importance of Software Architecture*, explains how to do so. This chapter uses the same code as Chapter 15, *Testing Your Code with Unit Test Cases and TDD*, which is available here: <https://github.com/PacktPublishing/Hands-On-Software-Architecture-with-CSharp-8>.

Understanding SaaS

Selling/using software as a service is connected with a wider set of solutions design principles called service design thinking. Service design thinking is not just a software development technique and/or a software deployment approach, but it impacts several business areas, namely, organization and human resources, software development processes, and finally, hardware infrastructures and software architecture.

In the subsections that follow, we will briefly discuss the implications for each of the business areas we listed, and in the last subsection, we will focus specifically on the SaaS deployment model.

Adapting your organization to a service scenario

The first organizational implication comes from the need to optimize the value of the software for the target organization. This requires a human resource or a team—in charge of planning and monitoring the impact of the software in the target organization—to maximize the value added by the software. This strategic role is not needed just during the initial design stage but during the whole lifetime of the application. In fact, this role is in charge of keeping the software fine-tuned with the ever-changing needs of the target organization.

Another important area of impact is **human resource management**. In fact, since the main priority is the value added by the software and not exploiting existing resources and competences, human resources must be adapted to the project needs. This means acquiring new resources as soon as they are needed and developing the required competencies through new human resources and/or adequate training of existing resources.

The next subsection deals with the implications of all processes involved in software development.

Developing software in a service scenario

The main constraint that impacts software development processes is the need to keep the software fine-tuned with the organization's needs. This need can be satisfied by any agile methodology based on a CI/CD approach. For a short review of CI/CD, please refer to the *Organizing your work using Azure DevOps* section of Chapter 3, *Documenting Requirements with Azure DevOps*, while for a detailed discussion of CI/CD, please refer to Chapter 17, *Deploying Your Application with Azure DevOps*, which is completely dedicated to CI/CD. It is worth pointing out that any well-designed CI/CD cycle should include the processing of user feedback and user satisfaction reports.

Moreover, to optimize the value added by the software, it is a good practice to organize stages where the development team (or part of it) is placed in close contact with the system users so that developers can better understand the impact of the software on the target organization.

Finally, the value added by the software must always be kept in mind when writing both functional and non-functional requirements. For this reason, it is useful to annotate *user stories* with consideration of *why* and *how* they contribute to value. The process of collecting requirements is discussed in Chapter 2, *Functional and Nonfunctional Requirements*.

More technical implications are discussed in the next subsection.

Technical implications of a service scenario

In a service scenario, both the hardware infrastructure and software architecture are constrained by the three main principles mentioned as follows, which are an immediate consequence of the requirement to keep the software fine-tuned with the organization's needs, namely, the following:

- There's the need to monitor the software to discover any kind of issue that might have been caused by system malfunctions or changes in software usage and/or user needs. This implies extracting health checks and load statistics from all hardware/software components. Good hints for discovering changes in the organization's needs are also given by statistics on the operations performed by the users—more specifically, the average time spent by both the user and the application on each operation instance, and the number of instances of each operation performed per unit of time (day, week, or month).
- There's also the need to monitor user satisfaction. Feedback on user satisfaction can be obtained by adding to each application screen a link to an easy-to-fill user-satisfaction report page.

- Finally, there's the need to adapt both hardware and software quickly, both to the traffic received by each application module and to the changes in the organization's needs. This means the following:
 - Paying extreme attention to software modularity
 - Keeping the door open for changes in the database engine and preferring SOA or microservices-based solutions to monolithic software
 - Keeping the door open to new technologies

Making the hardware easy to adapt means allowing hardware scaling, which in turn implies either the adoption of cloud infrastructures, hardware clusters, or both. It is also important to keep the door open to changes in cloud service suppliers, which in turn means encapsulating the dependence on the cloud platform in a small number of software modules.

The maximization of the value added by the software can be achieved by choosing the best technology available for the implementation of each module, which in turn means being able to mix different technologies. Here is where container-based technologies, such as Docker, come into play. Docker and related technologies were described in Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*.

Summing up, all of the requirements we have listed converge toward most of the advanced technologies we have described in this book, such as cloud services, scalable web applications, distributed/scalable databases, Docker, SOA, and microservices architectures.

More details on how to prepare your software for a service environment are given in the next section, while the next subsection focuses specifically on the advantages and disadvantages of SaaS applications.

Adopting a SaaS solution

The main attraction of SaaS solutions is their flexible payment model, which offers the following advantages:

- You can avoid abandoning big investments in favor of more affordable monthly payments.
- You can start with a cheap system and then move toward more expansive solutions only when the business grows.

However, SaaS solutions also offer other advantages, namely, the following:

- In all cloud solutions, you can easily scale up your solution.
- The application is automatically updated.
- Since SaaS solutions are delivered over the public internet, they are accessible from any location.

Unluckily, SaaS advantages come at a cost, since SaaS also has not negligible disadvantages, namely, the following:

- Your business is strictly tied to the SaaS provider, which might discontinue the service and/or modify it in a way that is not acceptable to you anymore.
- Usually, you can't implement any kind of customization, being limited to the few standard options offered by the SaaS supplier. However, sometimes SaaS suppliers also offer the possibility to add custom modules written either by them or by you.

Summing up, SaaS solutions offer interesting advantages but also some disadvantages, so you, as a software architect, must perform a detailed analysis to decide how to adopt them.

The next section explains how to adapt software to be used in a service scenario.

Preparing a solution for a service scenario

First of all, *preparing a solution for a service scenario* means designing it specifically for the cloud and/or for a distributed environment. In turn, this means designing it with scalability, fault tolerance, and automatic fault recovery in mind.

The main implications of the preceding three points are concerned with the way the *state* is handled. Stateless module instances are easy to scale and to replace, so you should carefully plan which modules are stateless and which ones have states. Moreover, as explained in Chapter 7, *How to Choose Your Data Storage in the Cloud*, you have to keep in mind that write and read operations scale in a completely different way. In particular, read operations are easier to scale with replication, while write operations do not scale well with relational databases and often require NoSQL solutions.

High scalability in a distributed environment prevents the usage of distributed transactions and of synchronous operations, in general. Therefore, data coherence and fault tolerance can be achieved only with more complex techniques based on asynchronous messages, such as the following:

- One technique is storing all messages to send in a queue so that asynchronous transmissions can be retried in the event of errors or timeouts. Messages can be removed from the queue either when confirmation of reception is received or when the module decides to abort the operation that produced the message.
- Another is handling the possibility that the same message is received several times because timeouts caused the same message to be sent several times.
- If needed, use techniques such as optimistic concurrency and event sourcing to minimize concurrency problems in databases. Optimistic concurrency is explained in *The data layer* subsection of the use case at the end of Chapter 13, *Presenting ASP.NET Core MVC*, while event sourcing is described together with other data layer stuff in the *Using SOLID principles to map your domains* section of Chapter 10, *Understanding the Different Domains in a Software Solution*.



The first two points in the preceding list are discussed in detail together with other distributed processing techniques in the *How does .NET Core deal with Microservices?* section of Chapter 5, *Applying a Microservice Architecture to Your Enterprise Application*.

Fault tolerance and automatic fault recovery require that software modules implement health check interfaces that the cloud framework might call, to verify whether the module is working properly or whether it needs to be killed and replaced by another instance. ASP.NET Core and all Azure microservices solutions offer off-the-shelf basic health checks, so the developer doesn't need to take care of them. However, more detailed custom health checks can be added by implementing a simple interface.

The difficulty increases if you have the goal of possibly changing the cloud provider of some of the application modules. In this case, the dependency from the cloud platform must be encapsulated in just a few modules, and solutions that are too strictly tied to a specific cloud platform must be discarded. Hence, for instance, you should avoid the use of stateful/stateless native Service Fabric services since their architecture is specific to Azure Service Fabric, so they can't be ported to a different cloud platform.

If your application is conceived for a service scenario, everything must be automated: new versions testing and validation, the creation of the whole cloud infrastructure needed by the application, and the deployment of the application on that infrastructure.

All cloud platforms offer languages and facilities to automate the whole software CI/CD cycle, that is, building the code, testing it, triggering manual version approvals, hardware infrastructure creation, and application deployment.

Azure Pipelines allows the complete automatization of all of the steps listed. The use case in Chapter 15, *Testing Your Code with Unit Test Cases and TDD*, shows how to automatize all steps up to and including software testing with Azure Pipelines. The use case in the next section will show how to automatize the application deployment on the Azure web app platform.

Automatization has a more fundamental role in SaaS applications since the whole creation of a new tenant for each new customer must be automatically triggered by the customer subscription. More specifically, multi-tenant SaaS applications can be implemented with three fundamental techniques:

- All customers share the same hardware infrastructure and data storage. This solution is the easiest to implement since it requires the implementation of a standard web application. However, it is possible just for very simple SaaS services since, for more complex applications, it becomes always more difficult to ensure that storage space and computation time are split equally between users. Moreover, as the database becomes more and more complex, it is always more difficult to keep the data of different users safely isolated.
- All customers share the same infrastructure but each customer has its own data storage. This option solves all database problems of the previous solution, and it is quite easy to automatize since the creation of a new tenant requires just the creation of a new database. This solution offers a simple way to define pricing strategies, by linking them to storage consumption.
- Each customer has their private infrastructure and data storage. This is the most flexible strategy. From the user's point of view, its only disadvantage is the higher price. Therefore, it is convenient only above a minimum threshold of computational power required by each user. It is more difficult to automate since a whole infrastructure must be created for each new customer and a new instance of the application must be deployed on it.

Whichever of the three strategies is chosen, you need the possibility to scale out your cloud resources as your consumers increase.

If you also need the possibility to ensure your infrastructure creation scripts work across several cloud providers, then, on the one hand, you can't use features that are too specific to a single cloud platform, and on the other, you need a unique infrastructure creation language that can be translated into the native languages of the more common cloud platforms. Terraform and Ansible are two very common choices for describing hardware infrastructures.

Use case – deploying our package-management application with Azure Pipelines

In this section, we will configure an automatic deployment to the Azure App Service platform for the DevOps project that we already defined in the use case at the end of Chapter 15, *Testing Your Code with Unit Test Cases and TDD*. Azure DevOps can also automatically create a new web app, but to prevent configuration errors (which might consume all your free credit), we will create it manually and let Azure DevOps just deploy the application. All of the required steps are organized into various subsections as follows.

Creating the Azure Web App and the Azure database

An Azure Web App can be defined by following the simple steps that follow:

1. Go to the Azure portal and select **App Services**, and then click the **Add** button to create a new Web App. Fill in all data as follows:

The screenshot shows the 'Create Web App' wizard in the Azure portal, specifically the 'Basics' step. The title 'Web App' is at the top, followed by a 'Create' button. Below the tabs 'Basics' (selected), 'Monitoring', 'Tags', and 'Review and create', is a descriptive text about App Service Web Apps. The 'Project Details' section includes fields for 'Subscription' (Visual Studio Ultimate con MSDN) and 'Resource Group' (Default-Web-WestEurope). The 'Instance Details' section includes fields for 'Name' (PackagesManagement.azurewebsites.net), 'Publish' (Code selected), 'Runtime stack' (Select a runtime stack), 'Operating System' (Windows selected), and 'Region' (West Europe). At the bottom are 'Review and create' and 'Next: Monitoring >' buttons.

2. Clearly, you may use a **Resource Group** you already have, and the most convenient region for you. For **Runtime stack**, please select the same .NET Core version you used in the Visual Studio solution.

3. Now, if you have enough credit, let's create a SQL Server database for the application, and let's call it `PackagesManagementDatabase`. If you don't have enough credit, don't worry—you can still test application deployment, but the application will return an error when it tries to access the database. Please refer to the *Relational databases* subsection of Chapter 7, *How to Choose Your Data Storage in the Cloud*, for how to create a SQL Server database.

Configuring your Visual Studio solution

Once you've defined the Azure Web App, you need to configure the application for running in Azure by following these simple steps:

1. If you defined an Azure database, you need two different connection strings in your Visual Studio solution, one of the local databases for development and one of the Azure database for the web app.
2. Now, open both `appsettings.json` and `appsettings.Development.json` in your Visual Studio solution, as follows:



3. Then, copy the whole `ConnectionStrings` node of `appsettings.json` into `appsettings.Development.json`, as follows:

```
"ConnectionStrings": {  
    "DefaultConnection": "Server=(localdb)...."  
},
```

Now you have the local connection string in the development settings, so you can change `DefaultConnection` in `appsettings.json` with one of the Azure databases.

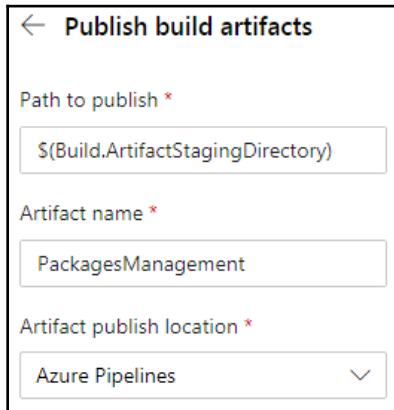
4. Go to the database in the Azure portal, copy the connection string, and fill it with the username and password you got when you defined the database server.
5. Finally, commit your changes locally and then synchronize with the remote repository. Now, your changes are on DevOps Pipelines, which is already processing them to get a new build.

Configuring Azure Pipelines

Finally, you can configure an Azure Pipeline for the automatic delivery of your application on Azure by following these steps:

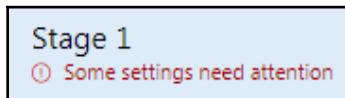
1. Connect Visual Studio with your DevOps project by clicking the **Manage Connections** link in the **Connection** tab of the Visual Studio **Team Server** window. Then, click the DevOps link to go to your online project.
2. Modify the `PackagesManagementWithTests` build pipeline by adding a further step after the unit test step. In fact, we need a step that prepares all files to be deployed in a ZIP file.
3. Click the **Edit** button of the `PackagesManagementWithTests` pipeline, and then go to the end of the file and write the following:

```
- task: PublishBuildArtifacts@1
```
4. When the **Settings** link appears above the new task, click it to configure the new task:



5. Accept the default **Path to publish** since it is already synchronized with the path of the task that will deploy the application, and just insert the artifact name, and then select **Azure Pipeline** as the location. As soon as you save, the pipeline will start, and the newly added task should succeed.

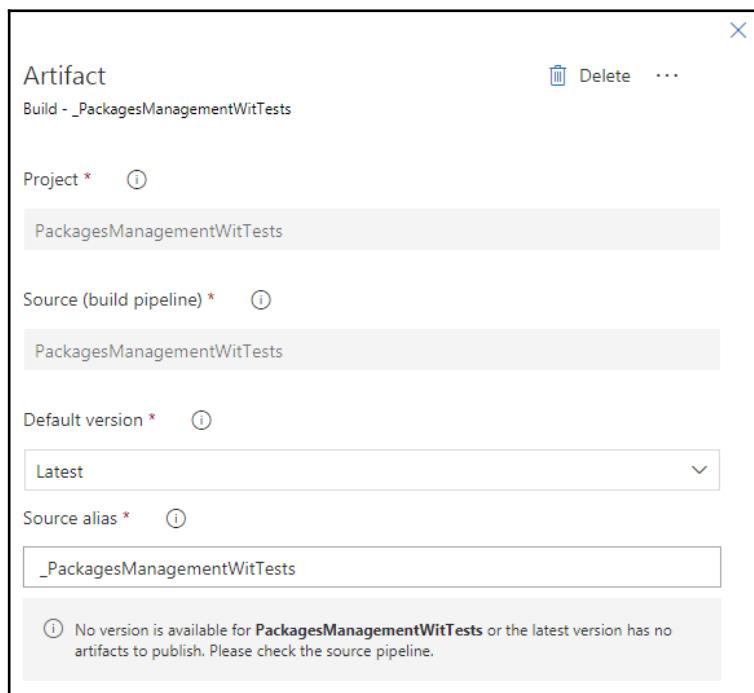
6. Deployments and other release artifacts are added to different pipelines called **Release Pipelines**, to decouple them from build related artifacts. With **Release Pipelines**, you cannot edit a `.yaml` file, but you will work with a graphic interface.
7. Click the **Releases** left menu tab to create a new **Release Pipeline**. As soon as you click **add a new pipeline**, you will be prompted to add the first task of the first pipeline stage. In fact, the whole release pipeline is composed of different stages, each grouping sequences of tasks. While each stage is just a sequence of tasks, the stages diagram can branch and we can add several branches after each stage. This way, we can deploy to different platforms that each require different tasks. In our simple example, we will use a single stage.
8. Select the **Deploy Azure App Service** task. As soon as you add this task, you will be prompted to fill in missing information:



9. Click the error link and fill in the missing parameters:

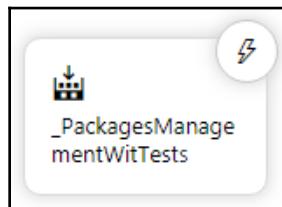
The screenshot shows the configuration for a "Deploy Azure App Service" task. It includes fields for "Stage name" (set to "Stage 1"), "Parameters" (with a "Unlink all" button), "Azure subscription" (set to "Visual Studio Ultimate con MSDN(ad55e9fd-c1c2-4de9-9547-e13e0d3496a7)"), "App type" (set to "Web App on Windows"), and "App service name" (set to "PackagesManagement"). Each field has a refresh button to its right.

10. Select your subscription, and then, if an authorization button appears, please click it to authorize Azure Pipelines to access your subscription. Then, select Windows as the deployment platform, and finally, select the App Service you created from the **App service name** drop-down list. Task settings are automatically saved while you write them, so you need just to click the **Save** button for the whole pipeline.
11. Now, we need to connect this pipeline to a source artifact. Click the **Add Artifact** button and then select **Build** as the source type, because we need to connect the new release pipeline with the ZIP file created by our build pipeline. A settings window appears:



12. Select our previous build pipeline from the drop-down list, and keep **Latest** as the version. Finally, accept the suggested name in **Source alias**.

Our release pipeline is ready and can be used as it is. The image of the source artifact you just added contains a trigger icon in its top-right corner, as follows:



If you click on the trigger icon, you are given the option to automatically trigger the release pipeline as soon as a new build is available:

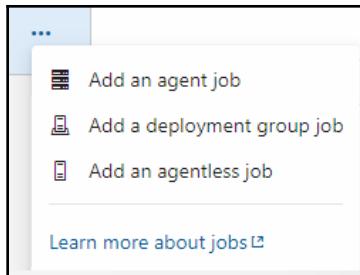


Keep it disabled; we can enable it after we have completed and manually tested the release pipeline. In preparation for an automatic trigger, we need to add a human approval task before the application is deployed.

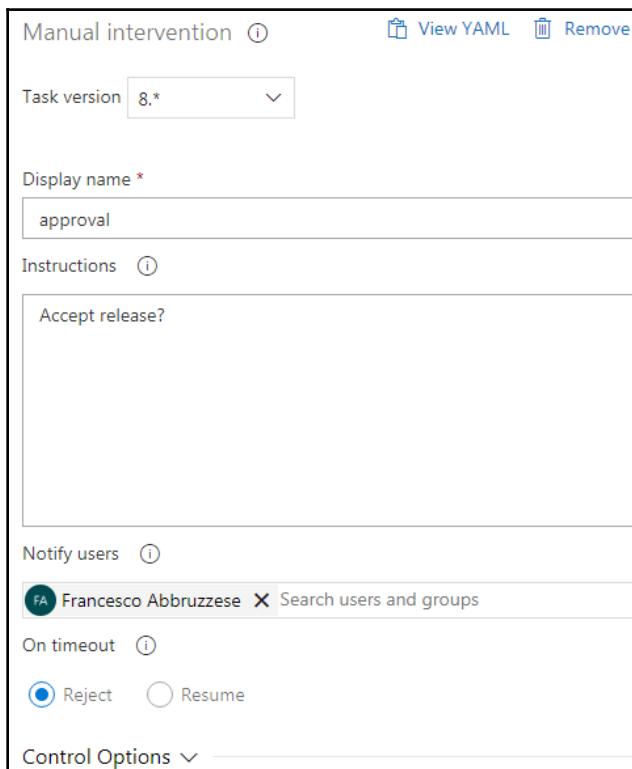
Adding a manual approval for the release

Since tasks are usually executed by software agents, we need to embed human approval in a manual job. Let's add it with the following steps:

1. Click the three dots on the right of the **Stage 1** header:

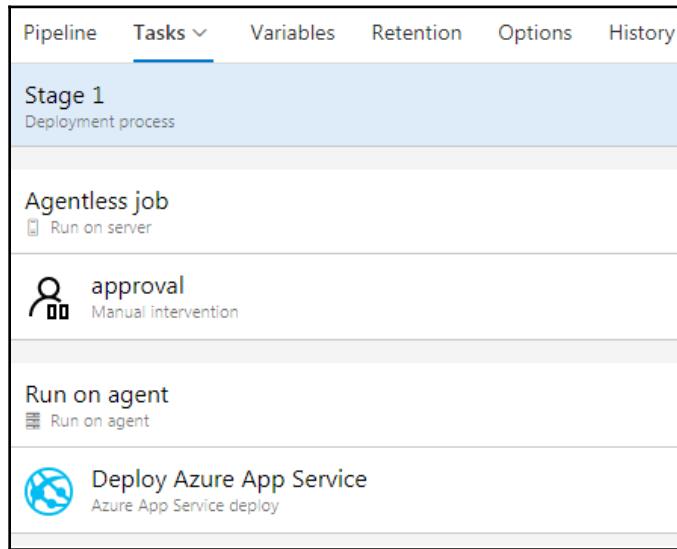


2. Then, select **Add an agentless job**. Once the agentless job has been added, click its add button and add a **Manual intervention** task. The following screenshot shows the **Manual intervention** settings:



3. Add instructions for the operator and select your account in the **Notify users** field.

4. Now, drag the whole **Agentless job** with the mouse, to place it before the application deployment task. The final screenshot should be as follows:



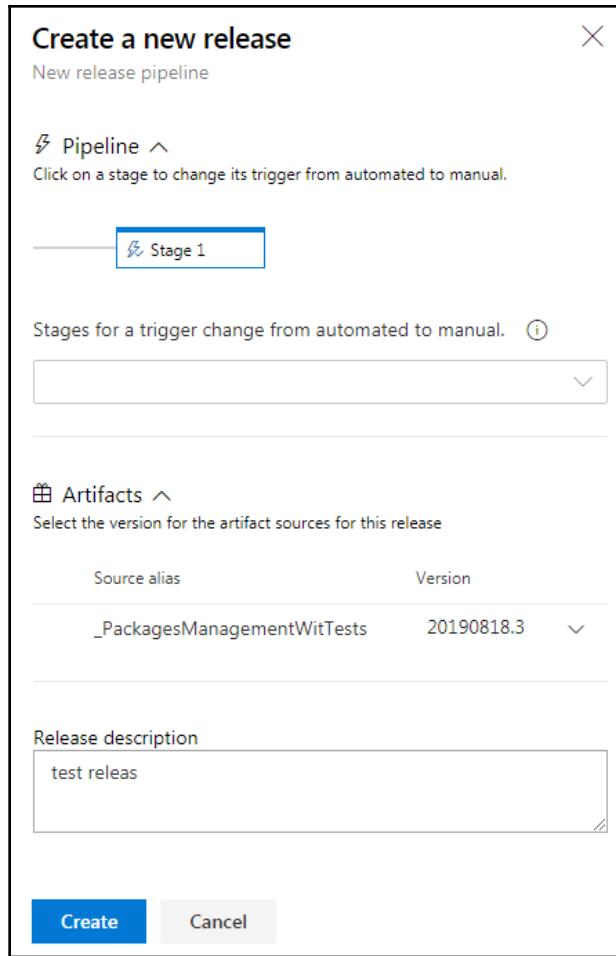
5. Finished! Click the save button in the top-left to save the pipeline.

Now, everything is ready to create our first automatic release.

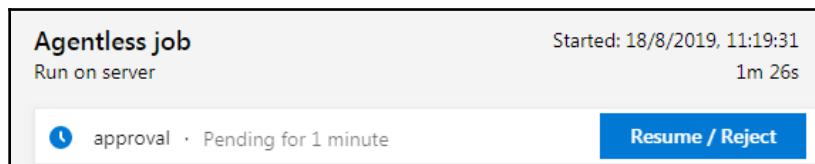
Creating a release

Once you have everything in place, a new release can be prepared and deployed as follows:

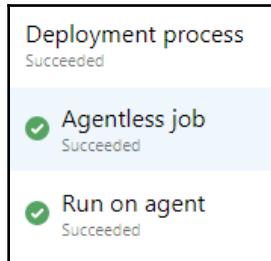
1. Let's click the **Create release** button to start the creation of a new release:



2. Verify that the **Source alias** is the last available, add a release description, and then click **Create**. In a short time, you should receive an email for the release approval. Click the link it contains, and go to the approval page:



3. Click the **Resume / Reject** button and then approve the release. Wait for the deployment to complete. You should have all of the tasks successfully completed, as shown in the following screenshot:



You have run your first successful release pipeline!

In a real-life project, the release pipeline would contain some more tasks. In fact, applications (before being deployed in the actual production environment) are deployed in a staging environment where they are beta-tested. Hence, probably, after this first deployment, there would be some manual tests, manual authorization for the deployment in production, and the final deployment in production.

Summary

We described *service design thinking* principles and the SaaS software deployment model. Now, you should be able to analyze all of the implications of these approaches for an organization, and you should be able to adapt pre-existing software development processes and hardware/software architectures to take advantage of the opportunities they offer.

We also explained the need for, and the techniques involved in, the automatization of the software cycle, cloud hardware infrastructure configuration, and application deployment.

Once you have implemented the example in the last use case section, you should be able to use Azure Pipelines to automate infrastructure configuration and application deployment.

The next chapter gives more insights into DevOps, which, together with CI/CD, which is discussed in detail in Chapter 19, *Challenges of Applying CI Scenarios in DevOps*, plays a fundamental role in service scenarios and, in particular, the maintenance of SaaS applications.

Questions

1. What is the main goal of service design thinking?
2. Is it true that service design thinking requires the optimal usage of all competencies already available in the company?
3. Why is a complete automatization fundamental in the life cycle of SaaS applications?
4. Is it possible to define hardware cloud infrastructures with a platform-independent language?
5. What is the preferred Azure tool for the automatization of the whole application lifecycle?
6. If two SaaS suppliers offer the same software product, should you prefer the most reliable or the cheapest one?
7. Is scalability the only important requirement in a service scenario?

Further reading

The main references in this chapter are references to other chapters/sections of this book and have already been given throughout this chapter. Here, we give just the link to the Azure Pipelines documentation: <https://docs.microsoft.com/en-us/azure/devops/pipelines/?view=azure-devops>, and to the two infrastructure description languages cited in this chapter, Terraform (<https://www.terraform.io/>) and Ansible (<https://www.ansible.com/>).

18

Understanding DevOps Principles

DevOps is a process that everybody is learning and putting into practice these days. But as a software architect, you need to understand and propagate DevOps not only as a process but as a philosophy. This chapter will cover the main concepts, principles, and tools you need to develop and deliver your software with DevOps.

The following topics will be covered in this chapter:

- Describing what DevOps is and looking at a sample of how to apply it in the WWTravelClub project
- Understanding DevOps principles and deployment stages to leverage the deployment process
- Learning DevOps tools that can be used together with Azure DevOps to improve tests and feedback

In contrast with other chapters, the WWTravelClub project will be presented during the topics and we will offer a conclusion at the end of the chapter, giving you the opportunity to understand how this philosophy can be implemented. All the screenshots exemplifying the DevOps principles come from the main sample of the book, so you will be able to understand the DevOps principles easily.

Technical requirements

This chapter requires the Visual Studio 2019 Community Edition or better. You may also need an Azure DevOps account, as described in Chapter 3, *Documenting Requirements with Azure DevOps*.

Describing DevOps

DevOps comes from a union of the words *Development and Operations*, so this process simply unifies actions in these areas. However, when you start to study a little bit more about it, you will realize that connecting these two areas is not enough to achieve the true goals of this philosophy.

We can also say that DevOps is the process that answers the current needs of humanity regarding software delivery.



Donovan Brown, Principal DevOps Manager of Microsoft, has a spectacular definition of what DevOps is: <http://donovanbrown.com/post/what-is-devops>.

A way to deliver value continuously to our end users, using process, people, and products—this is the best description of the DevOps philosophy. We need to develop and deliver customer-oriented software. As soon as all areas of the company understand that the key point is the end user, your task as a software architect is to present the technology that will facilitate the process of delivering.

It is worth mentioning that all the content of this book is connected to this approach. It is never a matter of knowing a bunch of tools and technologies. As a software architect, you have to understand that it is always a way to bring faster solutions easily to your end user, linked to their real needs. For this reason, you need to learn the DevOps principles, which will be discussed during the chapter.

Understanding DevOps principles

Considering DevOps as a philosophy, it is worth mentioning that you there some principles that enable the process to work well in your team. These principles are continuous integration, continuous delivery, and continuous feedback.



For more information, please visit: <https://azure.microsoft.com/en-us/overview/what-is-devops/>.

DevOps concept is represented by the symbol of infinity in many books and technical articles. This symbol represents the necessity to have a continuous approach in the software development life cycle. During the cycle, you will need to plan, build, continuously integrate, deploy, operate, get feedback, and start all over again. The process has to be a collaborative one, since everybody has the same focus—to deliver value to the end user. Together with these principles, you as a software architect will need to decide the best software development process that can fit this approach. We discussed these processes in Chapter 1, *Understanding the Importance of Software Architecture*.

Defining continuous integration

When you start building enterprise solutions, collaboration is the key to getting things done faster and to meeting the user needs. Version control systems, as we discussed in Chapter 14, *Best Practices in Coding C# 8*, are essential for this process, but the tool by itself does not do the job, especially if the tool is not well configured.

As a software architect, **continuous integration (CI)** will help you to have a concrete approach for software development collaboration. When you implement it, as soon as a developer commits its code, the main code is automatically built and tested.

The good thing when you apply it is that you can motivate developers to merge their changes as fast as they can in order to minimize merge conflicts. Besides, they can share unit tests, which will improve the quality of software.

It is very simple to set up CI in Azure DevOps. In the build pipeline, you will find its option by editing the configuration, as you can see in the following screenshot:

The screenshot shows the Azure DevOps interface for managing triggers. On the left, there's a sidebar with links like Overview, Boards, Repos, Pipelines, Builds, Releases, Library, Task groups, Deployment groups, Test Plans, and Project settings. The main area is titled 'wwtravelclub - CI'. It has tabs for Tasks, Variables, Triggers (which is selected), Options, Retention, and History. Under 'Triggers', there are sections for 'Continuous integration', 'Scheduled', and 'Build completion'. The 'Continuous integration' section contains a single item: 'wwtravelclub Enabled'. To the right of these sections are two filter panels: 'Branch filters' and 'Path filters'. The 'Branch filters' panel includes a dropdown for 'Type' (set to 'Include') and a dropdown for 'Branch specification' (set to 'master'). The 'Path filters' panel also has a 'Type' dropdown (set to 'Include') and a 'Path specification' dropdown containing a single slash character ('/').

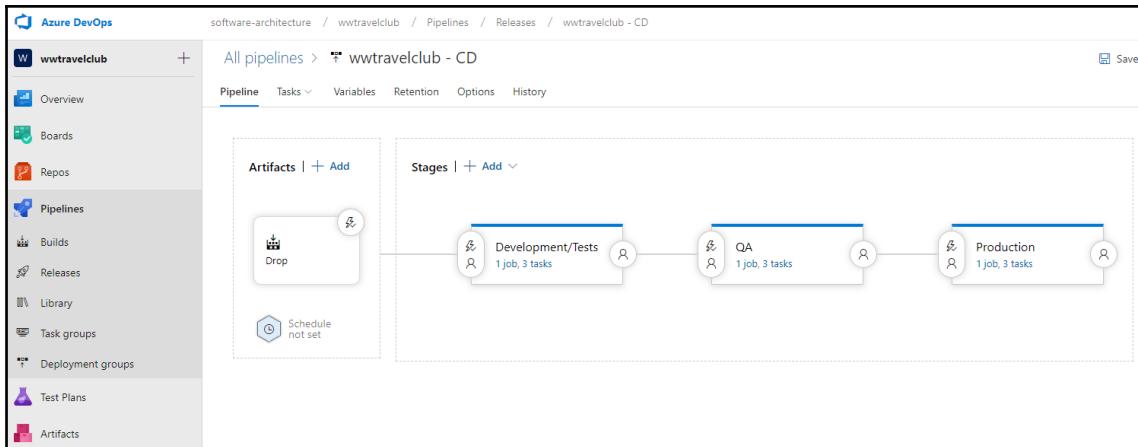
It is worth mentioning that if you have a solution set with unit and functional tests, as soon as you commit the code, it will automatically be compiled and tested. This will make your master branch stable and safe in every commit from your team.

The key point of CI is the ability to identify problems faster. You will have this opportunity when you allow the code to be tested and analyzed by others. The only thing the DevOps approach helps with is making sure this happens as fast as possible.

Understanding continuous delivery and multistage environment with Azure DevOps

Once every single commit of your application is built, and this code is tested with both unit and functional tests, you may also want to deploy it continuously. Doing this is not just a matter of configuring the tool. As a software architect, you need to be sure that the team and the process are ready to go to this step.

The approach associated with **continuous delivery (CD)** needs to guarantee that the production environment will be kept safe in each new deployment. To do so, a multistage pipeline needs to be adopted. The following screenshot shows an approach with common stages for this end:



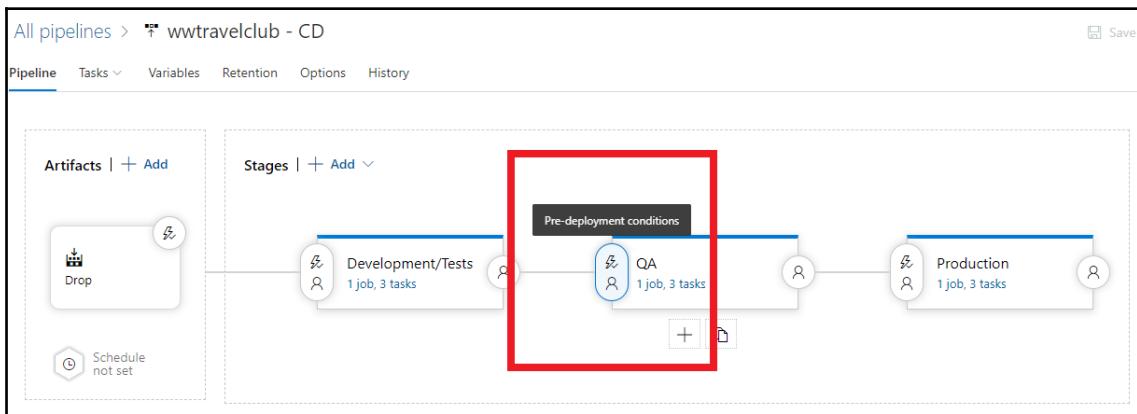
Release stages using Azure DevOps

As you can see, these stages were configured using the Azure DevOps release pipeline. Each stage has its own purpose, which will leverage the quality of the product delivered in the end. Let's look at the stages:

- **Development/tests:** This stage is used by developers and testers to build new functionality. This environment will certainly be the one that's most exposed to bugs and incomplete functions.
- **Quality assurance:** This environment gives a brief version of new functionalities to areas of the team not related to development and tests. Program managers, marketing, vendors, and others can use it as an area of study, validation, and even preproduction. Besides, the development and quality teams can guarantee that the new releases are correctly deployed, considering both functionality and infrastructure.
- **Production:** This is the stage where customers have their solution running. The goal for a good production environment, according to CD, is to have it updated as quickly as possible. The frequency will vary according to team size, but there are some approaches where this process happens more than once a day.

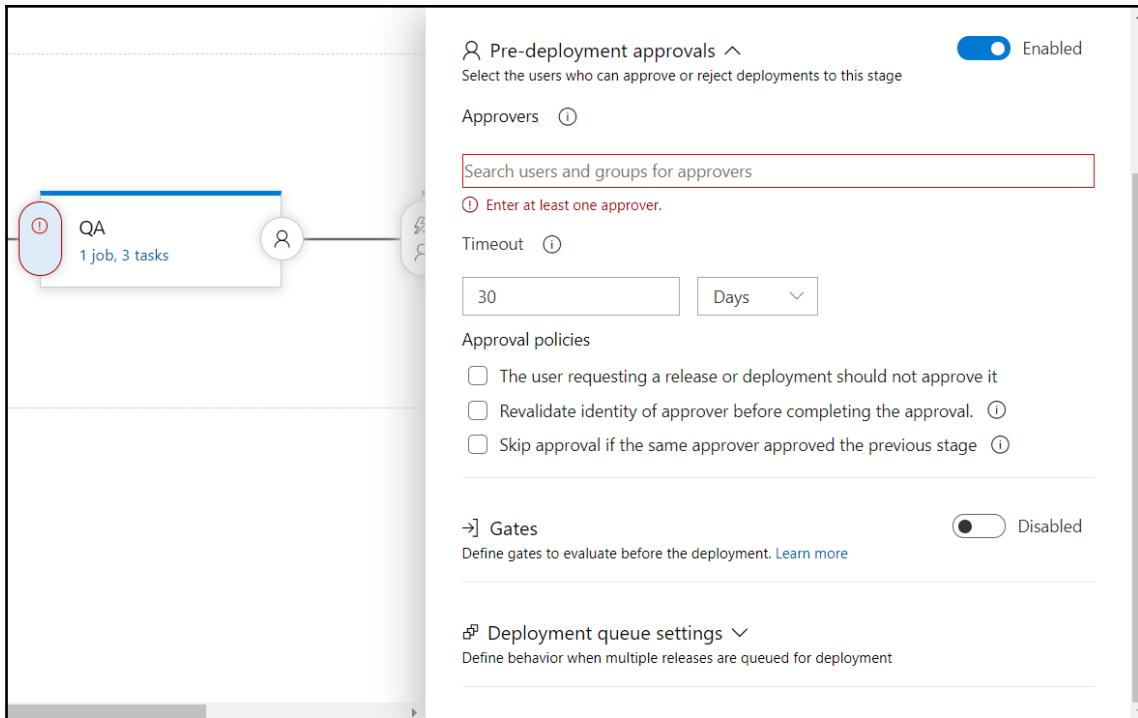
The adoption of three stages of deploying your application will impact on the quality of the solution. Besides, it will enable the team to have a safer process of deployment, with fewer risks and better stability of the product. This approach may look a bit expensive at first sight, but without it the results of bad deployment will generally be more expensive than this investment.

Besides all the safety, you will have to consider the multistage scenario. You can set up the pipeline in a way where only with defined authorizations will you be able to transition from one stage to another:



As you can see in the preceding screenshot, it is quite simple to set up pre-deployment conditions, and you can see in the following screenshot that there is more than a single option to customize the authorization method. This gives you the possibility to refine the CD approach, reaching exactly the needs of the project you are dealing with.

The following screenshot shows the options provided by Azure DevOps for pre-deployment approval. You can define the people who can approve the stage and set policies for them, that is, revalidate the approver identity before completing the process. You, as a software architect, will need to identify the configuration that fits the project you are creating with this approach:



It is worth mentioning that although this approach is far better than a single-stage deployment, a DevOps pipeline will direct you, as a software architect, to another stage of monitoring. Continuous feedback will be an incredible tool for this, and we will discuss this approach in the next section.

Defining continuous feedback and the related DevOps tools

Once you have a solution that is running perfectly in the deployment scenario described in the last section, feedback will be essential for your team to understand the results of the release and how the version is working for the customers. To get this feedback some tools can help both the developers and the customers, bringing these people together to fast-track process of feedback. Let's have a look at these tools.

Monitoring your software with Application Insights

Application Insights is definitely the tool a software architect needs to have for continuous feedback on their solution. As soon as you connect your app to it, you start receiving feedback on each request made to the software. This enables you to monitor not only the requests made but your database performance, the errors that the application may be suffering from, and the calls that take the most time to process.

Obviously, you will have costs relating to having this tool plugged into your environment, but the facilities that the tool provides will definitely be worth it. Besides, you need to understand that there is a very small performance cost since all the requests to store data in **Application Insights** run in a separate thread. The following screenshot shows how easily you can create a tool in your environment:

The screenshot shows the Microsoft Azure portal interface. On the left, a dark sidebar lists various services: Create a resource, Home, Dashboard, All services, FAVORITES (with a star icon), All resources, Resource groups, App Services, SQL databases, Azure Cosmos DB, Virtual machines, Load balancers, Storage accounts, Virtual networks, Azure Active Directory, and Monitor. The 'Create a resource' option is highlighted with a green plus sign icon. The main content area has a light background. At the top, a navigation bar shows 'Home > New > Application Insights > Application Insights'. A search bar at the top right contains the placeholder 'Search resources, services, and docs'. Below the navigation, the title 'Application Insights' is displayed with the subtitle 'Monitor web app performance and usage'. A sub-navigation bar below the title includes 'Basics' (which is underlined in purple), 'Tags', and 'Review + create'. A descriptive paragraph explains that Application Insights allows monitoring live web applications across complex distributed architectures, using powerful analytics tools for diagnosis and performance improvement. It supports .NET, Node.js, and Java EE, both on-premises and in public clouds. A 'Learn More' link is provided. The 'PROJECT DETAILS' section contains fields for 'Subscription' (set to 'Pay-As-You-Go'), 'Resource Group' (set to 'vwtravelclub-rg' with a 'Create new' link), 'Name' (set to 'AppInsightsSample' with a checkmark), and 'Region' (set to '(US) West US 2').

For instance, let's suppose you need to analyze the requests that take more time in your application. The process of attaching Application Insights to your web app is quite simple, considering it may be done as soon as you set up your web app. If you are not sure whether Application Insights is your web app, you can find out using the Azure portal. Navigate to **App Services** and look at the **Application Insights** settings, as shown in the following screenshot:

The screenshot shows the Azure portal interface for managing App Services. On the left, there is a navigation pane with 'Dashboard' > 'App Services' > 'wwtravelclub - Application Insights'. The main area displays the 'wwtravelclub - Application Insights' blade for an 'App Service'. The blade includes a search bar, a sidebar with links like 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Security', 'Deployment' (with 'Quickstart', 'Deployment slots', and 'Deployment Center'), 'Settings' (with 'Configuration' and 'Authentication / Authorization'), and 'Application Insights' (which is highlighted with a red dashed box). A message on the right encourages turning on Application Insights without redeploying.

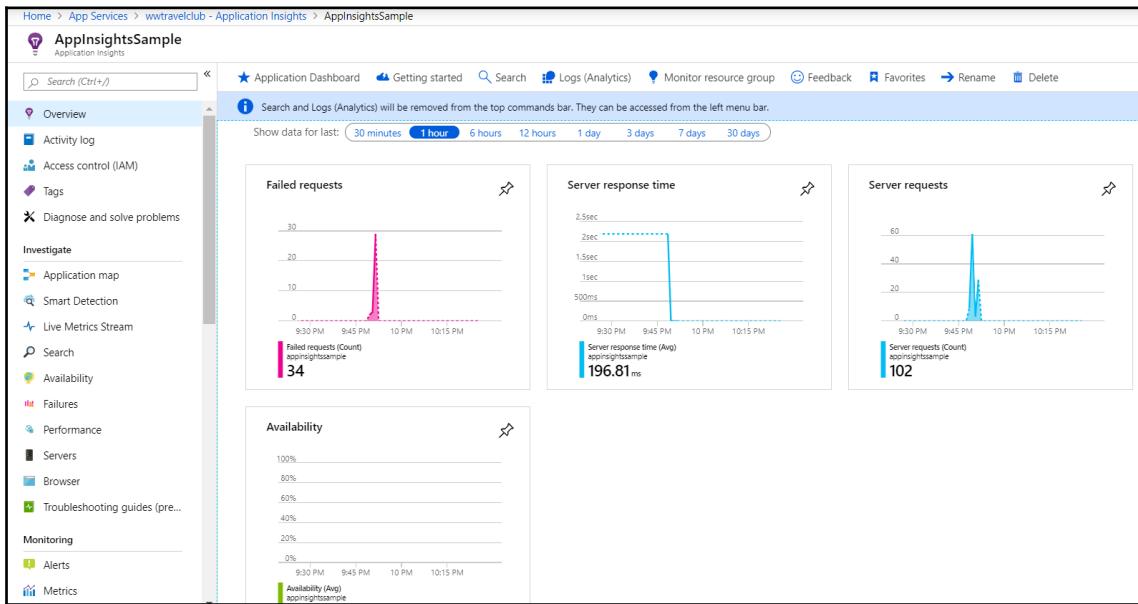
The interface will give you the opportunity to create or attach an already created monitor service to your web app. It is worth mentioning that you can connect more than one web app to the same Application Insights component. The following screenshot shows how to add a web app to an already created Application Insights resource:

The screenshot shows the Azure portal interface for managing resources. On the left, there's a sidebar titled 'All resources' with a list of resources including 'wwtravelclub', 'AppInsightsSample', 'ch16-dev', 'chapter16', 'software-architecture', 'wwtravelclub', 'wwtravelclub', 'wwtravelclub', 'wwtravelclub', 'wwtravelclubmail', and 'wwtravelclubstorage'. A red arrow points from the 'wwtravelclub' entry in the sidebar to the main content area. The main content area is titled 'wwtravelclub - Application Insights' and contains several sections: 'Application Insights' (with an 'Enable' button), 'Link to an Application Insights resource' (which states 'Your app will be connected to an auto-created Application Insights resource: **wwtravelclub**'), 'Change your resource' (with a 'Create new resource' section where a new resource named 'wwtravelclub' is being created and a 'Select existing resource' section where 'AppInsightsSample' is listed), 'Instrument your application' (with tabs for '.NET', '.NET Core', 'Node.js', and 'Java'), and 'Collection level' (with a note about gaining full APM visibility). A red arrow points from the 'Create new resource' section to the 'Name' input field, and another red arrow points from the 'Select existing resource' section to the 'AppInsightsSample' entry.

Once you have Application Insights configured for your web app, you will find the following screen in App Services:

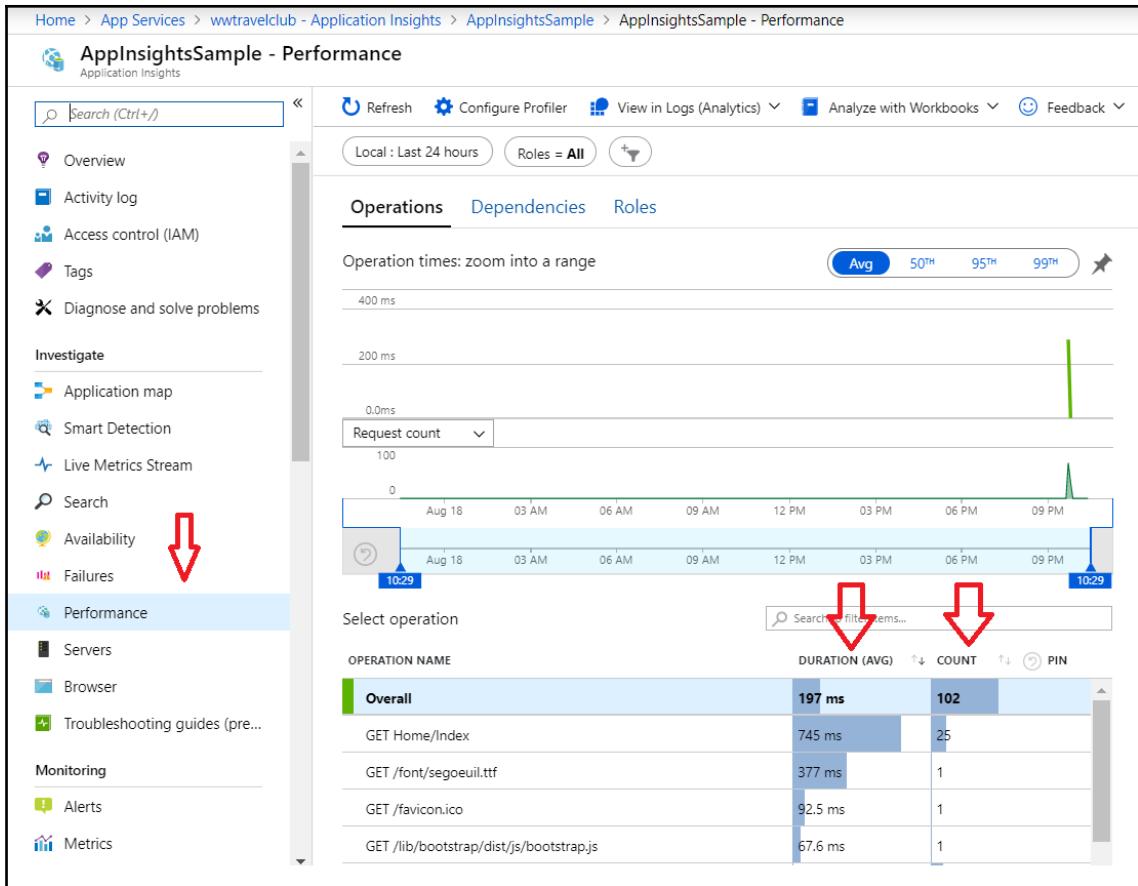
The screenshot shows the 'Application Insights' blade for the 'wwtravelclub' app service in the Azure portal. The left sidebar contains navigation links: Home, App Services, Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Security, Deployment (Quickstart, Deployment slots, Deployment Center), Settings (Configuration, Authentication / Authorization, Application Insights, Identity, Backups, Custom domains). The 'Application Insights' link is highlighted with a blue dashed border. The main content area includes a search bar, a 'View Application Insights data' button, a 'Collect application monitoring data using Application Insights' section with 'Enable' and 'Disable' buttons, a 'Feedback' button, a 'Link to an Application Insights resource' section showing the app is connected to 'ApplnightsSample', a 'Change your resource' dropdown, an 'Instrument your application' section for '.NET', and a 'Collection level' section with a 'How to get the most out of your APM data collection' link.

Once it is connected to your solution, the data collection will happen continuously and you will see the results in the dashboard provided by the component. You can find this screen in the same place as you configured Application Insights, inside the web app configurations, or in the Azure portal, navigating through the Application Insights resource:



This dashboard gives you an idea of failed requests, server response time, and server requests. You may also turn on the availability check, which will make requests to your selected URL from any of the Azure data centers.

But the beauty of Application Insights is related to how deeply it analyzes your system. In the following screenshot, for instance, it is giving you feedback on the number of requests done on the website. You can analyze it by ranking the ones that took more time to process or the ones that were called more often:



Considering this view can be filtered in different ways and you receive the info just after it happens in your web app, this is certainly a tool that defines continuous feedback. This is one of the best ways you can use the DevOps principles to achieve exactly what your customer needs.

Application Insights is a technical tool that does exactly what you as a software architect need to monitor modern applications in a real analytic model. It is a continuous feedback approach based on the behavior of users of the system you are developing.

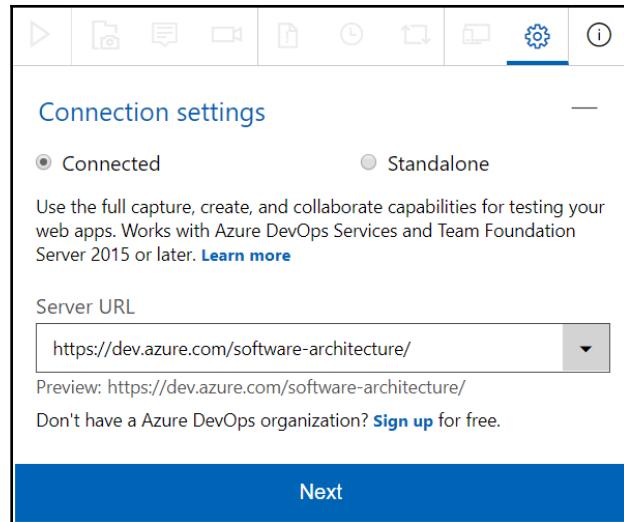
Using the Test and Feedback tool to enable feedback

Another really useful tool in the process of continuous feedback is the Test and Feedback tool, designed by Microsoft to help product owners and quality assurance users in the process of analyzing new features.

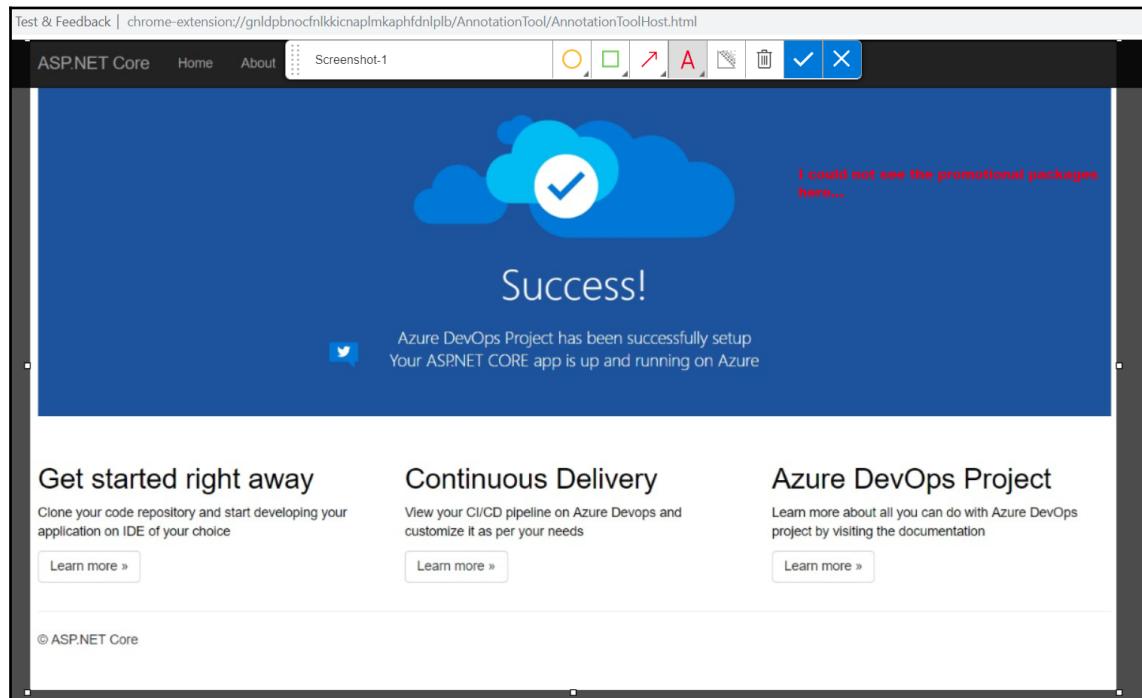
Using Azure DevOps, you may ask for feedback to your team by selecting an option inside each working item, as you can see in the following screenshot:

The screenshot shows the Azure DevOps interface for work items. A context menu is open on a specific work item, with the 'Request feedback' option highlighted by a red arrow. The menu also includes options like 'Follow', 'New linked work item', 'Change type...', 'Move to team project', 'Create copy of work item...', 'Email work item', 'Delete', 'Templates', 'New branch...', 'Do exploratory testing', and 'Customize'. The work item details include a description about promotional packages, acceptance criteria, and a discussion section. The 'Development' tab is selected, showing priority (2), effort (10), business value, and a value area labeled 'Business'.

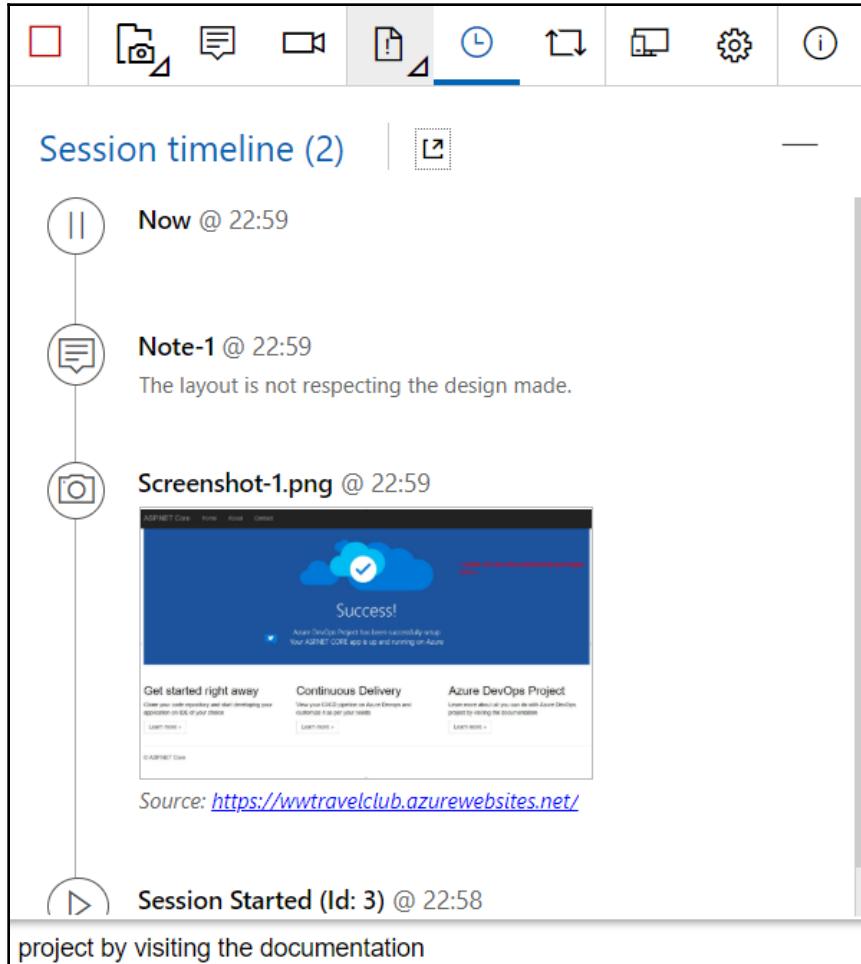
Once you receive a feedback request, you may use the Test and Feedback tool to analyze and give the correct feedback to the team. You will be able to connect the tool to your Azure DevOps project, giving you more features while analyzing the feedback request. The following screenshot shows how to set up Azure DevOps project URL for the Test and Feedback tool. You can download this tool from <https://marketplace.visualstudio.com/items?itemName=ms.vss-exploratorytesting-web>.



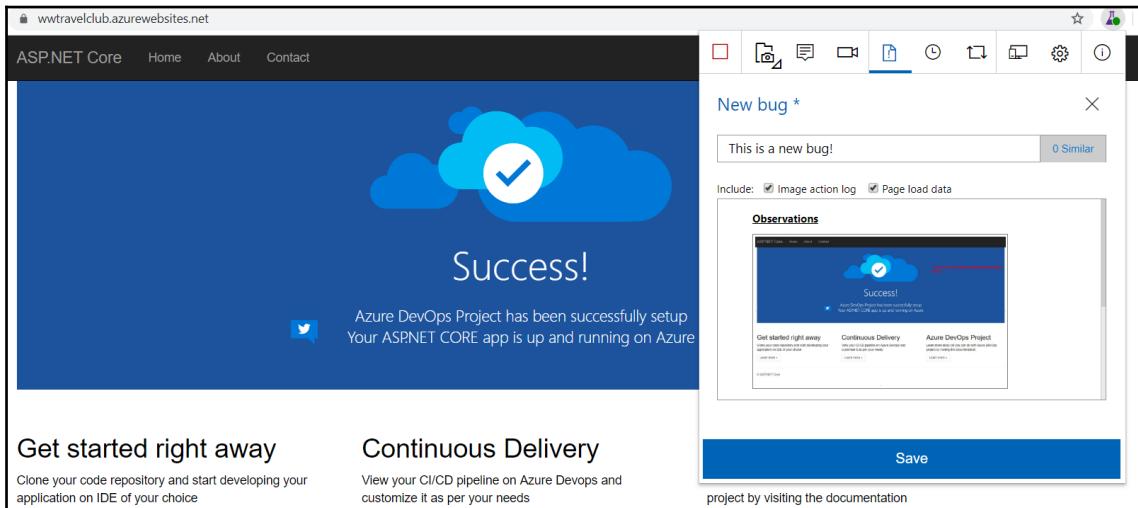
The tool is quite simple. You can take screenshots, record a process, or even make a note. The following picture shows how easily you can write a message inside the screenshot:



The good thing is that you record all this analysis in a session timeline. As you can see in the next screenshot, you can have more feedback in the same session, which is good for the analyzing process:



Once you have the analysis done and you are connected to Azure DevOps, you will be able to report a bug, create a task, or even start a new test case:



The result of the bug created can be checked in the **Work items** board in Azure DevOps. It is worth mentioning that you don't need a developer license of Azure DevOps to have access to this area of the environment. This enables you, as a software architect, to spread this basic and really useful tool to many key users of the solution you are building.

The following screenshot shows the bug created by the tool once you have connected it to your Azure DevOps project:

The screenshot shows the Azure DevOps interface for the 'software-architecture' project under the 'wwtravelclub' organization. The left sidebar is visible with options like Overview, Boards, Work items, Boards, Backlogs, Sprints, Queries, Repos, Pipelines, Test Plans, and Artifacts. The 'Work items' option is selected. The main area displays a table of work items with columns for ID, Title, and a small icon. A red arrow points to the last item in the list, which has the title 'This is a new bug'. The table data is as follows:

ID	Title
2	Customer View
3	Destination Expert View
4	Administrator View
5	Package Management
6	US_001: As a common user, I want to view promotional packages i...
7	Design Database
8	Implement MVC Structure
9	View promotional packages
21	This is a new bug

It is definitely important to have a tool like this to have good feedback on your project. But as a software architect, you may have to find the best solutions to accelerate this process. The tools explored in the book are good ways to do so. You may consider this approach every time you need to implement one more step in the development process.

The WWTravelClub project approach

During this chapter, screenshots from the WWTravelClub project have shown the steps needed to implement a good DevOps cycle. The WWTravelClub team has decided to use Azure DevOps because they understand that the tool is essential for getting the best DevOps experience for the whole cycle.

The requirements were written using user stories, which can be found in the work items section of Azure DevOps. The code is placed in the repository of the Azure DevOps project. Both concepts were explained in Chapter 3, *Documenting Requirements with Azure DevOps*.

The management life cycle used for getting things done is Scrum, presented in Chapter 1, *Understanding the Importance of Software Architecture*. This approach divides the implementation into Sprints, which forces the need to deliver value by the end of each cycle. Using the continuous integration facilities we learned in this chapter, code will be compiled each time the team concludes a development to the master branch of the repository.

Once the code is compiled and tested, the first stage of the deployment is done. The first stage is normally named Development/Test because you enable it for internal tests. Both Application Insights and Test and Feedback can be used to get the first feedback on the new release.

If the tests and the feedback of the new release pass, it is time to go to the second stage—Quality Assurance. Application Insights and Test and Feedback can be used again, but now in a more stable environment.

The cycle ends with the authorization to deploy in the production stage. This certainly is a tough decision, but DevOps indicates that you have to do it continuously so you can get better feedback from customers. Application Insights keeps being a really useful tool, since you are able to monitor the evolution of the new release in production, even comparing it to the past releases.

The WWTravelClub project approach described here can be used in many other modern application development life cycles. You, as a software architect, are in charge of making this happen. The tools are ready to go, and it depends on you to make things right!

Summary

In this chapter, we learned how DevOps is not only a bunch of techniques and tools used together to deliver software continuously but a philosophy to enable continuous delivery of value to the end user of the project you are developing.

Considering this approach, we saw how continuous integration, continuous delivery, and continuous feedback are essential to the purpose of DevOps. We also saw how Azure, Azure DevOps, and Microsoft tools help you to achieve your goals.

This chapter brought you this approach using WWTravelClub as an example, enabling CI/CD inside Azure DevOps, and using Application Insights and the Test and Feedback tool for both technical and functional feedback. In real life, these tools will enable you to understand the current behavior of the system you are developing faster, as you will have continuous feedback on it.

In the next chapter, we will learn about continuous integration in detail.

Questions

1. What is DevOps?
2. What is continuous integration?
3. What is continuous delivery?
4. What is continuous feedback?
5. What is the difference between the build and release pipelines?
6. What is the main purpose of Application Insights in the DevOps approach?
7. How can the Test and Feedback tool help in the process of DevOps?

Further Reading

These are some websites where you will find more information on the topics covered in this chapter:

- <http://donovanbrown.com/>
- <https://www.packtpub.com/networking-and-servers/devops-fundamentals-video>
- <https://docs.microsoft.com/en-us/azure/devops/learn/what-is-devops>
- <https://azuredavopslabs.com/labs/devopsserver/exploratorytesting/>
- <https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview>
- <https://marketplace.visualstudio.com/items?itemName=ms.vss-exploratorytesting-web>
- <https://docs.microsoft.com/en-us/azure/devops/test/request-stakeholder-feedback>

19

Challenges of Applying CI Scenarios in DevOps

Continuous Integration (CI) is a step ahead of DevOps. In the previous chapter, we discussed the basics of CI and how DevOps depends on it. Its implementation was presented in Chapter 18, *Understanding DevOps Principles*, too, but differently from the other practical chapters, the purpose of this chapter is discussing how to enable CI in a real scenario, considering the challenges that you, as a software architect, will need to deal with.

The topics covered in this chapter are as follows:

- Understanding CI
- Understanding the risks and challenges when using CI
- Understanding the WWTravelClub project approach for this chapter

Like in the previous chapter, the sample of the WWTravelClub will be presented during the explanation of the chapter, since all the screens captured to exemplify CI came from it. Besides this, we will offer a conclusion at the end of the chapter so you can understand CI principles easily.

By the end of the chapter, you will be able to decide whether or not to use CI in your project environment. Additionally, you will be able to define the tools needed for the successful use of this approach.

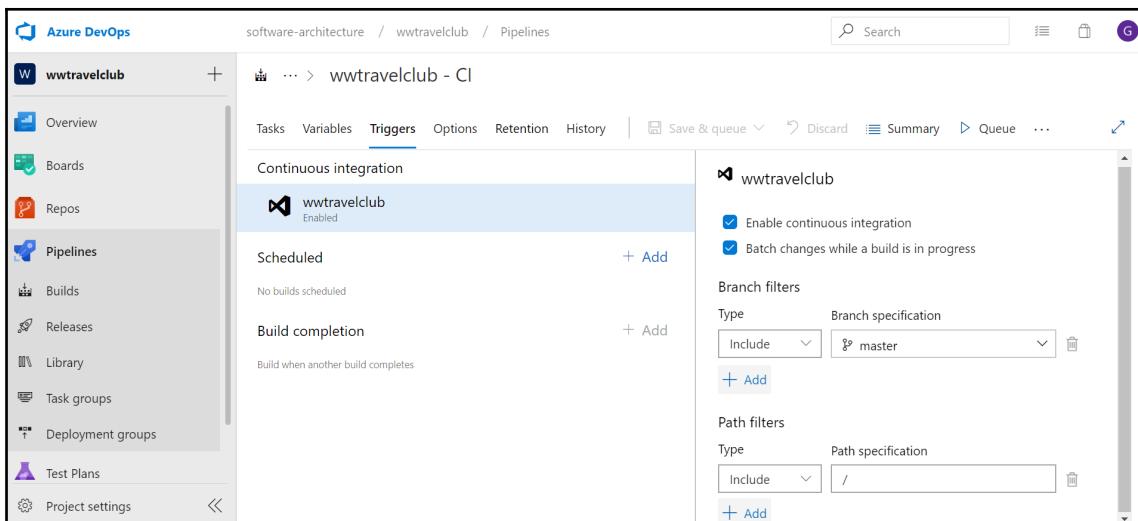
Technical requirements

This chapter requires Visual Studio 2019 Community Edition or better. You may also need an Azure DevOps account, as described in Chapter 3, *Documenting Requirements with Azure DevOps*.

Understanding CI

As soon as you start working with a platform such as Azure DevOps, enabling CI will definitely be easy when it comes to clicking on the options for doing so, as we saw in Chapter 18, *Understanding DevOps Principles*. So, technology is not the Achilles' heel for implementing this process.

The following screenshot shows an example of how easy it is to turn on CI using Azure DevOps. By clicking in the build pipeline and editing it, you will be able to set a trigger that enables CI after some clicks:



The truth is that CI will help you solve some problems. For instance, it will force you to test your code, since you will need to commit the changes faster, so other developers can make use of the code you are programming.

On the other hand, you will not enable CI just by clicking on the preceding screenshot. For sure, you will turn on the possibility of starting a build as soon as you get a commit done and the code is done, but this is far from saying you have CI available in your solution.

The reason why you as a software architect need to worry a bit more about it is related to the real understanding of what DevOps is. As discussed in Chapter 18, *Understanding DevOps Principles*, the need to deliver value to the end user will always be a good way to decide and draw the development life cycle. So, even if turning on CI is easy, what is the impact of this feature being enabled for your end user? Once you have all the answers to this question and you know how to reduce the risks of its implementation, then you will be able to say that you have a CI process implemented.

It is worth mentioning that CI is a principle that will make DevOps work better and faster, as was discussed in Chapter 18, *Understanding DevOps Principles*. However, DevOps surely can live without it, once you are not sure if your process is mature enough to enable code being continuously delivered. More than that, if you turn on CI in a team that is not mature enough to deal with its complexity, you will probably cause a bad understanding of DevOps, since you will start incurring some risks while deploying your solution.

This is the reason why we are dedicating an extra chapter on CI. You need to understand the risks and challenges you will have as a software architect once you turn CI on.

Understanding the risks and challenges when using CI

Now, you may be thinking about the risks and challenges as a way for you to avoid using CI. But why should we avoid using it if it will help you do a better DevOps process? This is not the purpose of the chapter. The idea of this section is to help you, as a software architect, to mitigate the risks and find a better way to pass through the challenges using good processes and techniques.

The list of risks and challenges that will be discussed in the chapter are as follows:

- Continuous production deployment
- Incomplete features in the production
- Unstable solutions for testing

Once you have the techniques and the processes defined to deal with them, there is no reason to not use CI. It is worth mentioning that DevOps does not depend on CI. However, it does make DevOps work more softly. Now, let's have a look at them.

Disabling continuous production deployment

Continuous production deployment is a process where, after a commit of a new piece of code and some pipeline steps, you will have this code in the **production** environment. This is not impossible but is really hard and expensive to do. Besides, you need to have a really mature team. The problem is that most of the demos and samples you will find on the internet presenting CI will show you a fast-track to deploy the code. The demonstrations of CI/CD look so simple and easy to do! This *simplicity* can suggest you work as soon as possible on its implementation. However, if you think a little more, this scenario can be dangerous if you deploy directly in production! In a solution that needs to be available 24 hours a day, 7 days a week, this is impractical. So, you will need to worry about that and think of different solutions.

The first one is the use of a multi-stage scenario, as described in Chapter 18, *Understanding DevOps Principles*. The multi-stage scenario can bring more security to the ecosystem of the deployment you are building. Besides, you will get more options to avoid wrong deployments into production, such as pre-deployment approvals:

The screenshot shows a pipeline configuration for the 'wwtravelclub - CD' pipeline. The pipeline consists of two stages: 'QA' and 'Production'. The 'QA' stage has 1 job and 3 tasks. The 'Production' stage also has 1 job and 3 tasks. On the right side of the screen, there are several configuration sections:

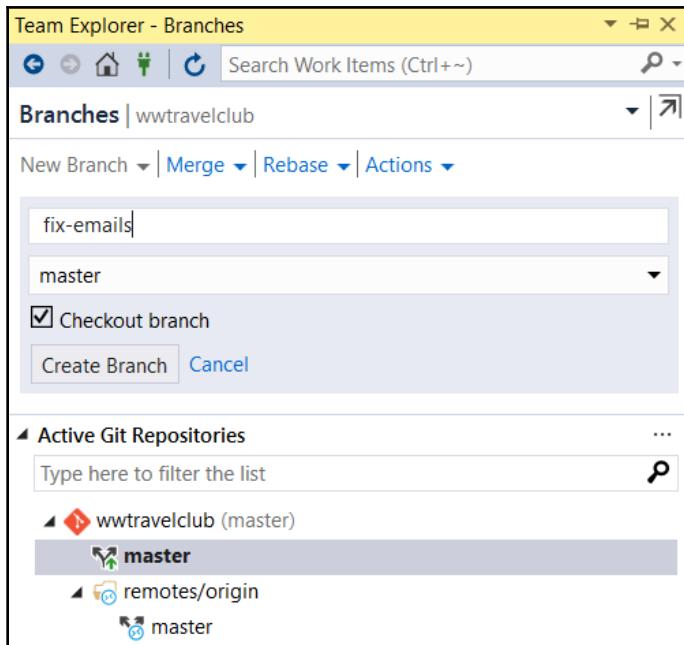
- Pre-deployment approvals**: A toggle switch is set to **Enabled**. A note says: "Select the users who can approve or reject deployments to this stage". Below it, under "Approvers", there is a search bar containing "Gabriel" and a placeholder "Search users and groups for approvers".
- Timeout**: Set to 30 Days.
- Approval policies**: Three checkboxes are present:
 - The user requesting a release or deployment should not approve it
 - Revalidate identity of approver before completing the approval. (i)
 - Skip approval if the same approver approved the previous stage (i)
- Gates**: A toggle switch is set to **Enabled**. A note says: "Define gates to evaluate before the deployment. Learn more".
- Deployment queue settings**: A note says: "Define behavior when multiple releases are queued for deployment".

It is worth mentioning, too, that you can build a deployment pipeline where all your code and software structure will be updated by this tool. However, if you have something out of this scenario, such as database scripts and environment configurations, a wrong publication into production may cause damage to end users. Besides, the decision of when the production will be updated needs to be planned and, in many scenarios, all the platform users need. Use a *change management* procedure in these cases needs to be decided.

So, the challenge of delivering code to production will make you think about a schedule to do so. It does not matter if your cycle is monthly, daily, or even at each commit. The key point here is that you need to create a process and a pipeline that guarantees that only good and approved software is in the production stage.

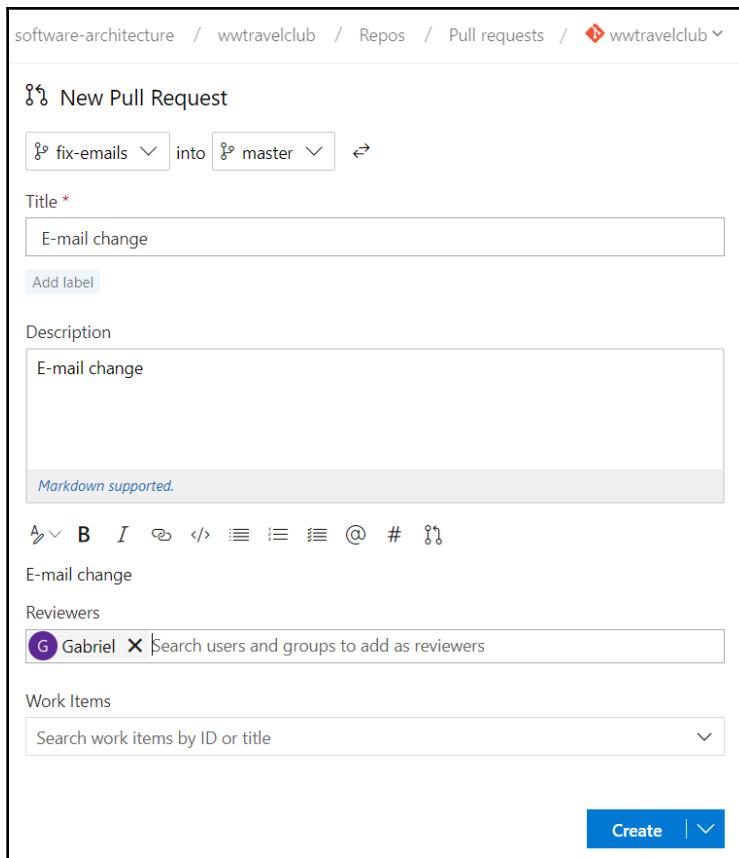
Incomplete features

While a developer of your team is creating a new feature or fixing a bug, you will probably consider generating a branch that can avoid the use of the branch designed for continuous delivery. A branch can be considered a feature available in code repositories to enable the creation of an independent line of development since it isolates the code. As you can see in the following screenshot, creating a branch using Visual Studio is quite simple:

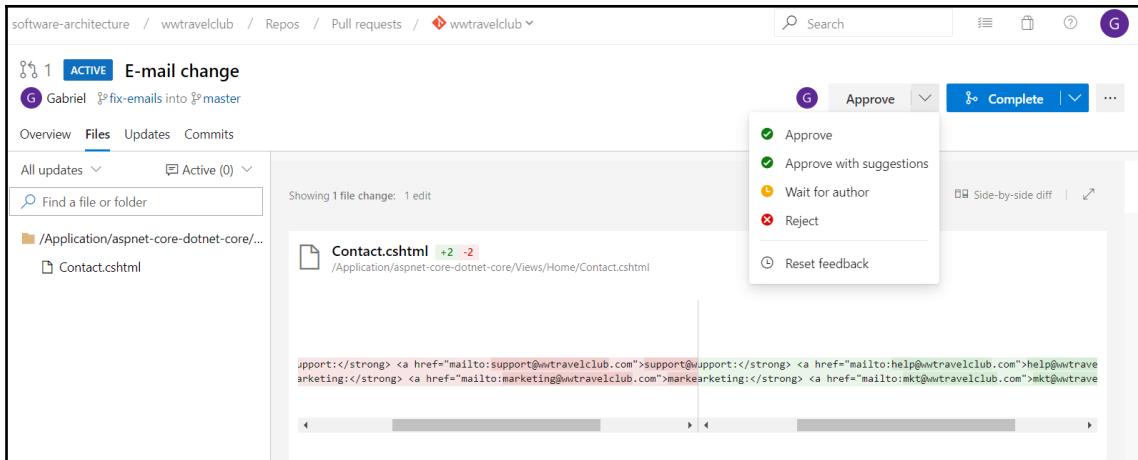


This seems to be a good approach, but let's suppose that the developer has considered the implementation ready for deploying and has just merged the code into the master branch. What if this feature is not ready yet, just because a requirement was omitted? What if the bug has caused an incorrect behavior? The result can be a release with an incomplete feature or an incorrect fix.

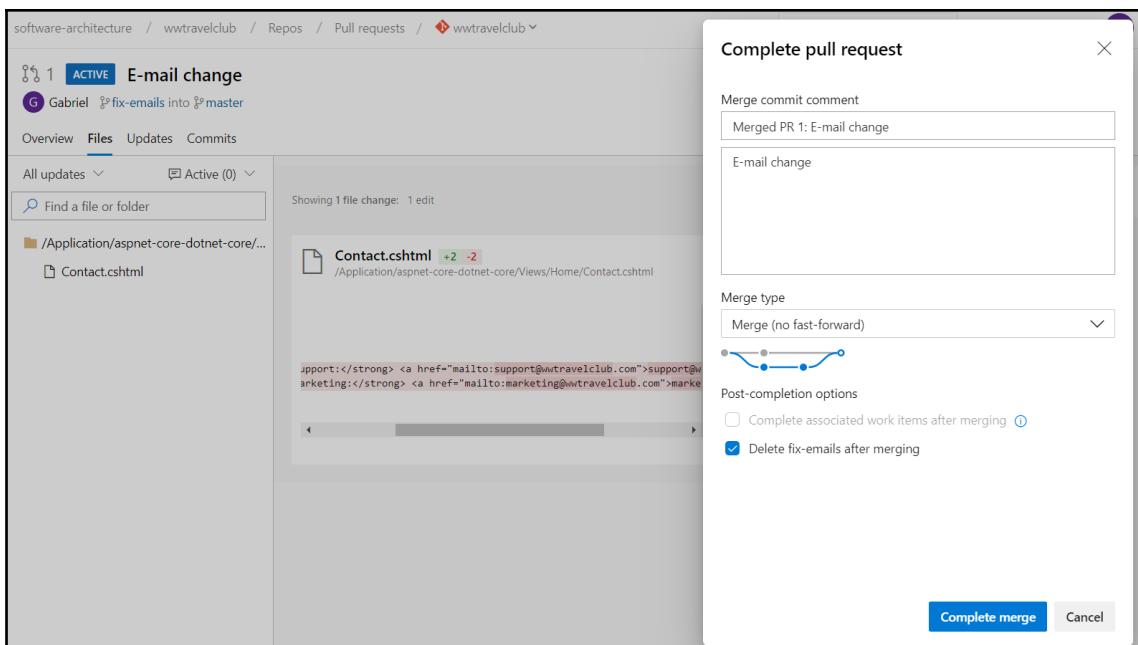
A good practice to avoid broken features and even wrong fixes in the master branch is the use of pull requests. Pull requests will let other team developers know that the code you developed is ready to be merged. The following screenshot shows how you can use Azure DevOps to create a **New Pull Request** for a change you have made:



Once the pull request is created and the reviewers are defined, each reviewer will be able to analyze the code and decide whether this code is healthy enough to be in the master branch. The following screenshot shows a way to check it by using the compare tool to analyze the change:



Once all approvals are done, you will be able to safely merge the code to the master branch, as you can see in the following screenshot. To merge the code, you will need to click on **Complete merge**. If the CI trigger is enabled, as shown earlier in the chapter, Azure DevOps will start a build pipeline:



There is no way to argue that without a process like this, the master branch will suffer from many bad code being deployed that can cause damage together with CD. It is worth mentioning that the code review is an excellent practice in CI/CD scenarios, and it is considered a wonderful practice for creating good quality in any software as well.

The challenge that you need to focus on here is guaranteeing that only entire features will appear to your end users. You may use for solving it the feature flag principle, which is a technique that makes sure only features that are ready are presented to end users. Again, we are not talking about CI as a tool, but as a process to be defined and used every single time you need to deliver code for production.

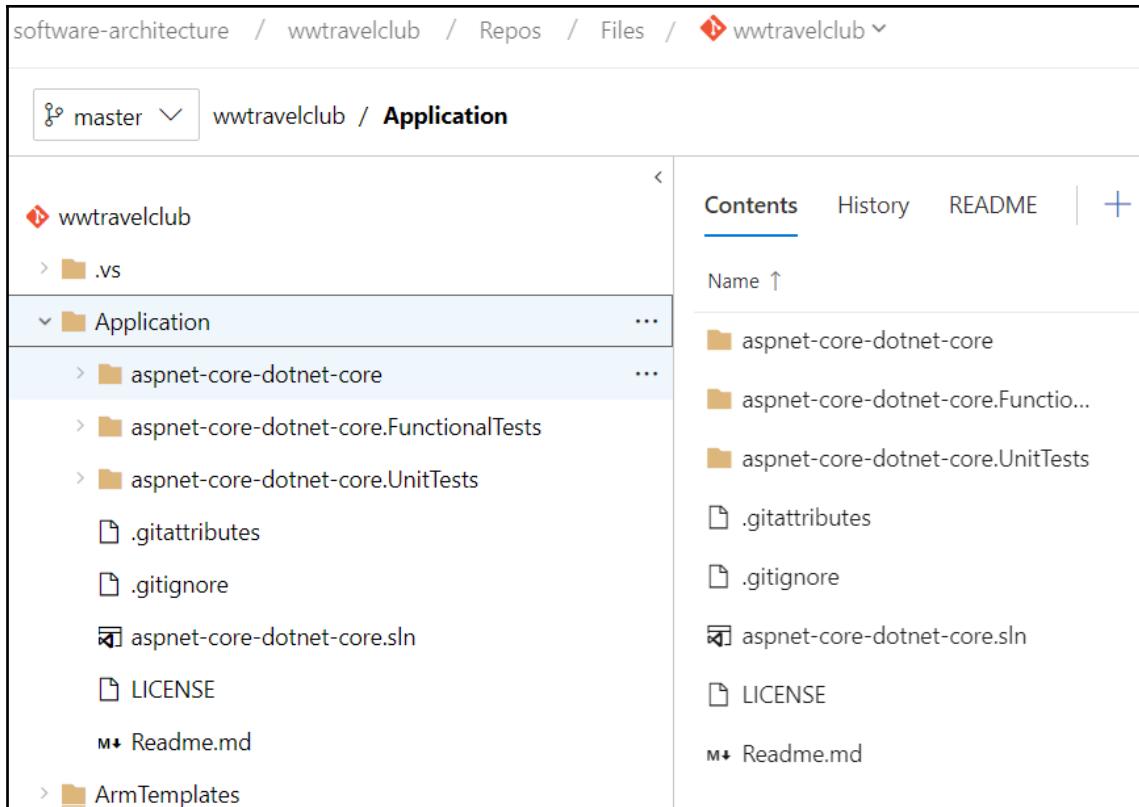
Unstable solution for testing

Considering that you have already mitigated the two other risks presented in this topic, you may find it pretty hard to have bad code after CI. It is true that the worries presented earlier will certainly be lower considering the fact that you are working with a multi-stage scenario and pull requests before pushing to the first stage.

But is there a way to accelerate the evaluation of release, being sure that this new release is ready for your stakeholder's tests? Yes, there is! Technically, the way you can do so is described in the use cases of Chapter 15, *Testing Your Code with Unit Test Cases and TDD*, and Chapter 20, *Automation for Software Testing*.

As discussed in both those chapters, it is impracticable to automate every single part of the software, considering the efforts needed to do so. Besides, the maintenance of automation can be more expensive in scenarios where the user interface or the business rules change a lot.

To exemplify it, let's have a look at the following screenshot, which shows the unit and functional tests created by Azure DevOps when the WWTravelClub project is started:



There are some architectural patterns, such as SOLID, presented in Chapter 9, *Design Patterns and .NET Core Implementation*, and quality assurance approaches, such as peer review, that will give you better results than software testing.

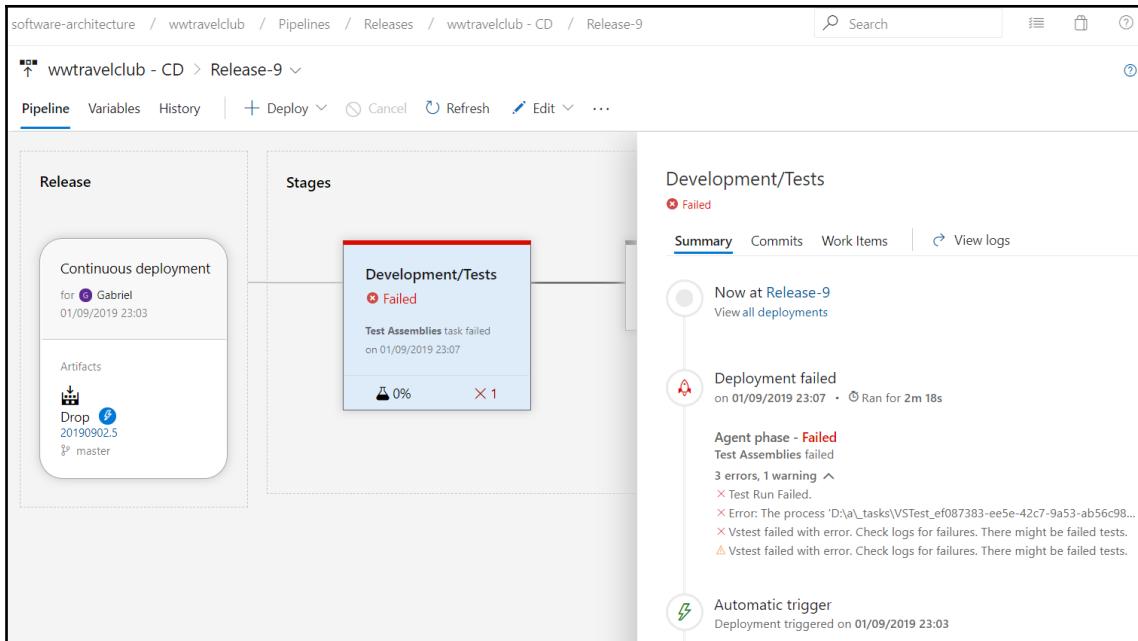
However, these approaches do not invalidate automation practice. The truth is that all of them will be useful for getting a stable solution, especially when you are running a CI scenario. In this environment, the best thing you can do is to detect errors and wrong behaviors as fast as you can. Both unit and functional tests, as shown earlier, will help you with this.

Unit tests will help you a lot while discovering business logic errors before deployment, during the building pipeline. For instance, in the following screenshot, you will find a simulated error that canceled the build since the unit test did not pass:

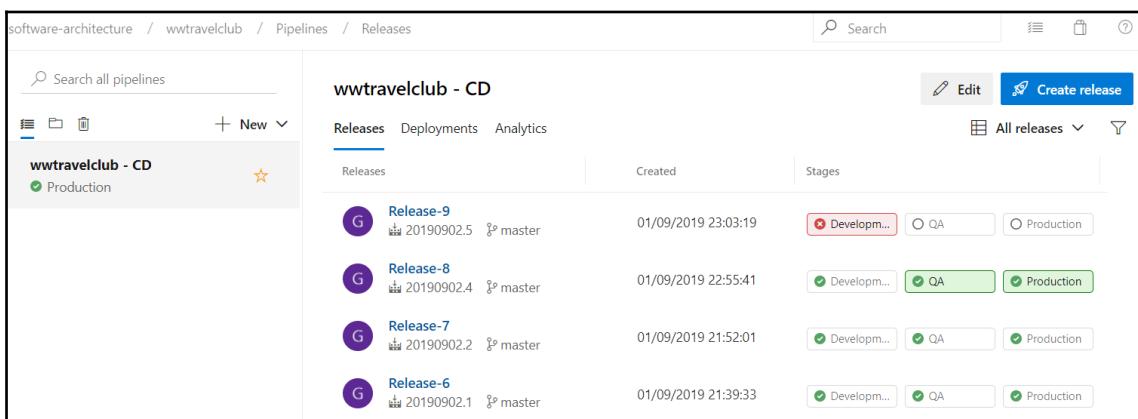
The screenshot shows a build log for a pipeline named 'Updated SampleUnitTests.cs'. The build was triggered today at 22:43 for Gabriel on the 'master' branch. The pipeline consists of several steps: 'Prepare job' (succeeded), 'Initialize job' (succeeded), 'Checkout' (succeeded), 'Restore' (succeeded), 'Prepare analysis on SonarCloud' (succeeded), 'Build' (succeeded), and 'Test' (failed). The 'Test' step failed with 2 errors, which are highlighted in a black box. The errors are: 'Error: The process 'c:\Program Files\dotnet\dotnet.exe' failed with exit code 1' and 'Dotnet command failed with non-zero exit code on the following projects : D:\a\1\s\Application\aspnet-core-dotnet-core.UnitTests\aspnet-core-dotnet-core.UnitTests.csproj'. The build started at 01/09/2019 22:43:54 and took 3m 41s.

The way to get this error is quite simple. You need to code something that does not respond according to what the unit tests are checking. Once you commit it, considering you have the trigger of continuous deployment on, you will have the code building in the pipeline. One of the last steps provided by the Azure DevOps Project Wizard we have created is the execution of the unit tests. So, after the build of the code, the unit tests will run. If the code does not match the tests anymore, you will get the error.

Meanwhile, the following screenshot shows an error during the functional tests in the **Development/Tests** stage. At this moment, the **Development/Tests** environment has a bug that was rapidly detected by functional tests:



But this is not the only good thing about applying functional tests in the process of CI/CD, once you have protected other deployment stages with this approach. For instance, let's take a look at the following screenshot from the **Release** pipeline interface in Azure DevOps. If you look at **Release-9**, you will realize that since this error happened after the publication in the **Development/Tests** environment, the multi-staged environment will protect the other stages of the deployment:



The key point to success in the CI process is to think about it as a useful tool for accelerating the delivery of software and to not forget that a team always needs to deliver value to their end users. With this approach, the techniques presented earlier will provide incredible ways to achieve the results that your team aims for.

Understanding the WWTravelClub project approach

During the chapter, WWTravelClub project screenshots were present, exemplifying the steps for getting a safer approach while enabling CI. Even considering WWTravelClub as a hypothetical scenario, some concerns were taken into account while building it:

- CI is enabled, but a multi-stage scenario is enabled too.
- Even with a multi-stage scenario, the pull request is a way to guarantee that only code with good quality will be presented in the first stage.
- To do a good job in the pull request, peer reviews are undertaken.
- The peer reviews check, for instance, the presence of a feature flag while creating a new feature.
- The peer reviews check both unit and functional tests developed during the creation of the new feature.

The preceding steps are not exclusively for WWTravelClub. You as a software architect will need to define the approach to guarantee a safe CI scenario. You may use this as a starting point.

Summary

This chapter covered the importance of understanding when you can enable CI in the software development life cycle, considering the risks and challenges you will take as a software architect once you decide to have it done in your solution.

Additionally, the chapter introduced some solutions and concepts that can make this process easier, such as multi-stage environments, pull request reviews, feature flags, peer reviews, and automated tests. Understanding these techniques and processes will enable you to guide your project to a safer behavior when it comes to CI in a DevOps scenario.

In the next chapter, we will see how automation for software testing works.

Questions

1. What is CI?
2. Can you have DevOps without CI?
3. What are the risks of enabling CI in a non-mature team?
4. How can a multi-stage environment help CI?
5. How can automated tests help CI?
6. How can pull requests help CI?
7. Do pull requests only work with CI?

Further reading

These are some websites where you will find more information on the topics covered in this chapter:

- <https://azure.microsoft.com/en-us/solutions/architecture/azure-devops-continuous-integration-and-continuous-deployment-for-azure-web-apps/>
- <https://docs.microsoft.com/en-us/azure/devops-project/azure-devops-project-github>
- <https://docs.microsoft.com/en-us/aspnet/core/azure/devops/cicd>
- <https://www.packtpub.com/virtualization-and-cloud/professional-microsoft-azure-devops-engineering>
- <https://www.packtpub.com/virtualization-and-cloud/hands-devops-azure-video>
- <https://www.packtpub.com/networking-and-servers/implementing-devops-microsoft-azure>
- <https://docs.microsoft.com/en-us/azure/devops/repos/git/pullrequest>
- <https://devblogs.microsoft.com/devops/whats-new-with-azure-pipelines/>
- <https://martinfowler.com/bliki/FeatureToggle.html>

20

Automation for Software Testing

In previous chapters, we discussed the importance of unit tests and integration tests in software development, and how they ensure the reliability of your code base. We also discussed how unit and integration tests are integral parts of all software production stages and are run each time the code base is modified.

There are also other important tests, called **functional/acceptation** tests. They are run only at the end of each sprint to verify that the output of the sprint actually satisfies the specifications that were agreed upon with the stakeholders.

This chapter is specifically dedicated to functional/acceptance tests and to the techniques for defining and executing them. More specifically, this chapter covers the following topics:

- Understanding the purpose of functional tests
- Using unit testing tools for automating functional tests in C#
- Use case – automating functional tests

By the end of this chapter, you will be able to design both manual and automatic tests to verify that the code produced by a sprint complies with its specifications.

Technical requirements

The reader is encouraged to read Chapter 15, *Testing Your Code with Unit Test Cases and TDD*, before proceeding with this chapter.

This chapter requires Visual Studio 2017 or the 2019 free Community Edition or better with all the database tools installed. Here, we will modify the code of Chapter 15, *Testing Your Code with Unit Test Cases and TDD*, which is available at <https://github.com/PacktPublishing/Hands-On-Software-Architecture-with-CSharp-8/tree/master/ch20>.

Understanding the purpose of functional tests

Functional/acceptance tests use techniques similar to unit and integration tests but differ from them in that they are run only at the end of each sprint. They have the fundamental role of verifying that the current version of the whole software complies with its specifications. This verification is turned into a formal process for the following purposes:

- Functional tests represent the most important part of the contract between stakeholders and the development team, the other part being the verification of non-functional specifications. The way this contract is formalized depends on the very nature of the relationship between the development team and stakeholders. In the case of a supplier-customer relationship, they become part of the supplier-customer business contract for each sprint, and they are written by a team that works for the customer. If the tests fail, then the sprint is rejected and the supplier must run a supplementary sprint to fix all problems. In case there is no formal business contract, the result of the tests is usually used to drive the specifications for the next sprints. However, also in this case, if the failure percentage is high, the sprint may be rejected and should be repeated.
- Formalized functional tests that run at the end of each sprint avoid that results achieved in previous sprints might be destroyed by new code.
- When using an agile development methodology, maintaining an updated battery of functional tests is the best way to get a formal representation of the final system specifications since, during agile development, the specifications of the final system are not decided before development starts but are the result of the system evolution.

Since the output of the first sprints may differ a lot from the final system in these early stages, it is not worth spending too much time writing detailed manual tests and/or automatized tests. Therefore, you may limit to add a few examples to the user stories that will be used both as inputs for software development and as manual tests.

As system functionalities become always more stable, it is worth investing time in writing detailed and formal functional tests for them. For each functional specification, we must write tests that verify their correct operation also in extreme cases. For instance, in a payment use case, we must write tests that verify all possibilities:

- Not enough funds
- Various digitization errors

- Card expired
- Wrong credentials and repeated wrong credentials

In the case of manual tests, for each of the preceding scenarios, we must give all details of all steps involved in each operation, and for each step, the expected result.

An important decision is if you want to automate all or a part of the acceptance/functional tests since it is very expansive to write automatic tests that simulate a human operator that interacts with a system's user interface. The final decision depends on the cost of the test implementation divided by the expected number of times it will be used.

In the case of CI/CD, the same functional test can be executed several times but, unluckily, functional/acceptance tests are strictly tied to the way the user interface is implemented, and, in modern systems, the user interface is frequently changed. Therefore, in this case, the same test is executed with exactly the same user interface not more than a couple of times.

In order to overcome all the problems related to the user interface, functional tests can be implemented as **subcutaneous tests**, that is, as tests that bypass the user interface.

However, subcutaneous tests are incomplete by their very nature since they can't detect errors in the user interface itself. Moreover, in the case of a web application, subcutaneous tests usually suffer from other limitations because they bypass the whole HTTP protocol. In the case of ASP.NET Core applications, this means that the whole ASP.NET Core pipeline must be bypassed and that requests are passed directly to ASP.NET controllers. Therefore, authentication, authorization, CORS, and the behavior of other modules in the ASP.NET Core pipeline will not be analyzed by the tests.

A complete automatic functional test of a web application should do the following things:

1. Start an actual browser on the URL to be tested.
2. Wait so that any JavaScript on the page completes its execution.
3. Then, send commands to the browser that simulate the behavior of a human operator.
4. Finally, after each interaction with the browser, automatic tests should wait so that any JavaScript that was triggered by the interaction completes.

While browser automation tools exist, tests implemented with browser automatization, as mentioned earlier, are very expensive and difficult to implement. Therefore, the suggested approach of ASP.NET Core MVC is to send actual HTTP requests to an actual copy of the web application, with a .NET HTTP client instead of using a browser. Once the HTTP client receives an HTTP response, it parses it in a DOM tree and verifies that it received the right response.

The only difference with the browser automatization tools is that the HTTP client is not able to run any JavaScript. However, other tests may be added to test the JavaScript code. These tests are based on test tools that are specific to JavaScript, such as **Jasmine** and **Karma**.

The next section explains how to automatize functional tests for web applications with .NET HTTP client, while a practical example of functional test automation is shown in the last section.

Using unit testing tools to automate functional tests in C#

Automated functional/acceptance tests use the same test tools as unit and integration tests. That is, these tests can be embedded in the same xUnit, NUnit, or MSTests projects that we described in Chapter 15, *Testing Your Code with Unit Test Cases and TDD*. However, in this case, we must add further tools that are able to interact with and inspect the user interface.

In the remainder of this chapter, we will focus on web applications since they are the main focus of this book. Accordingly, if we are testing web APIs, we just need `HttpClient` instances since they can easily interact with web API endpoints both in XML and JSON.

In the case of ASP.NET Core MVC applications that return HTML pages, the interaction is more complex, since we also need tools for parsing and interacting with the HTML page DOM tree. The AngleSharp NuGet package is a great solution since it supports state-of-the-art HTML and minimal CSS and has extension points for externally provided JavaScript engines, such as Node.js. However, we don't advise you to include JavaScript and CSS in your tests, since they are strictly tied to target browsers, so the best option for them is to use JavaScript-specific test tools that you can run directly in the target browsers themselves.

There are two basic options for testing a web application with the `HttpClient` class:

- An `HttpClient` instance connects with the actual *staging* web application through the internet/intranet, together with all other humans that are beta-testing the software. The advantage of this approach is that you are testing the *real stuff*, but tests are more difficult to conceive since you can't control the initial state of the application before each test.

- An `HttpClient` instance connects with a local application that is configured, initialized, and launched before every single test. This scenario is completely analogous to the unit test scenario. Test results are reproducible the initial state before each test is fixed, tests are easier to design, and the actual database can be replaced by a faster and easier-to-initialize in-memory database. However, in this case, you are far from the actual system's operation.

A good strategy is to use the second approach, where you have full control of the initial state, for testing all extreme cases, and then the first approach for testing random average cases on the *real stuff*.

The two sections that follow describe both approaches. The two approaches differ just in the way you define the fixtures of your tests.

Testing the staging application

In this case, your tests need just an instance of `HttpClient`, so you must define an efficient fixture that supplies `HttpClient` instances, avoiding the risk of running out of windows connections. We faced this problem in the *.NET Core HTTP clients* section of Chapter 12, *Applying Service-Oriented Architectures with .NET Core*. It can be solved by managing `HttpClient` instances with `IHttpClientFactory` and injecting them with dependency injection.

Once we have a dependency injection container, we can enrich it with the capability of efficiently handling `HttpClient` instances with the following code snippet:

```
services.AddHttpClient();
```

Here, the `AddHttpClient` extension belongs to the `Microsoft.Extensions.DependencyInjection` namespace and is defined in the `Microsoft.Extensions.Http` NuGet package. Therefore, our test fixture must create a dependency injection container, must call `AddHttpClient`, and finally, must build the container. The following fixture class does this job (please refer to the *Advanced test preparation/test tear down scenarios* section of Chapter 15, *Testing Your Code with Unit Test Cases and TDD*, if you don't remember fixture classes):

```
public class HttpClientFixture
{
    public HttpClientFixture()
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection
```

```
        .AddHttpClient();
        ServiceProvider = serviceCollection.BuildServiceProvider();
    }

    public ServiceProvider ServiceProvider { get; private set; }
}
```

After the preceding definition, your tests should look as follows:

```
public class UnitTest1:IClassFixture<HttpClientFixture>
{
    private ServiceProvider _serviceProvider;

    public UnitTest1(HttpClientFixture fixture)
    {
        _serviceProvider = fixture.ServiceProvider;
    }

    [Fact]
    public void Test1()
    {
        using (var factory =
            _serviceProvider.GetService<IHttpClientFactory>())
        {
            HttpClient client = factory.CreateClient();
            //use client to interact with application here
        }
    }
}
```

In `Test1`, once you get an HTTP client, you can test the application by issuing an HTTP request and then by analyzing the response returned by the application. More details on how to process the response returned by the server will be given in the *Use case* section.

The next section explains how to test an application that runs in a controlled environment.

Testing a controlled application

In this case, we create an ASP.NET Core server within the test application and test it with an `HttpClient` instance. The `Microsoft.AspNetCore.Mvc.Testing` NuGet package contains all that we need to create both an HTTP client and the server running the application. We also need to reference the whole web framework by referencing the `Microsoft.AspNetCore.App` NuGet package.

Finally, we must also transform the test project into a web project with the following steps:

1. Click on the test project icon in Visual Studio solution explorer, and select the edit project item from the context menu.
2. Replace the root XML node, which should be <Project Sdk="Microsoft.NET.Sdk">, with <Project Sdk="Microsoft.NET.Sdk.web">.

`Microsoft.AspNetCore.Mvc.Testing` contains a fixture class that does the job of launching a local web server and furnishing a client for interacting with it. The predefined fixture class is `WebApplicationFactory<T>`. The generic `T` argument must be instantiated with the `Startup` class of your web project.

Tests look like the following class:

```
public class UnitTest1
    : IClassFixture<WebApplicationFactory<MyProject.Startup>>
{
    private readonly
        WebApplicationFactory<RazorPagesProject.Startup> _factory;

    public UnitTest1 (WebApplicationFactory<MyProject.Startup> factory)
    {
        _factory = factory;
    }

    [Theory]
    [InlineData("/")]
    [InlineData("/Index")]
    [InlineData("/About")]
    ....
    public async Task MustReturnOK(string url)
    {
        var client = _factory.CreateClient();
        // here both client and server are ready
        var response = await client.GetAsync(url);
        //get the response
        response.EnsureSuccessStatusCode();
        // verify we got a success return code.
    }
    ...
---
```

If you want to analyze the HTML of the returned pages, you must also reference the AngleSharp NuGet package. We will see how to use it in the example of the next section. The simplest way to cope with databases in this type of tests is to replace them with in-memory databases that are faster and automatically cleared whenever the local server is shut down and restarted.

This can be done by creating a new deployment environment, say `AutomaticStaging`, and an associate configuration file that is specific for the tests. After having created this new deployment environment, go to the `ConfigureServices` method of your application's `Startup` class and locate the place where you add your `DbContext` configuration. Once located that place, add an `if` there, that, in case the application is running in the `AutomaticStaging` environment, replaces your `DbContext` configuration with something like this:

```
services.AddDbContext<MyDbContext>(options =>
    options.UseInMemoryDatabase(databaseName: "MyDatabase"));
```

As an alternative, you can also add all needed instructions to clear a standard database in the constructor of a custom fixture that inherits from `WebApplicationFactory<T>`.

Use case – automating functional tests

In this section, we will add a simple acceptance test to the ASP.NET Core test project of Chapter 15, *Testing Your Code with Unit Test Cases and TDD*. Our test approach is based on the `Microsoft.AspNetCore.Mvc.Testing` and `AngleSharp` NuGet packages. Please make a new copy of the whole solution.

As a first step, we must turn the test project into a web project by replacing the `sdk` attribute of the root node of its project file in `Sdk="Microsoft.NET.Sdk.web"`.

The test project already references the ASP.NET Core project under `test` and all the required xUnit NuGet packages, so we need to add just the `Microsoft.AspNetCore.Mvc.Testing` and `AngleSharp` NuGet packages.

Now, let's add a new class file called `UIExampleTest.cs`. We need `using` statements to reference all the necessary namespaces. More specifically, we need the following:

- `using PackagesManagement;`: This is needed for referencing your application classes.
- `using Microsoft.AspNetCore.Mvc.Testing;`: This is needed for referencing the client and server classes.

- `using AngleSharp;` and `using AngleSharp.Html.Parser;`: These are needed for referencing AngleSharp classes.
- `System.IO`: This is needed in order to extract HTML from HTTP responses.
- `using Xunit`: This is needed for referencing all xUnit classes.

Summing up, the whole `using` block is as follows:

```
using PackagesManagement;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Xunit;
using Microsoft.AspNetCore.Mvc.Testing;
using AngleSharp;
using AngleSharp.Html.Parser;
using System.IO;
```

We will use the standard fixture class we introduced in the previous *Testing a controlled application* section, that is, the following:

```
public class UIExampleTestcs:
    IClassFixture<WebApplicationFactory<Startup>>
{
    private readonly
        WebApplicationFactory<Startup> _factory;
    public UIExampleTestcs(WebApplicationFactory<Startup> factory)
    {
        _factory = factory;
    }
}
```

Now, we are ready to write a test for the home page! This test verifies that the home URL returns a successful HTTP result and that the home page contains a link to the package management page, which is the `/ManagePackages` relative link.

It is fundamental to understand that automatic tests must not depend on the details of the HTML, but that they must verify just logical facts, in order to avoid frequent changes after each small modification of the application HTML. That's why we just verify that the needed links exist without putting constraints on where they are.

Let's call `TestMenu` our home page test:

```
[Fact]
public async Task TestMenu()
{
    var client = _factory.CreateClient();
    ...
    ...
}
```

The first step of each test is the creation of a client. Then, if the test needs the analysis of some HTML, we must prepare the so-called AngleSharp browsing context:

```
//Create an angleSharp default configuration
var config = Configuration.Default;

//Create a new context for evaluating webpages
//with the given config
var context = BrowsingContext.New(config);
```

The configuration object specifies options such as cookie handling and other browser-related properties. At this point, we are ready to require the home page:

```
var response = await client.GetAsync("/");
```

As a first step, we verify that the response we received contains a success status code, as follows:

```
response.EnsureSuccessStatusCode();
```

The preceding method call throws an exception in case of unsuccessful status code, hence causing the test to fail. HTML analysis needs to be extracted from the response. The following code shows a simple way to do it:

```
var stream = await response.Content.ReadAsStreamAsync();
string source;
using (StreamReader responseReader = new StreamReader(stream))
{
    source = await responseReader.ReadToEndAsync();
}
```

`ReadAsStreamAsync` returns `Stream`, which we can use to build `StreamReader` (a stream specialized for reading text), which can read the whole response body.

Now, we must pass the extracted HTML to our previous AngleSharp browsing context object, so it can build a DOM tree. The following code shows how to do it:

```
var document = await context.OpenAsync(req => req.Content(source));
```

The `OpenAsync` method executes a DOM-building activity with the settings contained in `context`. The input for building the DOM document is specified by the lambda function passed as an argument to `OpenAsync`. In our case, `req.Content(...)` builds a DOM tree from the HTML string passed to the `Content` method, which is the HTML contained in the response received by the client.

Once a `document` object is obtained, we can use it as we would use it in JavaScript. In particular, we can use `QuerySelector` to find an anchor with the required link:

```
var node = document.querySelector("a[href=\"/ManagePackages\"]");
```

It remains to verify just that `node` is not null:

```
Assert.NotNull(node);
```

We have done it! If you want to analyze pages that require a user to be logged in or other more complex scenarios, you need to enable cookies and automatic URL redirects in the HTTP client. This way, the client will behave like a usual browser that stores and sends cookies and that moves to another URL whenever it receives a Redirect HTTP response. This can be done by passing an options object to the `CreateClient` method, as follows:

```
var client = _factory.CreateClient(
    new WebApplicationFactoryClientOptions
    {
        AllowAutoRedirect=true,
        HandleCookies=true
    });

```

With the preceding setup, your tests can do everything a usual browser can do. For instance, you can design tests where the HTTP client logs in and accesses pages that require authentication since `HandleCookies=true` lets the authentication cookie be stored by the client and be sent in all subsequent requests.

Summary

This chapter explains the importance of acceptance/functional tests, and how to define detailed manual tests to be run on the output of each sprint. At this point, you should be able to define automatic and/or manual tests to verify that, at the end of each sprint, your application complies with its specifications.

Then, this chapter analyzed when it is worth automating some or all acceptance/functional tests and describes how to automate them in ASP.NET Core applications.

A final example showed how to write, in practice, ASP.NET Core acceptance/functional tests with the help of AngleSharp to inspect the responses returned by the application.

Questions

1. Is it always worth automating user interface acceptance tests in the case of quick CI/CD cycles?
2. What is the disadvantage of the subcutaneous test for ASP.NET Core applications?
3. What is the suggested technique for writing ASP.NET Core acceptance tests?
4. What is the suggested way of inspecting the HTML returned by the server?

Further reading

More details on the `Microsoft.AspNetCore.Mvc.Testing` NuGet package and AngleSharp can be found in their respective official documentation at <https://docs.microsoft.com/en-US/aspnet/core/test/integration-tests?view=aspnetcore-3.0> and <https://angleSharp.github.io/>.

Readers interested in JavaScript tests can refer to the Jasmine documentation: <https://jasmine.github.io/>.

Assessments

Chapter 1

1. A software architect needs to be aware of any technology that can help them solve problems faster and ensure they have better quality.
2. Azure provides, and keeps evolving, lots of components that a software architect can implement in solutions.
3. The process model can help you understand the team you have, the kind of solution you will provide, and the budget that's available.
4. A software architect pays attention to any user or system requirement that can have an effect on performance, security, usability, and so on.
5. All of them, but the non-functional requirements need to be given more attention.
6. Design Thinking is a tool that helps software architects define exactly what users need.
7. User Stories are good when we want to define functional requirements.
8. Caching, asynchronous programming, and correct object allocation.
9. To check that the implementation is correct, a software architect compares it with models and prototypes that have already been designed and validated.

Chapter 2

1. Vertically and horizontally.
2. Yes, you can deploy automatically to an already defined web app or create a new one directly using Visual Studio.
3. To take advantage of available hardware resources by minimizing the time they remain idle.
4. Code behavior is deterministic, so it is easy to debug. No deadlocks are possible and the execution flow mimics the flow of sequential code, which means it's easier to design and understand.
5. Because the right order minimizes the number of gestures that are needed to fill in a form.

6. Because it allows for the manipulation of path files in a way that is independent of the operating system.
7. It can be used with several .NET Core versions, as well as with several versions of the classic .NET framework.
8. Console, .NET Core, and .NET standard class library; ASP.NET Core, Test, and Microservices.

Chapter 3

1. No, it is available for several platforms.
2. Yes, every step involved in the development/deployment process can be automated, including deployment to production.
3. Automatic, manual, and load test plans.
4. Yes, they can – through Azure DevOps feeds.
5. To manage requirements and to organize the whole development process.
6. Epic work items represent high-level system subparts that are made up of several features.
7. A children-father relationship.

Chapter 4

1. IaaS is a good option when you are migrating from an on-premise solution or if you have an infrastructure team.
2. PaaS is the best option for fast and safe software delivery in systems where the team is focused on software development.
3. If the solution you intend to deliver is provided by a well-known player, such as a SaaS, you should consider using it.
4. Serverless is definitely an option when you are building a new system where you don't have people who specialize in infrastructure and you don't want to worry about scalability.
5. Azure SQL Server Database can be up in minutes and you will have all the power of a Microsoft SQL Server afterward.
6. Azure provides a set of services called Azure Cognitive Services. These services provide solutions for vision, speech, language, search, and knowledge.

7. In a hybrid scenario, you have the flexibility to decide on the best solution for each part of your system while respecting the solution's path and driving it into the future.
8. To allow update/write parallelism.
9. The third argument that's passed to the `Create` method, which creates proxy instances, allows us to specify permitted targets for communication. In general, the third argument of the `ServiceReplicaListener` constructor specifies whether the listener will be created on secondary replicas or not.

Chapter 5

1. The modularity of code and deployment modularity.
2. No, other important advantages include handling the development team and the whole CI/CD cycle well, and the possibility of mixing heterogeneous technologies easily and effectively.
3. A library that helps us implement resilient communication.
4. It is in the `HostBuilder` method that you can declare dependency injection and hosted services.
5. Once you've installed Docker on your development machine, you can develop, debug, and deploy Dockerized .NET Core applications. You can also add Docker images to Service Fabric applications that are being handled with Visual Studio.
6. The one based on Kubernetes `.yaml` files.
7. The one that's exposed to traffic from outside the cluster and is accessible through the cluster's URI.

Chapter 6

1. With the help of database-dependent providers.
2. Either by calling them `Id` or by decorating them with the `Key` attribute.
3. With the `MaxLength` and `MinLength` attributes.
4. With something similar to: `builder.Entity<Package>().HasIndex(m => m.Name);`.

5. With something similar to `builder.Entity<Destination>():`
`.HasMany(m => m.Packages)`
`.WithOne(m => m.MyDestination)`
`.HasForeignKey(m => m.DestinationId)`
`.OnDelete(DeleteBehavior.Cascade);`
6. Add-Migration and Update-Database.
7. No, but you can forcefully include them with the `Include` LINQ clause.
8. Yes, it is, thanks to the `Select` LINQ clause.
9. By calling `context.Database.Migrate()`.

Chapter 7

1. No, it is an in-memory dictionary that can be used as a cache or for other in-memory storage needs.
2. Yes, they are. Most of this chapter's sections are dedicated to explaining why.
3. Write operations.
4. The main weaknesses of NoSQL databases are their consistency and transactions, while their main advantage is performance, especially when it comes to handling distributed writes.
5. Eventual, Consistency Prefix, Session, Bounded Staleness, Strong.
6. No, they are not efficient in a distributed environment. GUID-based strings perform better, since their uniqueness is automatic and doesn't require synchronization operations.
7. `OwnsMany` and `OwnsOne`.
8. Yes, they can. Once you use `SelectMany`, indices can be used to search for nested objects.

Chapter 8

1. Azure Functions is an Azure PaaS component that allows you to implement FaaS solutions.
2. You can program Azure Functions in different languages, such as C#, F#, and Node. You can also create functions using Azure Portal and Visual Studio VS Code.

3. There are two plan options in Azure Functions. The first plan is the Consumption Plan, where you are charged according to the amount you use. The second plan is the App Service Plan, where you share your App Service resources with the function's needs.
4. The process of deploying Functions in Visual Studio is the same as in web app deployment.
5. There are lots of ways we can trigger Azure Functions, such as using Blob Storage, Cosmos DB, Event Grid, Event Hubs, HTTP, Microsoft Graph Events, Queue storage, Service Bus, Timer, and Webhooks.
6. Azure Functions v1 needs the .NET Framework Engine, whereas v2 needs .NET Core.
7. The execution of every Azure Function can be monitored by Application Insights. Here, you can check the time it took to process, resource usage, errors, and exceptions that happened in each function call.

Chapter 9

1. Design patterns are good solutions to common problems in software development.
2. While design patterns give you code implementation for typical problems we face in development, design principles help you select the best options when it comes to implementing the software architecture.
3. The Builder Pattern will help you generate sophisticated objects without the need to define them in the class you are going to use them in.
4. The Factory Pattern is really useful in situations where you have multiple kinds of object from the same abstraction and you don't know which of them needs to be created by the time you start coding.
5. The Singleton Pattern is useful when you need a class that has only one instance during the software's execution.
6. The Proxy Pattern is used when you need to provide an object that controls access to another object.
7. The Command Pattern is used when you need to execute a *command* that will affect the behavior of an object.
8. The Publisher/Subscriber Pattern is useful when you need to provide information about an object to a group of other objects.
9. The DI Pattern is useful if you want to implement the Dependency Inversion principle.

Chapter 10

1. Changes in the language used by experts and changes in the meaning of words.
2. Domain mapping.
3. No; the whole communication passes through the entity, that is, the aggregate root.
4. Because aggregates represent part-subpart hierarchies.
5. Just one, since repositories are aggregate-centric.
6. The application layer manipulates repository interfaces. Repository implementations are registered in the dependency injection engine.
7. To coordinate in single transactions operations on several aggregates.
8. The specifications for updates and queries are usually quite different, especially in simple CRUD systems. The reason for its strongest form is mainly the optimization of query response times.
9. Dependency injection.
10. No; a serious impact analysis must be performed so that we can adopt it.

Chapter 11

1. No, since you will have lots of duplicate code in this approach, which will cause difficulties when it comes to maintenance.
2. The best approach for code reuse is creating libraries.
3. Yes. You can find components that have already been created in the libraries you've created before and then increase these libraries by creating new components that can be reused in the future.
4. The .NET Standard is a specification that allows compatibility between different frameworks of .NET, from .NET Framework to Unity. .NET Core is one .NET implementation and is open source.
5. By creating a .NET Standard library, you will be able to use it in different .NET implementations, such as .NET Core, the .NET Framework, and Xamarin.
6. You can enable code reuse using object-oriented principles (inheritance, encapsulation, abstraction, and polymorphism).
7. Generics is a sophisticated implementation that simplifies how objects with the same characteristics are treated by defining a placeholder that will be replaced with the specific type at compile time.

Chapter 12

1. No, since this would violate the principle that a service reaction to a request must depend on the request itself and not on other messages/requests that had previously been exchanged with the client.
2. No, since this would violate the interoperability constraint.
3. Yes, it can. The primary action of a POST must be creation, but a delete can be performed as a side-effect.
4. Three, that is, Base64 encoding of the header and body plus the signature.
5. From the request body.
6. With the ApiController attribute.
7. The ProducesResponseType attribute.
8. With the Route and Http<verb> attributes.
9. Something like services.AddHttpClient<MyProxy>().

Chapter 13

1. Developer error pages and developer database error pages, production error pages, hosts, HTTPS redirection, routing, authentication and authorization, and endpoint invokers.
2. No.
3. False. Several tag helpers can be invoked on the same tag.
4. ModelState.IsValid.
5. @RenderBody().
6. We can use @RenderSection("Scripts", required: false).
7. We can use return View("viewname", ViewModel).
8. Three.
9. No; there is also the ViewState dictionary.

Chapter 14

1. Maintainability gives you the opportunity to deliver the software you designed quickly. It also allows you to fix bugs easily.
2. Cyclomatic complexity is a metric that detects the number of nodes a method has. The higher the number, the worse the effect.
3. A version control system will guarantee the integrity of your source code, giving you the opportunity to analyze the history of each modification that you've made.
4. Try-catch is a way to control exceptions that have been invoked by the code you are writing. Try-finally is a way to guarantee that, even with an exception inside the try-block, the finally block will carry out its process. You can use try-catch-finally when you want to solve both situations in the same piece of code.
5. A garbage collector is a .NET Core/.NET Framework system that monitors your application and detects objects that you aren't using anymore. It disposes of these objects to release memory.
6. The `IDisposable` interface is important in classes that instantiate objects that need to be disposed of by the programmer since the garbage collector cannot dispose of them.
7. .NET Core encapsulates some design patterns in some of its libraries in a way that can guarantee safer code, such as with dependency injection and Builder.

Chapter 15

1. Because most of the tests must be repeated after any software changes occur.
2. Because the probability of exactly the same error occurring in a unit test and in its associated application code is very low.
3. `[Theory]` is used when the test method defines several tests, while `[Fact]` is used when the test method defines just one test.
4. `Assert`.
5. `Setup`, `Returns`, and `ReturnsAsync`.
6. Yes; with `ReturnAsync`.

Chapter 16

1. Well-written code is code that any person skilled in that programming language can handle, modify, and evolve.
2. Roslyn is the .NET Compiler that's used for code analysis inside Visual Studio.
3. Code analysis is a practice that considers the way the code is written to detect bad practices before compilation.
4. Code analysis can find problems that happen even with apparently good software, such as memory leaks and bad programming practices.
5. Roslyn can be programmed for code analysis.
6. Visual Studio Extensions are tools that have been programmed to run inside Visual Studio. These tools can help you out in some cases where Visual Studio IDE doesn't have the appropriate feature for you to use.
7. Microsoft Code Analysis, SonarLint, and Code Cracker.

Chapter 17

1. To maximize the value that the software provides for the target organization.
2. No; it requires the acquisition of all competencies that are required to maximize the value added by the software.
3. Because when a new user subscribes, its tenant must be created automatically, and because new software updates must be distributed to all the customer's infrastructures.
4. Yes; Terraform is an example.
5. Azure pipelines.
6. Your business depends on the SaaS supplier, so its reliability is fundamental.
7. No; scalability is just as important as fault tolerance and automatic fault recovery.

Chapter 18

1. DevOps is the approach of delivering value to the end user continuously. To do this with success, continuous integration, continuous delivery, and continuous feedback must be undertaken.
2. Continuous integration allows you to check the quality of the software you are delivering every single time you commit a change. You can do this by turning on this feature in Azure DevOps.
3. Continuous delivery allows you to deploy a solution once you are sure that all the quality checks have passed the tests you designed. Azure DevOps helps you with that by providing you with relevant tools.
4. Continuous Feedback is the adoption of tools in the DevOps life cycle that enable fast feedback when it comes to performance, usability, and other aspects of the application you are developing.
5. The build pipeline will let you run tasks for building and testing your application, while the release pipeline will give you the opportunity to define how the application will be deployed in each scenario.
6. Application Insights is a helpful tool for monitoring the health of the system you've deployed, which makes it a fantastic continuous feedback tool.
7. Test & Feedback is a tool that allows stakeholders to analyze the software you are developing and enables a connection with Azure DevOps to open tasks and even bugs.

Chapter 19

1. It is an approach that makes sure that every single commit to the code repository is built and tested.
2. Yes, you can have DevOps separately and then enable Continuous Delivery later. Your team and process need to be ready and attentive for this to happen.
3. All of these risks may cause damage to your production environment. You can have, for example, a feature that isn't ready but has been deployed, you can cause a stop at a bad time for your customers, or you can even suffer a bad collateral effect due to an incorrect fix.
4. A multi-stage environment protects production from bad releases.
5. Automated tests anticipate bugs and bad behaviors in preview scenarios.

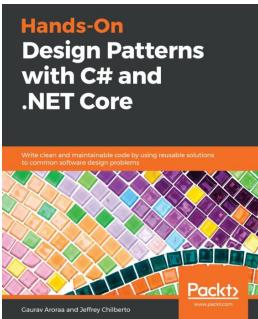
6. Pull requests allow code reviews before commits are made in the master branch.
7. No; pull requests can help you in any development approach where you have GIT as your source control.

Chapter 20

1. No; it depends on the complexity of the user interface and how often it changes.
2. The ASP.NET Core pipeline isn't executed, but inputs are passed directly to controllers.
3. Use of the `Microsoft.AspNetCore.Mvc.Testing` NuGet package.
4. Use of the `AngleSharp` NuGet package.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Hands-On Design Patterns with C# and .NET Core

Gaurav Arora, 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



C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development - Fourth Edition

Mark J. Price

ISBN: 978-1-78847-812-0

- Build cross-platform applications for Windows, macOS, Linux, iOS, and Android
- Explore application development with C# 8.0 and .NET Core 3.0
- Explore ASP.NET Core 3.0 and create professional web applications
- Learn object-oriented programming and C# multitasking
- Query and manipulate data using LINQ
- Use Entity Framework Core and work with relational databases
- Discover Windows app development using the Universal Windows Platform and XAML
- Build mobile applications for iOS and Android using Xamarin.Forms

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!

Index

- .NET Core 3.0
 - features 385, 386, 387
 - .Net Core HTTP clients 341, 342, 343
 - .NET Core, with microservices
 - about 120
 - Azure and Visual Studio support, for microservice orchestration 134, 135, 136
 - communication channels 121, 122
 - generic hosts 125, 126, 127, 128
 - resilient task execution 123, 124
 - Visual Studio support, for Docker 129, 131, 132, 133
 - .NET Core
 - about 11
 - dealing, with Service-Oriented Architecture (SOA) 324, 325, 326
 - design patterns 262, 263
 - interoperability 58, 59
 - project types 62
 - projects 61, 62
 - scalability 35
 - scalable web app, creating 42, 43, 44, 45, 46
 - tips and tricks, for coding 435
 - .NET Generic Host
 - reference link 262
 - .NET Standard 2.1
 - reference link 301
 - .NET Standard
 - library, creating 300
 - using, for code reusability 300
 - versions, reference link 300
- A**
- acceptance tests
 - writing 444
- Access Control 79
 - advanced test preparations 451, 452
 - aggregates 279, 280
 - agile 16
 - Agile Manifesto
 - URL 16
 - agile software development process models
 - about 16, 17
 - Scrum Model 17
 - Analyze and Code Cleanup 469
 - Application Insights monitor
 - creating 510
 - Application Insights
 - about 509
 - attaching 512
 - dashboard 513
 - enabling, to Web App 511
 - sample view 514
 - settings, in App Service 513
 - software, monitoring 509
 - Application Performance Management (APM) 234
 - Artificial Intelligence (AI) 96
 - ASP.NET Core MVC pipeline
 - defining 363, 364, 365, 367, 368
 - ASP.NET Core MVC
 - and design principles 387
 - configuration data, loading 358
 - configuration data, using with options framework 358
 - controllers, defining 368, 371, 372
 - pipeline 354, 356, 358
 - pipeline, defining 363
 - Razor Views 373
 - structure 354
 - view code, reusing 381, 382, 384
 - ViewModels, defining 368, 371, 372
 - web app, implementing 395

ASP.NET Core pipeline
advantages 388
client-side validation 388, 389
server-side validation 388, 389
working 354

ASP.NET Core
.NET Core 3.0, features 385
about 11, 326, 327, 328, 329
globalization 389, 391, 393
REST services, implementing with 330, 331, 332, 333
service authorization 334, 336
support, for OpenAPI 337, 338, 339, 340

asynchronous programming
applying, on ASP.NET Core applications 25
reference link 50

automated tests
about 440
execution and reporting tool 443
facilities 442
mock facilities 443
writing 442, 443

Azure account
creating 12, 13, 14

Azure Cognitive Services 96, 98, 99

Azure Cosmos DB
about 208, 209, 210, 211, 212, 213, 214
used, for implementing Destination/Packages database 218, 219, 220, 221, 222

Azure DevOps team
reference link 431

Azure DevOps
about 11, 65, 67, 69, 70
Epics work items 78
Features work items 78
multistage environment 505, 506, 507, 508
package feeds 73, 74
pipelines 76
Product Backlog work items 79
repository 71, 72, 73
system requisites, managing 77
use cases 80, 82, 84
used, for organizing work 70
User Story work items 79

Azure Functions App

about 226, 227
App Service Plan 227
Consumption Plan 227

Azure Functions templates
Blob Storage: 232
Cosmos DB 232
Event Grid 232
Event Hubs 232
HTTP 232
listing 232
Microsoft Graph Events 232
Queue storage 232
Service Bus 232
Timer 232
WebHooks 232

Azure Functions
implementing, to send emails 235, 236
maintaining 233, 234, 235
pricing, reference link 227
programming, with C# 228, 229, 230, 231

Azure Kubernetes Service (AKS) 146, 147, 148, 149

Azure Pipelines
packages-management application, deploying with 490

Azure Queue Storage
creating 237, 238, 239, 240, 241, 242

Azure Service Fabric
about 139
cluster configuration 141, 142
creating 140, 141
DNS 142
reverse proxy 143
security configuration 143, 144, 145

Azure SQL Server 94, 95, 96

Azure Storage Explorer
reference link 239

Azure
opportunities 89, 91
private Docker registry, defining 137, 138
scalability 35
scalable web app, creating 36, 37, 38

B

Blob Storage 232
Bounded Contexts 269
Builder pattern 249, 250
bulkhead isolation 113

C

C# code
evaluation, before publishing application 477, 478
evaluation, by applying tools 468, 469, 470, 471
C# test projects
defining 448
C#, performance issues
considerations 46
C#
about 11, 28
implementing, with code reusability 302
object-oriented analysis 302, 303
safe code, writing 431
used, for programming Azure Functions 228, 229, 230, 231
caching 25
cascading drop-down menus
example 57, 58
Cloud Design Patterns
URL 248
Code Cracker
obtaining, for Visual Studio 2017 475
Code Metrics tool
about 422
class coupling 428, 429, 430
cyclomatic complexity 423, 425, 426
depth of inheritance 427, 428
lines of code 430
maintainability index 423
reference link 423
code reusability
.NET Standard, using 300
about 296, 298
advantages 296
inserting, into development cycle 299
principles 296
use case 305
used, for implementing C# 302
command handlers 286, 288, 289
Command pattern 257, 258
Command Query Responsibility Segregation (CQRS) pattern 283, 284, 285
communication stack
options 311
configuration data
loading 358, 359, 361, 362, 363
using, with options framework 358, 359, 361, 362, 363
conformist pattern 272
connectors 175
consistency levels
Bounded Staleness 212
Consistency Prefix 212
eventual 212
session 212
strong 212
containers 114
Context Mapping 271
continuous delivery (CD) 506, 507, 508
continuous feedback
defining 508
continuous integration (CI) 504, 505, 524, 525
Continuous Integration (CI), risks and challenges
about 525
continuous production deployment, disabling 526, 527
incomplete features 527, 528, 529, 530
unstable solution, for testing 530, 531, 532, 533, 534
Continuous Integration/Continuous Deployment (CI/CD)
about 70
with microservices 108
controlled application
testing 541, 543
Cosmos DB client 215, 216, 232
Cosmos DB emulator
installation link 222
Cosmos DB Entity Framework Core provider 216, 217
customer 267
customer/supplier development teams pattern 271

D

data layer
 deploying 195, 196

data storage
 destination reviews 217
 destinations information 217
 packages information 217
 reservations 218
 use case 217, 218

Data Transport Objects (DTOs) 192

data
 querying, with Entity Framework Core 189, 190, 191
 returning, to presentation layer 192, 193
 updating, with Entity Framework Core 189, 190, 191

database access
 obtaining 26

Database Transaction Units (DTUs) 202

DB entities 179, 181, 182

DDD entities 273, 274, 276, 282, 283

Dependency Injection (DI) pattern 128, 260, 261, 356

design patterns
 about 248
 Builder pattern 249, 250
 Command pattern 257, 258
 Dependency Injection (DI) pattern 260, 261
 Factory pattern 251, 252
 in .NET Core 262, 263
 Proxy pattern 255, 256, 257
 Publisher/Subscriber pattern 259
 purpose 248
 Singleton pattern 252, 254, 255

design thinking 22

design thinking, steps
 define 22
 empathize 22
 ideate 22
 prototype 22
 test 22

Destination/Packages database
 implementing, with Cosmos DB 218, 219, 220, 221, 222

DevOps Azure

unit tests, automating in 455, 456, 457, 459, 460, 462, 463

DevOps principles 504

DevOps tools
 checking, for continuous feedback 508

DevOps
 about 503
 reference link 503

direct SQL commands
 about 188
 issuing 193, 194

disk memory 206

Docker
 about 114, 115
 reference link 115

domain events 270, 286, 288, 289

domain-driven design (DDD)
 about 265, 269, 270, 271, 272
 value objects 273, 274, 276

dynamic link libraries (DLL) 109

E

easy to learn user interfaces
 designing, tips 52
 requirements 53

entities 179

Entity Framework Core
 about 282, 283
 advanced feature 196, 197
 configuring 178, 179
 data, querying with 189, 190, 191
 data, updating with 189, 190, 191
 DB entities, defining 179, 181, 182
 direct SQL commands 188
 mapped collections, defining 182
 mapping configuration, completing 183, 184
 migrating 184, 185, 186, 187, 188
 options 185
 stored procedures 188

envelopes 313

Epics work items 78

Event Grid 232

Event Hubs 232

event sourcing 289, 290

exceptions 48, 49

extension tools
applying, for code analysis 472
Extreme Programming 446

F

Factory pattern 251, 252
fast selection logic
designing 53, 54, 55, 57
Features work items 78
final code
checking, after analysis 475, 476
fixture class instance 452
Function as a Service (FaaS) 100
functional tests
automating, in C# with unit testing tools 539, 540
automating, use case 543, 544, 545, 546
purpose 537, 538

G

Gang of Four (GoF) 248
Garbage Collector (GC) 25
General Data Protection Regulation (GDPR) 31
generics 304, 305
global filters 196, 197

H

high-quality software design
aspects, enabling for 18
horizontal scaling 41, 42
hosted services 125
HTTP 232
human resources management 484
hybrid applications
benefits 101
use case 102, 103

I

IdempotencyTools project 154
IDisposable interface 434
Incremental Model
analyzing 15, 16
URL 16
Infrastructure as a Service (IaaS)

about 89, 91
security responsibility 91, 92
installation scripts, in Linux and macOS
reference link 59
integration tests 441
Interaction library 157
interfaces
mocking, with Moq 453, 454
Internet Information Services (IIS) 328
interoperability
with .NET Core 58, 59
Inversion of Control (IoC) 261

L

Language Integrated Queries (LINQ) 176
lazy loading
reference link 257
Levenshtein algorithm 54, 55
Linux
service, creating 60
logging microservices
application, testing 171

M

mapped collections
defining 182
mapping configuration
completing 183, 184
memory leaks 432
microservice architectures
features 119, 120
microservices, logging
about 150, 151, 152, 153, 154
communication, with service 169, 170, 171
host, defining 168, 169
Interaction library 157, 158
message idempotency, ensuring 154, 155, 157
receiving side of communication, implementing 159, 160, 162
service logic, implementing 162, 163, 165, 167, 168
microservices
about 107
CI/CD improvement 108
design principles 111, 112, 113, 114

layered architectures 116, 117
managing, tools 137
role 116

Microsoft Azure

about 11, 12
URL 12

Microsoft Code Analysis 2019
using 472, 473

Microsoft Graph Events 232

mocking 453

Model View Controller (MVC) 352, 394
modules 109

Moq
used, for mocking interfaces 453, 454

multistage environment

with Azure DevOps 505, 506, 507, 508

multithreading environments 49

N

NoSQL databases 203, 204, 206, 207, 208

NoSQL storage 206, 207, 208

NSwag.MSBuild

reference link 340

NSwagStudio Windows program

reference link 340

O

object allocation

dealing with 25, 26

object-oriented analysis 302, 303

Object-Relational Mapping (ORM) 174, 175, 176, 177

OpenAPI standard 321

OpenAPI

about 321

ASP.NET Core support 337, 338, 339

ASP.NET Core, support for 340

operating system (OS) 19, 115

Orchestrators 116

organization

adapting, to service scenario 484

P

packages-management application, deploying with
Azure Pipelines

Azure database, creating 490, 492
Azure Pipelines, configuring 493, 494, 495, 496
Azure web app, creating 490, 492
manual approval, adding for release 496, 497, 498

release, creating 498, 500

scenarios 490

Visual Studio solution, configuring 492

partner pattern 271

Pencil Project

reference link 20

performance tests

writing 445

Platform as a Service (PaaS)

about 25

advantages 92, 93

Azure Cognitive Services 96, 98, 99

Azure SQL Server 94, 95, 96

web apps 93

pod 148

presentation layer

data, returning to 192, 193

of web applications 353

primary node 141

private Docker registry

defining, in Azure 137, 138

Product Backlog work items

tasks 79

test cases 79

User Story work items 79

Product Owner 17

providers

about 175

reference link 175

Proxy pattern 255, 256, 257

Publisher/Subscriber pattern 259

Q

Queue storage 232

R

Razor Views

about 373, 374

flow of control 374, 375, 376

properties 376

tag helpers 377, 378, 379, 381
Redis 204, 205
relational databases 201, 202
release stages, with Azure DevOps
 development/tests 506
 production 506
 quality assurance 506
reliable services 139
Remote Desktop Protocol (RDP) 91
repositories 200
repository pattern 280, 281
Representational State Transfer 318
requirements analysis
 about 20
 prototyping 20
 use cases 20
requirements engineering
 reference link 19
requirements gathering process
 about 18, 19
 elicitation of user needs, practicing 19
 impact, on system results 24
 performance principle, applying 23
 requirements analysis 20
 requirements, analyzing 20
 robustness principle, applying 23
 scalability principle, applying 23
 scenarios 24, 27, 28
 security principle, applying 23
 specification, reviewing 21
 specifications, writing 21
REST services
 implementing, with ASP.NET Core 330, 331, 332, 333
REST web services
 about 315, 316, 317, 318, 319, 320
 authentication 321, 323, 324
 authorization 321, 323, 324
 OpenAPI standard 321

S

SaaS solution
 adopting 486, 487
safe code
 writing, in C# 431

scalability
 with .NET Core 35
 with Azure 35
scalable web app
 creating, in Azure 36, 37, 38
 creating, with .NET Core 42, 43, 44, 45, 46
Scrum Master 17
Scrum Model 17
security responsibility
 in IaaS 91, 92
serverless solution 100
Service Bus 232
service design thinking 484
service scenario
 organization, adapting to 484
 software, developing 485
 solution, preparing for 487, 488, 489
 technical implications 485, 486
Service-Oriented Architecture (SOA)
 .NET Core, dealing with 324, 325, 326
 about 110, 308
service
 creating, in Linux 60
shard collection 207
Simple Object Access Protocol (SOAP) 313
Single Page Applications (SPA) 353
Single Responsibility, Open/Closed, Liskov
 Substitution, Interface Segregation, Dependency Inversion (SOLID) 441
Singleton pattern 252, 254, 255
SOA approach
 options 312
 principles 309, 310, 313
SOA solutions
 features 312
SOAP web services 313, 314, 315
software architect
 do's 51
 don'ts 51
software architecture
 about 10
Software as a Service (SaaS) 65, 89, 100, 484
software deployment models 89
software development process models 14
software domains 266, 267, 268, 269

software engineering 10
software
 developing, in service scenario 485
 monitoring, with Application Insights 509
SOLID design principles
 about 260
 Dependency Inversion 260
 Interface Segregation 260
 Liskov Substitution 260
 Open-Closed 260
 Single Responsibility 260
 used, for mapping domains 277, 278
Sonar Cloud 477
SonarLint
 applying, for Visual Studio 2019 474
Sprint 17
StackExchange.Redis
 references 204
staging application
 testing 540, 541
stored procedures 188
string concatenation 47
structured data 206, 207, 208
subcutaneous tests 444

T

tear-down scenarios 451, 452
technical implications
 of service scenario 485, 486
tenants 207
Test and Feedback tool
 download link 515
 used, for enabling feedback 515, 517, 518
test-driven development (TDD) 445, 447
tests
 integration tests 441
 unit tests 441
Timer 232
tools
 applying, for C# code evaluation 468, 469, 470, 471
 used, for managing microservices 137
traditional software development process models
 Incremental Model 15, 16
 reviewing 14

Waterfall Model, principles 14, 15
transactions
 handling 195
try-catch statement 431, 432
try-finally statement 432, 433

U

Ubiquitous Language 269
Unified Modeling Language (UML) 20
Unit of Work patterns 280, 281
unit testing tools
 used, for automating functional tests in C# 539, 540
unit tests
 about 441
 automating, in DevOps Azure 455, 456, 457, 459, 460, 462, 463
usability 52
user interface (UI) 52
user needs detection
 scenarios 28
User Story work items
 about 79
 tasks 79
 test cases 79
using statement 433

V

version control system
 dealing with 431
 using 430
vertical scaling 39
view components 384
ViewModel 371
Visual Studio 2017
 Code Cracker, obtaining for 475
Visual Studio 2019
 SonarLint, applying for 474
Visual Studio Team Services (VSTS) 100

W

Waterfall Model
 principles 14, 15
 URL 15
web app implementation, in ASP.NET Core MVC

about 395
application layer, defining 407, 409, 410, 412
architecture, defining 396, 397, 399
controllers, adding 412, 413, 415, 416, 418
data layer, defining 402, 403, 405, 406, 407
domain layer, defining 399, 401
specifications, defining 395
views 418
views, adding 412, 413, 414, 415, 416, 418
web applications
 presentation layer 353
web apps 93
WebHooks 232
well-written code
 identifying 466, 467
World Wild Travel Club (WWTravelClub)

about 28, 29, 30
user stories 30, 31
WWTravelClub project
 approach 520, 534
WWTravelClub use case
 domains 290, 292, 293
WWTravelClub
 best practices, for writing code 436
 cloud platform, selecting 103
 host, defining 347
 packages, exposing 344, 345, 346, 348

X

xUnit test framework
 using 449, 450, 451