# ASP.NET Core 3 and Angular 9

Third Edition

Full stack web development with .NET Core 3.1 and Angular 9

Packt>

www.packt.com

Valerio De Sanctis

# ASP.NET Core 3 and Angular 9
## *Third Edition*

Full stack web development with .NET Core 3.1 and Angular 9

**Valerio De Sanctis**

**BIRMINGHAM - MUMBAI**

# ASP.NET Core 3 and Angular 9
## *Third Edition*



**Commissioning Editor:** Ashwin Nair
**Acquisition Editor:** Larissa Pinto
**Content Development Editor:** Aamir Ahmed
**Senior Editor:** Hayden Edwards
**Technical Editor:** Sachin Sunilkumar
**Copy Editor:** Safis Editing
**Project Coordinator:** Kinjal Bari
**Proofreader:** Safis Editing
**Indexer:** Priyanka Dhadke
**Production Designer:** Arvindkumar Gupta

First published: October 2016
Second edition: November 2017
Third edition: February 2020

Production reference: 1130220

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

ISBN 978-1-78961-216-5

www.packt.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 author

**Valerio De Sanctis** is a skilled IT professional with more than 15 years of experience in lead programming, web-based development, and project management using ASP.NET, PHP, and Java. He has held senior positions at a range of financial and insurance companies, most recently serving as Chief Technology Officer, Chief Security Officer, and Chief Operating Officer at a leading after-sales and IT service provider for multiple top-tier life and non-life insurance groups.

During the course of his career, Valerio has helped many private organizations to implement and maintain .NET based solutions, working side by side with many IT industry experts and leading several frontend, backend, and UX development teams. He designed the architecture and actively oversaw the development of a wide number of corporate-level web application projects for high-profile clients, customers, and partners, including the London Stock Exchange Group, Zurich Insurance Group, Allianz, Generali, Harmonie Mutuelle, Honda Motor, FCA Group, Luxottica, ANSA, Saipem, ENI, Enel, Terna, Banzai Media, Virgilio.it, Repubblica.it, and Corriere.it.

He is an active member of the Stack Exchange network, providing advice and tips for .NET, JavaScript, HTML5, and web-related topics on the StackOverflow, ServerFault, and SuperUser communities. Most of his projects and code samples are available under open source licenses on GitHub, BitBucket, NPM, CocoaPods, JQuery Plugin Registry, and WordPress Plugin Repository. He's also a **Microsoft Most Valuable Professional (MVP) for Developer Technologies**, an annual award that recognizes exceptional technology community leaders worldwide who actively share their high-quality, real-world expertise with users and Microsoft.

Since 2014, he has operated an **IT-oriented, web-focused blog at www.ryadel.com**, featuring news, reviews, code samples, and guides designed to help developers and tech enthusiasts from all around the world. He has written various books on web development, many of which have become best-sellers on Amazon, with tens of thousands of copies sold worldwide.

You can reach him on LinkedIn at `https://www.linkedin.com/in/darkseal/`

# About the reviewers

**Anand Narayanaswamy** works as a freelance writer and reviewer based in Thiruvananthapuram. He has had articles published in leading print magazines and on online tech portals. He was the recipient of the Microsoft MVP award from 2002 to 2011. He is currently a Windows Insider MVP and also part of the prestigious ASPInsiders group.

Anand has also worked as a technical editor and reviewer for several publishers and authored *Community Server Quickly* for Packt Publishing. He has contributed content for Digit Magazine and Manorama Year Book.

Anand works as an Unacademy educator and also runs his own blogs – Netans and Learnxpress. He is active on social media and can be reached at @visualanand on Twitter and @netanstech on Instagram.

*I would like to thank God for giving me the power to work daily. I would also like to thank Amrita Venugopal, Kinjal Bari, and Manthan Patel for their excellent support and patience. The manner in which Packt editors handle reviewers from the start of a project is highly impressive. I am also grateful to my father, mother, and brother for their constant support and encouragement.*

**Santosh Yadav** is from Pune, India, and holds a bachelor's degree in computing. He has more than 11 years' experience as a developer, and has worked with multiple technologies, including .NET, Node.js, and Angular. He is a Google Developer Expert in Angular and web technologies. He is the author of the ng deploy library for Netlify and a writer for Angular In Depth. He is also a speaker and organizer for the Pune Tech Meetup, and a contributor to projects including Angular and NgRx.

# 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

ASP.NET Core is a free and open source modular web framework developed by Microsoft that runs on top of the full .NET Framework (Windows) or .NET Core (cross-platform). It has been made specifically for building efficient HTTP services that can be reached and consumed by a massive range of clients, including web browsers, mobile devices, smart TVs, web-based home automation tools, and more.

Angular is the successor of AngularJS, a world-renowned development framework born with the aim of providing the coder with the toolbox that is needed to build reactive and cross-platform web-based apps that are optimized for desktop and mobile. It features a structure-rich template approach based upon a natural, easy-to-write, and readable syntax.

Technically, these two frameworks have little or nothing in common: ASP.NET Core is mostly focused on the **server-side** part of the web development stack, while Angular is dedicated to covering all the **client-side** aspects of web applications, such as the **User Interface** (**UI**) and **User Experience** (**UX**). However, both of them came into being because of a common vision shared by their respective creators: *the HTTP protocol is not limited to serving web pages; it can also be used as a viable platform upon which to build web-based APIs to effectively send and receive data.* This is the notion that slowly made its way through the first 20 years of the World Wide Web's life and is now an undeniable, widely acknowledged statement and also a fundamental pillar of almost every modern web development approach.

As for the reasons behind this perspective switch, there are plenty of good ones, the most important of them being related to the intrinsic characteristics of the HTTP protocol: it's rather simple to use, and flexible enough to match most of the development needs of the ever-changing environment that the World Wide Web happens to be in. This is not to mention how universal it has become nowadays: almost any platform that we can think of has an HTTP library, so HTTP services can reach a broad range of clients, including desktop and mobile browsers, IoT devices, desktop applications, video games, and so on.

The main purpose of this book is to bring together the latest versions of ASP.NET Core and Angular within a single development stack to demonstrate how they can be used to create high-performance web applications and services that can be used seamlessly by any clients.

# Who this book is for

This book is for experienced ASP.NET developers who already know about ASP.NET Core and Angular and are looking to learn more about them and understand how to use them together to create a production-ready **Single-Page Application** (**SPA**) or **Progressive Web Application** (**PWA**). However, the fully documented code samples (also available on GitHub) and the step-by-step implementation tutorials make this book easy to understand even for beginners and developers who are just getting started.

# What this book covers

`Chapter 1`, *Getting Ready*, introduces some of the basic concepts of the frameworks that we are going to use throughout the book, as well as the various kinds of web applications that can be created (SPAs, PWAs, native web apps, and more).

`Chapter 2`, *Looking Around,* is a detailed overview of the various back-end and front-end elements provided by the .NET Core and Angular template shipped with Visual Studio 2019, backed up with some high-level explanations about how they can work together in a typical HTTP request-response cycle.

`Chapter 3`, *Front-end and Back-end Interactions,* provides a comprehensive tutorial for building a sample ASP.NET Core and Angular app that provides diagnostic info to the end user by querying health check middleware using a Bootstrap-based Angular client.

`Chapter 4`, *Data Model with Entity Framework Core,* constitutes a journey through Entity Framework Core and its capabilities as an **Object-Relational Mapping** (**ORM**) framework, from SQL database deployment (cloud-based and/or local instance) to data model design, including various techniques to read and write data from back-end controllers.

`Chapter 5`, *Fetching and Displaying Data,* covers how to expose Entity Framework Core data using the ASP.NET Core back-end web API, consume that data with Angular, and then show it to end users using the front-end UI.

`Chapter 6`, *Forms and Data Validation,* details how to implement the HTTP PUT and POST methods in back-end web APIs in order to perform insert and update operations with Angular, along with server-side and client-side data validation.

`Chapter 7`, *Code Tweaks and Data Services,* explores some useful refactoring and improvements to strengthen your app's source code and includes an in-depth analysis of Angular's data services to understand why and how to use them.

`Chapter 8`, *Back-end and Front-end Debugging* looks at how to properly debug the back-end and front-end stacks of a typical web application using the various debugging tools provided by Visual Studio to their full extent.

`Chapter 9`, *ASP.NET Core and Angular Unit Testing,* comprises a detailed review of the **Test-Driven Development** (**TDD**) and **Behavior-Driven Development** (**BDD**) development practices and goes into how to define, implement, and perform back-end and front-end unit tests using xUnit, Jasmine, and Karma.

`Chapter 10`, *Authentication and Authorization,* gives you a high-level introduction to the concepts of authentication and authorization and presents a narrow lineup of some of the various techniques, methodologies, and approaches to properly implementing proprietary or third-party user identity systems. A practical example of a working ASP.NET Core and Angular authentication mechanism based upon ASP.NET Identity and IdentityServer4 is included.

`Chapter 11`, *Progressive Web Apps,* delves into how to convert an existing SPA into a PWA using service workers, manifest files, and offline caching features.

`Chapter 12`, *Windows and Linux Deployment,* teaches you how to deploy the ASP.NET and Angular apps created in the previous chapters and publish them in a cloud-based environment using either a Windows Server 2019 or Linux CentOS virtual machine.

# To get the most out of this book

These are the software packages (and relevant version numbers) used to write this book and test the source code:

- Visual Studio 2019 Community Edition 16.4.3
- Microsoft .NET Core SDK 3.1.1
- TypeScript 3.7.5
- NuGet Package Manager 5.1.0
- Node.js 13.7.0 (we strongly suggest installing it using the **Node Version Manager**, also known as **NVM**)
- Angular 9.0.0 final

For deployment on **Windows**:

- ASP.NET Core 3.1 Runtime for Linux (YUM package manager)
- .NET Core 3.1 CLR for Linux (YUM package manager)
- Nginx HTTP Server (YUM package manager)

For deployment on **Linux**:

- ASP.NET Core 3.1 Runtime for Linux (YUM package manager)
- .NET Core 3.1 CLR for Linux (YUM package manager)
- Nginx HTTP Server (YUM package manager)


If you're on Windows, I strongly suggest installing Node.js using NVM for Windows-a neat Node.js version manager for the Windows system. You can download it from the following URL:

`https://github.com/coreybutler/nvm-windows/releases`.

We strongly suggest using the same version used within this book – or newer, but at your own risk! Jokes aside, if you prefer to use a different version, that's perfectly fine, as long as you are aware that, in that case, *you may need to make some manual changes and adjustments to the source code*.

# 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/ASP.NET-Core-3-and-Angular-9-Third-Edition`. 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!

# 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/9781789612165_ColorImages.pdf`.

# Conventions used

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

`CodeInText`: Indicates code words in the text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Navigate to the `/ClientApp/src/app/cities` folder."

A block of code is set as follows:

```
<mat-form-field [hidden]="!cities">
    <input matInput (keyup)="loadData($event.target.value)"
        placeholder="Filter by name (or part of it)...">
</mat-form-field>
```

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

```
import { FormGroup, FormControl } from '@angular/forms';

class ModelFormComponent implements OnInit {
  form: FormGroup;

  ngOnInit() {
    this.form = new FormGroup({
        title: new FormControl()
    });
  }
}
```

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

```
> dotnet new angular -o HealthCheck
```

**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: "A simple **Add a new City** button will fix both these issues at once."

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
# Getting Ready

In this chapter, we'll build the basics of our ASP.NET and Angular journey by mixing a theoretical coverage of their most relevant features and using a more practical approach. More specifically, in the upcoming part of this chapter, we'll briefly review the recent history of ASP.NET Core and Angular frameworks, while in the latter part, we'll learn how to configure our local development environment so we can assemble, build, and test a sample web application boilerplate.

By the end of this chapter, you'll have gained knowledge of the path taken by ASP.NET Core and Angular to improve web development in the last few years, and learned how to properly set up an ASP.NET and Angular web application.

Here are the main topics that we are going to cover:

- **The ASP.NET Core revolution**: A brief history of ASP.NET Core and Angular's most recent achievements.
- **A full-stack approach**: The importance of being able to learn how to design, assemble, and deliver a complete product.
- **Single-Page Applications** (**SPAs**), **Native Web Applications** (**NWAs**), **and Progressive Web Applications** (**PWAs**): Key features and the most important differences between the various types of web applications, as well as how well ASP.NET Core and Angular could relate to each one of them.
- **A sample SPA project**: What we're going to do throughout this book.
- **Preparing the workspace**: How to set up our workstation to achieve our first goal – implementing a simple Hello World boilerplate that will be further extended within the following chapters.

# Technical requirements

These are the software packages (and relevant version numbers) used to write this book and test the source code:

- Visual Studio 2019 Community Edition 16.4.3
- Microsoft .NET Core SDK 3.1.1
- TypeScript 3.7.5
- NuGet Package Manager 5.1.0
- Node.js 13.7.0 (we strongly suggest installing it using the **Node Version Manager**, also known as **NVM**)
- Angular 9.0.0 final

If you're on Windows, I strongly suggest installing Node.js using NVM for Windows-a neat Node.js version manager for the Windows system. You can download it from the following URL:

`https://github.com/coreybutler/nvm-windows/releases`.

We strongly suggest using the same version used within this book – or newer, but at your own risk! Jokes aside, if you prefer to use a different version, that's perfectly fine, as long as you are aware that, in that case, *you may need to make some manual changes and adjustments to the source code*.

# Two players, one goal

From the perspective of a fully functional web-based application, we can say that the Web API interface provided with the ASP.NET Core framework is a programmatic set of server-side handlers used by the server to expose a number of hooks and/or endpoints to a defined request-response message system. This is typically expressed in structured markup languages (XML), language-independent data formats (JSON), or query languages for APIs (GraphQL). As we've already said, this is achieved by exposing **application programming interfaces** (**APIs**) through HTTP and/or HTTPS protocols via a publicly available web server such as IIS, Node.js, Apache, Nginx, and so on.

Similarly, Angular can be described as a modern, feature-rich, client-side framework that pushes the HTML and ECMAScript's most advanced features, along with the modern browser's capabilities, to their full extent by binding the input and/or output parts of an HTML web page into a flexible, reusable and easily testable model.

Can we combine the *back-end* strengths of ASP.NET Core and the *front-end* capabilities of Angular in order to build a modern, feature-rich, and highly versatile web application?

The answer, in short, is yes. In the following chapters, we'll see how we can do that by analyzing all the fundamental aspects of a well-written, properly designed, web-based product, and how the latest versions of ASP.NET Core and/or Angular can be used to handle each one of them. However, before doing all that, it can be very useful to backtrack a bit and spend some valuable time recollecting what's happened in the last 3 years in the development history of the two frameworks we're going to use. It will be very useful to understand the main reasons why we're still giving them full credit, despite the valuable efforts of their ever-growing competitors.

# The ASP.NET Core revolution

To summarize what happened in the ASP.NET world within the last 4 years is not an easy task; in short, we can say that we've undoubtedly witnessed the most important series of changes in .NET Framework since the year it came to life. This was a revolution that changed the whole Microsoft approach to software development in almost every way. To properly understand what happened through these years, it can be useful to identify some distinctive key frames within a slow, yet constant, journey that allowed a company known (and somewhat loathed) for its proprietary software, licenses, and patents to become a driving force for open source development worldwide.

The first relevant step, at least in my humble opinion, was taken on April 3, 2014 at the annual Microsoft Build Conference, which took place at the Moscone Center (West) in San Francisco. It was there, during a memorable keynote speech, that Anders Hejlsberg – father of Delphi and lead architect of C# – publicly released the first version of the .NET Compiler Platform, known as Roslyn, as an open source project. It was also there that Scott Guthrie, executive vice president of the Microsoft Cloud and AI group, announced the official launch of the .NET Foundation, a non-profit organization aimed at improving open source software development and collaborative work within the .NET ecosystem.

From that pivotal day, the .NET development team published a constant flow of Microsoft open source projects on the GitHub platform, including: Entity Framework Core (May 2014), TypeScript (October 2014), .NET Core (October 2014), CoreFX (November 2014), CoreCLR and RyuJIT (January 2015), MSBuild (March 2015), .NET Core CLI (October 2015), Visual Studio Code (November 2015), .NET Standard (September 2016), and so on.

## ASP.NET Core 1.x

The most important achievement brought by these efforts towards open source development was the public release of ASP.NET Core 1.0, which came out in Q3 2016. It was a complete reimplementation of the ASP.NET Framework that we knew since January 2002 and that had evolved, without significant changes in its core architecture, up to version 4.6.2 (August 2016). The brand new framework united all the previous web application technologies, such as MVC, Web API, and web pages, into a single programming module, formerly known as MVC6. The new framework introduced a fully featured, cross-platform Component, also known as .NET Core, shipped with the whole set of open source tools mentioned previously, namely, a compiler platform (Roslyn), a cross-platform runtime (CoreCLR), and an improved x64 Just-In-Time compiler (RyuJIT).

Some of you may be wondering what happened to ASP.NET 5 and Web API 2, as these used to be quite popular names until mid-2016.

ASP.NET 5 was no less than the original name of ASP.NET Core before the developers chose to rename it to emphasize the fact that it is a complete rewrite. The reasons for that, along with the Microsoft vision about the new product, are further explained in the following Scott Hanselman blog post that anticipated the changes on Jan 16, 2016:
`http://www.hanselman.com/blog/ASPNET5IsDeadIntroducingASPNETCore10AndNETCore10.aspx`.

For those who don't know, Scott Hanselman is the outreach and community manager for .NET/ASP.NET/IIS/Azure and Visual Studio since 2007. Additional information regarding the perspective switch is also available in the following article by Jeffrey T. Fritz, program manager for Microsoft and a NuGet team leader:
`https://blogs.msdn.microsoft.com/webdev/2016/02/01/an-update-on-asp-net-core-and-net-core/`.

As for Web API 2, it was a dedicated framework for building HTTP services that returned pure JSON or XML data instead of web pages. Initially born as an alternative to the MVC platform, it has been merged with the latter into the new, general-purpose web application framework known as MVC6, which is now shipped as a separate module of ASP.NET Core.

The 1.0 final release was shortly followed by ASP.NET Core 1.1 (Q4 2016), which brought some new features and performance enhancements, and also addressed many bugs and compatibility issues affecting the earlier release. These new features include the ability to configure middleware as filters (by adding them to the MVC pipeline rather than the HTTP request pipeline), a built-in, host-independent URL rewrite module, made available through the dedicated `Microsoft.AspNetCore.Rewrite` NuGet package, View Components as tag helpers, View compilation at runtime instead of on demand, .NET native Compression and Caching middleware modules, and so on.

For a detailed list of all the new features, improvements, and bug fixes of ASP.NET Core 1.1, check out the following links:

**Release notes**: `https://github.com/aspnet/AspNetCore/releases/1.1.0`.

**Commits list**: `https://github.com/dotnet/core/blob/master/release-notes/1.1/1.1-commits.md`.

# ASP.NET Core 2.x

Another major step was taken with ASP.NET Core 2.0, which came out in Q2 2017 as a preview and then in Q3 2017 for the final release. The new version featured a wide number of significant interface improvements, mostly aimed at standardizing the shared APIs among .NET Framework, .NET Core, and .NET Standard to make them backward-compatible with .NET Framework. Thanks to these efforts, moving existing .NET Framework projects to .NET Core and/or .NET Standard became a lot easier than before, giving many traditional developers the chance to try and adapt to the new paradigm without losing their existing know-how.

Again, the major version was shortly followed by an improved and refined one: ASP.NET Core 2.1. This was officially released on May 30, 2018 and introduced a series of additional security and performance improvements, as well as a bunch of new features, including SignalR, an open source library that simplifies adding real-time web functionality to .NET Core apps; Razor class libraries; a significant improvement in Razor SDK that allows developers to build views and pages into reusable class libraries, and/or library projects that could be shipped as NuGet packages; Identity UI library and scaffolding, to add identity to any app and customize it to meet your needs, HTTPS support enabled by default; built-in **General Data Protection Regulation** (**GDPR**) support using privacy-oriented APIs and templates that give users control over their personal data and cookie consent; updated SPA templates for Angular and ReactJS client-side frameworks; and much more.

For a detailed list of all the new features, improvements, and bug fixes of ASP.NET Core 2.1, check out the following links:

**Release notes**: `https://docs.microsoft.com/en-US/aspnet/core/release-notes/aspnetcore-2.1`.

**Commits list**: `https://github.com/dotnet/core/blob/master/release-notes/2.1/2.1.0-commit.md`.

Wait a minute: did we just say Angular? Yeah, that's right. As a matter of fact, since its initial release, ASP.NET Core has been specifically designed to seamlessly integrate with popular client-side frameworks such as ReactJS and Angular. It is precisely for this reason that books such as this do actually exist. The major difference introduced in ASP.NET Core 2.1 is that the default Angular and ReactJS templates have been updated to use the standard project structures and build systems for each framework (Angular CLI and NPX's `create-react-app` command) instead of relying on task runners such as Grunt or Gulp, module builders such as webpack, or toolchains such as Babel, which were widely used in the past although were quite difficult to install and configure.

Being able to eliminate the need for these tools was a major achievement, which played a decisive role in revamping the .NET Core usage and growth rate among the developer communities since 2017. If you take a look at the two previous installments of this book – *ASP.NET Core and Angular 2*, published in mid-2016, and *ASP.NET Core 2 and Angular 5*, out in late 2017 – and compare their first chapter with this one, you will see the huge difference between having to manually use Gulp, Grunt, or webpack and relying on the integrated framework-native tools. This is a substantial reduction in complexity that would greatly benefit any developer, especially those less accustomed to working with those tools.

After 6 months from the release of the 2.1 version, the .NET Foundation came out with a further improvement: ASP.NET Core 2.2 was released on December 4, 2018 with several fixes and new features, such as an improved endpoint routing system for better dispatching of requests, updated templates featuring Bootstrap 4 and Angular 6 support, a new health checks service to monitor the status of deployment environments and their underlying infrastructures, including container orchestration systems such as Kubernetes, built-in HTTP/2 support in Kestrel, a new SignalR Java client to ease the usage of SignalR within Android apps, and so on.

For a detailed list of all the new features, improvements, and bug fixes of ASP.NET Core 2.2, check out the following links:

**Release notes**: `https://docs.microsoft.com/en-US/aspnet/core/release-notes/aspnetcore-2.2`.

**Commits list**: `https://github.com/dotnet/core/blob/master/release-notes/2.2/2.2.0/2.2.0-commits.md`.

# ASP.NET Core 3.x

ASP.NET Core 3 was released in September 2019 and came with another bunch of performance and security improvements and new features, such as Windows desktop applications support (Windows only) with advanced importing capabilities for Windows Forms and **Windows Presentation Foundation** (**WPF**) applications, C# 8 support, .NET Platform-Dependent Intrinsic access through a new set of built-in APIs that could bring significant performance improvements in certain scenarios, single-file executables support via the `dotnet publish` command using the `<PublishSingleFile>` XML element in project configuration or through the `/p:PublishSingleFile` command-line parameter, a new built-in JSON support featuring high performance and low allocation that's arguably 2x-3x faster than the JSON.NET third-party library (which became a de facto standard in most ASP.NET web projects), TLS 1.3 and OpenSSL 1.1.1 support in Linux, some important security improvements in the `System.Security.Cryptography` namespace, including AES-GCM and AES-CCM ciphers support, and so on.

A lot of work has also been done to improve the performance and reliability of the framework when used in a containerized environment. The ASP.NET Core development team put a lot of effort into improving the .NET Core Docker experience on .NET Core 3.0. More specifically, this is the first release featuring substantive runtime changes to make CoreCLR more efficient, honor Docker resource limits better (such as memory and CPU) by default, and offer more configuration tweaks. Among the various improvements, we could mention improved memory and GC heap usage by default, and PowerShell Core, a cross-platform version of the famous automation and configuration tool, which is now shipped with the .NET Core SDK Docker container images.

.NET Core Framework 3 also introduced Blazor, a free and open source web framework that enables developers to create web apps using C# and HTML.

Last but not least, it's worth noting that the new .NET Core SDK is much smaller than the previous installments, mostly thanks to the fact that the development team removed a huge set of unnecessary artifacts included in the various NuGet packages that were used to assemble the previous SDKs (including ASP.NET Core 2.2) from the final build, thus wasting a lot of space. The size improvements are huge for Linux and macOS versions, while less noticeable on Windows because that SDK also contains the new WPF and Windows Forms set of platform-specific libraries.

For a detailed list of all the new features, improvements, and bug fixes of ASP.NET Core 3.0, check out the following links:

**Release notes**: `https://docs.microsoft.com/en-us/dotnet/core/whats-new/dotnet-core-3-0`.

**ASP.NET Core 3.0 releases page**: `https://github.com/dotnet/core/tree/master/release-notes/3.0`.

ASP.NET Core 3.1, which is the most recent stable version at the time of writing, was released on December 3, 2019. The changes in the latest version are mostly focused on Windows desktop development, with the definitive removal of a number of legacy Windows Forms controls (*DataGrid, ToolBar, ContextMenu, Menu, MainMenu*, and *MenuItem*) and added support for creating C++/CLI Components (on Windows only).

Most of the ASP.NET Core updates were fixes related to Blazor, such as preventing default actions for events and stopping event propagation in Blazor apps, partial class support for Razor Components, additional *Tag Helper Component* features, and so on; however, much like the other *.1* releases, the primary goal of .NET Core 3.1 was to refine and improve the features already delivered in the previous version, with more than 150 performance and stability issues fixed.

A detailed list of the new features, improvements, and bug fixes introduced with ASP.NET Core 3.1 is available at the following URL:

**Release notes**: `https://docs.microsoft.com/en-us/dotnet/core/whats-new/dotnet-core-3-1`.

This concludes our journey through the recent history of ASP.NET Core. In the next section, we'll move our focus to the Angular ecosystem, which experienced a rather similar turn of events.

# What's new in Angular?

If following in the footsteps of Microsoft and the .NET Foundation in recent years has not been an easy task, things were not going to get any better when we turned our eyes to the client-side web framework known as Angular. In order to understand what happened there, we have to go back 10 years when JavaScript libraries such as jQuery and MooTools were dominating the client-side scenes, the first client-side frameworks such as Dojo, Backbone.js, and Knockout.js were struggling to gain popularity and reach wide adoption, and stuff such as React and Vue.js didn't even exist.


Truth be told, jQuery is still dominating the scene to a huge extent, at least according to Libscore (`http://libscore.com/#libs`) and w3Techs (`https://w3techs.com/technologies/overview/javascript_library/all`). However, despite being used by 74.1% of all websites, it's definitely a less chosen option for web developers than it was 10 years ago.

## GetAngular

The story of AngularJS started in 2009 when Miško Hevery (now senior computer scientist and Agile coach at Google) and Adam Abrons (now director of engineering at Grand Rounds) were working on their side project, an **end-to-end** (**E2E**) web development tool that would have offered an online JSON storage service and also a client-side library to build web applications depending on it. To publish their project, they took the `GetAngular.com` hostname.

During that time, Hevery, who was already working at Google, was assigned to the Google Feedback project with two other developers. Together, they wrote more than 17,000 lines of code in 6 months, slowly sinking into a frustrating scenario of code bloat and testing issues. Given the situation, Hevery asked his manager to rewrite the application using GetAngular (the side project mentioned previously), betting that he could do that alone within 2 weeks. His manager accepted and Hevery lost the bet shortly thereafter, as the whole thing took him 3 weeks instead of two; however, the new application had only 1,500 lines of code instead of 17,000. This was more than enough to get Google's interest for the new framework, which was given the name of AngularJS shortly thereafter.

To listen to the full story, take a look at the following Miško Hevery keynote speech at ng-conf 2014:

`https://www.youtube.com/watch?v=r1A1VR0ibIQ`.

# AngularJS

The first stable release of AngularJS (version 0.9.0, also known as dragon-breath) was released on GitHub in October 2010 under an MIT license; when AngularJS 1.0.0 (also known as temporal domination) came out on June 2012, the framework had already achieved huge popularity within the web development communities worldwide.

The reasons for such extraordinary success can hardly be summarized in a few words, but I'll try to do that nonetheless by emphasizing some fundamental key selling points:

- **Dependency injection**: AngularJS was the first client-side framework to implement it. This was undeniably a huge advantage over the competitors, including DOM-manipulating libraries such as jQuery. With AngularJS, developers could write loosely coupled and easily testable Components, leaving the framework with the task of creating them, resolving their dependencies, and passing them to other Components when requested.
- **Directives**: These can be described as markers on specific DOM items such as elements, attributes, styles, and so on: a powerful feature that could be used to specify custom and reusable HTML-like elements and attributes that define data bindings and/or other specific behaviors of presentation Components.
- **Two-way data binding**: The automatic synchronization of data between model and view Components. When data in a model changes, the view reflects the change; when data in the view changes, the model is updated as well. This happens immediately and automatically, which makes sure that the model and the view are updated at all times.
- **Single-page approach**: AngularJS was the first framework to completely remove the need for page reloads. This provided great benefits at both server-side (fewer and smaller network requests) and client-side level (smoother transitions, more responsive experience), and paved the way for the Single-Page Application pattern that would be also adopted by React, Vue.js, and the other runner-up frameworks later on.

- **Cache-friendly**: All the AngularJS magic was meant to happen on the client-side, without any server-side effort to generate the UI/UX parts. For this very reason, all AngularJS websites could be cached anywhere and/or made available through a CDN.

For a detailed list of AngularJS features, improvements, and bug fixes from 0.9.0 through 1.7.8, check out the following link:

Angularjs 1.x Changelog: `https://github.com/angular/angular.js/blob/master/CHANGELOG.md`.

# Angular 2

The new release of AngularJS, released on September 14, 2016 and known as Angular 2, was a complete rewrite of the previous one, entirely based upon the new ECMAScript version 6 (officially ECMAScript 2015) specifications. Just like the ASP.NET Core rewrite, the revolution brought such a number of breaking changes at architectural level, HTTP pipeline handling, app life cycle, and state management that porting the old code to the new one was nearly impossible. Despite keeping its former name, the new Angular version was a brand new framework with little or nothing in common with the previous one.

The choice of not making Angular 2 backward-compatible with AngularJS clearly demonstrated the intention of the author's team to adopt a completely new approach: not only in the code syntax, but also in their way of thinking and designing the client app. The new Angular was highly modular, Component-based, came with a new and improved dependency injection model, and a whole lot of programming patterns its older cousin had never heard of.

Here's a brief list of the most important improvements introduced with Angular 2:

- **Semantic versioning**: Angular 2 is the first release to use semantic versioning, also known as SemVer: a universal way of versioning the various software releases to help developers track down what's going on without having to dig into the changelog details. SemVer is based on three numbers – *X.Y.Z* – where *X* stands for a *major* version, *Y* stands for a *minor* version, and *Z* stands for a *patch* release. More specifically, the *X* number, representing the *major* version, gets incremented when incompatible API changes are made to stable APIs, the *Y* number, representing the *minor* version, gets incremented when backward-compatible functionality is added, and the *Z* number, representing a *patch* release, gets incremented when a backward-compatible bug is fixed. Such improvements can be easily underestimated, yet it's a must-have for most modern software development scenarios where **Continuous Delivery** (**CDE**) is paramount and new versions are released with great frequency.
- **TypeScript:** If you're a seasoned web developer, you probably already know what TypeScript is. In case you don't, no worries, you'll get way more on that later on since we're going to use it a lot during the Angular-related chapters of this book. For now, let's just say that TypeScript is a Microsoft-made superset of JavaScript that allows the use of all ES2015 features (such as Default-Rest-Spread Parameters, Template Literals, Arrow Functions, Promises, and more) and adds powerful type-checking and object-oriented features during development (such as class and type declarations). The TypeScript source code can be transpiled into standard JavaScript code that all browsers can understand.
- **Server-side rendering** (**SSR**): Angular 2 comes with Angular Universal, an open source technology that allows a *back-end* server to run Angular applications and serve only the resulting static HTML files to the client. In a nutshell, the server will render a first pass of the page for faster delivery to the client, then immediately refresh it with client code. SSR has its caveats, such as requiring Node.js to be installed on the host machine to execute the necessary pre-rendering steps, as well as having the whole `node_modules` folder there, but can greatly increase the app's response time for a typical internet browser, thus mitigating a known AngularJS performance issue.
- **Angular Mobile Toolkit** (**AMT**): A set of tools specifically designed for building high-performance mobile apps.
- **Command-line interface** (**CLI**): The new CLI introduced with Angular 2 could be used by developers to generate Components, routes, services, and pipes via console/Terminal commands, together with simple test shells.

- **Components**. These are the main building blocks of Angular 2, entirely replacing the Controllers and scopes of AngularJS, and also lifting most of the tasks previously covered by the former directives. Application data, business logic, templating, and the styling of an Angular 2 app can all be made using Components.

I did my best to explore most of these features in my first book, *ASP.NET Core and Angular 2*, which was published in October 2016, right after the final release of the two frameworks:

`https://www.packtpub.com/application-development/aspnet-core-and-angular-2`.

## Angular 4

On March 23, 2017, Google released Angular 4: the number 3 version was skipped entirely in order to unify all the major versions of the many Angular Components that had been developed separately until that date, such as Angular Router, which already was at version 3.x at the time. Starting with Angular 4, the entire Angular framework was then unified into the same MAJOR.MINOR.PATCH SemVer pattern.

The new major version brought a limited number of breaking changes, such as a new and improved routing system, TypeScript 2.1+ support (and requirement), and some deprecated interfaces and tags. There was also a good amount of improvements, including:

- **Ahead-of-time** (**AOT**) **compilation**: Angular 4 compiles the templates during the build phase and generates JavaScript code accordingly. That's a huge architectural improvement over the JIT mode used by AngularJS and Angular 2 where the app was compiled at runtime. For example, when the application starts, not only is the app faster since the client doesn't have to compile anything, but it throws/breaks at build time instead of during runtime for most Component errors, thus leading to more secure and stable deployments.
- **Animations npm package**: All the existing UI animations and effects – as well as new ones – have been moved to the `@angular/animations` dedicated package instead of being part of `@angular/core`. This was a smart move to give non-animated apps the chance to drop that part of code, thus being much smaller and arguably faster.

Other notable improvements included: a new form validator to check for valid email addresses, a new paramMap interface for URL parameters in the HTTP routing module, better internalization support, and so on.

## Angular 5

Released on November 1, 2017, Angular 5 featured TypeScript 2.3 support, another small set of breaking changes, many performance and stability improvements, and a couple of new features, such as the following:

- **New HTTP Client API**: Starting from Angular 4.3, the `@angular/http` module was put aside in favor of a new `@angular/common/http` package with better JSON support, interceptors and immutable request/response objects, and other stuff. The switch was completed in Angular 5 with the previous module being deprecated and the new one recommended for use in all apps.
- **State Transfer API**: A new feature that gives the developer the ability to transfer the state of the application between the server and the client.
- **A new set of router events for more granular control over the HTTP life cycle**: `ActivationStart`, `ActivationEnd`, `ChildActivationStart`, `ChildActivationEnd`, `GuardsCheckStart`, `GuardsCheckEnd`, `ResolveStart` and `ResolveEnd`.

November 2017 was also the release month of my *ASP.NET Core 2 and Angular 5* book, which covers most of the aforementioned improvements:

`https://www.packtpub.com/application-development/aspnet-core-2-and-angular-5`.

In June 2018, that book was made available as a video course:

`https://www.packtpub.com/web-development/asp-net-core-2-and-angular-5-video`.

# Angular 6

Released on April 2018, Angular 6 was mostly a maintenance release, more focused on improving the overall consistency of the framework and its toolchain than adding new features. Therefore, there were no major breaking changes. RxJS 6 supports a new way to register providers, the new `providedIn` injectable decorator, improved Angular Material support (a Component specifically made to implement material design in the Angular client-side UI), more CLI commands/updates, and so on.

Another improvement worth mentioning was the new CLI `ng add` command, which uses the package manager to download new dependencies and invoke an installation script to update our project with configuration changes, add additional dependencies, and/or scaffold package-specific initialization code.

Last, but not least, the Angular team introduced Ivy, a next-generation Angular rendering engine that aims to increase the speed and decrease the size of the application.

# Angular 7

Angular 7 came out in October 2018 and it definitely was a major update, as we can easily guess by reading the words written by Stephen Fluin, developer relations lead at Google and prominent Angular spokesman, on the official Angular development blog upon the official release:

> *"This is a major release spanning the entire platform, including the core framework, Angular Material, and the CLI with synchronized major versions. This release contains new features for our toolchain, and has enabled several major partner launches."*

Here's a list of the new features:

- **Easy upgrade**: Thanks to the groundwork made with version 6, the Angular team was able to reduce the steps that need to be done to upgrade an existing Angular app from an older version to the most recent one. The detailed procedure can be viewed by visiting `https://update.angular.io`, an incredibly useful Angular upgrade interactive guide that can be used to quickly recover the required steps, such as CLI commands, package updates, and so on. This needs to be done to upgrade an existing Angular app from an older version of Angular to a most recent one.

- **CLI update**: A new command that attempts to automatically upgrade the Angular application and its dependencies by following the procedure mentioned previously.
- **CLI prompts**: The Angular command-line interface has been modified to prompt users when running common commands such as `ng new` or `ng add @angular/material` to help developers discover built-in features such as routing, SCSS support, and so on.
- **Angular Material and CDK**: Additional UI elements such as virtual scrolling, a Component that loads and unloads elements from the DOM based on the visible parts of a list, making it possible to build very fast experiences for users with very large scrollable lists, CDK-native drag-and-drop support, improved drop-down list elements, and more.
- **Partner launches**: Improved compatibility with a number of third-party community projects such as: Angular Console; a downloadable console for starting and running Angular projects on your local machine, `AngularFire`, the official Angular package for Firebase integration, Angular for NativeScript; an integration between Angular and NativeScript – a framework for building native iOS and Android apps using JavaScript and/or JS-based client frameworks, some interesting new Angular-specific features for StackBlitz; an online IDE that can be used to create Angular and React projects, such as a tabbed editor and an integration with the Angular Language Service, and so on.
- **Updated dependencies**: Added support for TypeScript 3.1, RxJS 6.3, and Node 10, although the previous versions can still be used for backward compatibility.

The Angular Language Service is a way to get completions, errors, hints, and navigation inside Angular templates: think about it as a virtuous mix between a syntax highlighter, IntelliSense, and a real-time syntax error checker. Before Angular 7, which added the support for StackBlitz, such a feature was only available for Visual Studio Code and WebStorm.

For additional information about the Angular Language Service, take a look at the following URL:

`https://angular.io/guide/language-service`

# Angular 8

Angular 7 was quickly followed by Angular 8, which was released on May 29, 2018. The new release is mostly about Ivy, the long-awaited new compiler/runtime of Angular: although being an ongoing project since Angular 5, version 8 was the first one to officially offer a runtime switch to actually opt-in to using Ivy, which would become the default runtime starting from Angular 9.

> In order to enable Ivy on Angular 8, the developers had to add an `"enableIvy": true` property to the `angularCompilerOptions` section within the app's `tsconfig.json` file.

Those who want to know more about Ivy are encouraged to give an extensive look at the following post by Cédric Exbrayat, co-founder and trainer at the Ninja-Squad website and now part of the Angular developer team:

`https://blog.ninja-squad.com/2019/05/07/what-is-angular-ivy/`.

Other notable improvements and new features include:

- **Bazel support**: Angular 8 was the first version to support Bazel, a free software tool developed and used by Google for the automation of building and testing software. It can be very useful for developers aiming to automate their delivery pipeline as it allows incremental build and tests, and even the possibility to configure remote builds (and cache) on a build farm.
- **Routing**: A new syntax was introduced to declare the lazy-loading routes using the `import()` syntax from TypeScript 2.4+ instead of relying on a string literal. The old syntax has been kept for backward compatibility, but will be arguably dropped soon.
- **Service workers**: A new registration strategy has been introduced to allow developers to choose when to register their workers instead of doing it automatically at the end of the app's startup life cycle. It's also possible to bypass a service worker for a specific HTTP request using the new `ngsw-bypass` header.
- **Workspace API**: A new and more convenient way to read and modify the Angular workspace configuration instead of manually modifying the `angular.json` file.

In client-side development, a service worker is a script that the browser runs in the background to do any kind of stuff that doesn't require either a user interface or any user interaction.

The new version also introduced some notable breaking changes – mostly due to Ivy – and removed some long-time deprecated packages such as `@angular/http`, which was replaced by `@angular/common/http` in Angular 4.3 and then officially deprecated in 5.0.

A comprehensive list of all the deprecated APIs can be found in the official Angular deprecations guide at the following URL:

`https://angular.io/guide/deprecations`.

## Angular 9

Last, but not least, we come to Angular 9, which was released in February 2020 after a long streak of release candidates through 2019 Q4 and is currently the most recent version.

The new release brings the following new features:

- **JavaScript bundles and performance**: An attempt to fix the very large bundle files, one of the most cumbersome issues of the previous versions of Angular, has drastically increased the download time and brought down the overall performances.
- **Ivy compiler**: The new Angular build and render pipeline, shipped with Angular 8 as an opt-in preview, is now the default rendering engine.
- **Selector-less bindings**: A useful feature that was available to the previous rendering engine, but missing from the Angular 8 Ivy preview, is now available to Ivy as well.
- **Internationalization**: Another Ivy enhancement that makes use of the Angular CLI to generate most of the standard code necessary to create files for translators and to publish an Angular app in multiple languages, thanks to the new `i18n` attribute.

The new `i18n` attribute is a numeronym, which is often used as an alias of internationalization. The number 18 stands for the number of letters between the first i and the last n in the word internationalization. The term seems to have been coined by the **Digital Equipment Corporation** (**DEC**) around the 1970s or 1980s, together with `l10n` for localization, due to the excessive length of the two words.

The long-awaited Ivy compiler deserves a couple more words, being a very important feature for the future of Angular.

As you most likely already know, the rendering engine plays a major role in the overall performance of any *front-end* framework since it's the tool that translates the actions and intents performed by the presentation logic (in Angular, Components, and templates) into the instructions that will update the DOM. If the renderer is more efficient, it will arguably require less instructions, thus increasing the overall performance while decreasing the amount of required JavaScript code at the same time. Since the JavaScript bundles produced by Ivy are much smaller than the previous rendering engine, Angular 9's overall improvement is relevant in terms of both performance and size.

This concludes our brief review of the recent history of the ASP.NET Core and Angular ecosystems. In the next sections, we'll summarize the most important reasons that led us to actually choosing them in 2020.

# Reasons for choosing .NET Core and Angular

As we have seen, both frameworks have gone through three intense years of changes. This led to a whole refoundation of their core and, right after that, a constant strain to get back on top – or at least not lose ground against most modern frameworks that came out after their now departed golden age. They are eager to dominate the development scene: Python, Go, and Rust for the server-side part and React, Vue.js, and Ember.js for the client-side part, not to mention the Node.js and Express ecosystem, and most of the old competitors from the 1990s and 2000s, such as Java, Ruby, and PHP, which are still alive and kicking.

That said, here's a list of good reasons for picking ASP.NET Core in 2019:

- **Performance**: The new .NET Core web stack is considerably fast, especially since version 3.x.
- **Integration:** It supports most, if not all, modern client-side frameworks, including Angular, React, and Vue.js.
- **Cross-platform approach**: .NET Core web applications can run on Windows, macOS, and Linux in an almost seamless way.
- **Hosting**: .NET Core web applications can be hosted almost anywhere: from a Windows machine with IIS to a Linux appliance with Apache or Nginx, from Docker containers to edge-case, self-hosting scenarios using the Kestrel and WebListener HTTP servers.
- **Dependency injection**: The framework supports a built-in dependency injection design pattern that provides a huge number of advantages during development, such as reduced dependencies, code reusability, readability, and testing.
- **Modular HTTP pipeline**: ASP.NET Core middleware grants developers granular control over the HTTP pipeline, which can be reduced to its core (for ultra-lightweight tasks) or enriched with powerful, highly configurable features such as internationalization, third-party authentication/authorization, caching, routing, and so on.
- **Open source**: The whole .NET Core stack has been released as open source and is entirely focused on strong community support, thus being reviewed and improved by thousands of developers every day.
- **Side-by-side execution**: It supports the simultaneous running of multiple versions of an application or Component on the same machine. This basically means that it's possible to have multiple versions of the common language runtime, and multiple versions of applications and Components that use a version of the runtime, on the same computer at the same time. This is great for most real-life development scenarios as it gives the development team more control over which versions of a Component an application binds to, and more control over which version of the runtime an application uses.

As for the Angular framework, the most important reason we're picking it over other excellent JS libraries such as React, Vue.js, and Ember.js is the fact that it already comes with a huge pack of features out of the box, making it the most suitable choice, although maybe not as simple to use as other framework/libraries. If we combine that with the consistency benefits brought by the TypeScript language, we can say that Angular, from its 2016 rebirth up to the present day, embraced the framework approach more convincingly than the others. This has been consistently confirmed over the course of the past 3 years where the project underwent six major versions and gained a lot in terms of stability, performance, and features, without losing much in terms of backward compatibility, best practices, and overall approach. All these reasons are solid enough to invest in it, hoping it will continue to keep up with these compelling premises.

Now that we have acknowledged the reasons to use these frameworks, let's ask ourselves the best way to find out more about them: the next sections should give us the answers we need.

# A full-stack approach

Learning to use ASP.NET Core and Angular together would mean being able to work with both the *front-end* (client side) and *back-end* (server side) of a web application; to put it in other words, it means being able to design, assemble, and deliver a complete product.

Eventually, in order to do that, we'll need to dig through the following:

- *Back-end* programming
- *Front-end* programming
- UI styling and UX design
- Database design, modeling, configuration, and administration
- Web server configuration and administration
- Web application deployment

At first glance, it can seem that this kind of approach goes against common sense; a single developer should not be allowed to do everything by himself. Every developer knows that the *back-end* and the *front-end* require entirely different skills and experiences, so why in the world should we do that?

Before answering this question, we should understand what we really mean when we say *being able to*. We don't have to become experts on every single layer of the stack; no one expects us to. When we choose to embrace the full-stack approach, what we really need to do is raise our awareness level throughout the whole stack we're working on; this means that we need to know how the *back-end* works, and how it can and will be connected to the *front-end*. We need to know how the data will be stored, retrieved, and then served through the client. We need to acknowledge the interactions we will need to layer out between the various Components that our web application is made of, and we need to be aware of security concerns, authentication mechanisms, optimization strategies, load balancing techniques, and so on.

This doesn't necessarily mean that we have to have strong skills in all these areas; as a matter of fact, we hardly ever will. Nonetheless, if we want to pursue a full-stack approach, we need to understand the meaning, role, and scope of all of them. Furthermore, we should be able to work our way through any of these fields whenever we need to.

# SPAs, NWAs, and PWAs

In order to demonstrate how ASP.NET Core and Angular can work together to their full extent, we couldn't think of anything better than building some small SPA projects with most, if not all, Native Web Application features. The reason for such a choice is quite obvious: there is no better approach to show some of the best features they have to offer nowadays. We'll have the chance to work with modern interfaces and patterns such as HTML5 pushState API, webhooks, data transport-based requests, dynamic web Components, UI data bindings, and a stateless, AJAX-driven architecture capable of flawlessly encompassing all of these. We'll also make good use of some distinctive NWA features such as service workers, web manifest files, and so on.

If you don't know the meaning of these definitions and acronyms, don't worry, we are going to explore these concepts in the next couple of sections, which are dedicated to enumerating the most relevant features of the following types of web applications: SPAs, NWAs, and PWAs. While we're there, we'll also try to figure out the most common product owner's expectations for a typical web-based project.

# Single-page application

To put it briefly, an SPA is a web-based application that struggles to provide the same user experience as a desktop application. If we consider the fact that all SPAs are still served through a web server and thus accessed by web browsers just like any other standard website, we can easily understand how that desired outcome can only be achieved by changing some of the default patterns commonly used in web development, such as resource loading, DOM management, and UI navigation. In a good SPA, both contents and resources – HTML, JavaScript, CSS, and so on – are either retrieved within a single page load or are dynamically fetched when needed. This also means that the page doesn't reload or refresh, it just changes and adapts in response to user actions, performing the required server-side calls behind the scenes.

These are some of the key features provided by a competitive SPA nowadays:

- **No server-side round trips**: A competitive SPA can redraw any part of the client UI without requiring a full server-side round trip to retrieve a full HTML page. This is mostly achieved by implementing a **separation of concerns** (**SOC**) design principle, which means that the data source, the business logic, and the presentation layer will be separated.
- **Efficient routing**: A competitive SPA is able to keep track of the user's current state and location during its whole navigation experience using organized, JavaScript-based routers. We'll talk more about that in the upcoming chapters when we introduce the concepts of server-side and client-side routing.
- **Performance and flexibility**: A competitive SPA usually transfers all of its UI to the client, thanks to its JavaScript SDK of choice (Angular, JQuery, Bootstrap, and so on). This is often good for network performance as increasing client-side rendering and offline processing reduces the UI impact over the network. However, the real deal brought by this approach is the flexibility granted to the UI as the developer will be able to completely rewrite the application *front-end* with little or no impact on the server, aside from a few of the static resource files.

This list can easily grow, as these are only some of the major advantages of a properly designed, competitive SPA. These aspects play a major role nowadays, as many business websites and services are switching from their traditional **Multi-Page Application** (**MPA**) mindset to fully-committed or hybrid SPA-based approaches.

# Native web application

Multi-page applications, which have been increasingly popular since 2015, are commonly called NWAs because they tend to implement a number of small-scale, single-page modules bound together upon a multipage skeleton rather than building a single, monolithic SPA.

A – not to mention the fact that there are also a lot of enterprise-level SPAs and NWAs flawlessly serving thousands of users every day. Want to name a few? WhatsApp Web and Teleport Web, Flickr, plus a wide range of Google web services, including Gmail, Contacts, Spreadsheet, Maps, and more. These services, along with their huge user base, are the ultimate proof that we're not talking about a silly trend that will fade away with time; conversely, we're witnessing the completion of a consolidated pattern that's definitely meant to stay.

# Progressive web application

During 2015, another web development pattern pushed its way into light when Frances Berriman (a British freelance designer) and Alex Russel (a Google Chrome engineer) used the term PWAs for the first time to refer to those web applications that could take advantage of a couple of new important features supported by modern browsers: service workers and web manifest files. These two important improvements could be successfully used to deliver some functionalities usually only available on mobile apps – push notifications, offline mode, permission-based hardware access, and so on – using standard web-based development tools such as HTML, CSS, and JavaScript.

The rise of Progressive Web Apps began in March 19, 2018, when Apple implemented support for service workers in Safari 11.1. Starting from that date, PWAs have been widely adopted throughout the industry thanks to their undeniable advantages over MPAs, SPAs, and NWAs: faster load times, smaller application sizes, higher audience engagement, and so on.

Here are the main technical features of a Progressive Web App (according to Google):

- **Progressive**: Works for every user, regardless of browser choice, using progressive enhancement principles
- **Responsive**: Fits any form factor: desktop, mobile, tablet, or forms yet to emerge.
- Connectivity independent: Service workers allow offline uses, or on low-quality networks.
- **App-like**: Feels like an app to the user with app-style interactions and navigation.
- **Fresh**: Always up to date due to the service worker update process
- **Safe**: Served via HTTPS to prevent snooping and ensure content hasn't been tampered with
- **Discoverable**: Identifiable as an application by a web manifest (`manifest.json`) file, and a registered service worker, and discoverable by search engines
- **Re-engageable**: Ability to use push notifications to maintain engagement with the user
- **Installable**: Provides home screen icons without the use of an App Store
- **Linkable**: Can easily be shared via a URL and does not require complex installation

However, their technical baseline criteria can be restricted to the following subset:

- **HTTPS**: They must be served from a secure origin, which means over TLS with green padlock displays (no active mixed content).
- **Minimal offline mode**: They must be able to start, even if the device is not connected to the web, with limited functions or at least displaying a custom offline page.
- **Service workers**: They have to register a service worker with a fetch event handler (which is required for minimal offline support, as explained previously).
- **Web manifest file**: They need to reference a valid `manifest.json` file with at least four key properties (`name`, `short_name`, `start_url`, and `display`) and a minimum set of required icons.

For those interested in reading about this directly from the source, here's the original link from the Google Developers website:

```
https://developers.google.com/web/progressive-web-apps/.
```

In addition, here are two follow-up posts from Alex Russell's *Infrequently Noted* blog:

```
https://infrequently.org/2015/06/progressive-apps-escaping-
tabs-without-losing-our-soul/.
https://infrequently.org/2016/09/what-exactly-makes-
something-a-progressive-web-app/.
```

For those who don't know, Alex Russell has worked as a senior staff software engineer at Google since December 2008.

Although having some similarities, PWAs and SPAs are two different concepts, have different requirements, and differ in many important aspects. As we can see, none of the PWA requirements mentioned previously are referring to Single-Page Applications or server-side round trips. A Progressive Web App *can* work within a single HTML page and AJAX-based requests (thus also being an SPA), but it *could* also request other server-rendered (or static) pages and/or perform standard HTTP GET or POST requests, much like an MPA. It's also the opposite: any SPA can implement any single PWA technical criteria, depending on the product owner's requirements (more on that later), the server-side and client-side frameworks adopted, and the developer's ultimate goal.

Since we're going to use Angular, which is all about developing Single-Page Applications, and also ships with a strong and steady service worker implementation since version 5, we are fully entitled to take advantage of the best of both worlds. For this very reason, we're going to use service workers – along with the benefits of increased reliability and performance they provide – whenever we need to, all while keeping a solid SPA approach. Furthermore, we're definitely going to implement some strategic HTTP round trips (and/or other redirect-based techniques) whenever we can profitably use a microservice to lift off some workload from our app, just like any good Native Web Application is meant to do.

Are all these features able to respond to modern market needs? Let's try to find it out.

# Product owner expectations

One of the most interesting, yet underrated, concepts brought out by many modern Agile software development frameworks, such as Scrum, is the importance given to the meanings and definitions of roles. Among these, there's nothing as important as the product owner, also known as the customer in Extreme Programming methodology, or customer representative elsewhere. They're the ones who bring to the development table the expectations we'll struggle to satisfy. They will tell us what's most important to deliver and when they will prioritize our work based on its manifest business value rather than its underlying architectural value. They'll be entitled by management to make decisions and make tough calls, which is sometimes great, sometimes not; this will often have a big impact on our development schedule. To cut it short, they're the ones in charge of the project; that's why, in order to deliver a web application matching their expectancy, we'll need to understand their vision and feel it as if it were our own.

This is always true, even if the project's product owner is our dad, wife, or best friend: that's how it works.

Now that we have made that clear, let's take a look at some of the most common product owner's expectations for a typical web-based SPA project. We ought to see whether the choice of using ASP.NET Core and Angular will be good enough to fulfill each one of them, as follows:

- **Early release(s)**: No matter what we're selling, the customer will always want to see what he's buying. For example, if we plan to use an Agile development framework such as Scrum, we'll have to release a potentially shippable product at the end of each sprint. If we are looking to adopt a Waterfall-based approach, we're going to have milestones, and so on. One thing is for sure, the best thing we can do in order to efficiently organize our development efforts will be to adopt an iterative and/or modular-oriented approach. ASP.NET Core and Angular, along with the strong separation of concerns granted by their underlying MVC- or MVVM-based patterns, will gracefully push us into the mindset needed to do just that.

- **GUI over back-end**: We'll often be asked to work on the GUI and *front-end* functionalities because that will be the only really viewable and measurable thing for the customer. This basically means that we'll have to mock the data model and start working on the *front-end* as soon as possible, delaying everything that relies under the hood, even if that means leaving it empty; we can say that the hood is what we need the most. Note that this kind of approach is not necessarily bad; by all means, we won't do that just to satisfy the product owner's expectations. On the contrary, the choice of using ASP.NET Core along with Angular will grant us the chance to easily decouple the presentation layer and the data layer, implementing the first and mocking the latter, which is a great thing to do. We'll be able to see where we're going before wasting valuable time or being forced to make potentially wrong decisions. ASP.NET Core's Web API interface will provide the proper tools to do that by allowing us to create a sample web application skeleton in a matter of seconds using the Controller templates available within Visual Studio and in-memory data contexts powered by Entity Framework Core, which we'll be able to access using Entity models and code first. As soon as we do that, we'll be able to switch to GUI design using the Angular presentation layer toolbox as much as we want until we reach the desired results. Once we're satisfied, we'll just need to properly implement the Web API Controller interfaces and hook up the actual data.
- **Fast completion**: None of the preceding things will work unless we also manage to get everything done in a reasonable time span. This is one of the key reasons to choose to adopt a server-side framework and a client-side framework working together with ease. ASP.NET Core and Angular are the tools of choice, not only because they're both built on a solid, consistent ground, but also because they're meant to do precisely that – get the job done on their respective sides and provide a usable interface to the other partner.
- **Adaptability**: As stated by the Agile manifesto, being able to respond to change requests is more important than following a plan. This is especially true in software development where we can even claim that anything that cannot handle change is a failed project. That's another great reason to embrace the separation of concerns enforced by our two frameworks of choice, as this grants the developer the ability to manage—and even welcome, to some extent—most of the layout or structural changes that will be expected during the development phase.

A few lines ago, we mentioned Scrum, which is one of the most popular Agile software development frameworks out there. Those who don't know it yet should definitely take a look at what it can offer to any results-driven team leader and/or project manager. Here's a good place to start:

`https://en.wikipedia.org/wiki/Scrum_(software_development).`

For those who are curious about the Waterfall model, here's a good place to learn more about it:

`https://en.wikipedia.org/wiki/Waterfall_model.`

That's about it. Note that we didn't cover everything here as it will be impossible without knowing an actual assignment. We just tried to give an extensive answer to the following general question: if we were to build an SPA and/or a PWA, would ASP.NET Core and Angular be an appropriate choice? The answer is undoubtedly yes, especially when used together.

Does this mean that we're done already? Not a chance, as we have no intention of taking this assumption for granted. Conversely, it's time for us to demonstrate this by ceasing to speak in general terms and start to put things in motion. That's precisely what we're going to do in the next section: prepare, build, and test a sample Single-Page Application project.

# A sample SPA project

What we need now is to conceive a suitable test case scenario similar to the ones we will eventually have to deal with – a sample SPA project with all the core aspects we would expect from a potentially shippable product.

In order to do this, the first thing we need to do is to become our own customer for a minute and come up with an idea; a vision to share with our other self. We'll then be able to put our developer shoes back on and split our abstract plan into a list of items we'll need to implement; these will be the core requirements of our project. Finally, we'll set up our workstation by getting the required packages, adding the resource files, and configuring both the ASP.NET Core and Angular frameworks in the Visual Studio IDE.

# Not your usual Hello World!

The code we're going to write within this book won't be just a shallow demonstration of full-stack development concepts; we won't throw some working code here and there and expect you to connect the dots. Our objective is to create solid, realistic web applications – with server-side web APIs and client-side UIs – using the frameworks we've chosen, and we're also going to do that following the current development best practices.

Each chapter will be dedicated to a single core aspect. If you feel like you already know your way there, feel free to skip to the next one. Conversely, if you're willing to follow us through the whole loop, you'll have a great journey through the most useful aspects of ASP.NET Core and Angular, as well as how they can work together to deliver the most common and useful web-development tasks, from the most trivial ones to the more complex beasts. It's an investment that will pay dividends as it will leave you with a maintainable, extensible, and well-structured project, plus the knowledge needed to build your own. The following chapters will guide us through such a journey. During that trip, we'll also learn how to take care of some important high-level aspects such as SEO, security, performance issues, best coding practices, and deployment, as they will become very important if/when our applications will be eventually published in a production environment.

To avoid making things too boring, we'll try to pick enjoyable themes and scenarios that will also have some usefulness in the real world: to better understand what we mean – no spoilers here – you'll just have to keep reading.

# Preparing the workspace

The first thing we have to do is set up our workstation; it won't be difficult because we only need a small set of essential tools. These include Visual Studio 2019, an updated Node.js runtime, a development web server (such as the built-in IIS Express), and a decent source code control system such as Git, Mercurial, or Team Foundation. We will take the latter for granted as we most likely already have it up and running.


In the unlikely case you don't, you should really make amends before moving on! Stop reading, go to `www.github.com`, `www.bitbucket.com` or whichever online SCM service you like the most, create a free account, and spend some time learning how to effectively use these tools; you won't regret it, that's for sure.

During the next sections, we'll set up the web application project, install or upgrade the packages and libraries, and build and eventually test the result of our work. However, before doing that, we're going to spend a couple of minutes in order to understand a very important concept that is required to properly use this book without getting (emotionally) hurt – at least in my opinion.

# Disclaimer – do (not) try this at home

There's something very important that we need to understand before proceeding. If you're a seasoned web developer, you will most likely know about it already; however, since this book is for (almost) everyone, I feel like it's very important to deal with this matter as soon as possible.

This book will make extensive use of a number of different programming tools, external Components, third-party libraries, and so on. Most of them, such as TypeScript, NPM, NuGet, most .NET Core frameworks/packages/runtimes, and so on, are shipped together with Visual Studio 2019, while others, such as Angular, its required JS dependencies and other third-party server-side and client-side packages will be fetched from their official repositories. These things are meant to work together in a 100% compatible fashion; however, they are all subject to changes and updates during the inevitable course of time. As time passes by, the chance that these updates might affect the way they interact with each other and the project's health will decrease.

## The broken code myth

In an attempt to minimize the chances that this can occur, this book will always work with fixed versions/builds of any third-party Component that can be handled using the configuration files. However, some of them, such as Visual Studio and/or .NET Framework updates, might be out of that scope and might bring havoc to the project. The source code might cease to work, or Visual Studio could suddenly be unable to properly compile it.

When something like that happens, the less-experienced person will always be tempted to put the blame on the book itself. Some of them may even start thinking something like this:

> *There are a lot of compile errors, hence the source code must be broken!*

Alternatively, they may think like this:

> *The code sample doesn't work: the author must have rushed things here and there, and forgot to test what he was writing.*

It goes without saying that such hypotheses are rarely true, especially considering the amount of time that the authors, editors, and technical reviewers of these books spend in writing, testing, and refining the source code before building it up, making it available on GitHub, and often even publishing working instances of the resulting applications to worldwide public websites.

The GitHub repository for this book can be found here:

`https://github.com/PacktPublishing/ASP.NET-Core-3-and-Angular-9-Third-Edition`

It contains a Visual Studio solution file for each chapter (`Chapter_01.sln`, `Chapter_02.sln` and so on), as well as an additional solution file (`All_Chapters.sln`) containing the source code for all the chapters.

Any experienced developer will easily understand that most of these things couldn't even be done if there was some broken code somewhere; there's no way this book can even attempt to hit the shelves unless it comes with a 100% working source code, except for a few possible minor typos that will quickly be reported to the publisher and thus fixed within the GitHub repository in a short while. In the unlikely case that it looks like it doesn't, such as raising unexpected compile errors, the novice developer should spend a reasonable amount of time trying to understand the root cause.

Here's a list of questions they should try to answer before anything else:

- Am I using the same development framework, third-party libraries, versions, and builds adopted by the book?
- If I updated something because I felt like I needed to, am I aware of the changes that might affect the source code? Did I read the relevant changelogs? Have I spent a reasonable amount of time looking around for breaking changes and/or known issues that could have had an impact on the source code?
- Is the book's GitHub repository also affected by this issue? Did I try to compare it with my own code, possibly replacing mine?

If the answer to any of these questions is *No*, then there's a good chance that the problem is not ascribable to this book.

## Stay hungry, stay foolish, yet be responsible as well

Don't get me wrong: whenever you want to use a newer version of Visual Studio, update your Typescript compiler or upgrade any third-party library, which you are encouraged to do. This is nothing less than the main scope of this book – making you fully aware of what you're doing and capable of, way beyond the given code samples.

However, if you feel you're ready to do that, you will also have to adapt the code accordingly; most of the time, we're talking about trivial stuff, especially these days when you can Google the issue and/or get the solution on StackOverflow. They changed the typings? Then you need to load the new typings. They moved the class somewhere else? Then you need to find the new namespace and change it accordingly, and so on.

That's about it – nothing more, nothing less. The code reflects the passage of time; the developer just needs to keep up with the flow, performing minimum changes to it when required. You can't possibly get lost and blame someone other than yourself if you update your environment and fail to acknowledge that you have to change a bunch of code lines to make it work again.

Am I implying that the author is not responsible for the source code of this book? It's the exact opposite; the author is always responsible. They're supposed to do their best to fix all the reported compatibility issues while keeping the GitHub repository updated. However, you should also have your own level of responsibility; more specifically, you should understand how things work for *any* development book and the inevitable impact of the passage of time on any given source code. No matter how hard the author can work to maintain it, the patches will never be fast or comprehensive enough to make these lines of code always work on any given scenario. That's why the most important thing you need to understand – even before the book topics – is the most valuable concept in modern software development: being able to efficiently deal with the inevitable changes that *will* always occur.

Whoever refuses to understand that is doomed; there's no way around it.

# Setting up the project

Assuming that we have already installed Visual Studio 2019 and Node.js, here's what we need to do:

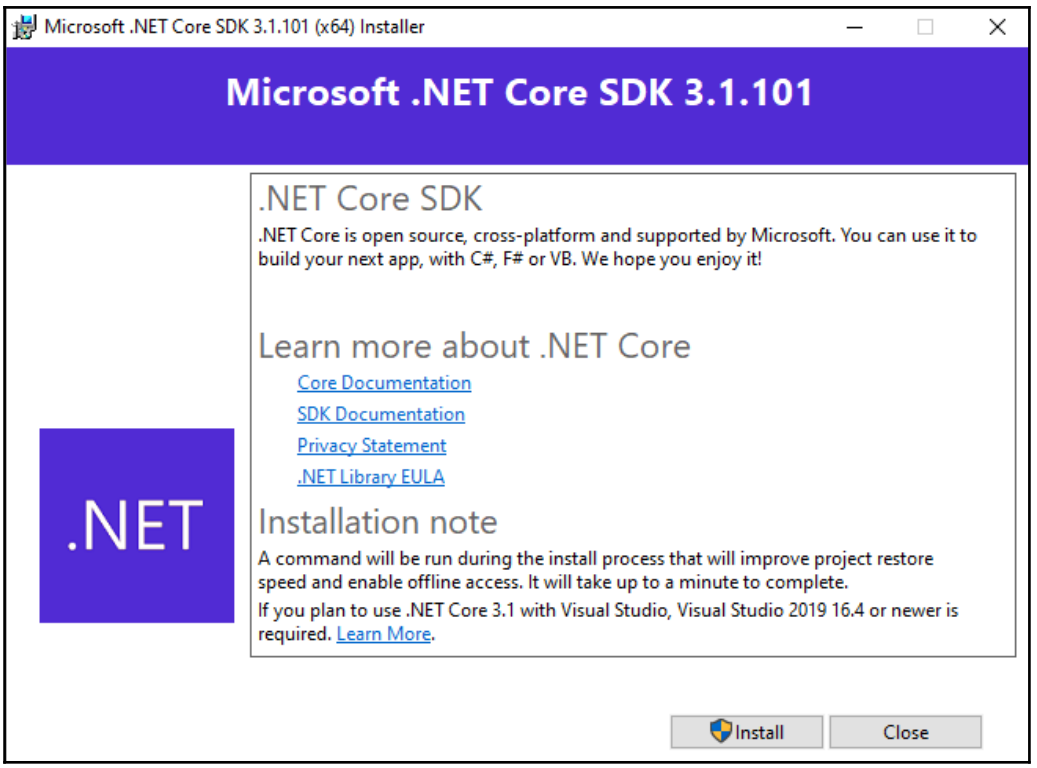
1. Download and install the .NET Core SDK
2. Check that the .NET CLI will use that SDK version
3. Create a new .NET Core and Angular project
4. Check out the newly created project within Visual Studio
5. Update all the packages and libraries to our chosen versions

Let's get to work.

## Installing the .NET Core SDK

We can download the latest version from either the official Microsoft URL (`https://dotnet.microsoft.com/download/dotnet-core`) or from the GitHub official release page (`https://github.com/dotnet/core/blob/master/release-notes/README.md`).

The installation is very straightforward – just follow the wizard until the end to get the job done, as follows:




The whole installation process shouldn't take more than a couple of minutes.

## Checking the SDK version

Once the .NET Core SDK has been installed, we need to confirm that the new SDK PATH has been properly set and/or that the .NET CLI will actually use it. The fastest way to check that is opening a Command Prompt and typing the following:

```
> dotnet --help
```

Be sure that the .NET CLI executes without issue and that the given version number is the same as we installed a moment ago.

If the prompt is unable to execute the command, go to **Control Panel** | **System** | **Advanced System Settings** | **Environment Variables** and check that the `C:\Program Files\dotnet\` folder is present within the `PATH` environment variable; manually add it if needed.

# Creating the .NET Core and Angular project

The next thing we have to do is create our first .NET Core plus Angular project – in other words, our first app. We'll do that using the Angular project template shipped with the .NET Core SDK as it provides a convenient starting point by adding all the required files and also a general-purpose configuration that we'll be able to customize later on to better suit our needs.

From the command line, create a root folder that will contain all our projects and get inside it.

In this book, we're going to use `\Projects\` as our root folder: non-experienced developers are strongly advised to use the same folder to avoid possible path errors and/or issues related to path names being too long (Windows 10 has a 260-character limit that can create some issues with some deeply nested NPM packages). It would also be wise to use something other than the `C:` drive to avoid permission issues.

Once there, type the following command to create the Angular app:

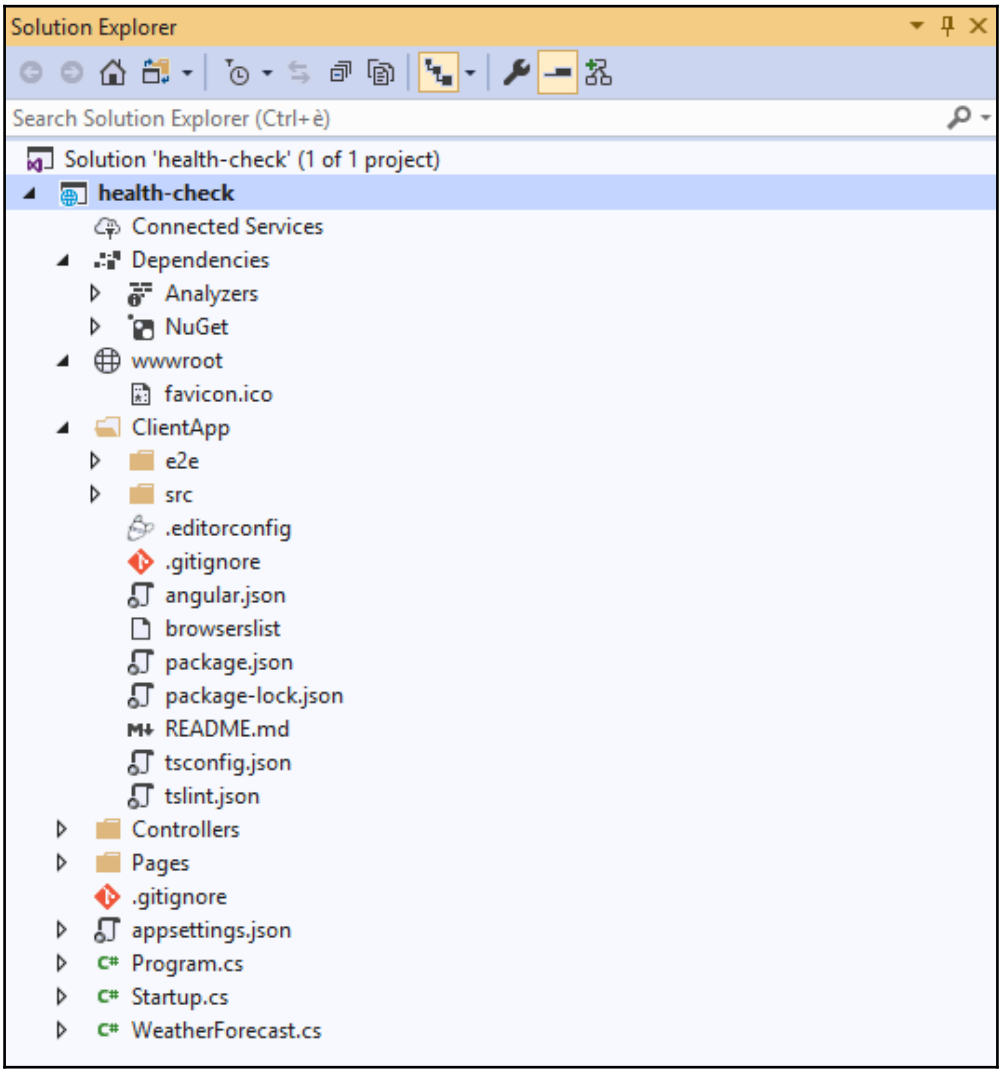
```
> dotnet new angular -o HealthCheck
```

This command will create our first Angular app in the `C:\Projects\HealthCheck\` folder. As we can easily guess, its name will be `HealthCheck`: there's a good reason for such a name, as we're going to see in a short while (no spoilers, remember?).

# Opening the new project in Visual Studio

It's now time to launch Visual Studio 2019 and perform a quick checkup of our newly created project. This can be done by either double-clicking on the `HealthCheck.csproj` file or through the VS2019 main menu (**File** | **Open** | **Project/Solution**).

Once done, we should be able to see our project's source tree in all its lightweight glory, as shown in the following screenshot:




As we can see from the previous screenshot, it's a rather compact boilerplate that only contains the required .NET Core and Angular configuration files, resources, and dependencies: just what we need to start coding!

However, before doing that, let's continue our brief review. As we can see by looking at the various folders, the working environment contains the following:

- The default ASP.NET MVC `/Controllers/` and `/Pages/` folders, both containing some working samples.
- The `/ClientApp/src/` folder with some TypeScript files containing the source code of a sample Angular app.
- The `/ClientApp/e2e/` folder containing some sample E2E tests built with the Protractor testing framework.
- The `/wwwroot/` folder, which will be used by Visual Studio to build an optimized version of the client-side code whenever we need to execute it locally or have it published elsewhere. That folder is initially empty, but it will be populated upon the project's first run.

If we spend some time browsing through these folders and taking a look at their content, we will see how the .NET Core developers did a tremendous job in easing the .NET with the Angular project setup process. If we compare this boilerplate with the built-in Angular 2.x/5.x templates shipped with Visual Studio 2015/2017, we will see huge improvement in terms of readability and code cleanliness, as well as a better file and folder structure. Also, those who fought with task runners such as Grunt or Gulp and/or client-side building tools such as webpack in the recent past will most likely appreciate the fact that this template is nothing like that: all the packaging, building, and compiling tasks are entirely handled by Visual Studio via the underlying .NET Core and Angular CLIs, with specific loading strategies for development and production.

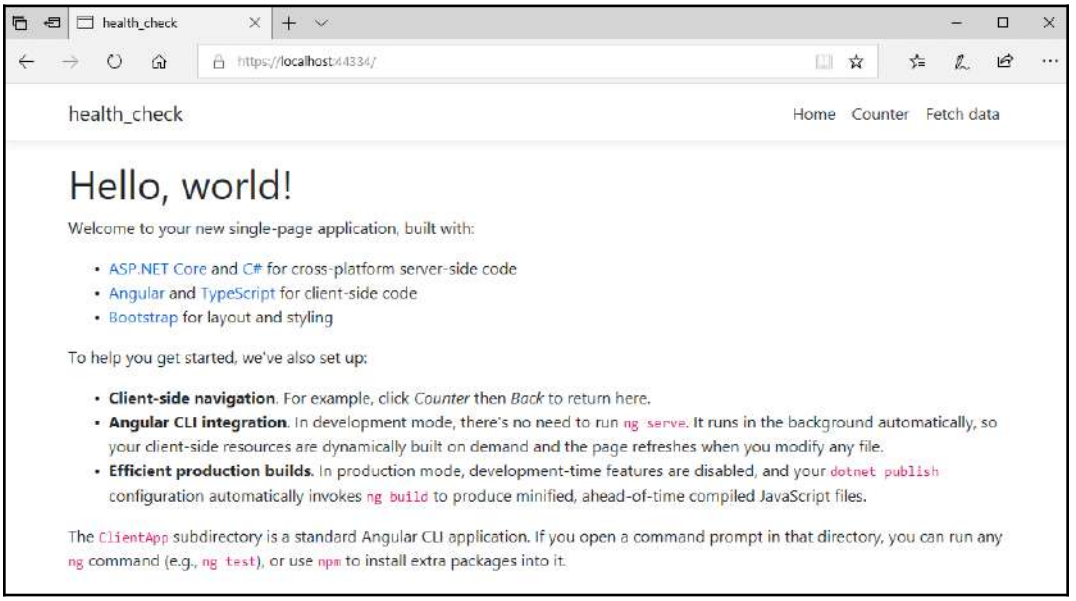
Truth be told, the choice to use a pre-made template such as this one comes with its flaws. The fact that the *back-end* (the .NET Core APIs) and the *front-end* (the Angular app) are both hosted within a single project can be very useful, and will greatly ease up the learning and development phase, but it's not a recommended approach for production.

Ideally, it would be better to split the server-side and the client-side parts into two separate projects to enforce decoupling, which is paramount when building microservice-based architectures. That said, being able to work with the *back-end* and the *front-end* within the same project is a good approach for learning, thus making these templates an ideal approach for the purpose of a programming book – and that's why we're going to always use them.

Before moving on, we should definitely perform a quick test run to ensure that our project is working properly. This is what the next section is all about.

## Performing a test run

Luckily enough, performing a test run at this point is just as easy as hitting the **Run** button or the *F5* key:




This is an excellent consistency check to ensure that our development system is properly configured. If we see the sample Angular SPA up and running, as shown in the preceding screenshot, it means that we're good to go; if we don't, it probably means that we're either missing something or that we've got some conflicting software preventing Visual Studio and/or the underlying .NET Core and Angular CLIs from properly compiling the project.

In order to fix that, we can try to do the following:

- Uninstall/reinstall Node.js, as we can possibly have an outdated version installed.
- Uninstall/reinstall Visual Studio 2019, as our current installation might be broken or corrupted. The.NET Core SDK should come shipped with it already; however, we can try reinstalling it as well.

If everything still fails, we can try to install VS2019 and the previously mentioned packages in a clean environment (be it either a physical system or a VM) to overcome any possible issue related to our current operating system configuration.


If none of these work, the best thing we can do is to ask for specific support on the .NET Core community forum at `https://forums.asp.net/1255.aspx/1?ASP+NET+Core`.

If we manage to successfully perform the test run, it means that the sample app is working: we're ready to move on.

# Summary

So far, so good; we've just set up a working skeleton of what's about to come. Before moving on, let's do a quick recap of what we just did (and arguably learned) in this chapter.

We briefly described our platforms of choice – ASP.NET Core and Angular – and acknowledged their combined potential in the process of building a modern web application. We spent some valuable time recollecting what's happened in these last 3 years, and summarizing the efforts of both development teams to reboot and improve their respective frameworks. These recaps were very useful to enumerate and understand the main reasons why we're still using them over their ever-growing competitors.

Right after that, we did our best to understand the differences between the various approaches that can be adapted to create web apps nowadays: SPAs, MPAs, and PWAs. We also explained that, since we'll be using .NET Core and Angular, we'll stick to the SPA approach, but we'll also implement most PWA features such as service workers and web manifest files. In an attempt to reproduce a realistic production-case scenario, we also went through the most common SPA features, first from a technical point of view, and then putting ourselves in the shoes of a typical product owner while trying to enumerate their expectations.

Last, but not least, we learned how to properly set up our development environment; we chose to do that using the latest Angular SPA template shipped with the .NET Core SDK, thus adopting the standard ASP.NET Core approach. We created our app using the .NET Core CLI and then tested it on Visual Studio to ensure it was working properly.

In the next chapter, we'll take an extensive look at the sample app we just created in order to properly understand how the .NET Core *back-end* and the Angular *front-end* perform their respective tasks and what they can do together.

# Suggested topics

Agile development, Scrum, Extreme Programming, MVC and MVVM architectural patterns, ASP.NET Core, .NET Core, Roslyn, CoreCLR, RyuJIT, Single-Page Application (SPA), Progressive Web Application (PWA), Native Web Application (NWA), Multi-Page Application (MPA), NuGet, NPM, ECMAScript 6, JavaScript, TypeScript, webpack, SystemJS, RxJS, Cache-Control, HTTP Headers, .NET middleware, Angular Universal, server-side rendering (SSR), Ahead-of-Time (AOT) compiler, service workers, web manifest files.

# References


- *Native Web Apps, Henrik Joreteg, 2015*: `https://blog.andyet.com/2015/01/22/native-web-apps/`
- *Manifesto for Agile Software Development, Kent Beck, Mike Beedle, and many others, 2001*: `https://agilemanifesto.org/`
- *ASP.NET 5 is dead – Introducing ASP.NET Core 1.0 and .NET Core 1.0*: `http://www.hanselman.com/blog/ASPNET5IsDeadIntroducingASPNETCore10AndNETCore10.aspx`

- *An Update on ASP.NET Core and .NET Core*: `https://blogs.msdn.microsoft.com/webdev/2016/02/01/an-update-on-asp-net-core-and-net-core/`
- *ASP.NET Core 1.1.0 release notes*: `https://github.com/aspnet/AspNetCore/releases/1.1.0`
- *ASP.NET Core 1.1.0 Commits list*: `https://github.com/dotnet/core/blob/master/release-notes/1.1/1.1-commits.md`
- *ASP.NET Core 2.1.0 release notes*: `https://docs.microsoft.com/en-US/aspnet/core/release-notes/aspnetcore-2.1`
- *ASP.NET Core 2.1.0 Commits list*: `https://github.com/dotnet/core/blob/master/release-notes/2.1/2.1.0-commit.md`
- *ASP.NET Core 2.2.0 release notes*: `https://docs.microsoft.com/en-US/aspnet/core/release-notes/aspnetcore-2.2`
- *ASP.NET Core 2.2.0 Commits list*: `https://github.com/dotnet/core/blob/master/release-notes/2.2/2.2.0/2.2.0-commits.md`
- *ASP.NET Core 3.0.0 release notes*: `https://docs.microsoft.com/en-us/dotnet/core/whats-new/dotnet-core-3-0`
- *ASP.NET Core 3.0 releases page*: `https://github.com/dotnet/core/tree/master/release-notes/3.0`
- *ASP.NET Core 3.1.0 release notes*: `https://docs.microsoft.com/en-us/dotnet/core/whats-new/dotnet-core-3-1`
- *Libscore: JavaScript library usage stats*: `http://libscore.com/#libs`
- *Usage of JavaScript libraries for websites*: `https://w3techs.com/technologies/overview/javascript_library/all`
- *Miško Hevery and Brad Green - Keynote - NG-Conf 2014:* `https://www.youtube.com/watch?v=r1A1VR0ibIQ`
- *AngularJS 1.7.9 Changelog*: `https://github.com/angular/angular.js/blob/master/CHANGELOG.md`
- *ASP.NET Core and Angular 2*: `https://www.packtpub.com/application-development/aspnet-core-and-angular-2`
- *ASP.NET Core 2 and Angular 5*: `https://www.packtpub.com/application-development/aspnet-core-2-and-angular-5`
- *ASP.NET Core 2 and Angular 5 - Video Course*: `https://www.packtpub.com/web-development/asp-net-core-2-and-angular-5-video`
- *Angular Update Guide*: `https://update.angular.io`
- *Angular Language Service*: `https://angular.io/guide/language-service`