

# ASP.NET Core and Vue.js

---

Build real-world, scalable, full-stack applications  
using Vue.js 3, TypeScript, .NET 5, and Azure

Devlin Basilan Duldulao



BIRMINGHAM—MUMBAI

# ASP.NET Core and Vue.js

Copyright © 2021 Packt Publishing

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

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

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

**Group Product Manager:** Pavan Ramchandani

**Publishing Product Manager:** Ashitosh Gupta

**Senior Editor:** Hayden Edwards

**Content Development Editor:** Abhishek Jadhav

**Technical Editor:** Joseph Aloocaran

**Copy Editor:** Safis Editing

**Project Coordinator:** Manthan Patel

**Proofreader:** Safis Editing

**Indexer:** Pratik Shirodkar

**Production Designer:** Roshan Kawale

First published: June 2021

Production reference: 1140621

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80020-669-4

[www.packt.com](http://www.packt.com)

*I'd like to thank all the people who have contributed to my achievements in life.*



*To my loving wife, Ruby Jane, for supporting me to achieve my career goals, starting from career shifting to being a full-stack web/mobile engineer and writing a technical book. This wouldn't be possible without you.*

*To my mother, Lucy Basilan, and my father, Alberto Duldulao, for constantly reminding me that I can do anything. Salamat sa inyo (thank you in Filipino).*

*To Inmeta, the company that I'm working for right now, and to my managers, Mohammad "Moa" Yassin and Jon Sandvand, for giving me a chance to share my skills here in Scandanavia. Takk til dere alle (thanks to all of you in Norwegian).*

*To the developer communities in Manila and Oslo, where I usually do talks, presentations, and demos of what's new in web, mobile, and cloud technologies. You guys contributed to my growth.*

*And to the Packt team, Hayden, Divij, Abhishek, Ashwin, Ashitosh, Manthan, Deepesh, and all those involved in this project. Thank you for trusting me to write my first ever book.*

– *Devlin Basilan Duldulao*

## Contributors

### About the author

**Devlin Basilan Duldulao** is a full-stack engineer with over 8 years of web, mobile, and cloud development experience. He has been a recipient of Microsoft's **Most Valuable Professional (MVP)** award since 2018 and earned the title of Auth0 ambassador for his passion for sharing best practices in application securities. Devlin has passed some prestigious exams in software and cloud development such as MSCD, Azure Associate Developer, AWS Associate Developer, and Terraform Associate.

Perhaps it was serendipity that made him venture into the coding world after a short stint in the medical field; however, once he stepped into it, he fell for it hook, line, and sinker – but in the right way, he claims. Devlin often finds himself engrossed in solving coding problems and developing apps, even to the detriment of his once-active social life.

One of the things that motivates him is ensuring the long-term quality of his code, including looking into ways to transform legacy code into more maintainable and scalable applications.

Devlin enjoys tackling challenging projects or applications for high-level clients and customers, as he currently does at his company based in Norway. He also provides training and consultation for international corporations.

One of his other interests is giving talks at IT conferences worldwide and meeting unique people in the industry.

Devlin is currently based in Oslo, Norway, with his wife. He is a senior software engineer at Inmeta Consulting Company, a subsidiary of the Crayon Group of Companies.

### About the reviewer

**Sebastian Nilsson** is a professional problem-solver with a proven track record of improving software, processes, and people at multiple companies, including start-ups, mid-sized companies, and international corporations. His expertise is within Azure, .NET, the web, DevOps, Agile, and leadership.

Going from a freelancing technical expert to walking the complete path from full-stack web developer to the role of CTO, he has gathered extensive knowledge and experience within all the steps of software engineering and product development.

Having a wide range of experiences has given him a unique insight into building great technical cultures and improving creativity, productivity, and work processes, which benefits both the organization and the people in it.



*First off, thank you to Apurv, Manthan, and Packt Publishing for allowing me to be part of the work on the book and for having the patience to let me use my full registry of skills in the technical feedback on this based book.*

*Thank you to my wife and family for supporting me with my late evenings and weekends of reading the book and writing the feedback.*

*All this was made possible by my friend, Rouzbeh Delavari, who nudged me and coached me into an actual career in software development, not just a hobby.*

*Thanks to all my employers throughout my career, who have allowed me to grow my skills, knowledge, and experience. Reviewing this book gave me great insight into how diverse, but also deep, my experiences have actually been.*

## Table of Contents

[ASP.NET Core and Vue.js](#)

[Contributors](#)

[About the author](#)

[About the reviewer](#)

[Preface](#)

[Who this book is for](#)

[What this book covers](#)

[To get the most out of this book](#)

[Download the example code files](#)

[Download the color images](#)

[Conventions used](#)

[Get in touch](#)

[Reviews](#)

## Section 1: Getting Started

## [Chapter 1: Getting Started with ASP.NET Core and Vue.js](#)

[Technical requirements](#)

[Introducing ASP.NET Core](#)

[What's new in .NET?](#)

[What's new in ASP.NET Core?](#)

[Breaking changes in migration from ASP.NET Core 3.1 to ASP.NET Core 5.0](#)

[When to use ASP.NET Core](#)

[Why should you learn ASP.NET Core?](#)

[Introducing Vue.js](#)

[What's new in Vue.js?](#)

[Why is learning Vue.js the right choice?](#)

[Summary](#)



## [Chapter 2: Setting Up a Development Environment](#)

### [Technical requirements](#)

### [Installing and configuring VS Code, Visual Studio 2019, VS for Mac, and Rider](#)

#### [VS Code](#)

#### [Visual Studio 2019](#)

#### [Visual Studio for Mac](#)

#### [Rider](#)

### [Installing .NET SDK, Node.js, and npm](#)

#### [.NET SDK](#)

#### [Node and npm](#)

### [Setting up the .NET Core CLI and Vue CLI](#)

### [Installing Postman and Vue DevTool](#)

### [Installing Entity Framework Core tools](#)

### [Installing a database provider](#)

### [Installing Git version control](#)

### [Summary](#)

## Section 2: Backend Development

## [Chapter 3: Starting Your First ASP.NET Core Project](#)

[Technical requirements](#)

[Creating an ASP.NET Core project](#)

[First command – dotnet new sln](#)

[Second command – dotnet new webapi --name Web](#)

[Third command – dotnet sln add \[csproj-file-location\]](#)

[The last command to run – dotnet run Web.csproj](#)

[Understanding the Program.cs file](#)

[Demystifying the Startup.cs file](#)

[Getting started with Swashbuckle](#)

[Introducing OpenAPI and Swagger](#)

[Built-in Swagger integration](#)

[Swagger documentation and Swagger UI in action](#)

[Summary](#)

## [Chapter 4: Applying Clean Architecture to an ASP.NET Core Solution](#)

[Technical requirements](#)

[Introducing clean architecture](#)

[The core layer – directory](#)

[Domain – project](#)

[Application – project](#)

[Shared Kernel – NuGet project](#)

[The infrastructure layer – directory](#)

[Data – project](#)

[Shared – project](#)

[The presentation layer – directory](#)

[WebApi – project](#)

[client-app – non-project web application](#)

[Managing tests – directories](#)

[Unit test – project](#)

[Integration test – project](#)

[Structuring a clean architecture solution](#)

[Visual Studio 2019](#)

[Visual Studio for Mac](#)

[Rider](#)

[Summary](#)

## [Chapter 5: Setting Up DbContext and Controllers](#)

[Technical requirements](#)

[Writing entities and enums](#)

[Creating entities and enums for the Travel Tour application](#)

[Setting up a database, EF Core, and DbContext](#)

[EF Core](#)

[DbContext](#)

[Setup](#)

[Writing controllers and routes](#)

[TourPackagesController](#)

[TourListsController](#)

[Testing controllers with Swagger UI](#)

[Summary](#)

## Chapter 6: Diving into CQRS

Technical requirements

What is CQRS?

What is the mediator pattern?

What is the MediatR package?

Why learn CQRS?

When to use CQRS

Drawbacks of CQRS

Summary.

## [Chapter 7: CQRS in Action](#)

[Technical requirements](#)

[Implementing CQRS](#)

[Adding the MediatR package](#)

[Creating MediatR pipeline behaviors](#)

[Using FluentValidation](#)

[Using AutoMapper](#)

[Writing queries](#)

[Writing commands](#)

[Writing IServiceCollection](#)

[Summary](#)



## [Chapter 8: API Versioning and Logging in ASP.NET Core](#)

### [Technical requirements](#)

### [API versioning](#)

### [What is API versioning?](#)

### [API versioning strategies](#)

### [Deprecating an API](#)

### [API versioning integration with OpenAPI](#)

### [Logging in ASP.NET Core](#)

### [Logging in ASP.NET Core](#)

### [What is structured logging?](#)

### [Serilog versus NLog](#)

### [Configuring Serilog](#)

### [Summary](#)

## [Chapter 9: Securing ASP.NET Core](#)

[Technical requirements](#)

[Understanding ASP.NET Core Identity](#)

[ASP.NET Core Identity features](#)

[Introducing IdentityServer4](#)

[Customer identity and access management \(CIAM\)](#)

[Authentication implementation using JWT](#)

[Implementing token-based authentication](#)

[Checking Swagger UI](#)

[Summary](#)

## [Chapter 10: Performance Enhancement with Redis](#)

[Technical requirements](#)

[In-memory caching in ASP.NET Core](#)

[Enabling in-memory caching in ASP.NET Core](#)

[Distributed caching](#)

[Setting up and running Redis](#)

[For Windows users](#)

[For macOS users](#)

[For Linux or Ubuntu users](#)

[Implementing Redis in ASP.NET Core](#)

[Code update](#)

[Summary](#)

## Section 3: Frontend Development

## [Chapter 11: Vue.js Fundamentals in a Todo App](#)

[Technical requirements](#)

[Starting a project using the Vue CLI](#)

[Files and folders generated by the Vue CLI](#)

[Getting started with a Vue component](#)

[Writing a Vue component](#)

[Common features in a Vue component](#)

[Writing local states in a Vue component](#)

[Adding a function in a Vue component](#)

[Looping in an array in a Vue component](#)

[If-else conditions in a Vue component](#)

[Creating and passing props](#)

[Life cycle hooks in a Vue component](#)

[Summary](#)

[Further reading](#)

## [Chapter 12: Using a UI Component Library and Creating Routes and Navigations](#)

[Technical requirements](#)

[Using a third-party UI component library](#)

[Setting up a Vue.js project and installing a UI component library](#)

[Other third-party UI libraries](#)

[Adding navigation bars](#)

[Writing page components](#)

[Setting up Vue Router with lazy loading and eager loading](#)

[Summary](#)

## [Chapter 13: Integrating a Vue.js Application with ASP.NET Core](#)

### [Technical requirements](#)

### [Putting ASP.NET Core Web API and a Vue.js app together as a single unit](#)

### [Introducing Cross-Origin Resource Sharing or CORS](#)

### [Enabling a CORS policy in ASP.NET Core](#)

### [Summary](#)



## Chapter 14: Simplifying State Management with Vuex and Sending GET HTTP Requests

Technical requirements

Understanding complex state management

Understanding global state

Sending an HTTP request in Vue.js

Setting up state management using Vuex

Step 1 – Writing a store

Step 2 – Writing a module

Step 3 – Writing a module if we are using TypeScript

Step 4 – Writing an API service

Step 5 – Writing an action type

Step 6 – Writing an action

Step 7 – Writing a state

Step 8 – Writing a mutation

Step 9 – Writing a getter

Step 10 – Updating the store by inserting the module

Step 11 – Updating components with mapGetters and mapActions

Summary

## [Chapter 15: Sending POST, DELETE, and PUT HTTP Requests in Vue.js with Vuex](#)

### [Technical requirements](#)

### [Removing a tour list using Axios and Vuex](#)

### [Adding a tour list using Axios and Vuex](#)

### [Using a non-async action in Vuex](#)

### [Removing a tour package using Axios and Vuex](#)

### [Adding a tour package using Axios and Vuex](#)

### [Updating a tour package using Axios and Vuex](#)

### [Summary](#)

## [Chapter 16: Adding Authentication in Vue.js](#)

[Technical requirements](#)

[Setting up Vuex for authentication](#)

[Writing an auth guard](#)

[HTTP interceptor](#)

[Auto login](#)

[Summary](#)

## Section 4: Testing and Deployment

## *Chapter 17: Input Validations in Forms*

Technical requirements

Installing an input validation library

Using validators in forms

Summary

## [Chapter 18: Writing Integration Tests Using xUnit](#)

[Technical requirements](#)

[Getting started with automated testing](#)

[Benefits of automated testing](#)

[Installing MS SQL Server in a Docker container](#)

[Understanding xUnit](#)

[Features of xUnit](#)

[Using xUnit in ASP.NET Core](#)

[Understanding unit testing](#)

[Writing unit tests](#)

[Understanding integration testing](#)

[Writing integration tests](#)

[Summary](#)

## [Chapter 19: Automatic Deployment Using GitHub Actions and Azure](#)

[Technical requirements](#)

[Introducing GitHub Actions – a CI/CD tool](#)

[Understanding GitHub Actions](#)

[GitHub Actions for .NET apps](#)

[Understanding where to deploy](#)

[When to deploy to Azure App Service?](#)

[When to deploy to Azure Functions?](#)

[When to deploy to Azure Static Web Apps?](#)

[When to deploy to Azure Kubernetes Service?](#)

[Automated deployment to Azure App Service using GitHub Actions](#)

[Syntax of the workflow file](#)

[Creating an Azure App Service instance in the Azure portal](#)

[Summary](#)

[Why subscribe?](#)

[Other Books You May Enjoy](#)

[Packt is searching for authors like you](#)

[Leave a review - let other readers know what you think](#)

# Preface

Vue.js 3 is faster and smaller than the previous version, and TypeScript's full support out of the box makes it a more maintainable and easier-to-use version of Vue.js. Then, there's ASP.NET Core 5, which is the fastest .NET web framework today. Together, Vue.js for the frontend and ASP.NET Core 5 for the backend make a powerful combination. This book follows a hands-on approach to implementing practical methodologies for building robust applications using ASP.NET Core 5 and Vue.js 3. The topics here are not in depth, and the book is intended for busy .NET developers who have limited time and want a quick implementation of a clean architecture with popular libraries.

You'll start by setting up your web app's backend, guided by clean architecture, **Command Query Responsibility Segregation (CQRS)**, mediator patterns, and Entity Framework Core 5. The book then shows you how to build the frontend application using best practices, state management with Vuex, Vuetify UI component libraries, Vuelidate for input validations, lazy loading with Vue Router, and JWT authentication. Later, you'll focus on testing and deployment, performing tasks such as load testing in ASP.NET Core 5 and deploying containerized apps to the cloud. All the tutorials in this book support Windows 10, macOS, and Linux users.

By the end of this book, you'll be able to build an enterprise full-stack web app, use the most common npm packages for Vue.js and NuGet packages for ASP.NET Core, and deploy Vue.js and ASP.NET Core to Azure App Service using GitHub Actions.

## Who this book is for

This book is for busy .NET developers who want to get started with Vue.js and build full-stack real-world enterprise web applications. Developers looking to build a proof-of-concept application quickly and pragmatically



using their existing knowledge of ASP.NET Core, as well as developers who want to write readable and maintainable code using TypeScript and the C# programming language will also find this book useful. The book assumes intermediate-level .NET knowledge, along with an understanding of C# programming, JavaScript, and ECMAScript.

## What this book covers

[Chapter 1](#), *Getting Started with ASP.NET Core and Vue.js*, serves as a short recap regarding the current state of ASP.NET Core and Vue.js to give you a glimpse of what lies ahead in the web development of ASP.NET Core and Vue.js.

[Chapter 2](#), *Setting Up a Development Environment*, will teach you how to set up your computer's development environment to build backend and frontend web applications. You will go through different IDEs and text editors to write code and make sure everything has been set up before proceeding with the app development.

[Chapter 3](#), *Starting Your First ASP.NET Core Project*, shows the step-by-step process of creating an ASP.NET Core 5 Web API project. This chapter also describes the default folders and files in a newly created ASP.NET Core 5 Web API, particularly **Program.cs** and **Start.cs**, including the dependency services and middleware.

[Chapter 4](#), *Applying Clean Architecture to an ASP.NET Core Solution*, teaches you the real-world organization of files, folders, projects, and ASP.NET Core app dependencies, preparing you for future big and scalable ASP.NET Core 5 enterprise applications.

[Chapter 5](#), *Setting Up DbContext and Controllers*, will teach you how to set up a database, Entity Framework Core, DbContext, and how to write entities and enums in a clean architectural way. This chapter also teaches you how to write controllers and routes with Swagger UI to test controllers.

[Chapter 6](#), *Diving into CQRS*, is all about the CQRS pattern, the mediator pattern, and the popular MediatR NuGet package for CQRS and pipeline behavior.

[Chapter 7](#), *CQRS in Action*, shows you how to implement CQRS, use FluentValidation and AutoMapper, and write queries, commands, and **IServiceCollection**.

[Chapter 8](#), *API Versioning and Logging in ASP.NET Core*, teaches you about API versioning, which is sometimes necessary to create maintainable APIs but can be problematic if not done correctly.

[Chapter 9](#), *Securing ASP.NET Core*, discusses the integration of the ASP.NET Core 5 backend with the Vue.js 3 frontend. The chapter explores authentication and authorization in the ASP.NET Core 5 Web API by creating and handling JWT. This chapter then explains how to use JWT builder, writing custom JWT middleware, developing basic authentication, and adding role-based authorizations on GET, POST, PUT, and DELETE methods.

[Chapter 10](#), *Performance Enhancement with Redis*, covers in-memory caching in ASP.NET Core, distributed caching, and implementing Redis.

[Chapter 11](#), *Vue.js Fundamentals in a Todo App*, is entirely devoted to Vue.js, the **Node Package Manager (npm)**, and the Vue CLI. These tools help developers to scaffold Vue.js projects with different configurations based on the user's options. This chapter also describes the Vue component's features and what you can do with them.

[Chapter 12](#), *Using a UI Component Library and Creating Routes and Navigations*, teaches you how to use open source UI libraries built by different Vue.js communities. You will use one of the popular libraries in Vue.js, which will save you from spending countless hours building your components. Then you will set up the navigation and routing of your Vue.js 3 app with best practices in mind.

[Chapter 13](#), *Integrating a Vue.js Application with ASP.NET Core*, explains how to put the ASP.NET Core Web API and the Vue.js application together

as a single unit. You will look at how the CORS policy works and how to enable it.

[Chapter 14](#), *Simplifying State Management with Vuex and Sending GET HTTP Requests*, is about sending HTTP requests and solving the most common problem in big web applications—the problem of syncing the state of a component with another component. In large and complex applications, you need a tool that centralizes your application's state and makes the data flow transparent and predictable.

[Chapter 15](#), *Sending POST, DELETE, and PUT HTTP Requests in Vue.js with Vuex*, shows the step-by-step process of synchronizing fetching, removing, creating, and updating data in the frontend and backend. This chapter explains the effective state management of a Vue.js 3 application in the easiest way possible.

[Chapter 16](#), *Adding Authentication in Vue.js*, explains the setting up of Vuex for authentication and writing an Auth Guard. This chapter also covers writing HTTP interceptors and setting up auto-login in the app.

[Chapter 17](#), *Input Validations in Forms*, discusses the installation of an input validation library called Vuelidate and explains how to use validators in forms to prevent users from typing invalid inputs.

[Chapter 18](#), *Writing Integration Tests Using xUnit*, explores ways to efficiently test ASP.NET Core 5 and Vue.js applications. This chapter serves as a guide for detecting bugs in applications before users use them.

[Chapter 19](#), *Automatic Deployment Using GitHub Actions and Azure*, explains what GitHub Actions is, where to deploy apps, and how to implement automated deployment to Azure App Service using GitHub Actions.

## To get the most out of this book

You can install Node.js, VS Code, Vue CLI, and the .NET 5 SDK or wait for further instructions in the relevant chapters.

Please take note of the required OS versions of your machine and the required versions of the software.

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

## Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js>. 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/9781800206694\\_ColorImages.pdf](https://static.packt-cdn.com/downloads/9781800206694_ColorImages.pdf).

## Conventions used

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

**Code in text:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded **WebStorm-10\*.dmg** disk image file as another disk in your system."

A block of code is set as follows:

```
html, body, #map {  
  
height: 100%;  
  
margin: 0;  
  
padding: 0  
  
}
```

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

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

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

```
$ mkdir css  
  
$ cd css
```

**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: "Select **System info** from the **Administration** panel."

Tips or important notes

Appear like this.

## Get in touch

Feedback from our readers is always welcome.

**General feedback:** If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at [customercare@packtpub.com](mailto: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](http://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](mailto: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](http://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](http://packt.com).

# Section 1: Getting Started

This section deals with a real-world scenario: how to start a web application project. The following chapters are included in this section:

- [Chapter 1](#), *Getting Started with ASP.NET Core and Vue.js*
- [Chapter 2](#), *Setting Up a Development Environment*



# Chapter 1: Getting Started with ASP.NET Core and Vue.js

First of all, I would like to thank you for getting a copy of this book. This book is designed to teach *busy developers* how to build a real-world full-stack web application, from development to deployment. The book is tailored based on the step-by-step process I have developed throughout the years from my workshops. So, let's start the journey.

This chapter serves as a short recap regarding the current state of ASP.NET Core and Vue.js to give you a glimpse of what lies ahead in the web development of ASP.NET Core and Vue.js. You will also see how stable and reliable Vue.js is as an app and learn about the team behind writing and maintaining the Vue.js framework.

In this chapter, we will cover the following topics:

- Introducing ASP.NET Core
- What's new in .NET?
- What's new in ASP.NET Core?
- Introducing Vue.js

## Technical requirements

You will find the repository for the application we will build at this URL:

<https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js>.

Every chapter has its directory, and each directory has one folder named **start** and one folder named **finish**.

The **start** folders are the state of the repository before any code was written to it. The **finish** folders are the state of the repository at the end of every chapter.

## Introducing ASP.NET Core

**ASP.NET Core** is an open source web app framework from Microsoft built to be fast, performant, and to work across platforms such as Windows, macOS, and Linux, for building modern cloud services and internet-connected apps. You can use the cross-platform VS Code to build your applications without installing virtualization software such as Parallels or VMware. You only need to install another instance of VS Code on another operating system, **git clone** your repository, install .NET Core SDK, and continue writing code.

The benefits that developers can get from smaller application surface areas brought by ASP.NET Core's framework structure are tighter security, improved performance, and reduced serving.

However, before we discuss what's new in ASP.NET Core 5, we must first know what .NET 5 is.

## What's new in .NET?

**.NET** is an open source development platform created by Microsoft for building many different types of applications.

Microsoft now uses a single framework that unifies all .NET platforms, from developing for web apps, mobile, and the cloud, to desktop. .NET 5 includes both Xamarin and its web assembly platform, and to make it better, Microsoft was also able to move the support for **Windows Presentation Foundation (WPF)** and Windows Forms to the framework.

Look at *Figure 1.1*, which shows that the new .NET 5 platform provides a common set of APIs supporting the different runtime implementations:

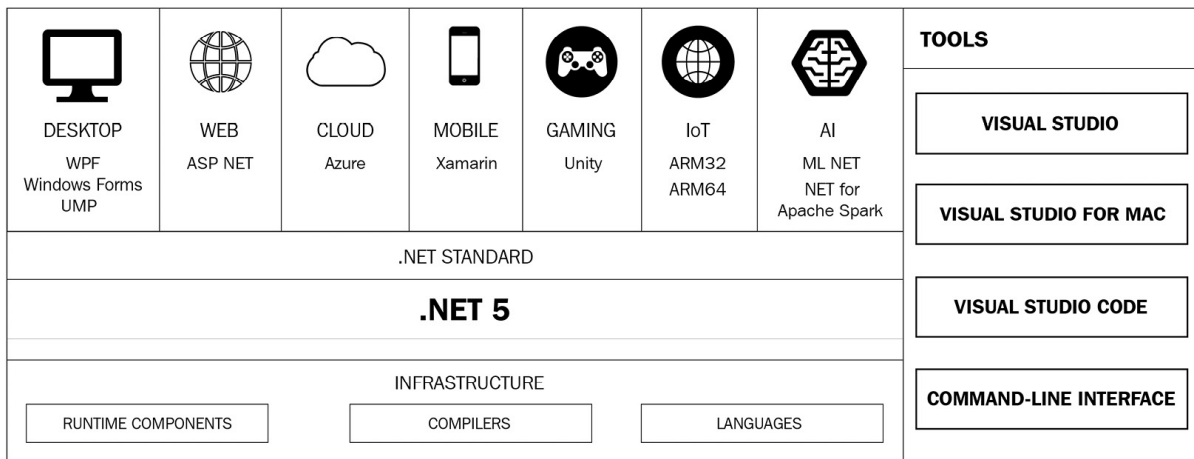


Figure 1.1 – .NET: A unified platform

You can use the same APIs of .NET 5 and target different OSes, application types, and chip architectures. Plus, you will be able to configure or edit your build configuration using your favorite **Integrated Development Environment (IDE)** and text editors—you can use popular IDEs such as Visual Studio, Visual Studio for Mac, or Rider, or text editors such as Visual Studio Code or the plain old command line to build your application.

The highlights of .NET 5 are as follows:

- It includes the new C# 9 and F# 5.
- A new single-file publish type that executes your app out of a single binary.
- Runs .NET natively on Windows ARM64.

- Improves ARM64 performance (Linux and Windows) in the JIT and BCL libraries.
- Reduces the container image size and implements new container APIs to enable .NET to stay up to date with container runtime evolution.
- It enables easier migration from **Newtonsoft.Json** to **System.Text.Json**.

Now we can take a look at what's new in ASP.NET Core 5.

## What's new in ASP.NET Core?

Here is a rough list of what has been added to the new ASP.NET Core web framework:

- **Performance Improvements to HTTP/2:** .NET 5 improves the performance of HTTP/2 by adding support for HPack dynamic compression of HTTP/2 response headers in Kestrel.
- **Reduction in container image sizes:** Sharing layers between two images dramatically reduces the size of the aggregate images that you pull. This reduction is achieved by re-platting the SDK image on the ASP.NET runtime image.
- **Reloadable endpoints via configuration for Kestrel:** Kestrel can now observe changes to configurations passed to **KestrelServerOptions.Configure**. Then it can be applied to any new endpoints without restarting your application.
- **JSON extension methods for HttpRequest and HttpResponse:** Using the new **ReadFromJsonAsync** and **WriteAsJsonAsync** extension methods, you can now easily consume and use JSON data from **HttpRequest** and **HttpResponse**. The JSON extension methods can also be written with an endpoint routing to create JSON APIs like so:

```
endpoints.MapGet("/weather/{city:alpha}",
    async context =>
{
    var city = (string)context.Request
        .RouteValues["city"];

    var weather = GetFromDatabase(city);

    await context.Response.WriteAsJsonAsync(weather);
});
```

- **An extension method allows anonymous access to an endpoint:** The **AllowAnonymous** extension allows anonymous access to an endpoint when using endpoint routing. In the following code, the extension method, **AllowAnonymous()**, is chained after calling the **MapGet** method:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();

    app.UseAuthentication();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
{

```

```

endpoints.MapGet("/", async context =>
{
    await context.Response
        .WriteAsync("Hello Packt!");
    }).AllowAnonymous();
});
}

```

- **Custom handling of authorization failures:** With the new **IAuthorizationMiddlewareResultHandler** interface invoked by **AuthorizationMiddleware**, custom handling of authorization failures is now easier than before. You can now register a custom handler in the dependency injection container that allows developers to customize HTTP responses.
- **SignalR Hub filters:** Similar to how middleware lets you run code before and after an HTTP request, Hub Pipelines in ASP.NET SignalR is the feature that allows you to run code before and after Hub methods are called.
- **Updated debugging for Blazor WebAssembly:** No need for a VS Code JS debugger extension for developing Blazor WebAssembly apps.
- **Blazor accessibility improvements:** Input components that derive from **InputBase** now automatically render **aria-invalid** (an HTML validation attribute) on failed validation.
- **Blazor performance improvements:** This includes optimized .NET runtime execution, JSON serialization, JavaScript interop, and component rendering.
- **Kestrel socket transport support for additional endpoint types:** The **System.Net.Sockets** transport in Kestrel now allows you to bind to both Unix domain sockets and existing file handles.
- **Azure Active Directory authentication with Microsoft.Identity.Web:** Any ASP.NET Core project templates can now easily integrate with **Microsoft.Identity.Web** to handle authentication with Azure AD.
- **Sending HTTP/2 PING frames:** Microsoft added the ability to send periodic PING frames in Kestrel by setting limits on **KestrelServerOptions**, which are **Limits.Http2.KeepAlivePingInterval** and **Limits.Http2.KeepAlivePingTimeout**, as shown in the following code:

```

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.ConfigureKestrel(options =>
            {
                options.Limits.Http2.KeepAlivePingInterval = TimeSpan
                    .FromSeconds(10);
                options.Limits.Http2.KeepAlivePingTimeout = TimeSpan
                    .FromSeconds(1);
            });
            webBuilder.UseStartup<Startup>();
        });

```

```
});
```

- **Custom header decoding in Kestrel:** Microsoft also added the ability to specify which **System.Text.Encoding** to use to interpret incoming headers based on the header name instead of defaulting to UTF-8, like so:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.ConfigureKestrel(options =>
            {
                options.RequestHeaderEncodingSelector = encoding =>
                {
                    switch (encoding)
                    {
                        case "Host":
                            return System.Text
                                .Encoding
                                .Latin1;
                        default:
                            return System.Text
                                .Encoding
                                .UTF8;
                    }
                }
            });
        });
    webBuilder.UseStartup<Startup>();
});
```

- **CSS isolation for Blazor components:** Blazor now supports scoped CSS styles inside a component.
- **Lazy loading in Blazor WebAssembly:** Use the **OnNavigateAsync** event on the **Router** component to lazy load assemblies for a specific page.
- **Set UI focus on Blazor apps:** Use the **FocusAsync** method on **ElementReference** to set the UI focus on an element.
- **Control Blazor component instantiation:** **IComponentActivator** can be used to control how Blazor components are instantiated.
- **Influencing the HTML head in Blazor apps:** Add dynamic link and meta tags by using the built-in **Title**, **Link**, and **Meta** components in the **head** tags of a Blazor app.

- **Protected browser storage:** **ProtectedLocalStorage** and **ProtectedSessionStorage** can be used to create a secure persisted app state in local or session storage.
- **Model binding and validation with C#9 record types:** You can use **Record** types to model data transmitted over the network like so:

```
public record Person([Required] string Name,
                    [Range(0, 150)] int Age);

public class PersonController
{
    public IActionResult Index() => View();

    [HttpPost]
    public IActionResult Index(Person person)
    {
        // ...
    }
}
```

You can see the **record** type after the public access modifier.

- **Improvements to DynamicRouteValueTransformer:** You can now pass state to **DynamicRouteValueTransformer** and filter the set of chosen endpoints.
- **Auto-refresh with dotnet watch:** The ASP.NET Core project will now both launch the default browser and auto-refresh it as you make changes to your code while running **dotnet watch**.
- **Console** Logger Formatter: The console logger formatter gives the developer complete control over the formatting and colorization of the console output.
- **JSON** Console Logger: Microsoft added a built-in JSON formatter that emits structured JSON logs to the console.

That was the list of what's new in ASP.NET Core 5. What about breaking changes? Are there any breaking changes in ASP.NET Core 5? Yes, and let's check them out in the next section.

## Breaking changes in migration from ASP.NET Core 3.1 to ASP.NET Core 5.0

If you are planning to migrate your existing app or a project under development in .NET Core 3.1 to Core 5, you might need to take a pause and read the following quick list of breaking changes.

### Authentication

There's a new behavior in integrating Azure and ASP.NET Core to determine a user's identity. The **AzureAD.UI** and **AzureADB2C.UI** APIs and packages are now obsolete in the framework.

**AzureAD.UI** and **AzureADB2C.UI** migrated to the **Microsoft Authentication Library** (or **MSAL**), which is under **Microsoft.Identity.Web**.

## Authorization

There is a little change in the endpoint routing of ASP.NET Core. The resource passed to the authorization endpoint is now guaranteed to be of the type **HttpContext**. The new change will allow developers to use the functionalities of **HttpContext** from non-endpoint routing.

## Azure

Azure prefixes replaced the Microsoft prefixes in integration packages. These packages are as follows:

- **Microsoft.Extensions.Configuration.AzureKeyVault**, which developers use to connect Azure Key Vault to the configuration system.
- **Microsoft.AspNetCore.DataProtection.AzureKeyVault**, which connects Azure Key Vault to the ASP.NET Core data protection system.
- **Microsoft.AspNetCore.DataProtection.AzureStorage**, which lets developers port Azure Blob storage into the ASP.NET Core data protection system.

## Blazor

This new framework for browser-based .NET apps of Microsoft has some recent changes:

- The compiler will trim any whitespaces in components of Blazor during compile time. Trimming of the compiler improves the performance of rendering and DOM diffing, which is comparing the previous version of the virtual DOM to the new version of the virtual DOM.
- The **ProtectedBrowserStorage** feature is now part of the ASP.NET Core shared framework for a better developer experience. The shared frameworks are **Microsoft.NETCore.App**, **Microsoft.AspNetCore.App**, and **Microsoft.AspNetCore.All**.
- .NET 5.0 is the new target framework of Blazor Server and Blazor WebAssembly projects to better align with .NET target framework requirements.

## HTTP

There are some changes in how you would handle bad HTTP request exceptions and log HTTP requests and responses:

- **Microsoft.AspNetCore.Http.BadHttpRequestException** is the new derived class of **Microsoft.AspNetCore.Server.Kestrel.BadHttpRequestException** and **Microsoft.AspNetCore.Server.IIS.BadHttpRequestException**. These packages are tagged as obsolete and are set for removal in the future release to consolidate duplicate types and to unify the packages across server implementations.
- Code as integers is now the status code used by the **IHttpClientFactory** interface to log HTTP instead of names, to offer developers more flexibility on querying ranges of values.

## Kestrel

Here are the changes to Kestrel, the cross-platform web server for ASP.NET Core:

- **SslProtocols.None** is now the default TLS protocol version of **HttpsConnectionAdapterOptions.SslProtocols** instead of **SslProtocols.Tls12** | **SslProtocols.Tls11**, to support TLS 1.3 and future versions by default.
- Since socket-based transport was the default transport of Kestrel, the libuv APIs are now tagged as obsolete and will be removed in the next version.

## Middleware

The **middleware**, which is a pipeline to handle requests and responses, has a new behavior.

**DatabaseErrorPageMiddleware** and its related extensions are marked as obsolete and replaced with **DatabaseDeveloperPageExceptionFilter**.

## SignalR

The **SignalR** library, which uses real-time web functionality in an application, has a couple of changes:

- ASP.NET Core 5.0 upgrades the package version of the **MessagePack** hub protocol from 1.x to 2.x, which has the latest improvements.
- The **UseSignalR** and **UseConnections** methods are no longer available because they had custom logic that didn't interact with other routing components in ASP.NET Core.

## Static files

Serving **text/csv**, a static file, directly to client applications has a new header value. **text/csv** replaced **application/octet-stream** as the Content-Type header value of Static File Middleware for **.csv** files for compliance with the RFC 7111 standard. You can find the full details of the RFC 7111 standard at <https://tools.ietf.org/html/rfc7111#section-5.1>.

## When to use ASP.NET Core

Since ASP.NET Core provides a web framework that can be used in different use-case scenarios, you can use the framework to build dynamic web applications. This includes web applications such as online stores, internal enterprise apps, content-base, multi-tenant applications, **Content Management Systems (CMSes)**, **Software as a Service (SaaS)**, or just a RESTful service with ASP.NET Core. We will be focusing on building a RESTful service in ASP.NET Core because this is the backend that we will integrate with the Vue.js application in the third part of the book.

ASP.NET Core also contains features for managing authentication, authorization, data protection, HTTPS enforcement, app secrets, XSRF/CSRF prevention, CORS management, and enabling developers to build robust yet secure ASP.NET Core apps.

## Why should you learn ASP.NET Core?



Aside from ASP.NET Core's performance, ASP.NET Core is a popular choice with enterprise businesses, insurance, banks, and other types of businesses. Using IT JobsWatch (<https://www.itjobswatch.co.uk/>), you can search jobs by date in 2020. The .NET Core job vacancy trend has been increasing since 2019, and with average earnings of \$95,657/year according to ZipRecruiter (<https://www.ziprecruiter.co.uk/>).

Based on Stackoverflow's 2020 survey (<https://insights.stackoverflow.com/survey/2020>), ASP.NET Core is the winner of their most loved and wanted web framework. It received the highest number of votes, which went up to 70.7%, meaning these are the developers who are developing with specific languages or technologies and have shown interest in continuing to create with it, followed by React, Vue, and Express. These are reasons to try and use ASP.NET Core because of the huge availability of jobs, and ASP.NET Core is here to stay for the next several years.

This completes the quick overview of ASP.NET Core and what is new in ASP.NET Core 5. You have learned about the current state of ASP.NET Core and how it is the right choice for building performant RESTful services. Now it is time to meet Vue.js.

Let's see why, suddenly, Vue.js became one of the hottest JavaScript frameworks.

## Introducing Vue.js

**Vue.js** is a JavaScript framework for building user interfaces. In a nutshell, Vue.js gives frontend developers everything that they would want. Vue.js has the characteristics of being performant, size-efficient, progressive, developer-friendly, and has one of the slimmest barriers to entry if you are new to frontend development.

Today, Vue.js has more than 1.3 million weekly active users (based on the statistics of Vue.js Devtool extensions) and more than 8 million **npm** downloads per month.

Today, Vue.js is being used by some of the most iconic and influential organizations around the world, such as Apple, IBM, Microsoft, Google, Nike, Wikimedia, NASA, Pixar, Louis Vuitton, L'Oréal, and tens of thousands of businesses of all sizes.

In the next couple of sections of this chapter, we will check out what the Vue.js core team has added in the new Vue.js 3, and whether learning Vue.js a good investment of your time.

Let's find out.

## What's new in Vue.js?

After 2 years of development, the Vue.js core team has finally released the latest Vue.js, version 3, codenamed One Piece. The changes are listed here:

- **More maintainable:** The Vue.js code base has been re-written in TypeScript for maintainability, and the internals are more modular.
- **Faster:** Vue.js 3 is faster and has better performance than Vue.js 2. The new version has a new proxy-based reactivity system.
- **Smaller:** Vue.js has tree shaking; tree shaking is a method to remove unused libraries from the project automatically. This capability is essential to make the file size smaller than the previous version. Vue.js 3 also has some compile-time flags that allow you to drop things that cannot automatically be tree-shaken.
- **Scales better:** Vue.js now provides the Composition API, an easier way to reuse a segment of Vue.js component logic. The Composition API is an exciting new feature that solves complex use cases such as sharing business logic between components.
- **A better developer experience:** For me, Vue.js already provided unparalleled developer experience, but Vue.js has improved it in Vue.js 3 (by introducing the new single-file component improvements, type checking for template expressions, and props of sub-components).

## Why is learning Vue.js the right choice?

Along with Angular and React, Vue.js makes up one of the big three JavaScript tools for building modern web applications. Vue.js is not backed by a tech company such as Microsoft or any of the **Facebook, Amazon, Apple, Netflix, and Alphabet (FAANG)** companies. However, through the years of excellent tooling and the great documentation Vue.js provides, it has acquired numerous sponsors worldwide (you can see the list of sponsors at <https://github.com/vuejs/vue>). Having several sponsorships is good because there will be constant maintenance and improvements in Vue.js.

Third-party libraries such as UI libraries, routing libraries, forms, state management, static site generators, are getting better. Hence, making Vue.js a dependable, trustworthy, solid, stable, reliable, and developer-friendly framework for building enterprise applications. Not to mention that Vue.js has 100+ contributors right now, adding new features, improvements, and fixing all issues that appear on GitHub Vue.js.

This ends our quick overview of Vue.js and what is new in Vue.js 3. You have learned the current state of Vue.js and why considering Vue.js 3 as your frontend application best fits developing modern web applications nowadays.

## Summary

To summarize everything you have gained from finishing this first chapter, you have learned that ASP.NET Core is an open source, cross-platform web framework trusted by enterprise companies worldwide because of its security and performance. And most importantly, you have learned that ASP.NET Core is a battle-tested web framework that gives you peace of mind with different business logic scenarios in the future.

Vue.js, on the other hand, is open source, easy to use and learn, easy to integrate, has excellent documentation, is fast, small, performant, stable, and well suited for any web application. You will never go wrong with Vue.js 3 if you pick it for your applications, be they small or large.

In the next chapter, you will learn the necessary software to install on your computer and set up the development environment in a step-by-step process.

## Chapter 2: Setting Up a Development Environment

In the last chapter, you learned about ASP.NET Core in a nutshell and its latest features. The same goes for Vue.js; you had an overview of Vue.js and its most recently added features in Vue.js 3.

This chapter will teach you how to set up your computer's development environment to build backend and frontend web applications. We will go through different IDEs and text editors to write code and make sure everything has been set up before we proceed with the app development.

Installing everything from the beginning will keep us writing code without interruptions.

In this chapter, we will cover the following topics:

- Installing VS Code, Visual Studio 2019, VS for Mac, and Rider
- Installing .NET 5 SDK, Node.js, and **npm**
- Setting up .NET Core CLI and Vue CLI
- Installing Postman and Vue DevTool
- Installing Entity Framework Core tools
- Installing different database providers
- Installing Git version control

## Technical requirements

These are the links to the software you have to install:

- **Download Visual Studio Code (for Windows, Mac, and Linux):** <https://code.visualstudio.com/download>
- **Visual Studio 2019:** <https://visualstudio.microsoft.com/vs/>
- **Visual Studio for Mac:** <https://visualstudio.microsoft.com/vs/mac/>
- **Rider:** <https://www.jetbrains.com/rider/>
- **Download .NET 5.0:** <https://dotnet.microsoft.com/download/dotnet/5.0>
- **Node.js and Node Package Manager:** <https://nodejs.org/en/>
- **Download Postman:** <https://www.postman.com/downloads/>
- **Vue.js DevTools:** <https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnanhbldajbpd>
- **Vue.js DevTools (Firefox Browser Addadd-ons):** <https://addons.mozilla.org/en-US/firefox/addon/vue-js-devtools/>
- **DB Browser for SQLite:** <https://sqlitebrowser.org/dl/>
- **SQLiteStudioSQLite Studio:** <https://sqlitestudio.pl/>
- **Git source control:** <https://git-scm.com/>

# Installing and configuring VS Code, Visual Studio 2019, VS for Mac, and Rider

This section will guide you in installing and configuring the IDE or text editor of your choice. Here is a quick breakdown of what you can use depending on your machine or operating system.

## VS Code

Download the VS Code installer by going to <https://code.visualstudio.com/download>. I would suggest installing VS Code regardless of your machine's OS because this is the ideal text editor that I recommend when writing JavaScript applications.

The editor has built-in support for TypeScript, IntelliSense, formatting, code navigation, and has tons of extensions that you can use. Add the following basic VS extensions after installing VS Code:

- **Code Spell Check:** This is a spelling checker for source code, which helps you avoid bugs caused by typo errors.
- **Prettier:** This is a code formatter, which reformats the code of your file on each save.
- **Vetur:** This extension gives VS Code features such as syntax-highlighting, snippet, emmet, linting, and IntelliSense for writing Vue.js applications.

This ends the VS Code installation and configuration here. Your VS Code text editor is now set for Vue.js development.

## Visual Studio 2019

This IDE is the preferred IDE of every .NET developer when writing applications using C# or F#. Visual Studio 2019 is packed with different features for your needs. The following is a list of the powerful tools in Visual Studio 2019:

- **Develop:** Use IntelliSense for code suggestions if you get stuck.
- **Analyze:** Use CodeLens to see changes in your code, unit tests, commit history, and so on.
- **Debug:** A breakpoint for inspecting bugs.
- **Test:** A test suite viewer so you can easily navigate and organize your tests.
- **Version Control:** You have two version control engines to choose from – Git, the default version control, and Team Foundation.
- **Collaborate:** Use Live Share to edit and debug the code of your colleagues in real time. This feature is helpful for developers who work remotely.
- **Deploy:** Use the **Publish** button to deploy applications such as ASP.NET Core without leaving Visual Studio.

You should go to <https://visualstudio.microsoft.com/vs/> to download the installer. You can choose the community version, which is the free version of Visual Studio.

After downloading, run the installer and only install the parts you need for ASP.NET Core 5 development. Choose the following options:

- **ASP.NET & web development**
- **Data storage & processing**
- **.NET Core cross-platform development**

The total file size of all the component bundles is around 20 GB, so don't get a little trigger-happy with the bundles, or you may end up over installing the IDE components on your machine.

Another thing to remember is that a bad internet connection will add hours to the installation of Visual Studio 2019 and the IDE's components.

After installing, create a **Console App** (.NET Core) project. Please don't run the project; you only need to open it up and click on the **Extensions** tab on the top navigation bar, then click on **Manage Extensions**, which pops up. Scroll down and download **Productivity Power Tools 2017/2019**. The Productivity Power Tools extension will install the following extensions, which will increase your development productivity:

- **Align Assignments:** Adds a command to Visual Studio to format assignments.
- **Copy As HTML:** Adds support to copy text to the clipboard in HTML format.
- **Double-Click Maximize:** Allows you to maximize and dock window headers.
- **Fixed Mixed Tabs:** A tool that can fix mixed tabs and spaces.
- **Match Margin:** Draws markers in the scroll bar for matches of the word under the caret.
- **Middle Click Scroll:** Lets you scroll in the editor pane using the middle-click mouse button.
- **Peek Help:** Shows a small *F1-Help* guide in the code.
- **Power Commands:** Formats documents on save, clears all panes, collapses projects, opens the containing folder (meaning you will see your current file's location in the directory), clears your recent file list, and more.
- **Solution Error Visualizer:** Shows and highlights errors and warnings in Solution Explorer.
- **Shrink Empty Lines:** Shrinks lines that contain numbers or text to display more lines in the editor.

After downloading the **Productivity Power** tool, close Visual Studio 2019 to initiate the extension installation. You will see the VSIX installer initializing the extensions you have downloaded; click **Ok** and **Modify** to start the installation.

This ends the installation and configuration of Visual Studio 2019; your IDE in your Windows machine is all set.

## Visual Studio for Mac

**Visual Studio for Mac** is Visual Studio 2019 for Mac machines. You can use this IDE to develop apps and games for iOS, Android, and the web using .NET. The following are the primary features that you will notice in Visual Studio for Mac:

- You can use this IDE to write C#, F#, Razor, HTML, CSS, JS, TypeScript, XAML, and XML.
- The IDE has advanced IntelliSense powered by Roslyn, a .NET compiler platform analyzer.
- The IDE has a powerful debugging tool that lets you step into and out of functions and inspect a code stack state.
- The IDE has powerful built-in refactoring options, such as the **extract** method and renaming functions or methods in your code.
- The Visual Studio for Mac IDE has an integrated source control to manage your code in Git or SVN repositories.
- The IDE supports major testing frameworks such as NUnit, MSTest, and xUnit. This IDE allows you to run and debug unit tests and UI tests efficiently.
- The IDE allows you to share your C# and F# projects with your teammates using Windows or macOS.

To download Visual Studio for Mac, you need to go to <https://visualstudio.microsoft.com/vs/mac/>, then install it. Unlike Visual Studio 2019 with extensions, Visual Studio for Mac has few extensions you can install. Unfortunately, in my opinion, none of the extensions are useful for developing ASP.NET Core.

This ends the section on the installation of Visual Studio for Mac. Now let's see how to install Rider.

## Rider

If you are using Linux, Rider fits you best. **Rider**, from JetBrains, is a cross-platform .NET IDE, meaning you can develop .NET, .NET Core, Xamarin, Unity, or ASP.NET applications on Windows, Mac, and Linux. JetBrains is also the creator of ReSharper, a popular Visual Studio extension.

Since Visual Studio 2019 does not exist in a Linux-based operating system, and we need a fast and powerful IDE to write our backend, Rider would be the IDE candidate for Linux machines.

The features in Rider are almost the same as what Visual Studio 2019 gives you. You will have access to code analysis, code editing, refactorings, unit test runners, debuggers, database management tools, code navigation, a frontend framework ready, and plugins.

To download Rider, please go to <https://www.jetbrains.com/rider/> and start installing it. There is no need to configure Rider or add plugins because Rider is feature-rich and fast right off the bat.

Now, you are done with installing and configuring a text editor for writing a Vue.js 3 application and an IDE for building ASP.NET Core 5 APIs. You have learned where to get the downloadable installers for VS Code, Visual Studio IDE, and Rider, and which specific operating systems they should use. You also know what extensions of Visual Studio and VS Code to install to increase your development efficiency.

In the next section, you will install the .NET Core SDK, Node.js, and npm (Node Package Manager).

## Installing .NET SDK, Node.js, and npm

This section will explain what .NET Core SDK, Node.js runtime, and **npm** are. This section will also guide you in installing .NET Core SDK and Node.js runtime in Windows, Mac, and Linux.

### .NET SDK

.NET Core SDK is composed of libraries and tools that let you write .NET Core applications. If you are using Visual Studio 2019 or Visual Studio for Mac, you don't need to install the .NET 5 runtime since Visual Studio includes .NET Core SDK in the IDE installation. .NET Core SDK also has the .NET command-line interface and the .NET Core runtime (the underlying layer for the .NET app to run) for you to run a .NET Core app.

So if you are using Linux and have installed Rider, you can start by going to <https://dotnet.microsoft.com/download> to download and install the .NET 5 SDK.

### Node and npm

**Node** (or Node.js) is an open source and cross-platform runtime environment for executing JavaScript code outside of the browser. This is more important to know for building Vue.js is building the application with the help of **npm**.

**npm** or **Node Package Manager** is a CLI tool and a registry for third-party JavaScript libraries that you can add to a node application. So, for any functionality that you want to include in your application, I am sure that there is an open source library or module that you can use from the **npm** registry.

Here, I have combined the installation of Node.js and **npm** because installing Node.js also installs **npm** automatically. So, go to <https://nodejs.org/en/> to download the installer. Please download and install the **LTS version**, which is short for Long Term Support, and not the current version because some cloud services and **npm** libraries only use LTS versions, and that would not make your application compatible with theirs.

Finally, for Linux users, you can run the following command in your Terminal:

```
sudo apt install nodejs
```

It will install the latest LTS version of Node.js on your machine. There are also other ways of installing Node.js in Linux, but I found this more straightforward than the rest.



Now you have learned what .NET Core SDK is and where to get .NET Core SDK 5, the latest version. You were able to understand what Node.js runtime is and how **npm** works. You have also experienced installing them.

We will now proceed to the next section to set up your .NET Core command-line interface and Vue command-line interface.

## Setting up the .NET Core CLI and Vue CLI

The .NET Core CLI is the cross-platform command-line interface for writing, building, running, and publishing .NET Core applications. The .NET Core SDK installation also installs the .NET Core CLI behind the scenes.

Vue CLI, on the other hand, is the standard tooling for developing Vue.js applications. You will use Vue CLI when creating a project and adding third-party libraries such as the Vue.js UI library and Vue.js state management library. To install the Vue.js CLI, just run the **npm install -g @vue/cli** command, which will install the Vue.js CLI globally on your machine.

You will be able to use Vue CLI in [Chapter 10, Performance Enhancement with Redis](#).

Now you have learned what the .NET Core CLI and Vue CLI are and how they can speed up your development, and you have experienced installing them.

In the next section, you will install Postman, a tool for testing APIs, and Vue DevTool, a browser extension for Vue.js apps.

## Installing Postman and Vue DevTool

**Postman** is a platform for API development. Whenever you are building a RESTful service, Postman is an excellent tool for sending HTTP requests to the APIs you are writing to see how the controllers of your APIs behave. You will learn how to use this when you start writing your APIs in [Chapter 9, Securing ASP.NET Core](#).

Switching to Vue DevTool, this tool helps you debug your application by providing a user interface in DevTool where you can view your Vuex store, events, routing, and the performance of your Vue app. Vue DevTool is available in Chrome and Firefox:

- Here is the link for the Google Chrome extension: <https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd>
- Here is the link for the Firefox add-on: <https://addons.mozilla.org/en-US/firefox/addon/vue-js-devtools/>.

You will use this in action when you start building your routes in [Chapter 12, Using a UI Component Library and Creating Routes and Navigations](#), and the Vue.js store in [Chapter 14, Simplifying State](#)

*Management with Vuex and Sending GET HTTP Requests.*

Now you know which tool to use for testing HTTP requests while developing your backend. You have also learned what browser extension to install and what it does to help you debug in a Vue.js application.

In the next section, you will switch over to the database part and install Entity Framework Core tools.

## Installing Entity Framework Core tools

The **Entity Framework Core** tool is the CLI tool for Entity Framework Core. In short, it enables you to run commonly used Entity Framework commands such as the following:

- **Add-Migration:** Adds a new migration
- **Drop-Database:** Drops the database
- **Get-DbContext:** Lists and gets info about **DbContext** types
- **Script-Migrations:** Creates SQL scripts for migration
- **Update-Database:** Updates the database to the latest migration

You can install this tool globally by running the following command in your Terminal on a Windows, Mac, or Linux machine:

```
dotnet tool install --global dotnet-ef
```

You will see the EF Core CLI tool in action in [Chapter 5, Setting Up DbContext and Controllers](#), of this book.

Now you have learned what an EF Core tools CLI is and the commands that it can run.

In the next section, we will discuss which database provider to install, why you should install it, and how.

## Installing a database provider

EF Core can use several different databases by using NuGet libraries as plugin database providers. You can find providers for MS SQL Server, MySQL, PostgreSQL, Oracle DB, and SQLite.

Since installing any SQL database server can be cumbersome, you will use SQLite, which also has the same query commands you would use in other databases. Entity Framework, being an ORM, handles all the compatibility of the query commands across different databases.

SQLite does not require a database server to run it and runs on Windows, Macbook, and Linux without any hassle. The easy setup and portability of SQLite make the perfect fit for your database provider while learning about EF Core and **DbContext**.

So what software do you need now? You can download and install DB Browser for SQLite from <https://sqlitebrowser.org/> or SQLiteStudio from <https://sqlitestudio.pl/>, which is both cross-platform and open source.

## IMPORTANT NOTE

*Installing a database server in a Docker container to promote the same version and environment across all the developers in a team is recommended. But using a database server in a Docker container is out of this book's scope.*

*However, we will still explore more information about Docker and containers in [Chapter 19](#), Automatic Deployment Using GitHub Actions and Azure.*

Now we are almost at the end of this chapter. Next, we will look at Git version control.

## Installing Git version control

Installing Git, a distributed version control system, will be the last setup you will do. But this does not mean that it is of the least importance. You will need this to save different versions and stages of your repository. Git also helps you roll back your repository's recent working version if you made changes in your code that broke your app, and you can't fix them anymore.

Go to <http://git-scm.com/> and click the **Download** button on the screen to download and install Git.

Now you have learned from this final section about Git version control, where to get it, what it does, and why it is crucial, let's summarize everything.

## Summary

With this, we have reached the end of this chapter. Let's have a recap of the essential things that you have learned. You have learned that the VS Code text editor and Rider IDE can be used on any operating system. Visual Studio 2019 is suited for ASP.NET Core development in Windows, while Visual Studio for Mac fits ASP.NET Core development for macOS.

You have also learned that .NET 5 SDK is needed to build .NET Core applications. Node Package Manager, on the other hand, is a JavaScript library package manager. On top of that, it is also a CLI tool for creating modern JavaScript applications like Vue.js.

Postman is an API tool for testing REST API calls, while Vue CLI is a tool for scaffolding Vue.js applications.

There are also different database providers, such as MS SQL Server, MySQL, PostgreSQL, Oracle DB, and SQLite, that have NuGet package libraries available for ASP.NET Core 5.

And last but not least, Git version control is a must-have tool for saving and creating histories of your code where you can easily roll back or create a new version of your application.

In the next chapter, you will be building your first ASP.NET Core 5 project.

## Section 2: Backend Development

This section deals with the real-world scenario of developing an ASP.NET Core 5 application. The following chapters are included in this section:

- [Chapter 3](#), *Starting Your First ASP.NET Core Project*
- [Chapter 4](#), *Applying Clean Architecture to an ASP.NET Core Solution*
- [Chapter 5](#), *Setting Up DbContext and Controllers*
- [Chapter 6](#), *Diving into CQRS*
- [Chapter 7](#), *CQRS in Action*
- [Chapter 8](#), *API Versioning and Logging in ASP.NET Core*
- [Chapter 9](#), *Securing ASP.NET Core*
- [Chapter 10](#), *Performance Enhancement with Redis*

# Chapter 3: Starting Your First ASP.NET Core Project

**REST (Representational State Transfer)** is an interface between systems using HTTP to fetch data and generate HTTP operations in all possible formats, such as JSON, which is the commonly used format for sending and getting data.

By the end of this chapter, you will have created an ASP.NET Core 5 Web API and understood the **Program.cs** file's responsibility inside the ASP.NET Core project, learned how to use the **Startup.cs** file of that project, and tried the new built-in API documentation out of the ASP.NET Core Web API.

In this chapter, we will cover the following topics:

- Creating an ASP.NET Core project
- Understanding the **Program.cs** file
- Demystifying the **Startup.cs** file
- Getting started with Swashbuckle

## Technical requirements

Here is what you need to complete this chapter.

- **For building a backend:** Visual Studio 2019, Visual Studio for Mac, or Rider
- **For scaffolding a project:** The .NET Core CLI

## Creating an ASP.NET Core project

The project you will be creating here is not yet the real-world backend ASP.NET Core application that you will connect with Vue.js. The purpose of this is to see what default files and folders are in a newly created ASP.NET Core project.

You will be using the command line instead of the IDE to ensure that other developers trying out ASP.NET Core from this book on Windows, Mac, or Linux will get the same results.

To start the project, follow these simple instructions:

1. Create a folder anywhere on your computer; it could be on your desktop or your **Download** directory. I usually put my test or demo apps in the **Download** folder to easily find them and delete them.
2. Name the folder **DemoProject**, and then open up the folder.
3. Next, open your command-line terminal and navigate to the directory. There are many ways to do this efficiently. The following are the tools that I can recommend for opening a command line in a specific folder:

**Windows users:** <https://hyper.is/>.

**Mac users:** <https://hyper.is/> or <https://iterm2.com/> and the OpenInTerminal tool: <https://github.com/Ji4n1ng/OpenInTerminal>.

**Ubuntu Linux users:** OpenInTerminal is built in by default.

As shown in *Figure 3.1*, I will open my command line by right-clicking the mouse button on the specific folder:

 Figure 3.1 – Open Hyper in this folder

Figure 3.1 – Open Hyper in this folder

After navigating to the **DemoProject** folder where you will create the application, run the following commands using the .NET Core CLI installed in the previous chapter:

- **dotnet new sln**
- **dotnet new webapi --name Web**

- `dotnet sln add [csproj-file-location]`
- `dotnet run Web.csproj`

## First command – `dotnet new sln`

The first command creates a solution file that will track and organize projects in Visual Studio. Your IDE will use this solution file as the container for different projects, such as class libraries, executable applications, websites, or unit test projects. The CLI uses the name of the folder you have made, which is **DemoProject**:

```
dotnet new sln
```

You should see a **DemoProject.sln** file in the **DemoProject** folder after running this command using the .NET CLI.

## Second command – `dotnet new webapi --name Web`

The second command creates an ASP.NET Core 5 Web API project containing the boilerplate to run the app:

```
dotnet new webapi --name Web
```

The command is a little self-explanatory, instructing the CLI to create a new Web API project and name it **Web**.

You should see a folder named **Web** and inside of that is the ASP.NET Core Web API project called **Web** as well, which matches its folder.

## Third command – `dotnet sln add [csproj-file-location]`

The third command adds the **Web** project to the solution file. It's normal to see a multiple-project solution in real-world enterprise development:



```
dotnet sln add [csproj-file-location]
```

**dotnet sln add ./Web/Web.csproj** is for Linux and Mac users, and **dotnet sln add .\Web\Web.csproj** for Windows users.

Now navigate to the **Web** directory. You can do that by running the following command:

```
cd Web
```

The **cd** command will not be available for you if you are using the native CMD in Windows. I suggest using the **bash terminal** that was included in the Git installation for Windows. The bash terminal uses forward slashes in the directory tree.

OK, you should already be on the path – **./DemoProject/Web**. Now let's move to the next command.

## The last command to run – dotnet run Web.csproj

This command, which is useful while developing your app, will run your source code application. The command automatically resolves the NuGet dependencies of your app through running **dotnet restore** implicitly before the **dotnet run** command:

```
dotnet run Web.csproj
```

Now go to your browser and enter the following URL:

**https://localhost:5001/weatherforecast**. Your browser may prompt you with a message, saying that this site is not secure. To permit the development certificate that ASP.NET Core created, click the **Accept** button that pops up on the browser. You will only need to do this once because your browser will remember the certificate for you.

The JSON response, formatted by the **JSONview** Chrome/Firefox extension, should be visible in your browser. Don't expect the same values

for each property of each object; the **controller** method randomizes the values here before returning the response. If you are interested in the **JSONView** extension, you can get it from <https://jsonview.com/>.

The following code is the JSON response you will get from hitting **<https://localhost:5001/weatherforecast>** using your browser:

```
[
  {
    "date": "2020-10-12T12:21:55.5119059+02:00",
    "temperatureC": -19,
    "temperatureF": -2,
    "summary": "Scorching"
  },
  {
    "date": "2020-10-13T12:21:55.5123846+02:00",
    "temperatureC": 32,
    "temperatureF": 89,
    "summary": "Sweltering"
  },
  {
    "date": "2020-10-16T12:21:55.5123876+02:00",
    "temperatureC": 6,
```

```
"temperatureF": 42,  
  
"summary": "Sweltering"  
  
},  
  
]
```

You can take a peek at how the JSON response is generated and is returned to the client by looking inside the **WeatherForecastController.cs** file, which you can find in **Web/Controllers/**. We will talk more about controllers in [Chapter 5, Setting Up DbContext and Controllers](#).

We have created an ASP.NET Core 5 project using the .NET CLI. Now let's move to the next section to learn about the **Program.cs** file and its job in an ASP.NET Core Web API project.

## Understanding the Program.cs file

The **Program.cs** file, located in the **Web** project's root folder, is responsible for executing the **Main** method, which serves as the app's entry point and starts the application.

The following code is present inside the **Program.cs** file, and we will break it down into digestible content:

```
namespace Web  
  
{  
  
    public class Program  
  
    {  
  
        public static void Main(string[] args)  
  
        {
```

```

        CreateHostBuilder(args).Build().Run();
    }

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

```

As you can see, **namespace** is named after the project's name. Next, there's the **Program** class. You may notice that the **Program** class with the **Main** method is similar to the .NET console application because the ASP.NET Core Web API is indeed a console project with different dependencies for building web applications.

Take a look inside the **static Main** method. You can see that **Main** is simply calling **CreateHostBuilder(args).Build().Run()**, which is this block of the following code:

```

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

```

```
});
```

The preceding code creates an instance of **WebHostBuilder** with a configuration from the **Startup** class, which we will discuss in the next section. The static **CreateWebHostBuilder** method eventually gets built and run by the **Main** method.

Simplifying the initial bootstrapping of the web app is not the only job of the **Program.cs** file. You can set other configurations, such as custom logging, environment variables, Azure Key Vault configuration, in-memory .NET objects, Azure App configuration, command-line arguments, directory files, customer providers, and more.

Now that you know what **Program.cs** contributes to an ASP.NET Core project, we can move on to the next section, which concerns the **Startup.cs** file.

## Demystifying the Startup.cs file

The **Startup.cs** file, located in the web project's root folder, is first executed when the application starts. The code inside the file is as follows:

```
namespace Web
{
    public class Startup
    {
        public Startup(IConfiguration configuration)...

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)...

        public void Configure(IApplicationBuilder app,
```

```
        IWebHostEnvironment env)...  
  
    }  
  
}
```

So let's check this out one by one.

The first block of code you will see is the **Startup** constructor, which takes an **IConfiguration** interface and assigns it to the public **IConfiguration Configuration** getter. The **Configuration** property signifies that a **Configuration** argument will always be present whenever the **Startup** class gets instantiated.

Moving on, you will find two public methods included in the **Startup.cs** file – **ConfigureServices** and **Configure**.

Let's take a look at **ConfigureServices** first. You'll find many blogs online that describe the **ConfigureServices** method as a location where you can insert your classes with the built-in **Inversion of Control (IoC)** container. I've found this hard for my audience to understand whenever I do some workshops and training. So, I will make it as simple as possible.

ASP.NET Core provides a built-in IoC container that helps you use a service in any part of your classes. I'll give you a use case where you can use the **ConfigureServices** method. Suppose you like to use libraries or SDKs, such as GraphQL, OData, and Identity, in your ASP.NET Core application. You would likely go to Google and search how to implement the mentioned libraries or SDKs, right? You will end up in NuGet packages repositories.

In most cases, you will have to install the NuGet package and then register the package inside the **ConfigureServices** method. Normally, you would write **services.AddTheFeatureOfTheInstalledNuget**. You type the word **services**, then a full stop, and you will see the interfaces of the NuGet packages you installed through the help of the IntelliSense feature of your IDE.

**services.AddSomething** is the IoC part of ASP.NET Core, where you are injecting interfaces of the tools you will use. Injecting dependencies in a program is IoC in a nutshell. IoC itself is a big topic, so I would recommend reading more about it at <https://auth0.com/blog/dependency-injection-in-dotnet-core/>, to avoid us from drifting too far from the main topic in this chapter.

The **ConfigureServices** method also has built-in services or utilities of ASP.NET Core. Some good examples of these that you can see right now in your **Startup.cs** file are as follows:

- **services.AddControllers()**, which configures the MVC services with controllers for an API.
- **services.AddSwaggerGen()**, whose purpose is to add API documentation in your ASP.NET Core automatically. You will learn more about this in the *Getting started with Swashbuckle* section.

The following is the block of code or logic of the **ConfigureServices** method in the **Startup.cs** file:

[illegible]

Here, **ConfigureServices** takes an **IServiceCollection** interface, which exposes all the services available in your ASP.NET Core, where any services are just one call away from being enabled. **IServiceCollection** enables **Controllers** and **SwaggerGen**, as you can see inside the **ConfigureServices** method.

Now let's check out the **Configure** method.

The **Configure** method allows you to add middleware to your application's request pipeline. This method is where you would want to modify the HTTP requests of a frontend application or another service before it lands in the controller.

The process is that a request gets modified by a function or a component you've installed or created. Examples of these functions or components are logger or authentication, and they may or may not be in order depending on the logic that you are building for a solution. Now let's check out the code block of the **Configure** method:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseSwagger();
        app.UseSwaggerUI(c =>
            c.SwaggerEndpoint("/swagger/v1/swagger.json",
                "Web v1"));
    }
}
```



```
app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
});
}
```

The **Configure** method here takes an **IApplicationBuilder** interface, which gives the mechanisms to modify or change the request pipeline of ASP.NET Core, and **IWebHostEnvironment**, which offers information about the web hosting environment where your ASP.NET Core is running.

Now let's check out this default middleware that ASP.NET Core 5 provides:

- **UseDeveloperExceptionPage()**, which shows a helpful error page for debugging your app.
- **UseSwagger()** for API documentation. I will discuss this further in the *Built-in Swagger integration* subsection of the *Getting started with Swashbuckle* section.
- **UseSwaggerUI()**, which allows some meta information of your APIs to be viewed.

You can see that the **if env.IsDevelopment** conditional only includes the preceding three pieces of middleware in the development environment.

And here is the rest of the default middleware in the ASP.NET Core Web API:

- **UseHttpsRedirection()**: Redirects the HTTP requests to HTTPS
- **UseRouting()**: Adds routing capabilities that define your endpoints

- **UseAuthorization():** Enables authorization of HTTP requests
- **UseEndpoints():** Define the endpoints and maps the controllers to the respective endpoints by using **MapControllers()**

Now we have finished discussing what's inside **Startup.cs**. Let's move to this chapter's last topic, the latest features added by default in ASP.NET Core 5 OpenAPI, also known as Swagger.

## Getting started with Swashbuckle

What is Swashbuckle? **Swashbuckle** is a NuGet package that provides an easy way of adding Swagger to ASP.NET Core Web API projects.

One of the things that a backend developer should do is to create developer-friendly APIs. You can do this by documenting APIs while developing them. Documenting APIs is essential and helpful to the developers who will use your APIs. For instance, documentation saves everyone's time by removing the unnecessary questions and answers between you and the consumer of your APIs.

To give you another scenario where you will see how the API documentation can be substantial, imagine working in a big team. By documenting your APIs, your teammates will have opportunities to quickly check existing APIs, which avoids wasting time and money writing the same APIs repeatedly.

OK, I already mentioned the main benefits you can get from the API documentation. Now here is a list of what we will uncover.

## Introducing OpenAPI and Swagger

The **OpenAPI Specification**, formerly known as the **Swagger Specification**, is a specification for describing APIs' capabilities in text, JSON, and YAML format while being language-agnostic.

Now we know that the OpenAPI is a specification, there's a popular tool that implements it using the JSON format it produces. This tool is also Swagger, but don't get confused with the Swagger Specification; this Swagger that I'm talking about now is the tool that will make it easier for you to create API documentation and help you integrate an API quickly.

Now let's go to the next section to see the built-in Swagger integration that ASP.NET Core 5 offers.

## Built-in Swagger integration

ASP.NET Core 5 has officially included the **Swashbuckle** package, which integrates Swagger into the ASP.NET Core Web API in the **Microsoft.Extensions.DependencyInjection** namespace. Swagger is automatically installed for you by the .NET CLI while scaffolding the app. You can see it inside **ItemGroup** of **Web.csproj**:

```
<PropertyGroup>

  <TargetFramework>net5.0</TargetFramework>

</PropertyGroup>

<ItemGroup>

  <PackageReference
    Include="Microsoft.AspNetCore.Authentication.JwtBearer"
    Version="5.0.2" NoWarn="NU1605" />

  <PackageReference
    Include="Microsoft.AspNetCore.Authentication.OpenIdConnect"
    Version="5.0.2" NoWarn="NU1605" />

  <PackageReference Include="Swashbuckle.AspNetCore"
    Version="5.6.3" />

</ItemGroup>
```

In the preceding code, you can see that there is a package reference to **Swashbuckle.AspNetCore version 5.\*.\***. Hence, it is ready to be used.

Also, Swagger is enabled in the ASP.NET Core 5 Web API project by default. Open up **Startup.cs** file and see the **AddSwaggerGen** extension in the following code:

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title="Web", Version="v1" });
});
```

The **AddSwaggerGen** is an extension for services that enables the Swagger generator to build **SwaggerDocument** objects. You will also find the **SwaggerDoc** extension, which adds a URL-friendly name to identify a document by passing the following two arguments:

- "v1"
- **new OpenApiInfo { Title = "Web", Version = "v1" };**

This **v1** string type and the **OpenApiInfo** object are necessary parameters when using the **SwaggerDoc** extension. Moreover, the **OpenApiInfo** requires you to add values to the **Title** and **Version** properties to add some metadata to the Swagger documentation. Now let's go to the middleware part.

There are two pieces of middleware included inside the **Configure** method out of the box:

- **UseSwagger():** A middleware that generates the OpenAPI document
- **UseSwaggerUI():** Renders the Swagger UI web page

You can see the **UseSwagger** and **UseSwaggerUI** middleware in action here:

```
if (env.IsDevelopment())
```

```
{  
  
    app.UseDeveloperExceptionPage();  
  
    app.UseSwagger();  
  
    app.UseSwaggerUI(  
        c => c.SwaggerEndpoint("/swagger/v1/swagger.json",  
                                "Web v1"));  
  
}
```

There's one more thing to add to the **UseSwaggerUI** middleware, though. You have to specify the documentation's route and version in **UseSwaggerUI** through the **SwaggerEndpoint** extension.

In the next section, we will find out how to open up or view a sample of Swagger documentation provided by ASP.NET Core.

## Swagger documentation and Swagger UI in action

After seeing the **Startup.cs** file's code, let's check out the Swagger documentation and Swagger UI.

We will run the application to see the Swagger documentation and Swagger UI. We will also test the API to determine the user interface it shows to developers like us. Perform the following steps in turn to complete this section:

1. Check whether your ASP.NET Core 5 is still running. If your project is not running, run the following command inside the **Web** folder:

**dotnet run Web.csproj**

2. Head to the following URL,  
**<https://localhost:5001/swagger/v1/swagger.json>**, in your browser. As shown in *Figure 3.2*, you will see the generated Swagger JSON document describing the **WeatherForecast** endpoint:


Figure 3.2 – Swagger documentation

Figure 3.2 – Swagger documentation

If you notice in the preceding screenshot, the URL version matches the Swagger document version, the version of OpenAPI, the title, and the version defined by the **OpenApiInfo** object.

You will also find the single default controller in the ASP.NET Core 5 project, **WeatherForecast**, along with its **GET** method.

3. Now, navigate to the Swagger UI:  
**<https://localhost:5001/swagger/index.html>**. The Swagger UI in *Figure 3.3* is the interactive web form of the Swagger documentation. You should see on your screen the same image as *Figure 3.3*:


Figure 3.3 – Swagger UI

Figure 3.3 – Swagger UI

The Swagger UI lets developers discover the details of the controllers and schemas of the controllers. In the preceding screenshot, you only see one for the controller and one model because that's what ASP.NET Core 5 has provided for you out of the box.

4. Click the **GET** method to see its info. Your screen should look like this:


Figure 3.4 – Swagger UI

Figure 3.4 – Swagger UI

The screenshot describes the selected HTTP method's information, such as parameters, response code, media type, example value, and the schema. You will notice here that the media type is set to **text/plain** by default. You can change that to **application/json** if you want.

5. Click the **Try it out** button to show the UI to test the **GET** method:


 Figure 3.5 – Swagger UI

Figure 3.5 – Swagger UI

After clicking the **Try it out** button, the wide **Execute** button should appear, as you can see in *Figure 3.5*. We can try the HTTP **GET** method to see whether it hits the endpoint and responds with **200**.

6. Click the **Execute** button, and then observe the UI:


 Figure 3.6 – Swagger UI

Figure 3.6 – Swagger UI

After clicking the **Execute** button, you should see a screen similar to *Figure 3.6*. You can spot the curl CLI, a tool for URL manipulation in the top area of the image, meaning you can hit the endpoint by running the **curl** command on your terminal and will see the same response.

You can also spot the exact request URL or endpoint after the **curl** command, and for the server response, you will find on the image the response body in JSON format.

7. In this part, you don't need to do anything aside from scrolling. Scroll down inside the UI to see the rest of the response. You can also find the response headers here, which tell you the content type, data, and server.
8. Now let's move down to the Swagger UI schemas to investigate what we can find there. Click the **WeatherForecast** dropdown, as shown in *Figure 3.7*. You should see that the properties of **WeatherForecast** are the same as the image, which are **date string**, **temperatureC integer**, **temperatureF integer**, and **summary string**:


 Figure 3.7 – Swagger UI

Figure 3.7 – Swagger UI

9. Lastly, go back to your ASP.NET Core 5 project and see the **WeatherForecast** class in the **WeatherForecast.cs** file:

```
public class WeatherForecast
{
    public DateTime Date { get; set; }

    public int TemperatureC { get; set; }

    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);

    public string Summary { get; set; }
}
```

10. Compare the shape of the **WeatherForecast** class with the **WeatherForecast** schema from the Swagger UI. They should be the same.

That concludes our discussion of the Swagger documentation and Swagger UI from the **Swashbuckle** package. Now, let's summarize what we learned in the chapter.

## Summary

With that, you have learned how to easily create an ASP.NET Core 5 project without an IDE.

You've understood how the **Program.cs** file works, executing the main method, and starting the application. You've also found out what the **Startup.cs** file's responsibilities are: to enable services through the built-in dependency injection and collect all the middleware running in order.

Lastly, you've seen what's new in an ASP.NET Core Web API project.

ASP.NET Core 5 sets up the Swagger documentation for you right off the bat when you create a Web API project. You have seen the importance of



Swagger documentation in helping developers see the existing APIs and check their details.

In the next chapter, we will create another .NET solution and project with clean architecture in mind.

# *Chapter 4: Applying Clean Architecture to an ASP.NET Core Solution*

With the previous chapter still fresh in your memory, this is the chapter where you will apply clean architecture to an ASP.NET Core 5 solution. This chapter teaches you the real-world organization of files, folders, projects, and ASP.NET Core app dependencies, preparing you for future big and scalable ASP.NET Core 5 enterprise applications.

Organizing your code, files, and folders helps other developers to understand your code, refactor it, and add future features, something that happens most of the time in a real-world application.

We will cover the following topics:

- Introducing clean architecture
- The core layer
- The infrastructure layer
- The presentation layer
- Managing the **Tests** folder
- Structuring a clean architecture solution

## Technical requirements

Here is what you need to complete this chapter:

- **For building a backend:** Visual Studio 2019, Visual Studio for Mac, or Rider
- **For scaffolding a project:** The .NET Core CLI
- Bash terminal for Mac and Linux users
- PowerShell or the Git Bash terminal for Windows users

Here is the link to this chapter's finished repository:  
<https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter04>.

## Introducing clean architecture

So, what on earth is clean architecture? In a nutshell, **clean architecture** is a principle of organizing software architecture to keep developers away from the difficult refactoring of the application in the future. Clean architecture helps you build a service for a specific domain model that prepares it for microservice architecture.

Here is a diagram of clean architecture in ASP.NET applications:


 Figure 4.1 – Clean architecture diagram

Figure 4.1 – Clean architecture diagram

In clean architecture, two layers must be the core or center of the structure. These layers are the domain layer, which contains most of your entities, enums, and settings. The application layer keeps most of your **Data Transfer Objects (DTOs)**, interfaces, mappings, exceptions, behaviors, and business logic.

The difference is that the enterprise logic could be shared across many systems, whereas the application logic or business logic is specific. Now, rather than having the core dependent on data access and infrastructure, we invert those dependencies. So, presentation and infrastructure depend on the core, but the core has no dependency on either layer. This structure is done by adding abstractions or interfaces within the application layer, which are implemented outside of the application layer in other layers. For instance, if we wanted to implement the repository pattern, we would add an **IRepository** interface within the infrastructure application and implementation.

With this design principle, all dependencies point toward the middle. Hence, you can see that the core has no dependencies on other layers. Subsequently, presentation and infrastructure depend on the core, but not on one another. This is very important because we want to make sure that the logic we create for this system stays within the core. For example, if the presentation took a dependency on infrastructure to send some notifications, it is a logic. This logic now has to appear within the presentation layer. Hence it has to orchestrate the interaction between presentation and infrastructure. Typically, we don't want that to happen because we can't reuse that logic. We want a system that's going to last for a decade at least.

Here is the thing. If we have a frontend web app with a backend application, there is a high probability that it is not going to be there in 10 or 15 years. We need the logic inside the core, where it is isolated from all of those things.

If the circular diagram (see *Figure 4.1*) of the clean architecture is a bit difficult to understand, we can look at the following illustration:

 Figure 4.2 – A flat diagram of clean architecture

Figure 4.2 – A flat diagram of clean architecture

The preceding diagram is a simplified and flattened version of the clean architecture.

This diagram illustrates that your application core is at the very bottom, which is the center of the clean architecture circle diagram (see *Figure 4.1*).

As you can see, the app core has no dependencies. On the other hand, the UI or the presentation layer is dependent on the application core. You will also notice that the project infrastructure is dependent on the app core.

In the end, the whole application becomes highly testable. You can quickly write unit tests, integration tests, and functional tests.

Now let's break these down into more digestible content. We'll separate the layers into three folders with .NET projects inside:

- The **core** folder will have **Application** and **Domain** projects.
- The **infrastructure** folder will have **Data** and **Shared** projects.
- The **presentation** folder will have a **WebApi** project.

This is the setup of the projects and folder arrangement of the ASP.NET Core 5 solution.

## The core layer – directory

The **core layer** is the center of clean architecture. We will implement this as a directory or folder inside a Visual Studio solution that can be created using the .NET Core CLI. All other project dependencies must point toward the core to the domain and application project to be specific. Similarly, the core layer will never depend on any other layers.

To set up the core layer, we need two projects – the **Domain** and **Application** projects.

### Domain – project

This clean architecture part is a .NET Standard 2.1 class library project with entities, interfaces, enums, DTOs, and so on.

The **Domain** project must have an empty project reference, which shows that it does not have any dependencies on any project.

### Application – project

This part of the clean architecture is also a .NET Standard 2.1 class library project. It has defined interfaces, but the implementations are outside of this layer. This project also has the command and queries of the CQRS

pattern, the behaviors of MediatR, the profile of AutoMapper, exceptions, models, and so on.

We will create the aforementioned projects in the *Structuring a clean architecture solution* section of this chapter.

What if you just started to build microservices and realized that there is code in the core layer you want to reuse in other services or domains?

## Shared Kernel – NuGet project

Ideally, the **Shared Kernel** project is a private NuGet package for sharing code between multiple projects, in microservices, for instance. This project helps us easily swap out which version we depend on without breaking the work of our peers or other teams building other solutions.

Now let's take a look at the infrastructure layer.

## The infrastructure layer – directory

The **infrastructure layer** has the class implementations of interfaces defined in the **Application** project. Resources such as SMTP, filesystems, or web services are samples of the application's external dependencies but implemented in this layer.

This layer is another directory inside the solution while holding multiple projects. The projects we will add here are for data and shared projects. We can also add a project named **Identity for authentication**, but we will do this in [Chapter 9, Securing ASP.NET Core](#), to keep the structuring part of this clean architecture minimal. So here they are.

To set up the infrastructure layer, we will need two projects – **Data** and **Shared**.

## Data – project

This part of the infrastructure layer is a .NET 5.0 class library project intended for a database. You can also name this data project to a persistence project; persistence and data are intractable.

## Shared – project

This part of the infrastructure layer is also a NET 5.0 class library project. This project will include shared code between different services, such as Email, SMS, or Date Time.

Again, you will create the infrastructure layer projects in the *Structuring a clean architecture solution* section of this chapter. In the meantime, let's move to the next section, which is the presentational layer.

## The presentation layer – directory

The **presentation layer** is where you build your web application. The web application can use ASP.NET Core MVC, ASP.NET Core Web API, a **Single-Page Application (SPA)**, or a mobile application.

The real-world application that we will be building throughout the chapters of this book requires a Web API project and a website.

## WebApi – project

**WebApi** is an ASP.NET Web API project that uses .NET 5.0. This project interacts with any client-side applications, such as on the web, mobile, desktop, and **Internet of Things (IoT)**.

Also, **WebApi** depends on both the **Application** and **infrastructure** layers.

## client-app – non-project web application

**client-app** is the web application that serves as the user interface of the application. We will create the Vue.js 3 application in [Chapter 11](#), *Vue.js Fundamentals in a Todo App*, and it will be inside the **presentational** folder. Name it **client-app**.

That's it for the presentation layer. So, where do you think we should place the tests for the application? Let's move on to the next section.

## Managing tests – directories

This section is not part of the clean architecture principle, but the best practice is to use separation of concerns. Hence, organizing the test projects based on the tests they do, such as unit tests, functional tests, integration tests, and load testing, is the best practice.

I will lay out the test projects in the following subsections, but we will not create them in this chapter yet. The following are the test projects.

### Unit test – project

A **Unit test** project tests small and specific parts of your code. This project can be created using an XUnit, NUnit, or MSTest project.

### Integration test – project

An **Integration test** project tests whether the components are working together. This project can be created using an XUnit, NUnit, or MSTest project.

Now that we've finished the part where developers write their tests in ASP.NET Core, it is time to create the layers and projects for an ASP.NET Core 5 solution.



# Structuring a clean architecture solution

This section is that part where we will create a solution, directories, and projects to build a clean architecture solution for the application. The application is about travel lists and places worldwide where an admin can add new, delete, update, and read places from the travel destination list.

Before we begin, open up your terminal and navigate to your desktop. Make sure your terminal or command line is in the **Desktop** directory.

For Windows users, use PowerShell or the Git Bash terminal. The Git Bash terminal is part of the Git version control installer you used in [Chapter 2, Setting Up a Development Environment](#). If you are going to use PowerShell, remember to use a backslash instead of a forward slash:

1. Let's start with the folder of the solution by running the following command:

```
mkdir Travel
```

This command creates a folder named **Travel**.

2. The next step is to go inside the **Travel** folder:

```
cd Travel
```

Your terminal should be inside the **Travel** directory by now.

3. Now, use the **dotnet** CLI:

```
dotnet new sln
```

This command creates a solution file and gets its name from the folder where it is located.

4. The next step is to create a folder named **src**:

```
mkdir src
```

The **src** directory is inside the **Travel** directory.

5. Now, navigate to the **src** directory or folder:

```
cd src
```

While inside the **src** directory, we will create three folders here, the three layers, namely – **core**, **infrastructure**, and **presentation**.

6. Now let's create the **core** folder:

```
mkdir core
```

This command creates the **core** directory.

7. Now create the folder for **infrastructure**:

```
mkdir infrastructure
```

This command creates the **infrastructure** directory or folder.

8. And now, create the **presentation** directory or folder:

```
mkdir presentation
```

This command creates the **presentation** directory or folder.

9. Go inside the **core** directory:

```
cd core
```

While inside the **core** directory, we will create two projects. The first project will be for the **Domain** layer:

```
dotnet new classlib -f netstandard2.1 --name Travel.Domain
```

This command creates a new .NET Standard 2.1 class library named **Travel.Domain**. The second project will be for the **Application** layer:

```
dotnet new classlib -f netstandard2.1 --name Travel.Application
```

This command creates a new .NET Standard 2.1 class library named **Travel.Application**.

10. Then, go inside the **Travel.Application** directory:

```
cd Travel.Application
```

This command navigates you to the **Travel.Application** folder.

11. Then we need to add a reference to the **Travel.Domain** project:

```
dotnet add reference ../Travel.Domain/Travel.Domain.csproj
```

Now, the **Travel.Application** project has a dependency on the **Travel.Domain** project, but only this project.

12. Now, navigate to the **infrastructure** directory:

```
cd ../../infrastructure
```

While inside the **infrastructure** directory, we will create the projects for **Data** and **Shared**.

13. Let's now create the infrastructure for **Data** or persistence:

```
dotnet new classlib -f net5.0 --name Travel.Data
```

This command creates a new .NET 5.0 class library project named **Travel.Data**.

14. Let's now create the infrastructure for **Shared**:

```
dotnet new classlib -f net5.0 --name Travel.Shared
```

This command creates another .NET 5.0 class library project named **Travel.Shared**.

15. Then, go inside the **Travel.Data** directory:

```
cd Travel.Data
```

While inside the **Travel.Data** directory, we need to make **Travel.Data** dependent on the **Domain** and **Application** layers.

16. Let's add a reference to the **Travel.Domain** project first:

```
dotnet add reference  
../../core/Travel.Domain/Travel.Domain.csproj
```

Now, **Travel.Data** is dependent on the **Travel.Domain** project.

17. Now, let's add a reference to the **Travel.Application** project:

```
dotnet add reference  
../../core/Travel.Application/Travel.Application.csproj
```

This command creates a dependency on **Travel.Data** to **Travel.Application**.

18. Now, go to the **Travel.Shared** project:

```
cd ../Travel.Shared
```

This command takes you to the directory of the **Travel.Shared** project.

19. Now we can add a reference to the **Travel.Application** project:

```
dotnet add reference  
../../core/Travel.Application/Travel.Application.csproj
```

This command creates a dependency on the **Travel.Shared** project to the **Travel.Application** project.

20. Now, let's go to the **presentation** layer:

```
cd ../../presentation
```

Now that we are in the **presentation** layer, we will create a Web API project here.

21. We will create the latest ASP.NET Core 5 **webapi** here:

```
dotnet new webapi --name Travel.WebApi
```

This command creates a new Web API project named **Travel.WebApi**.

22. Let's navigate inside the **Travel.WebApi** folder:

```
cd Travel.WebApi
```

Now, while inside **Travel.WebApi**, we will add some references here.

23. Let's add a reference to the **Application** layer:

```
dotnet add reference  
../../core/Travel.Application/Travel.Application.csproj
```

This command adds a dependency to the **Travel.Application** project.

24. Now let's add a reference to **Data** for persistence:

```
dotnet add reference  
../../infrastructure/Travel.Data/Travel.Data.csproj
```

This command adds a dependency to the **Travel.Data** project.

25. The next step is to add a reference to the **Shared** project:

```
dotnet add reference  
../../infrastructure/Travel.Shared/Travel.Shared.csproj
```

This command adds a dependency to the **Travel.Shared** project.

26. Now we can go back to the root folder of the application solution:

```
cd ../../..
```

Since the solution file is in the root folder, we can register all the projects to the solution file.

27. The first project we will add to the solution file is the **Domain** layer:

```
dotnet sln add src/core/Travel.Domain/Travel.Domain.csproj
```

This command lets the solution file know the **Travel.Domain** project.

28. We will also add the **Application** layer of the app:

```
dotnet sln add  
src/core/Travel.Application/Travel.Application.csproj
```

This command lets the solution file know the **Travel.Application** project.

29. The next project to add is **Data** for persistence:

```
dotnet sln add src/infrastructure/Travel.Data/Travel.Data.csproj
```

This command lets the solution file know the **Travel.Data** project.

30. We also have to add the **Shared** project:

```
dotnet sln add  
src/infrastructure/Travel.Shared/Travel.Shared.csproj
```

This command lets the solution file know the **Travel.Shared** project.

31. Now, the last project to add is the Web API project:

```
dotnet sln add  
src/presentation/Travel.WebApi/Travel.WebApi.csproj
```

This command lets the solution file know the **Travel.WebApi** project.

You have now finished structuring an ASP.NET Core 5 application with a clean architecture. Close down your terminal and double-click the solution file to open up the whole application.

Let's see the application in either Visual Studio 2019, Visual Studio for Mac, or Rider.

## Visual Studio 2019

In Visual Studio 2019, this is how it should look:

 Figure 4.3 – Folder structure in Visual Studio 2019

Figure 4.3 – Folder structure in Visual Studio 2019

Visual Studio 2019 shows **Travel** as the solution that has the **src** directory. The **core**, **infrastructure**, and **presentation** layers must be under the **src** directory. You will also see that the **Application** and **Domain** projects are under the **core** directory, while the **Data** and **Shared** projects are under the **infrastructure** directory.

Similarly, you can find the ASP.NET Core 5 **WebApi** under the **presentation** directory.

Now let's see the application for Mac users.

## Visual Studio for Mac

In Visual Studio for Mac, this is how it should look if you followed along correctly:

 Figure 4.4 – Folder structure in Visual Studio for Mac

Figure 4.4 – Folder structure in Visual Studio for Mac

**Travel** is the name of the application solution, and below that is the **src** directory, which has the **core** layer, **infrastructure** layer, and **presentation** layer in the clean architecture.

Moreover, the **Application** and **Domain** projects are under the **core** directory, while the **Data** project and the **Shared** project are under the **infrastructure** directory. Finally, the ASP.NET Core 5 **WebApi** project must be inside the **presentation** directory.

That's all for Mac users. The structure should be the same as Visual Studio 2019. Likewise, the structure should be the same as the next section, which is for users of Rider. Now let's see how the Rider IDE shows the application solution.

## Rider

Lastly, if you are using Rider, the following screenshot shows you the structure of your application solution:

 Figure 4.5 – Folder structure in Rider

Figure 4.5 – Folder structure in Rider

Again, the screenshot displays **Travel** as the solution's name, and it has the **src** directory, which holds the **core** layer, **infrastructure** layer, and **presentation** layer.

The **Application** and **Domain** projects are inside the **core** directory, while the **Data** and **Shared** projects are inside the **infrastructure** directory.

In the **presentation** folder, you should see the ASP.NET Core **WebApi** project

OK, so that was a well-layered and structured clean architecture solution. Here is the GitHub repository of the source code after finishing this chapter:



<https://github.com/PacktPublishing/ASP.NET-Core-5-and-Vue.js-3/tree/master/Chapter-4/Finish/Travel>.

Now let's summarize everything we have learned here.

## Summary

With that, we have learned what clean architecture is and how it will help developers build applications that can last for a decade or more. We also learned that clean architecture makes our service testable and ready for microservices.

We've tackled what comprises clean architecture. There is the core layer that does not have any dependencies on the infrastructure and presentation layers. An infrastructure layer communicates with external sources and the presentation layer, which your users use and interact with.

You've also learned how to structure tests in clean architecture and lastly, we learned how to use the **dotnet** CLI to build an ASP.NET Core solution and project.

Now that you've improved your skills by learning how to apply clean architecture to an ASP.NET Core 5, this will help you to build a highly scalable and testable application in the future.

In the next chapter, we will set up our database and build routing and controllers to see how they go about processing HTTP requests.

## Chapter 5: Setting Up DbContext and Controllers

Routing and controllers are responsible for receiving, validating, and processing incoming HTTP requests. During the processing, the controllers may or may not persist and read records in the database through **DbContext**. Learning how a controller and **DbContext** work together is essential in building web application projects that work as expected.

At the end of this chapter, you will be able to understand the following topics:

- Writing entities and enums
- Setting up a database, EF Core, and **DbContext**
- Writing controllers and routes
- Testing controllers with Swagger UI

## Technical requirements

Here is what you need to complete this chapter:

- Visual Studio 2019, Visual Studio for Mac, or Rider
- The **dotnet-ef** tool
- SQLite Browser or SQLiteStudio

Here is the starter repository for this chapter, which is also the finished source code of [Chapter 4](https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter05), *Applying Clean Architecture ASP.NET Core Solution*: <https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter05>.

## Writing entities and enums

Here are quick definitions of entity and enum before we start writing them in the domain project. What is an entity? An **entity** is a representation of a domain object model in an application. A database translates an entity into a row in a database table. Similarly, the properties of an entity are the columns in a database table.

On the other hand, an **enum** is a type of class representing a set of constants or read-only variables.

We have looked at a quick definition of an entity and an enum; now, let's see them in action.

## Creating entities and enums for the Travel Tour application

In your solution application, go to the **Travel.Domain** project and create two directories:

- **Entities**
- **Enums**

The **Entities** directory will be for all the Travel Tour application entities, while the **Enums** directory is for all the Travel Tour application enums:

1. In the **Entities** folder, create a C# file, **TourPackage.cs**, and write this code:

```
namespace Travel.Domain.Entities
{
    public class TourPackage
    {
        public int Id { get; set; }
        public int ListId { get; set; }
        public string Name { get; set; }
        public string WhatToExpect { get; set; }
        public string MapLocation { get; set; }
        public float Price { get; set; }
        public int Duration { get; set; }
        public bool InstantConfirmation { get; set; }
    }
}
```

The class is a **TourPackage** entity with a **namespace** of **Travel.Domain.Entities**. The **TourPackage** entity represents the package tour data in the database, which has **Id**, **ListId**, **Name**, **WhatToExpect**, **MapLocation**, **Price**, **Duration**, and **InstantConfirmation**. It also needs an enum for **Currency** and an object relationship for **List**. Now let's create these two classes.

2. In your **Enums** folder, create **Currency.cs** and write the following code:

```
namespace Travel.Domain.Enums
{
    public enum Currency
    {
        PHP,
        USD,
        JPY,
        EUR,
        NOK
    }
}
```

```
}  
}
```

The **Currency** enum file has a **namespace** of **Travel.Domain.Enums** and enumeration of **PHP**, **USD**, **JPY**, **EUR**, and **NOK**.

3. Now switch back to the **TourPackage** class to add the **Currency** enum like so:

```
using Travel.Domain.Enums;  
  
namespace Travel.Domain.Entities  
{  
    public class TourPackage  
    {  
        ...  
        public int Duration { get; set; }  
        public bool InstantConfirmation { get; set; }  
        public Currency Currency { get; set; }  
    }  
}
```

Bring in the **Travel.Domain.Enums namespace** so you can use the **Enum** class you just created by using **Travel.Domain.Enums**. Now add the **Currency** property after the **InstantConfirmation** property.

4. Next, let's create a **TourList.cs** C# file inside the **Entities** folder and write the following code:

```
using System.Collections.Generic;  
  
namespace Travel.Domain.Entities  
{  
    public class TourList  
    {  
        public TourList()  
        {  
            Tours = new List<TourPackage>();  
        }  
        public IList<TourPackage> Tours { get; set; }  
        public int Id { get; set; }  
        public string City { get; set; }  
        public string Country { get; set; }  
        public string About { get; set; }  
    }  
}
```

```
}  
}
```

Let's bring in **System.Collections.Generic** and define the **TourList** entity. The **TourList** entity has a constructor that initializes the **Tours** property with a new **List** type of **TourPackage**, **new List<TourPackage>()**, which sets a one-to-many relationship. And the rest of the properties are **Id**, **City**, **Country**, and **About**.

5. Now go back to **TourPackage.cs** and add a **List** property of type **TourList** below the **Currency** property:

```
using Travel.Domain.Enums;  
  
namespace Travel.Domain.Entities  
{  
    public class TourPackage  
    {  
        ...  
        public bool InstantConfirmation { get; set; }  
        public Currency Currency { get; set; }  
        public TourList List { get; set; }  
    }  
}
```

The **List** type **TourList** signifies a one-to-one relationship, also known as an object-to-object relationship.

Writing the entities and enums for the domain project ends here. Now let's move on to the next section to set up **Entity Framework Core (EF Core)**, **DbContext**, and the database we will use in the application.

## Setting up a database, EF Core, and DbContext

You often require a persistence framework, a middleware that helps developers store data in a database if there's a requirement to access a database in our application. Using a persistence framework, you can easily use your entities to query or save objects to a database. Now let's look at what EF Core and **DbContext** are.

## EF Core

In ASP.NET Core, you can build a persistence framework from scratch, but you don't have to because it is time-consuming and expensive. Why? Writing many stored procedures is hard to maintain;

reading data through ADO.NET objects then mapping them to your tables in your application is painful.

So, here comes Entity Framework to the rescue. **Entity Framework** is a persistence framework that does all the plumbing for you. Typically, you don't need to write any store procedures or map tables to your entities.

So what is EF Core? **EF Core** is a cross-platform **object-relational mapper (O/RM)**. EF Core itself is a version of Entity Framework, which is a persistence framework that uses the O/RM behind the scene to store and retrieve objects.

Okay, so that's an overview of EF Core; let's head to **DbContext**.

## DbContext

EF Core provides a class called **DbContext**, which is an interface to our database. **DbContext** can have one or more **DbSet** that represents tables in our database. We use **Language Integrated Query (LINQ)** to query **DbSet** entities, and then EF Core will translate our LINQ queries to SQL queries at runtime.

**DbContext** opens a connection to your database, reads and maps data to objects, and adds them to **DbSet** of **DbContext**.

As we save, update, or delete objects in **DbSet**, EF Core keeps track of the changes. When we ask to persist the changes, EF Core automatically generates SQL statements and executes them in the database.

So that's a quick overview of **DbContext**; now it's time to write **DbContext** and some **DbSet** entities in the application.

## Setup

This setup is going to be a simple EF Core and **DbContext** setup. However, we will refactor and improve this along the way:

1. First things first, install the **dotnet-ef** tool globally in your Terminal or CMD:

```
dotnet tool install --global dotnet-ef
```

The **dotnet-ef** tool is an EF Core tool for the .NET CLI. You can go to the link [nuget.org/packages/dotnet-ef](https://nuget.org/packages/dotnet-ef) to see the latest version of the **dotnet-ef** tool.

We also need to install two NuGet packages. We will be using **dotnet cli** because it runs on any operating system, but if you prefer to use **Manage NuGet Packages**, a NuGet packages installer

UI in your IDE, you can do so.

2. Now go to the **Travel.WebApi** project and run this command:

```
dotnet add package Microsoft.EntityFrameworkCore.Design
```

**Microsoft.EntityFrameworkCore.Design** is required in database migration using EF Core at runtime.

3. Next, go to the **Travel.Data** project and run this command:

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

**Microsoft.EntityFrameworkCore.Sqlite** is the SQLite database provider of EF Core.

4. Now, still inside the **Data** project, let's write **DbContext** we will need here. Create a folder named **Contexts** inside the **Travel.Data** project. Inside the folder, create a **TravelDbContext.cs** file and write the following code:

```
using Microsoft.EntityFrameworkCore;
using Travel.Domain.Entities;
namespace Travel.Data.Contexts
{
    public class TravelDbContext : DbContext
    {
        public TravelDbContext
            (DbContextOptions<TravelDbContext> options)
            : base(options)
        {
        }

        public DbSet<TourList> TourLists { get; set; }
        public DbSet<TourPackage> TourPackages { get;
            set; }
    }
}
```

The preceding code imports the **Microsoft.EntityFrameworkCore** package as well as **Travel.Domain.Entities**. We named the class **TravelDbContext** and derived it from **DbContext** of EF Core.

The constructor of **TravelDbContext** has a parameter option of the **DbContextOptions** type of **TravelDbContext**, which goes to the base keyword.

You will see the two **DbSet** entities – **TourLists** and **TourPackages** – below the constructor.

5. Now go to the **Startup.cs** file of the **Travel.WebApi** project and bring in these two namespaces:

```
using Microsoft.EntityFrameworkCore;
using Travel.Data.Contexts;
```

6. Write the following code inside the **ConfigureServices** method:

```
services.AddDbContext<TravelDbContext>(options =>
    options
        .UseSqlite("Data
            Source=TravelTourDatabase.sqlite3"));
```

The code registers **TravelDbContext** as a service in **IServiceCollection** while **UseSqlite** configures the context to connect to the SQLite database.

The **ConfigureServices** method should look like this:

```
public void ConfigureServices(IServiceCollection
    services)
{
    services.AddDbContext<TravelDbContext>(options
        => options
            .UseSqlite("Data
                Source=TravelTourDatabase.sqlite3"));
    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title =
            "Travel.WebApi", Version = "v1" });
    });
}
```

7. Now let's go back to the **Travel.Data** project for database migration.

## NOTE

*All the commands are one-liner commands. The width of a book's page adds another line to any commands that are too long.*

8. After navigating to the **Travel.Data** project, you are now ready to run your first migration:

```
dotnet ef migrations add InitialCreate --startup-project
../../presentation/Travel.WebApi/
```

EF Core will create a folder named **Migrations** in your project with generated C# files. These files are the migration builder named **InitialCreate** and a snapshot of your database's current schema.



9. Next, create a database and schema from the migration files:

```
dotnet ef database update --startup-project ../../presentation/Travel.WebApi/
```

This command creates an SQLite database file, **TravelTourDatabase.sqlite3**, in the Web API project.

10. To see the successful migration that you did, use SQLiteStudio or SQLite Browser.

11. Add the **TravelTourDatabase** SQLite database file in SQLiteStudio or SQLite Browser to view the tables:

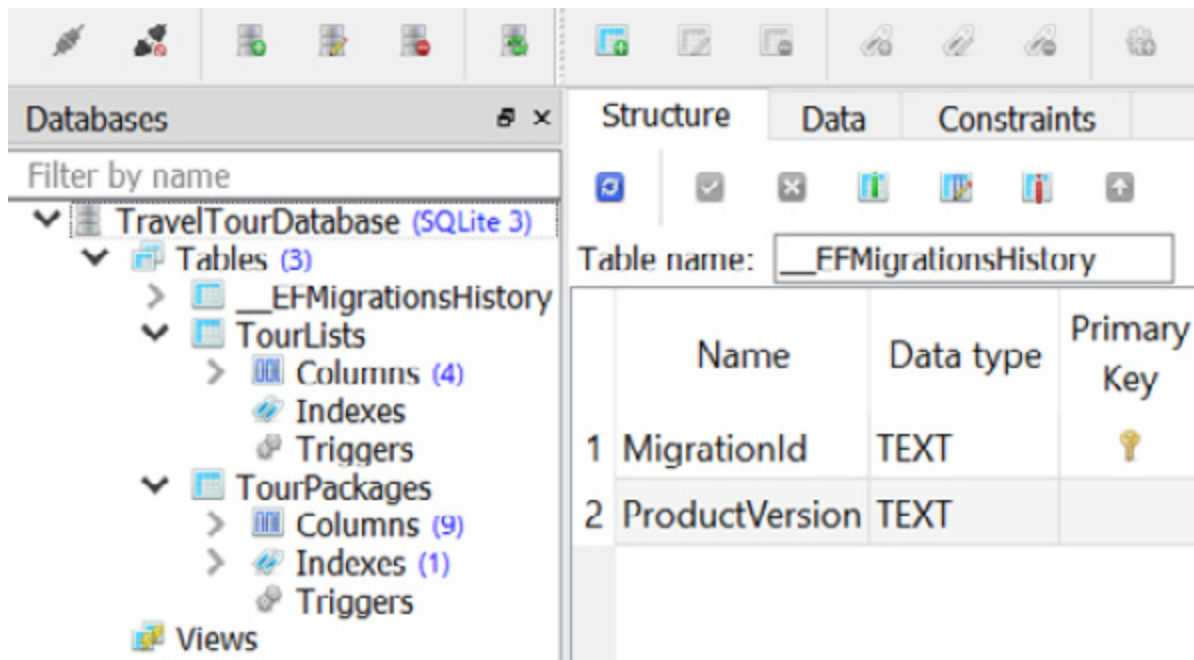


Figure 5.1 – SQLiteStudio

The preceding screenshot of SQLiteStudio shows **TravelTourDatabase**, while the following screenshot of SQLite Browser shows **TravelTourDatabase**:

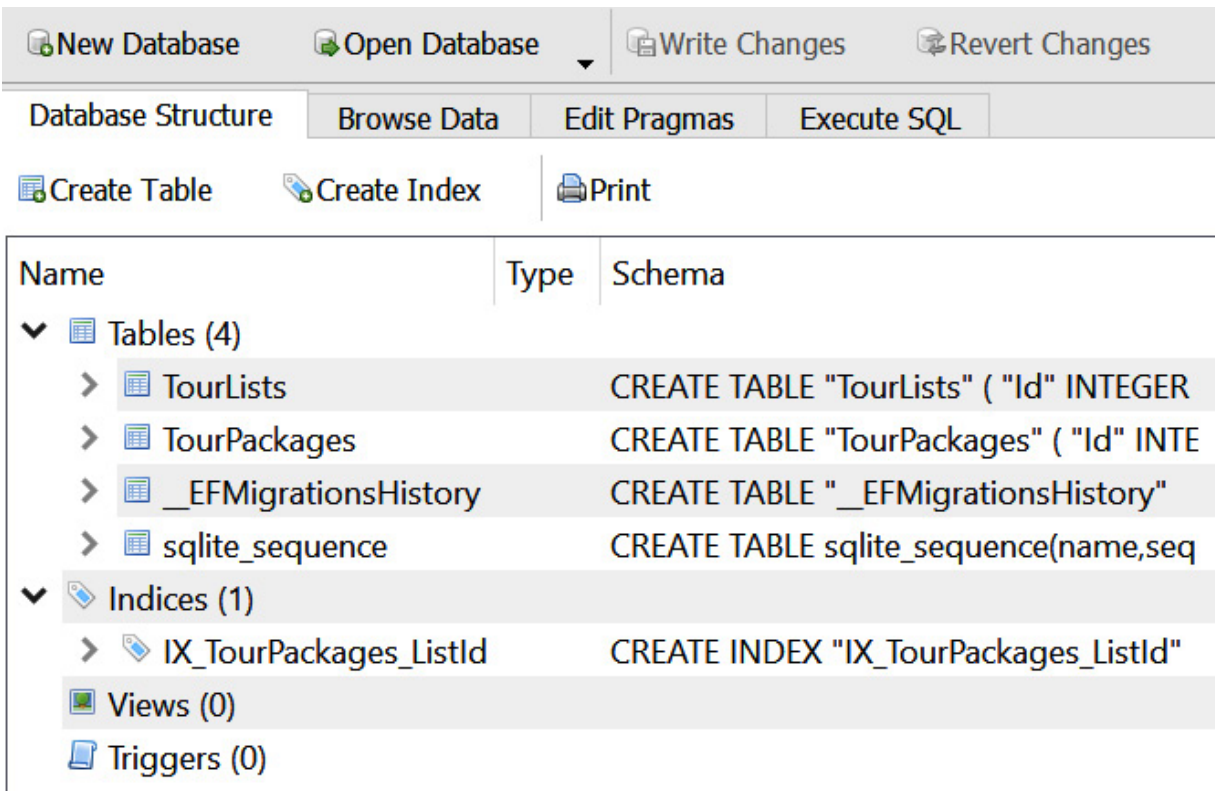


Figure 5.2 – SQLite Browser

You've just learned what entities, Entity Framework, EF Core, and **DbContext** are. You also learned where to get them and how to set them up. Now we have a database, let's move on to the next section, *Writing controller and routes*.

## Writing controllers and routes

We are now in the section where we will be writing controllers that process HTTP requests from the client application. We are also going to write routes that redirect an HTTP request to its respective controller.

These will be two simple controllers, but we will refactor them in the next chapter, [Chapter 6, Diving into CQRS](#).

## TourPackagesController

We will be creating the controller for **TourPackages** that will handle and process any requests that will retrieve and update **TourPackages** table in the database:

1. Now go back to the **Travel.WebApi** project and create **TourPackagesController.cs** inside of the **Controllers** directory and write the following code:

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;
using Travel.Data.Contexts;
using Travel.Domain.Entities;

```

The code imports the preceding **namespaces**, which are needed in this controller.

2. Next, write the **controller** class:

```

namespace Travel.WebApi.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class TourPackagesController : ControllerBase
    {
    }
}

```

We will use the **ApiController** attribute, which gives the decorated class some API behaviors such as HTTP API responses, followed by another attribute, **Route** – the routes that redirects HTTP requests.

The **TourPackagesController** class is derived from **ControllerBase**, a base class for an MVC controller, but without the view support.

3. Next, write the following code inside **TourPackagesController**:

```

    private readonly TravelDbContext _context;

    public TourPackagesController(TravelDbContext
        context)
    {
        _context = context;
    }

```

We inject **TravelDbContext** in the constructor of **TourPackagesController**, which is then assigned to a backing field **\_context**. Now let's move on to our first public method in the controller.

4. Next, we need to write the **Get** method under the constructor method:

```

    [HttpGet]
    public IActionResult Get()

```

```

{
    return Ok(_context.TourPackages);
}

```

The **HttpGet** attribute identifies an action that can support an **HttpGet** method. Also, the **Get** method returns a **200** status code along with a collection of **TourPackage**. This method gets triggered when the request's endpoint is **api/tourpackages** while using the **GET HTTP** verb request.

Now let's move on to the next public method.

- Next, we need to write the **Create** method under the **Get** method:

```

[HttpPost]
public async Task<IActionResult> Create([FromBody]
    TourPackage tourPackage)
{
    await _context.TourPackages.AddAsync
        (tourPackage);
    await _context.SaveChangesAsync();
    return Ok(tourPackage);
}

```

The **HttpPost** attribute on top of the **Create** method identifies an action that can support an **HttpPost** method. The **post** method takes a **TourPackage** object decorated with **FromBody**, specifying that a parameter should be bound using a request body.

The **Create** method returns a **200** status code along with newly created **TourPackage**. This method gets triggered when the request's endpoint is **api/tourpackages** while using the **POST HTTP** verb request.

You will also notice here **Task** of type **IActionResult**. **IActionResult** is an interface that defines a contract representing the result of an action method, while **Task** is used for asynchronous operations.

Now let's move on to the next public method of **TourPackagesController**.

- Next, we write the **Delete** method under the **Create** method:

```

[HttpDelete("{id}")]
public async Task<IActionResult>
    Delete([FromRoute] int id)
{

```

```

var tourPackage = await
    _context.TourPackages.SingleOrDefaultAsync
        (tp => tp.Id == id);
if (tourPackage == null)
{
    return NotFound();
}

_context.TourPackages.Remove(tourPackage);
await _context.SaveChangesAsync();
return Ok(tourPackage);
}

```

The **HttpDelete** attribute on top of the **Delete** method identifies an action that can support an **HttpGet** method. This method gets triggered when the request's endpoint is **api/tourpackages/{id}** while using the **DELETE HTTP** verb request.

You will notice in the **HttpDelete** attribute of the method that there's an "{id}" string. **SingleOrDefaultAsync** uses the **id** parameter, matches it with an existing object, and then returns that object to be removed from the table. **DbContext** must call **SaveChangesAsync()** to finalize the deletion of the specified **TourPackage** object.

And finally, **controller** includes deleted **TourPackage** in the **200 - OK** response.

Now let's move on to the method.

7. We need to write the **Update** method under the **Delete** method:

```

[HttpPut("{id}")]
public async Task<IActionResult>
    Update([FromRoute] int id,
        [FromBody] TourPackage tourPackage)
{
    _context.Update(tourPackage);
    await _context.SaveChangesAsync();
    return Ok(tourPackage);
}

```

The preceding code is a method that updates a **TourPackage** object.

And that's it for **TourPackagesController**. We can now move on to **TourListsController**.

## TourListsController

Now create **TourListsController** inside the **Controllers** directory. **TourListsController** has the same structure as **TourPackagesController**.

Here is the summary of the code for **TourListsController**. Go check out the code inside [Chapter 5 | Finish directory](#) of the application's GitHub repo and write it in your file:

```
using Microsoft.AspNetCore.Mvc;
...
using Travel.Domain.Entities;
namespace Travel.WebApi.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class TourListsController : ControllerBase
    {
        private readonly TravelDbContext _context;

        public TourListsController(TravelDbContext context){...}

        [HttpGet]
        public IActionResult Get(){...}

        [HttpPost]
        public async Task<IActionResult> Create([FromBody]
            TourList tourList){...}

        [HttpDelete("{id}")]
        public async Task<IActionResult> Delete([FromRoute]
            int id){...}

        [HttpPut("{id}")]
        public async Task<IActionResult> Update([FromRoute]
            int id, [FromBody] TourList tourList){...}
    }
}
```

As you can see, the code is also a class that is derived from **ControllerBase** and has **Get**, **Create**, **Delete**, and **Update** public methods.

Now we've finished writing the controllers for the application, let's run it to see our work in action, by running the following command still inside the **WebApi** project:

```
dotnet run
```

The preceding command will start or run the application. While the application is running, we can move on to the next section, *Testing controllers with Swagger UI*, to see the controllers in action.

## Testing controllers with Swagger UI

We've built controllers and run the application, but we haven't tested it yet, right? In this section, we will send **GET**, **POST**, **PUT**, and **DELETE** requests to **TourList** controller.

First of all, make sure that you are running the application by going to the **Web Api** project and running this command:

```
dotnet run
```

Now check your Swagger UI by going to the link <https://localhost:5001/swagger/index.html> while the application is running.

You will see the **TourLists** and **TourPackages** endpoints. The Swagger UI is showing you the documentation of the HTTP methods for each controller. There are **GET**, **POST**, **DELETE**, and **PUT** methods ready to be tried out:

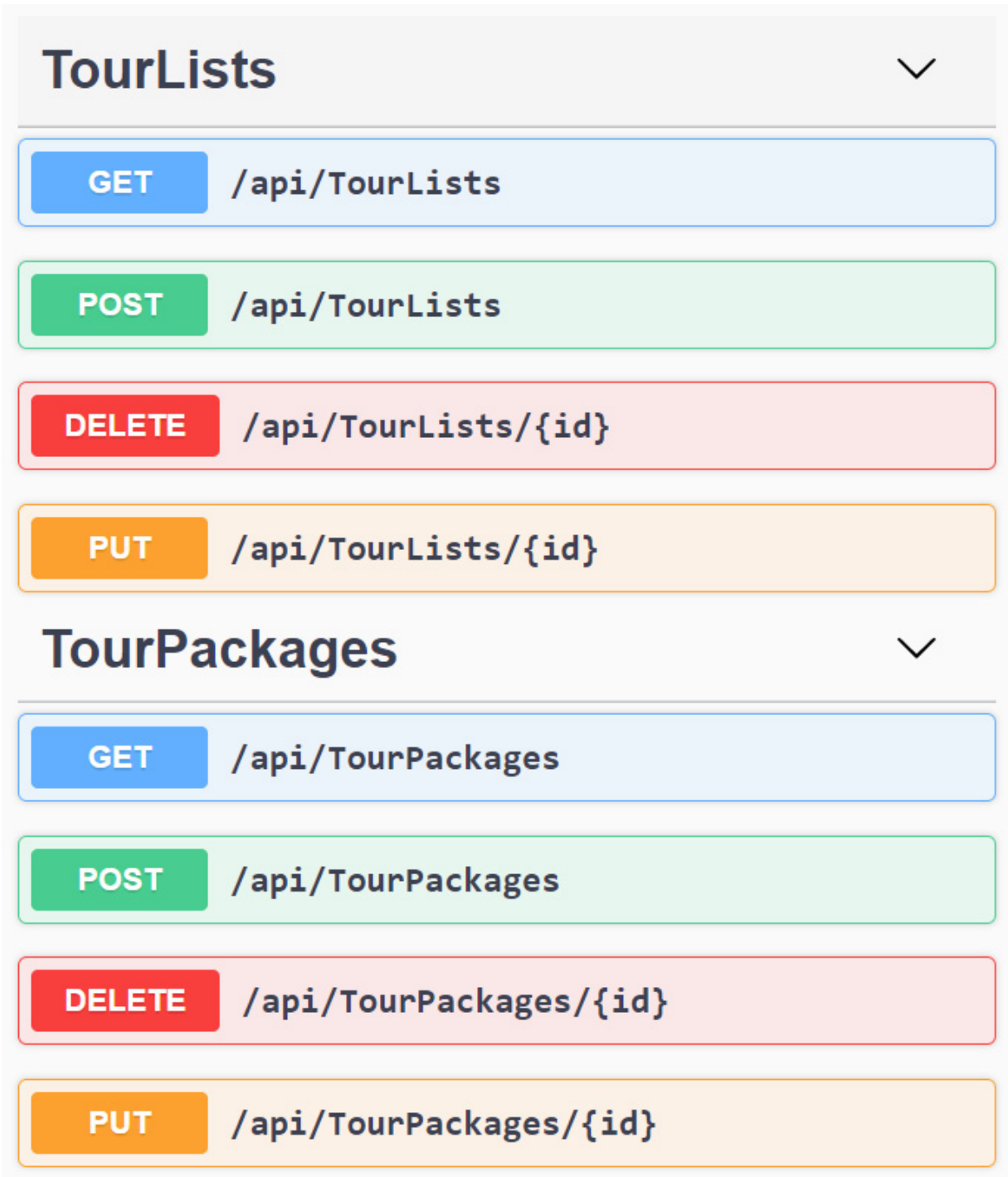


Figure 5.3 – Swagger UI

The following screenshot shows the schemas of the application, which are also generated for you by Swagger UI:



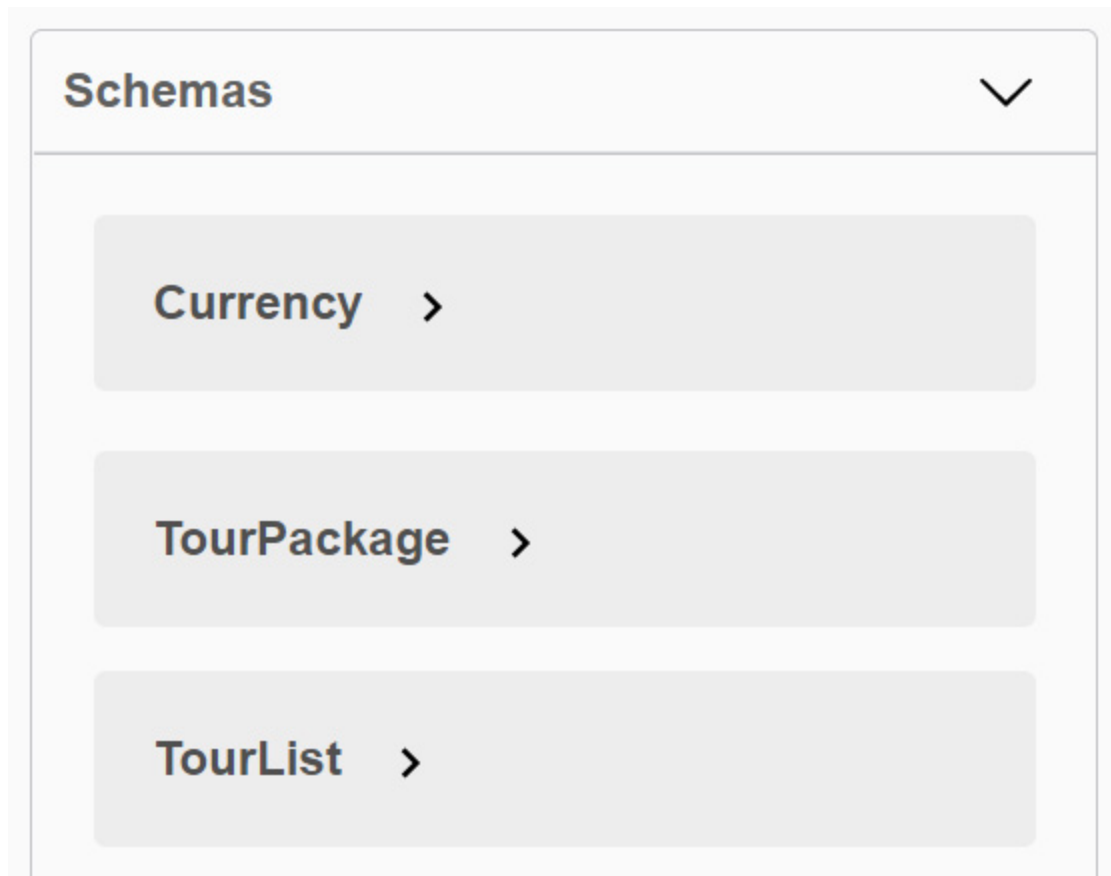


Figure 5.4 – Schemas in Swagger UI

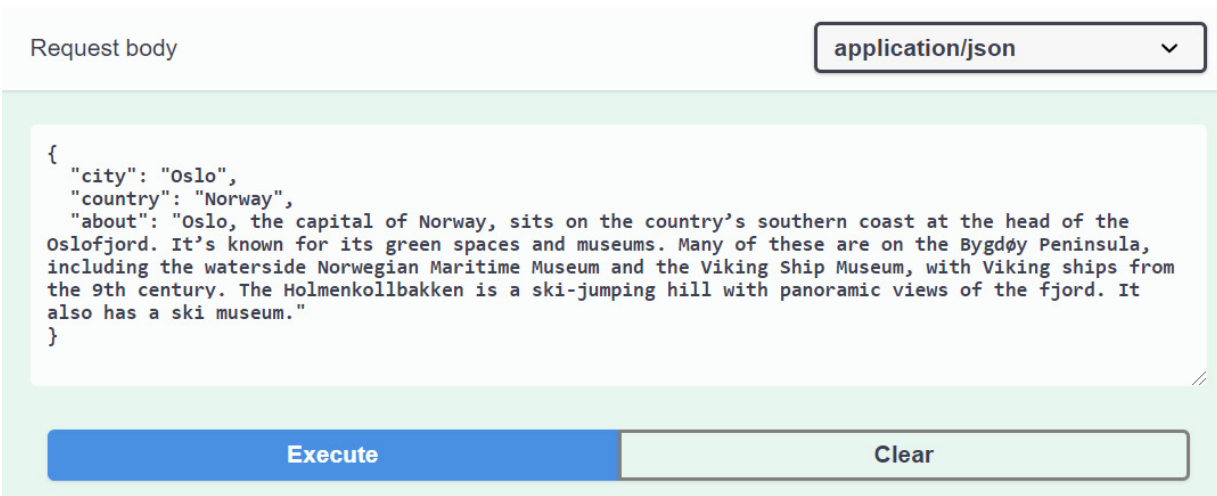
Now let's try the **GET** method of the **TourLists** controller. It will return an empty collection or array since we haven't added any data there yet.

Now go to **POST** and click the **Try it out** button. Before clicking the **Execute** button, replace the request body with the following code:

```
{  
  "city": "Oslo",  
  "country": "Norway",  
  "about": "Oslo, the capital of Norway, sits on the country's southern coast at the  
head of the Oslofjord. It's known for its green spaces and museums. Many of these are on  
the Bygdøy Peninsula, including the waterside Norwegian Maritime Museum and the Viking  
Ship Museum, with Viking ships from the 9th century. The Holmenkollbakken is a ski-  
jumping hill with panoramic views of the fjord. It also has a ski museum."  
}
```

You can copy the preceding request body, which we will use to send a **POST** request to the server from the **Commands.txt** file that you can find inside the [Chapter 5](#) folder of the application's GitHub repo.

We will only include the **city** property, the **country** property, and the **about** property in the text area for the request body, like so:



The image shows the Swagger UI interface for editing a POST request body. At the top, there's a label 'Request body' and a dropdown menu set to 'application/json'. Below this is a large text area containing a JSON object: 

```
{  "city": "Oslo",  "country": "Norway",  "about": "Oslo, the capital of Norway, sits on the country's southern coast at the head of the Oslofjord. It's known for its green spaces and museums. Many of these are on the Bygdøy Peninsula, including the waterside Norwegian Maritime Museum and the Viking Ship Museum, with Viking ships from the 9th century. The Holmenkollbakken is a ski-jumping hill with panoramic views of the fjord. It also has a ski museum."}
```

 At the bottom of the text area are two buttons: 'Execute' (in blue) and 'Clear'.

Figure 5.5 – Request body for the POST method

After replacing the **POST** part's default request body, like you can see in *Figure 5.5*, click the **Execute** button:



The image shows the Swagger UI interface for viewing a response body. At the top, there are tabs for 'Code' and 'Details', with 'Details' selected. Below the tabs, the status '200' is shown next to the label 'Response body'. The main area contains a JSON object: 

```
{  "tours": [],  "id": 1,  "city": "Oslo",  "country": "Norway",  "about": "Oslo, the capital of Norway, sits on the country's southern coast at the head of the Oslofjord. It's known for its green spaces and museums. Many of these are on the Bygdøy Peninsula, including the waterside Norwegian Maritime Museum and the Viking Ship Museum, with Viking ships from the 9th century. The Holmenkollbakken is a ski-jumping hill with panoramic views of the fjord. It also has a ski museum."}
```

 To the right of the JSON text are two buttons: a 'Download' button with a download icon and a 'Download' button with the text 'Download'. Below the JSON text is the label 'Response headers'.

Figure 5.6 – Response body from the TourLists controller

You can see in *Figure 5.6* the response body has the properties we posted and an **id** of **1**.

If you want to update the entry for **Oslo City** or delete it, you can go to Swagger UI and make the request there. However, since we'll also be using this in the application, I will leave it there as it is.

You can check out the whole finished solution of this chapter in the GitHub repository of this book.

Now we are done with this section, let's have a recap of what you have learned.

## Summary

To summarize this chapter, we have covered that entities represent domain models in an application, and we write entities in the domain layer of the application.

We've also covered EF Core, which helps developers persist and query data from the database without using SQL queries through an O/RM.

We've learned that **DbContext** of Entity Framework is an interface of the database and has **DbSet** representing a table in the database.

And lastly, we've written two controllers with **DbContext** and tested them by sending HTTP requests through Swagger UI.

You now have the skills to handle any HTTP requests sent by any client-side applications and process and save them in any type of relational database using EF Core.

In the next chapter, we will learn how to write maintainable and testable code using **CQRS**, or **Command and Query Responsibility Segregation**. So, see you later.

## Chapter 6: Diving into CQRS

If you are going to build big applications, separating the read and write operations of a data source can help the application scale up quickly in the future. The **Command and Query Responsibility Segregation (CQRS)** pattern will help you write maintainable and testable code. The pattern would also be an ideal candidate for how to separate all the requests for reading and all the requests for modifying data.

This chapter is about the CQRS pattern, the mediator pattern, and the popular MediatR NuGet package for CQRS and pipeline behavior.

We will cover the following topics:

- What is CQRS?
- What is the mediator pattern?
- What is the MediatR package?
- Why learn CQRS?
- When to use CQRS
- Drawbacks of CQRS

## Technical requirements

You will need Visual Studio 2019, Visual Studio for Mac, or Rider to complete this chapter.

## What is CQRS?

**CQRS** stands for **Command and Query Responsibility Segregation**. What this means is that commands and queries should have separate responsibilities and clear domain boundaries.

Let say you have a controller, and the controller has a few endpoints for getting data, creating data, updating data, and removing data, and these are your *GET*, *POST*, *PUT*, and *DELETE* methods in short.

Everything in the controller that gets data or does not mutate data falls under *Query*; while everything else that mutates data, such as *POST*, *PUT*, and *DELETE* requests, is classified as *Command*.

Now, your application should have a query model that handles the query for getting data from the database. It should also have a command model that handles the command for writing or deleting data in the database.

The following is a diagram of a command that goes to your application:

 Figure 6.1 – An app sending an HTTP request

Figure 6.1 – An app sending an HTTP request

The preceding *Figure 6.1* shows the UI sends a POST/UPDATE/DELETE request to an API endpoint, passing through the command handler. After the command handler, the request goes to the domain model and infrastructure to modify the database.

The segregation of queries and commands is the separation we are after. However, a mediator, which is a pattern, will help us to decouple everything.

Now, let's see another example where we can see a controller:

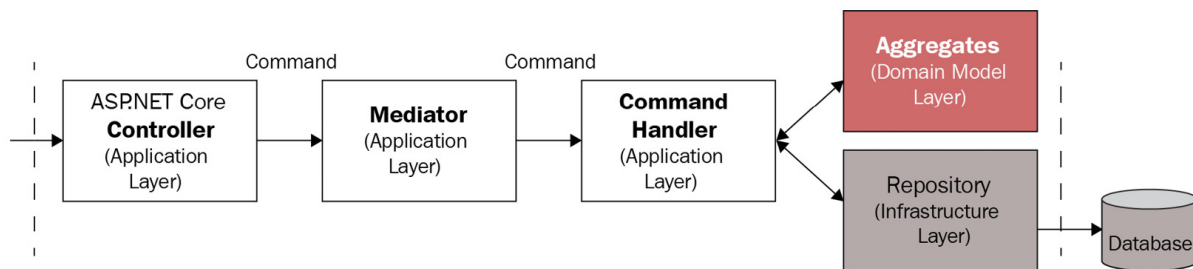


Figure 6.2 – A controller sending a command

In *Figure 6.2*, the ultimate goal is to bring a GET request and have a post, like in the preceding figure, to end up in a mediator. This mediator will find a query handler and a command handler to handle the requests, and through that, the controller won't need to know if any service is injected into handlers.

The design makes the application more manageable and way more testable because you can unit test the handlers isolated from the whole controller. After all, there is nothing in the controller. The controller just has a query sent to the mediator and eventually lands in a handler, but it's a very decoupled approach and remarkably predictable for your application.

So that is CQRS. Now, let's move on to the next section to find out what the mediator pattern is.

## What is the mediator pattern?

Suppose you have four services, objects, or elements in your application, and these services, objects, or elements need to communicate generally to each other. *ServiceA* needs to talk to *ServiceB* and *ServiceD*, and *ServiceC* needs to talk to *ServiceB* and *ServiceD*, while *ServiceB* also needs to talk to *ServiceD*. These services, objects, and elements are now tightly coupled with each other.

Here comes the Mediator to the rescue. Instead of services, objects, or elements calling each other, we will put a mediator in the middle. The mediator acts like an airport traffic control tower that knows how we want our services to communicate with each other, as shown in the following figure:



Figure 6.3 – Mediator as a traffic control tower

The mediator in *Figure 6.3* is the traffic control tower that handles the communication between the services, objects, or elements (the airplanes in this case).

So how do you do or implement the mediator pattern in ASP.NET Core? Let's find out in the next section.

## What is the MediatR package?

So, what does the NuGet MediatR package do? The MediatR package is an implementation of the mediator pattern in .NET that is ready to use.

The MediatR package is not complicated to use. You only need to remember that one request has a respective handler, which returns one response, as shown in the following figure:



Figure 6.4 – MediatR is an implementation of the mediator pattern

You can see in *Figure 6.4* that you can have several requests. One request enters the MediatR package and is followed by another one. You have the mediator in the middle, which accepts the requests and finds the appropriate handlers that need to handle those requests. Each request has one handler; the handler does all the magic and returns the right response for the correct request.

The MediatR package is like a magic box where you push the request and you don't know downstream who will handle the request. Addressing the requests now becomes the responsibility of MediatR. Hence, a reason to use MediatR is that it completely decouples the controller from the application and the business logic.

So, that is the MediatR package, a package that you can use for CQRS. By why learn CQRS? Let's find out in the next section.

## Why learn CQRS?

All your controllers will only have a single dependency, which is the MediatR package. Each one of your actions will just call a method, **Mediator.send**, and return a result, making your controllers slimmer than writing it without the CQRS pattern.

Okay, so what about when to use CQRS? Let's see in the next section.

## When to use CQRS

Why would you want to use the mediator or the CQRS patterns? There are a lot of reasons for sure, but we will only go over the apparent reasons, which are as follows:

- **Services call each other:** All the read and write requests go into the box in the middle (the mediator) and then come out. If you want to trigger any request from anywhere, the request has to hit the box in the middle, the mediator.
- **Clean code in large projects:** The use of the mediator and mediator pipelines will help you shrink your controller sizes and move your business logic into their respective files. Hence, it will be effortless to traverse the folder structure and find the logic you're looking for.
- **A boundary between writes and reads:** Any UI team can efficiently implement a UI that requires more data from the database due to the separation of reads and writes.

A UI team can work freely without worrying about affecting the backend work and how logic may distribute notifications or write to some other channel to trigger further services.

- **Portability of the code:** The mediator can trigger services that you reused in a separate project. You can do this by bringing the mediator and importing the project with services and see that your services are accessible through this box.

Okay, so these are the obvious reasons to use CQRS; the next question is whether there are any cons to using CQRS. Of course, pros have cons in most cases. The cons are covered in the next section.

## Drawbacks of CQRS

CQRS will bring extra code if you combine CQRS with the repository pattern. My suggestion is not to add another abstraction such as the repository pattern for the following reasons:

- Entity Framework already implements the repository pattern.
- There is a 99% chance that you will not change your Entity Framework implementation with NHibernate ORM or Dapper Micro ORM.

Okay, so we just finished exploring what CQRS, Mediator, and the MediatR package are. Let's summarize what you have learned here.

## Summary

You have learned that CQRS separates commands, which are requests that mutate or write data, and queries, which are requests to read data. You also learned how CQRS helps you to write slimmer controllers.

You have also learned that the Mediator design pattern acts like an air traffic controller between commands/queries and the handlers. You learned the implementation of a mediator pattern that you can use in .NET, the MediatR NuGet package, a time saver because you don't need to implement it yourself, and using Mediator makes your code cleaner and more maintainable.

You also have seen drawbacks of using CQRS, including writing extra code, but the additional code means implementing CQRS to make cleaner and more maintainable code.

In the next chapter, we will apply the CQRS pattern and the mediator pattern and use the MediatR NuGet package in ASP.NET Core 5 to build a highly scalable and maintainable Web API.



## Chapter 7: CQRS in Action

Having learned what the CQRS pattern, the Mediator pattern, and the **MediatR** NuGet package are, it's time to apply them in the ASP.NET Core 5 Web API that we are building. The patterns and package mentioned will bring value to our application by making it very scalable, testable, and readable.

In this chapter, we will cover the following topics while writing code in our application:

- Implementing CQRS
- Adding the **MediatR** package
- Creating **MediatR** pipeline behaviors
- Using **FluentValidation**
- Using **AutoMapper**
- Writing queries
- Writing commands
- Writing **IServiceCollection**

## Technical requirements

Here is what you need to complete this chapter:

- Visual Studio 2019, Visual Studio for Mac, or Rider
- The .NET CLI

You can go to the following link to see the finished source code for this chapter:

<https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter07>.

## Implementing CQRS

Here are the steps on how to use the **MediatR** package in the ASP.NET Core application. The task of the **MediatR** package is to help you implement both the CQRS and Mediator patterns with ease.

Let's clean up our solution first by deleting all the **Class1.cs** files you can find in the application. The **Class1.cs** files were generated when we created the projects.

## Adding the MediatR package

We are now going to install the **MediatR** package:

1. Navigate to your **Travel.Application** project using the **dotnet** CLI. Then we need to install some NuGet packages by running the following commands:

```
dotnet add package MediatR
```

The preceding command installs the mediator implementation in .NET into the **Travel.Application** project.

The following command installs the **MediatR** extensions for ASP.NET Core:

```
dotnet add package MediatR.Extensions.Microsoft.DependencyInjection
```

The following command installs the logging abstractions for **Microsoft.Extensions.Logging**:

```
dotnet add package Microsoft.Extensions.Logging.Abstractions
```

The preceding command installs **Microsoft.Extensions.Logging.Abstractions**, a package for creating loggers in a .NET application.

2. While in the **Travel.Application** project, create a directory and name it **Common**. Then, create a directory in the **Common** folder, and name it **Behaviors**.

Now, let's create four C# files and name them **LoggingBehavior.cs**, **PerformanceBehavior.cs**, **UnhandledExceptionBehavior.cs**, and **ValidationBehavior.cs**. All of them will be inside the **Behaviors** folder, which will be a pipeline for pre-processing and post-processing of any requests that go to our handlers.

## **NOTE**

*I am going to truncate any unnecessary code here. Please go to the GitHub repository of this chapter and refer to see each file's whole source code.*

## Creating MediatR pipeline behaviors

A pipeline behavior in **MediatR** acts like middleware between the requests and the handlers. An excellent example of this is validation, so that the handlers will only deal with necessary and valid requests. You can also do your logging and other stuff here, depending on the problem you are solving.

We will write our first **MediatR** pipeline behavior here:

```
// LoggingBehavior.cs
using MediatR.Pipeline;

...

namespace Travel.Application.Common.Behaviors
{
    public class LoggingBehavior<TRequest> :
```

```

IRequestPreProcessor<TRequest>
{
    ...

    public async Task Process(TRequest request,
        CancellationToken cancellationToken)
    {
        var requestName = typeof(TRequest).Name;
        _logger.LogInformation("Travel Request: {@Request}",
            requestName, request);
    }
}

```

The code for **LoggingBehavior.cs** is for logging requests using the **Microsoft.Extensions.Logging** and **MediatR.Pipeline** namespaces.

The **IRequestPreProcessor** is an interface for a defined request to be pre-processed for a handler, while **IRequest** is a marker to represent a request with a response that you will also find in the commands and queries for mapping later.

The following code logs the responses' elapsed time in milliseconds of **PipelineBehavior**. **PipelineBehavior** wraps the inner handler and adds an implementation for additional behavior in the request:

```

// PerformanceBehavior.cs

using MediatR;

...

namespace Travel.Application.Common.Behaviors
{
    internal class PerformanceBehavior<TRequest, TResponse> :
        IPipelineBehavior<TRequest, TResponse>
    {
        ...

        public async Task<TResponse> Handle(TRequest request,
            CancellationToken cancellationToken,
            RequestHandlerDelegate<TResponse> next)
        {

```

```

...
_logger.LogWarning("Travel Long Running Request:
    {Name} ({ElapsedMilliseconds} milliseconds)
    {@Request}",
    requestName, elapsedMilliseconds, request);
return response;
}
}
}

```

You will also find here **RequestHandlerDelegate next**, which calls the next thing in the pipeline. The **next** keyword is common in a middleware implementation that means go to the next function that modifies the request:

```

// UnhandledExceptionBehavior.cs
using MediatR;
...
namespace Travel.Application.Common.Behaviors
{
    public class UnhandledExceptionBehavior<TRequest,
        TResponse> : IPipelineBehavior<TRequest, TResponse>
    {
        private readonly ILogger<TRequest> _logger;
        ...
        public async Task<TResponse> Handle(TRequest request,
            CancellationToken cancellationToken,
            RequestHandlerDelegate<TResponse> next)
        {
            Try { return await next(); }
            catch (Exception ex)
            {
                var requestName = typeof(TRequest).Name;
                _logger.LogError(ex, "Travel Request: Unhandled
                    Exception for Request {Name} {@Request}",
                    requestName, request);
            }
        }
    }
}

```

```

        throw;
    }
}
}
}
}

```

There is one more thing that is missing in **PipelineBehavior**. That is a validation mechanism, **ValidationBehavior.cs**, which we will add next.

We are still in the **Travel.Application** project. Let's add two more NuGet packages, which involve **FluentValidation**.

## Using FluentValidation

**FluentValidation** gives you full control when creating data validation. Hence, it is very useful for all validation scenarios:

1. Let's add the **FluentValidation** package to our application:

```
dotnet add package FluentValidation
```

This preceding command installs the popular validation library for .NET. The package uses a fluent interface to build strongly typed rules.

2. The following command installs the dependency injection extensions for **FluentValidation**:

```
dotnet add package FluentValidation.DependencyInjectionExtensions
```

Now we can create another behavior, and this will be for validation. The following code validates the requests inside **PipeLineBehavior** by using **IValidator**, which defines a validator for a specific type, and **ValidationContext**, which creates an instance of the new validation context, from the **FluentValidation** namespace:

```

// ValidationBehavior.cs
using FluentValidation;
using MediatR;
using ValidationException = Travel.Application.Common.Exceptions.ValidationException;
...
namespace Travel.Application.Common.Behaviors
{
    public class ValidationBehavior<TRequest, TResponse> :
        IPipelineBehavior<TRequest, TResponse>
        where TRequest : IRequest<TResponse>

```

```

{
    private readonly IEnumerable<IValidator<TRequest>>
        _validators;

    public ValidationBehavior
        (IEnumerable<IValidator<TRequest>> validators)
    { _validators = validators; }

    public async Task<TResponse> Handle(TRequest request,
        CancellationToken cancellationToken,
        RequestHandlerDelegate<TResponse> next)
    {
        if (!_validators.Any()) return await next();

        var context = new
            ValidationContext<TRequest>(request);

        var validationResults = await Task.WhenAll
            (_validators.Select(v => v.ValidateAsync(context,
                cancellationToken)));

        var failures = validationResults.SelectMany(r =>
            r.Errors).Where(f => f != null).ToList();

        ...

        return await next();
    }
}

```

Let's now create another directory and name it **Exceptions** inside the **Common** directory of the **Travel.Application** project because this folder will be the location for our *not-found* and *validation* exceptions.

After creating the **Exceptions** folder, create two C# files – **NotFoundException.cs** and **ValidationException.cs**:

```

// NotFoundException.cs

using System;

namespace Travel.Application.Common.Exceptions
{
    public class NotFoundException : Exception

```

```

{
    public NotFoundException()
        : base() { }

    public NotFoundException(string message)
        : base(message) { }

    public NotFoundException(string message, Exception
        innerException)
        : base(message, innerException) { }

    public NotFoundException(string name, object key)
        : base($"Entity \"{name}\" ({key}) was not found.") { }
}
}

```

The preceding code is an overloading method for throwing **NotFoundException** with the meaning inside the commands. You will do this shortly.

The following code is for the exception of one or more validation failures that have occurred:

```

// ValidationException.cs

using FluentValidation.Results;
...

namespace Travel.Application.Common.Exceptions
{
    public class ValidationException : Exception
    {
        public ValidationException()
            : base("One or more validation failures have
                occurred.")
        {
            Errors = new Dictionary<string, string[]>();
        }

        public ValidationException(
            IEnumerable<ValidationFailure> failures)
            : this()
        {
            var failureGroups = failures

```

```

        .GroupBy(e => e.PropertyName, e => e.ErrorMessage);
    foreach (var failureGroup in failureGroups)
    { ... }
}

public IDictionary<string, string[]> Errors { get; }
}
}

```

Let's also create an **Interfaces** directory inside the **Common** folder. This directory will be the location of the interfaces for our two simple services.

After creating the **Interfaces** folder, let's create the two C# files: **IDateTime.cs** and **IEmailService.cs**:

```

// IDateTime.cs

using System;

namespace Travel.Application.Common.Interfaces
{
    public interface IDateTime
    {
        DateTime NowUtc { get; }
    }
}

```

The preceding code is a contract of the **DateTime** service that we will create later:

```

// IEmailService.cs

...

namespace Travel.Application.Common.Interfaces
{
    public interface IEmailService
    {
        Task SendAsync(EmailDto emailRequest);
    }
}

```

The preceding code is a contract of the **EmailService** service that we will create later.

Now, let's create an interface in the **Common** folder of **Travel.Application** for **DbContext**, but first we need to install the **EntityFrameworkCore** package:



```
dotnet add package Microsoft.EntityFrameworkCore
```

The preceding **dotnet** CLI command will install the Entity Framework Core NuGet package.

Now let's create the interface in **IapplicationDbContext**:

```
...
namespace Travel.Application.Common.Interfaces
{
    public interface IApplicationDbContext
    {
        DbSet<TourList> TourLists { get; set; }
        DbSet<TourPackage> TourPackages { get; set; }
        Task<int> SaveChangesAsync(CancellationToken
            cancellationToken);
    }
}
```

The preceding code is a contract for **TravelDbContext**, which we created in [Chapter 5, Setting Up DbContext and Controllers](#). We will update **TravelDbContext** later once we are going to implement the contract.

We just finished setting up validations using Fluent API, now let's proceed with mapping data transfer objects to entities and vice versa.

## Using AutoMapper

**AutoMapper** is a popular library that uses a convention-based, object-to-object mapper. Hence, AutoMapper lets you map objects without writing a ton of code, which you will see in a bit.

Now we need to set up our mapping of objects to objects automatically using the **AutoMapper** NuGet package, which is authored by the same person who wrote the **MediatR** library. I love **AutoMapper** because it simplifies mapping and projections.

The following command installs **AutoMapper**:

```
dotnet add package AutoMapper
```

The following command installs the **AutoMapper** extensions for ASP.NET Core:

```
dotnet add package AutoMapper.Extensions.Microsoft.DependencyInjection
```

Now let's create the files for the mapper. Create a new directory and name it **Mappings** inside **Common** of the **Travel.Application** project.

After creating the **Mappings** folder, create two C# files: **ImapFrom.cs** and **MappingProfile.cs**:

// IMapFrom.cs

```
using AutoMapper;

namespace Travel.Application.Common.Mappings
{
    public interface IMapFrom<T>
    {
        void Mapping(Profile profile) =>
            profile.CreateMap(typeof(T), GetType());
    }
}
```

The preceding code is an interface for applying mappings from the assembly. You will notice that there is a **Profile** type that is being passed here from **AutoMapper**. **Profile** is a configuration that will do the mapping for you based on naming conventions.

The preceding code allows us to group mapping configurations:

// MappingProfile.cs

```
using AutoMapper;

...

namespace Travel.Application.Common.Mappings
{
    public class MappingProfile : Profile
    {
        public MappingProfile()
        {
            ApplyMappingsFromAssembly(
                Assembly.GetExecutingAssembly());
        }

        private void ApplyMappingsFromAssembly(Assembly
            assembly)
        {
            var types = assembly.GetExportedTypes()
                .Where(t => t.GetInterfaces().Any(i =>
                    i.IsGenericType && i.GetGenericTypeDefinition()

```

```

        == typeof(IMapFrom<>)))
    .ToList();
foreach (var type in types)
{
    var instance = Activator.CreateInstance(type);
    var methodInfo = type.GetMethod("Mapping")
        ??
        type.GetInterface("IMapFrom`1").GetMethod
            ("Mapping");
    methodInfo?.Invoke(instance, new object[] { this
    });
}
}
}
}

```

Next, we create a directory named **TourLists** inside the **Travel.Application** project. Then, create another directory with the name **Queries** inside the **TourLists** folder. Finally, we create a directory named **ExportTours** inside the **Queries** folder.

After creating the three nested directories, create a C# file named **TourItemFileRecord.cs** inside the **Queries** folder:

```

using Travel.Application.Common.Mappings;
...
namespace Travel.Application.TourLists.Queries.ExportTours
{
    public class TourPackageRecord : IMapFrom<TourPackage>
    {
        public string Name { get; set; }
        public string MapLocation { get; set; }
    }
}

```

The **CsvFileBuilder** file that we will create will need the preceding code to create the type of parameter.

Let's create another C# interface file, **ICsvFileBuilder.cs**, in the **Interfaces** folder of the **Common** directory:

```
...
namespace Travel.Application.Common.Interfaces
{
    public interface ICsvFileBuilder
    {
        byte[] BuildTourPackagesFile
            (IEnumerable<TourPackageRecord> records);
    }
}
```

The preceding code is a contract for **CsvFileBuilder**, which we are going to create later.

Now let's create two more C# files inside the **ExportTours** folder. One is the **ExportToursVm.cs** file, and the other is **ExportToursQuery.cs**, which we will see in the following section:

```
// ExportToursVm.cs
namespace Travel.Application.TourLists.Queries.ExportTours
{
    public class ExportToursVm
    {
        public string FileName { get; set; }
        public string ContentType { get; set; }
        public byte[] Content { get; set; }
    }
}
```

The preceding code is a View Model that we will use for the file builder later.

So now, after writing the **ExportToursVm** class, we can move on to the next section, which is how to write queries.

## Writing queries

We will now create our first query, which is a request for reading data, and a query handler, which will resolve what the query needs. Again, the controller is responsible for sending or dispatching the queries to the relevant handlers.

The following is a block of code for **ExportToursQuery** and its handler. You will see later how the controller will use **ExportToursQuery** as an argument in the mediator's **Send** method:

```
// ExportToursQuery.cs

using AutoMapper;
using AutoMapper.QueryableExtensions;
using MediatR;
...
namespace Travel.Application.TourLists.Queries.ExportTours
{
    public class ExportToursQuery : IRequest<ExportToursVm>
    {
        public int ListId { get; set; }
    }

    public class ExportToursQueryHandler :
        IRequestHandler<ExportToursQuery, ExportToursVm>
    {
        ...
        public async Task<ExportToursVm>
            Handle(ExportToursQuery request, CancellationToken
                cancellationToken)
        {
            var vm = new ExportToursVm();
            ...
            vm.ContentType = "text/csv";
            vm.FileName = "TourPackages.csv";
            return await Task.FromResult(vm);
        }
    }
}
```

You create a handler by deriving from **IRequestsHandler** of the **MediatR** package. If you notice, we are using **DbContext** and **AutoMapper** here to process the request and create the response for the **MediatR** package.

Next, create a **Dtos** folder in the **root** directory of the **Travel.Application** project. The **Dtos** folder will have the same level as the **Common** directory.

After creating the **Dtos** folder, let's now create two C# files inside it:

```
// TourListDto.cs

...

namespace Travel.Application.Dtos.Tour
{
    public class TourListDto : IMapFrom<TourList>
    {
        public TourListDto()
        {
            Items = new List<TourPackageDto>();
        }
        public IList<TourPackageDto> Items { get; set; }
        public int Id { get; set; }
        public string City { get; set; }
        public string About { get; set; }
    }
}
```

The preceding code is a **Data Transfer Object**, or **DTO**, for **TourList**. The following code is a DTO for **TourPackage**:

```
// TourPackageDto.cs

using AutoMapper;

...

namespace Travel.Application.Dtos.Tour
{
    public class TourPackageDto : IMapFrom<TourPackage>
    {
        public int Id { get; set; }

        ...

        public void Mapping(Profile profile)
        {
            profile.CreateMap<TourPackage, TourPackageDto>()

```

```

        .ForMember(d =>
            d.Currency, opt =>
                opt.MapFrom(s =>
                    (int)s.Currency));
    }
}
}

```

In the same way as the **Dtos** directory, let's also create another directory inside the **Travel.Application** project and name it **TourLists**. Then, create a folder named **GetTours** inside the **TourLists** directory.

After creating the **GetTours** folder, create two C# files inside it:

```

// ToursVm.cs

using System.Collections.Generic;
using Travel.Application.Dtos.Tour;
namespace Travel.Application.TourLists.Queries.GetTours
{
    public class ToursVm
    {
        public IList<TourListDto> Lists { get; set; }
    }
}

```

The preceding code is a View Model for **Tours** for **GetToursQuery** that we are about to create now:

```

// GetToursQuery.cs

...
namespace Travel.Application.TourLists.Queries.GetTours
{
    public class GetToursQuery : IRequest<ToursVm> { }
    public class GetToursQueryHandler :
        IRequestHandler<GetToursQuery, ToursVm>
    {
        ...
        public async Task<ToursVm> Handle(GetToursQuery
            request, CancellationToken cancellationToken)

```

```

{
    return new ToursVm
    {
        Lists = await _context.TourLists
            .ProjectTo<TourListDto>
                (_mapper.ConfigurationProvider)
            .OrderBy(t => t.City)
            .ToListAsync(cancellationToken)
    };
}
}
}

```

The preceding code is a handler for **GetToursQuery**, which the mediator sends from the controller. We will update **TourListsController** later to be able to do that.

Now let's move to the next interesting section, *Writing commands*.

## Writing commands

We will now create our first command, which is a request for saving, updating, or deleting data, and a command handler, which will resolve what the command needs. Again, the controller is responsible for sending or dispatching the commands to the relevant handlers.

Now we are in the part where we are going to create commands for **TourLists** and **TourPackages**.

Let's create a folder inside the **TourLists** directory of the **Tour.Application** project and name it **Commands**. Then, let's create three folders inside that folder and name them **CreateTourList**, **DeleteTourList**, and **UpdateTourList**.

Now it's time to create some commands and command validators. Create two C# files inside the **CreateTourList** folder:

```

// CreateTourListCommand.cs

using MediatR;

...

namespace Travel.Application.TourLists.Commands.CreateTourList
{
    public partial class CreateTourListCommand :
        IRequest<int>

```



```

{
...
}

public class CreateTourListCommandHandler :
    IRequestHandler<CreateTourListCommand, int>
{
    private readonly IApplicationDbContext _context;

    public CreateTourListCommandHandler
        (IApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<int> Handle(CreateTourListCommand
        request, CancellationToken cancellationToken)
    {
        var entity = new TourList { City = request.City };
        _context.TourLists.Add(entity);
        await _context.SaveChangesAsync(cancellationToken);
        return entity.Id;
    }
}
}

```

The preceding code is a command for creating new **TourList**, and we are using the **MediatR** package here:

```

// CreateTourListCommandValidator.cs
using FluentValidation;
...
namespace Travel.Application.TourLists.Commands.CreateTourList
{
    public class CreateTourListCommandValidator :
        AbstractValidator<CreateTourListCommand>
    {
        private readonly IApplicationDbContext _context;

        public CreateTourListCommandValidator
            (IApplicationDbContext context)
    {
    }
}

```

```

    {
        _context = context;
        RuleFor(v => v.City)
            ...
            .NotEmpty().WithMessage("About is required");
    }
}
}

```

The preceding code is a validator of **CreateTourListCommand**, and we are using **RuleFor** here from the **FluentValidation** package.

**RuleFor** is a builder of validation rules for a particular property. That said, **FluentValidation** is a validation library that replaces Data Annotations. You should use this instead of Data Annotations because it helps you to write clean and maintainable code.

Now, create a C# file inside the **DeleteTourList** folder:

```

// DeleteTourListCommand.cs
using MediatR;
...
namespace Travel.Application.TourLists.Commands.DeleteTourList
{
    public class DeleteTourListCommand : IRequest
    {
        public int Id { get; set; }
    }

    public class DeleteTourListCommandHandler :
        IRequestHandler<DeleteTourListCommand>
    {
        private readonly IApplicationDbContext _context;
        public DeleteTourListCommandHandler
            (IApplicationDbContext context)
        {
            _context = context;
        }

        public async Task<Unit> Handle(DeleteTourListCommand

```

```

        request, CancellationToken cancellationToken)
    {
        var entity = await _context.TourLists
            .Where(l => l.Id == request.Id)
            .SingleOrDefaultAsync(cancellationToken);

        ...

        _context.TourLists.Remove(entity);

        await _context.SaveChangesAsync(cancellationToken);

        return Unit.Value;
    }
}

```

The preceding code is a command for deleting **TourList**, and we are using the **MediatR** package here. The **Unit** type here is from the **MediatR** package, which signals a no return value. Since **void** is not a valid return type, the **Unit** type represents a void type.

Now, create two C# files inside the **UpdateTourList** folder:

// UpdateTourListCommand.cs

```

using MediatR;

...

namespace Travel.Application.TourLists.Commands.UpdateTourList
{
    public class UpdateTourListCommand : IRequest
    {
        ...
    }

    public class UpdateTourListCommandHandler :
        IRequestHandler<UpdateTourListCommand>
    {
        private readonly IApplicationDbContext _context;

        public UpdateTourListCommandHandler
            (IApplicationDbContext context)
        { _context = context; }

        public async Task<Unit> Handle(UpdateTourListCommand

```

```

        request, CancellationToken cancellationToken)
    {
        var entity = await
            _context.TourLists.FindAsync(request.Id);
        ...
        entity.City = request.City;
        await _context.SaveChangesAsync(cancellationToken);
        return Unit.Value;
    }
}

```

The preceding code is a command for updating existing **TourList** data in the database. Again, the **Unit** type is being used here:

// UpdateTourListCommandValidator.cs

```

using FluentValidation;
...
namespace Travel.Application.TourLists.Commands.UpdateTourList
{
    public class UpdateTourListCommandValidator :
        AbstractValidator<UpdateTourListCommand>
    {
        private readonly IApplicationDbContext _context;
        public UpdateTourListCommandValidator
            (IApplicationDbContext context)
        {
            _context = context;
            RuleFor(v => v.City)
                .NotEmpty().WithMessage("City is required.")
                ...
        }
    }
}

```

This preceding code is a validator of **UpdateTourListCommand**, and we are using the **FluentValidation** package here.

Now for **TourPackage**. Let's create a folder in the root folder of the **Travel.Application** project, just as we did for **TourLists**, and name it **TourPackages**.

After creating the **TourPackages** directory, let's create a folder inside the **TourPackages** directory and name it **Commands**. Then, let's create four folders inside that folder and name them **CreateTourPackage**, **DeleteTourPackage**, **UpdateTourPackage**, and **UpdateTourPackageDetail**.

Now, create two C# files inside the **CreateTourPackage** folder:

```
// CreateTourPackageCommand.cs

using MediatR;

...

namespace Travel.Application.TourPackages.Commands.CreateTourPackage
{
    public class CreateTourPackageCommand : IRequest<int>
    {
        ...
    }

    public class CreateTourPackageCommandHandler :
        IRequestHandler<CreateTourPackageCommand, int>
    {
        private readonly IApplicationDbContext _context;

        public CreateTourPackageCommandHandler
            (IApplicationDbContext context)
        { _context = context; }

        public async Task<int> Handle(CreateTourPackageCommand
            request, CancellationToken cancellationToken)
        {
            var entity = new TourPackage { ... };
            _context.TourPackages.Add(entity);

            await _context.SaveChangesAsync(cancellationToken);

            return entity.Id;
        }
    }
}
```

```
}
```

The preceding code is a command for creating new **TourPackage**, and we are using the **MediatR** package here:

```
// CreateTourPackageCommandValidator.cs

using FluentValidation;

...

namespace Travel.Application.TourPackages.Commands.CreateTourPackage
{
    public class CreateTourPackageCommandValidator :
        AbstractValidator<CreateTourPackageCommand>
    {
        private readonly IApplicationDbContext _context;

        public CreateTourPackageCommandValidator
            (IApplicationDbContext context)
        {
            _context = context;

            RuleFor(v => v.Name)
                .NotEmpty().WithMessage("Name is required.")
                ...
        }

        public async Task<bool> BeUniqueName(string name,
            CancellationToken cancellationToken)
        {
            return await _context.TourPackages
                .AllAsync(l => l.Name != name);
        }
    }
}
```

The preceding code is a validator of **CreateTourPackageCommand**, and we are using the **FluentValidation** package here.

Now create a C# file inside the **DeleteTourList** folder:

```
// DeleteTourPackageCommand.cs

using MediatR;
```

```

...
namespace Travel.Application.TourPackages.Commands.DeleteTourPackage
{
    public class DeleteTourPackageCommand : IRequest
    {
        public int Id { get; set; }
    }

    public class DeleteTourPackageCommandHandler :
        IRequestHandler<DeleteTourPackageCommand>
    {
        private readonly IApplicationDbContext _context;

        public DeleteTourPackageCommandHandler
            (IApplicationDbContext context)
        {
            _context = context;
        }

        public async Task<Unit> Handle(DeleteTourPackageCommand
            request, CancellationToken cancellationToken)
        {
            var entity = await
                _context.TourPackages.FindAsync(request.Id);

            ...

            _context.TourPackages.Remove(entity);

            await _context.SaveChangesAsync(cancellationToken);

            return Unit.Value;
        }
    }
}

```

The preceding code is a command for deleting **TourPackage**, and we are using the **MediatR** package here.

Now create two C# files inside the **UpdateTourPackage** folder:

```
// UpdateTourPackageCommand.cs
```

```
using MediatR;
```

```

...
namespace Travel.Application.TourPackages.Commands.UpdateTourPackage
{
    public partial class UpdateTourPackageCommand : IRequest
    {
        ...
    }

    public class UpdateTourPackageCommandHandler :
        IRequestHandler<UpdateTourPackageCommand>
    {
        private readonly IApplicationDbContext _context;

        public UpdateTourPackageCommandHandler
            (IApplicationDbContext context)
        { _context = context; }

        public async Task<Unit> Handle(UpdateTourPackageCommand
            request, CancellationToken cancellationToken)
        {
            var entity = await
                _context.TourPackages.FindAsync(request.Id);

            ...

            entity.Name = request.Name;

            await _context.SaveChangesAsync(cancellationToken);

            return Unit.Value;
        }
    }
}

```

The preceding code is a command for updating **TourPackage**, and we are using the **MediatR** package here.

```

// UpdateTourPackageCommandValidator.cs
using FluentValidation;

...
namespace Travel.Application.TourPackages.Commands.UpdateTourPackage
{

```



```

public class UpdateTourPackageCommandValidator :
    AbstractValidator<UpdateTourPackageCommand>
{
    private readonly IApplicationDbContext _context;

    public UpdateTourPackageCommandValidator
        (IApplicationDbContext context)
    {
        _context = context;

        RuleFor(v => v.Name)

            ...

            .MustAsync(BeUniqueName).WithMessage("The specified
                name already exists.");
    }

    public async Task<bool> BeUniqueName(string name,
        CancellationToken cancellationToken)
    {
        return await _context.TourPackages
            .AllAsync(l => l.Name != name);
    }
}

```

The preceding code is a validator of **UpdateTourPackageCommand**, and we are using the **FluentValidation** package here.

Now create two C# files inside the **UpdateTourPackageDetail** folder:

```

// UpdateTourPackageDetail.cs
using MediatR;

...

namespace Travel.Application.TourPackages.Commands.UpdateTourPackageDetail
{
    public class UpdateTourPackageDetailCommand : IRequest
    {
        ...
    }
}

```

```

public class UpdateTourPackageDetailCommandHandler :
    IRequestHandler<UpdateTourPackageDetailCommand>
{
    private readonly IApplicationDbContext _context;
    public UpdateTourPackageDetailCommandHandler
        (IApplicationDbContext context)
    {
        _context = context;
    }
    public async Task<Unit> Handle
        (UpdateTourPackageDetailCommand request,
         CancellationToken cancellationToken)
    {
        var entity = await
            _context.TourPackages.FindAsync(request.Id);
        ...
        await _context.SaveChangesAsync(cancellationToken);
        return Unit.Value;
    }
}

```

The preceding code is another command for updating **TourPackage**, and we are using the **MediatR** package here.

// UpdateTourPackageDetailCommandValidator.cs

```

using FluentValidation;
...
namespace Travel.Application.TourPackages.Commands.UpdateTourPackageDetail
{
    public class UpdateTourPackageDetailCommandValidator :
        AbstractValidator<UpdateTourPackageDetailCommand>
    {
        private readonly IApplicationDbContext _context;
        public UpdateTourPackageDetailCommandValidator

```

```

        (IApplicationDbContext context)
    {
        _context = context;
        ...
        RuleFor(v => v.Currency)
            .NotEmpty().WithMessage("Currency is required");
    }

    public async Task<bool> BeUniqueName(string name,
        CancellationToken cancellationToken)
    {
        return await _context.TourPackages
            .AllAsync(l => l.Name != name);
    }
}

```

The preceding code is the validator of **UpdateTourPackageDetailCommand**, and we are using the **FluentValidation** package here.

Now let's move on to the next section, which is about writing **IServiceCollection**.

## Writing IServiceCollection

**IServiceCollection** is an interface from the **DependencyInjection** namespace. We are going to use **IServiceCollection** for our dependency injection.

And finally, there is a dependency injection for the **Travel.Application** project. Create a C# file in the **root** folder of the **Travel.Application** project:

```

// DependencyInjection.cs
...
namespace Travel.Application
{
    public static class DependencyInjection
    {
        public static IServiceCollection AddApplication(this
            IServiceCollection services)
        {

```

```

        services.AddAutoMapper
            (Assembly.GetExecutingAssembly());
        services.AddValidatorsFromAssembly
            (Assembly.GetExecutingAssembly());
        services.AddMediatR(Assembly.GetExecutingAssembly());
        services.AddTransient(typeof(IPipelineBehavior<,>),
            typeof(PerformanceBehavior<,>));
        services.AddTransient(typeof(IPipelineBehavior<,>),
            typeof(ValidationBehavior<,>));
        services.AddTransient(typeof(IPipelineBehavior<,>),
            typeof(UnhandledExceptionBehavior<,>));
        return services;
    }
}
}

```

The preceding code is a dependency injection container method. You will see here that **IServiceCollection** is adding different kinds of services to the collection of the service descriptors.

We will inject the static method, **AddApplication**, later in the **Startup** file so that the Web API project, particularly the **Startup** file, won't need to declare any dependencies on third-party libraries such as AutoMapper because they are already declared in this file.

Now let's go to the **Travel.Domain** project to add a **settings** file for the **Mail** service. Create a folder inside the **Travel.Domain** project and name it **Settings**.

After creating the **Settings** directory, create a C# file inside it:

```

// MailSettings.cs
namespace Travel.Domain.Settings
{
    public class MailSettings
    {
        public string EmailFrom { get; set; }
        ...
        public string DisplayName { get; set; }
    }
}

```

The preceding code is for the settings of an email service that we are going to create later.

Now let's add one more exception file to the **Travel.Application** project. Go to the **Common** directory of the **Travel.Application** project and create a C# file:

```
// ApiException.cs
...
namespace Travel.Application.Common.Exceptions
{
    public class ApiException : Exception
    {
        public ApiException() : base() { }
        public ApiException(string message) : base(message) { }
        public ApiException(string message, params object[]
            args)
            : base(String.Format(CultureInfo.CurrentCulture,
                message, args)) { }
    }
}
```

The preceding code is for an exception that we are going to use in our email service later.

Let's go to the **Travel.Shared** project to add some NuGet packages to this project:

```
dotnet add package MailKit
```

The preceding CLI command installs **MailKit**, a .NET mail-client library. The following CLI command installs a package that provides an additional configuration in the ASP.NET Core app:

```
dotnet add package Microsoft.Extensions.Options.ConfigurationExtensions
```

The following CLI command installs **Mimekit**, a package for creating and parsing S/MIME, MIME, and PGP messages:

```
dotnet add package MimeKit
```

The following CLI command installs **CsvHelper**, a package for reading and writing CSV files:

```
dotnet add package CsvHelper
```

After installing the NuGet packages, let's create two new folders – **Files** and **Services**, in the **root** directory of the **Travel.Shared** project.

Inside the **Files** folder, let's create a C# file:

```
// CsvFileBuilder.cs
```

```

using CsvHelper;
using System.IO;
...
namespace Travel.Shared.Files
{
    public class CsvFileBuilder : ICsvFileBuilder
    {
        public byte[] BuildTourPackagesFile
            (IEnumerable<TourPackageRecord> records)
        {
            using var memoryStream = new MemoryStream();
            ...
            return memoryStream.ToArray();
        }
    }
}

```

The preceding code is an implementation of **CsvFileBuilder**.

Now let's create two C# files in the **Services** folder:

// EmailService.cs

```

using MimeKit;
using MailKit.Net.Smtp;
...
namespace Travel.Shared.Services
{
    public class EmailService : IEmailService
    {
        ...
        public async Task SendAsync(EmailDto request)
        {
            try
            {
                var email = new MimeMessage { Sender =
                    MailboxAddress.Parse(request.From ??
                        MailSettings.EmailFrom) };
            }
        }
    }
}

```

```

        email.To.Add(MailboxAddress.Parse(request.To));

        ...

        await smtp.DisconnectAsync(true); }
    catch (System.Exception ex)
    { Logger.LogError(ex.Message, ex);
        throw new ApiException(ex.Message); }
    }
}
}

```

The preceding code is an implementation of **IEmailService**.

// DateTimeService.cs

```

...

namespace Travel.Shared.Services
{
    public class DateTimeService : IDateTime
    {
        public DateTime NowUtc => DateTime.UtcNow;
    }
}

```

The preceding code is an implementation of **IDateTime**. It only returns **DateTime.UtcNow**.

And in the case of the dependency injection of the **Travel.Shared** project, let's create a C# file in the **root** directory of the **Travel.Shared** project:

// DependencyInjection.cs

```

using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
...

namespace Travel.Shared
{
    public static class DependencyInjection
    {
        public static IServiceCollection
            AddInfrastructureShared(this IServiceCollection
                services, IConfiguration config)
    }
}

```

```

    {
        services.Configure<MailSettings>
            (config.GetSection("MailSettings"));
        services.AddTransient<IDateTime, DateTimeService>();
        services.AddTransient<IEmailService, EmailService>();
        services.AddTransient<ICsvFileBuilder,
            CsvFileBuilder>();
        return services;
    }
}

```

The preceding code is another dependency injection container method that we will add later in the **Startup** file.

Now it's time for the final step. Go to the **Travel.WebApi** project and update the **appSettings.json** file:

```

// appSettings.json
{
    "Logging": {
        "LogLevel": {
            ...
        }
    },
    "MailSettings": {
        "EmailFrom": "",
        ...
        "DisplayName": ""
    },
    "AllowedHosts": "*"
}

```

This code adds **MailSettings** in **appsettings.json**.

Next, we create a C# file inside the **Controllers** folder of the **Travel.WebApi** project:

```

// ApiController.cs
using MediatR;

```



```

using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.DependencyInjection;
namespace Travel.WebApi.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public abstract class ApiController : ControllerBase
    {
        private IMediator _mediator;

        protected IMediator Mediator => _mediator ??=
            HttpContext.RequestServices.GetService<IMediator>();
    }
}

```

The preceding code is a property injection to allow **ApiController** to use Mediator. I prefer this approach over the constructor injection because of its simplicity. Property injection frees you up from maintaining all the controllers' parameters and signatures using the constructor injection.

Next, we update the **TourPackagesController** and **TourListController** files in the **Travel.WebApi** project:

```

// TourPackagesController.cs
...
namespace Travel.WebApi.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class TourPackagesController : ApiController
    {
        [HttpPost]
        public async Task<ActionResult<int>>
            Create(CreateTourPackageCommand command)
        { return await Mediator.Send(command); }

        [HttpPut("{id}")]
        public async Task<ActionResult> Update(int id,
            UpdateTourPackageCommand command)
        { }
    }
}

```

```

{
    ...
    await Mediator.Send(command);
}
[HttpPut("[action]")]
public async Task<ActionResult> UpdateItemDetails(int
    id, UpdateTourPackageDetailCommand command)
{
    ...
    await Mediator.Send(command);
}
[HttpDelete("{id}")]
public async Task<ActionResult> Delete(int id)
{
    await Mediator.Send(new DeleteTourPackageCommand { Id
        = id });
}
}
}

```

This updated code of **TourPackagesController** uses Mediator to send commands and is derived from **ApiController**.

// TourListsController.cs

```

...
namespace Travel.WebApi.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class TourListsController : ApiController
    {
        [HttpGet]
        public async Task<ActionResult<ToursVm>> Get()
        {
            return await Mediator.Send(new GetToursQuery());
        }
    }
}

```

```

    }

    [HttpGet("{id}")]
    public async Task<FileResult> Get(int id)
    {
        var vm = await Mediator.Send(new ExportToursQuery {
            ListId = id });
        return File(vm.Content, vm.ContentType, vm.FileName);
    }

    [HttpPost]
    public async Task<ActionResult<int>>
        Create(CreateTourListCommand command)
    {
        return await Mediator.Send(command);
    }

    [HttpPut("{id}")]
    public async Task<ActionResult> Update(int id,
        UpdateTourListCommand command)
    {
        await Mediator.Send(command);
    }

    [HttpDelete("{id}")]
    public async Task<ActionResult> Delete(int id)
    {
        await Mediator.Send(new DeleteTourListCommand { Id =
            id });
    }
}
}

```

This updated code of **TourListsController** uses Mediator to send commands and is derived from **ApiController**.

After updating **TourListsController**, we will add a filter for our API by creating a new folder in the **root** directory of the **Travel.WebApi** project and name it **Filter**. Then, create a C# file, **ApiExceptionHandler.cs**, inside the newly created folder and add the following code for **ApiExceptionHandler**:

```

using System;

...

using Microsoft.AspNetCore.Mvc.Filters;
using Travel.Application.Common.Exceptions;
namespace Travel.WebApi.Filters
{
    public class ApiExceptionFilter :
        ExceptionFilterAttribute
    {
        private readonly IDictionary<Type,
            Action<ExceptionContext>> _exceptionHandlers;

        public ApiExceptionFilter()
        {
            ...
        }

        public override void OnException(ExceptionContext
            context)
        {
            HandleException(context);
            base.OnException(context);
        }

        private void HandleException(ExceptionContext
            context)
        {
            ...
        }

        private void HandleUnknownException
            (ExceptionContext context)
        {
            ...
        }

        private void HandleValidationException
            (ExceptionContext context)
        {

```

```

        ...
    }
    private void HandleNotFoundException
        (ExceptionContext context)
    {
        ...
    }
}

```

There is nothing special in either method of the preceding class. Please go to the GitHub repository of the preceding code to see the implementations of the methods. The API exception filter that we created will handle any **NotFoundException**, **ValidationException**, and **UnknownException** that our Web API will generate by catching errors and treating them in a consistent manner.

And finally, the last thing to do is to update the **Startup.cs** file of the Web API project:

```

// Startup.cs
using Microsoft.Extensions.DependencyInjection;
using Travel.Application;
using Travel.Data;
using Travel.Shared;
using Microsoft.AspNetCore.Mvc;
using Travel.WebApi.Filters;
...
namespace Travel.WebApi
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        { Configuration = configuration; }
        public IConfiguration Configuration { get; }
        public void ConfigureServices(IServiceCollection
            services)
        {
            services.AddApplication();

```

```

        services.AddInfrastructureData();
        services.AddInfrastructureShared(Configuration);
        services.AddHttpContextAccessor();
        services.AddControllersWithViews(options =>
            options.Filters.Add(new ApiExceptionHandler()));
        services.Configure<ApiBehaviorOptions>(options =>
            options.SuppressModelStateInvalidFilter = true
        );
        ...
    }

    // This method gets called by the runtime. Use this
    // method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        ...
    }
}

```

This updated code of the **Startup.cs** file registers the services of the application, data, and shared projects through dependency injections. The updated code is an elegant way of keeping our container bootstrapping in our startup project without violating the object-oriented design principles.

Now we can check whether our controllers are wired up correctly. Run the application and then go to Swagger UI. Try out the **GET** request of the **TourLists** controller:

 Figure 7.1 – Using Swagger UI to test the TourLists controller

Figure 7.1 – Using Swagger UI to test the TourLists controller

The response from testing the **GET** request of **TourLists** here can be seen in *Figure 7.1*. You can see that the controller responded with **200** with a **TourList** object.

Now let's wrap this chapter up by summarizing everything that we've learned.

## Summary

The entire contents here amounts to quite a chapter. Let's summarize the essential parts.

You have finally seen how to apply **CQRS**, **MediatR**, and **Pipeline Behavior**. The **MediatR** package makes the CQRS pattern easy to do in ASP.NET Core. The **Pipeline Behavior** package

allows you to run a number of methods, such as validations or loggings, in a command before and after a handler processes it.

You learned how to use the **FluentValidation** package, a powerful library for validating your models.

You also learned how to use the **AutoMapper** package, a library that allows you to map an object to another object by writing a few lines of code.

Lastly, you saw how to use **IServiceCollection** to create a clean dependency injection in the **Startup.cs** file.

With this, we have made the ASP.NET Core 5 application more testable and scalable. In the next chapter, we will use Serilog for logging in ASP.NET Core 5, and we will also implement API versioning.

## Chapter 8: API Versioning and Logging in ASP.NET Core

Welcome to another chapter. This chapter will teach you about logging API requests, which is an essential part of any application because logging brings benefits to developers and business people. You will also learn about API versioning, which is sometimes necessary to create maintainable APIs but can be problematic if not done correctly.

We will cover the following topics:

- API versioning
- Logging in ASP.NET Core

### Technical requirements

Here is what you need to complete this chapter:

- Visual Studio 2019, Visual Studio for Mac, or Rider
- SQLite Browser or SQLiteStudio

The link to the finished repository can be found here: <https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter08>.

### API versioning

So what we are going to learn about in this section is a few things about versioning ASP.NET Core Web API. We will start by talking about what API versioning is and some versioning strategies in APIs, then we will dive into some code and integrate API versioning with OpenAPI so you can see the versioned APIs in your Swagger UI.

Now let's start by defining API versioning.

### What is API versioning?

In a nutshell, **API versioning** is how you want to deal with API changes over time. Most of us backend developers concentrate on building and deploying a web service because that is our job. They're usually the written tasks that we have to implement in a typical project. But sometimes we forget that we're going to have to support the web service for the next 5 to 10 years, or the developers replacing us will.



Once the web service is deployed and developers have coded against the API, there will be users who are relying on the APIs that we have built. For instance, a mobile app with a specific version would expect the web service's endpoints to not ever change. Having a plan and strategy for dealing with APIs that don't change is vital because your requirements and the features of your application will continue to evolve. You need a way to change these APIs without breaking existing client-side applications.

And that's the idea or core of what API versioning is about.

So how do we version our APIs? Let's go to the next section.

## API versioning strategies

There are many ways to do this, but we will only tackle the top three most used API versioning strategies or schemes.

Which one should you choose? It depends on who uses your API, how your API is going to evolve, and what kind of technical requirements you need. Find something that works for your use case.

Okay, so how do we deprecate some of our APIs?

### URI path versioning

The **URI path strategy** is popular because it is simple to implement. Somewhere in the URI path, you insert a version indicator such as **v1** or **v2** near the root, and then the rest of the path follows. Here's an example:

```
https://traveltour.xyz/api/v1/customers
```

The preceding example would be version 1 of the API and version 2 would be given as follows:

```
https://traveltour.xyz/api/v2/customers
```

### NOTE

*The cache uses the URI as a key to invalidate when switching the API version so that the content itself in the cache will match the specific version of the API is returning.*

URI path versioning is also more common in other frameworks such as Laravel in PHP, Spring Boot in Java, and Ruby on Rails than query-based versioning or header-based versioning.

We will apply URI versioning in our application in a bit. Now let's go on to the next popular API scheme or strategy.

### Header versioning

Versioning with headers is where you have a verb, and you have a header value that indicates the version developers looking for.

Here is sample content of the header where you can specify the version:

```
GET /api/customers HTTP/1.1
Host: localhost:5001
Content-Type: application/json
x-api-version: 2
```

The preceding header content shows no filter is added to the URI.

This strategy can be useful as it doesn't corrupt the URIs. However, consuming these kinds of APIs on the client side would require a bit of sophistication to send the correct requests and headers to the web service.

## Query string versioning

Query string versioning is where you use a query string to specify the version of your API. It allows the consumers of the APIs to change the version as they need to. They can either opt into an old version or opt into a new version of the APIs.

Remember that you should have an implicit default version of your APIs if there is no query string in the request. Here's an example of query string versioning:

```
https://traveltour.xyz/api/customers?api-version=2
```

As you can see, **?api-version=2** means this request is for version 2 of the customer's route and controller.

Okay, so is there a way to let the developers know if an API is no longer recommended for use? Yes, it is a deprecated API, as you will see in the next section.

## Deprecating an API

A deprecated API is an API that is no longer recommended for developers to use. And once a version of an API has zero users for a number of months, that's the time when you can remove that version from your application.

So, to mark an API as deprecated is dead simple. Here's how:

```
[ApiVersion("1.0", Deprecated = true)]
```

This is an annotation from the **Microsoft.AspNetCore.Mvc** namespace telling developers that API version 1 is now deprecated.

Let's see in the next section how to implement versioning, the deprecation of an API, and integrating them with OpenAPI for visual representation in Swagger UI.

# API versioning integration with OpenAPI

Now we are in the part where we will implement the API versioning, deprecate an API, and integrate it with OpenAPI. All changes are going to happen inside the **Travel.WebAPI** project:

1. Go to **Travel.WebAPI** and install two packages. The following NuGet library adds an API versioning service for ASP.NET Core:

```
Microsoft.AspNetCore.Mvc.Versioning
```

The following NuGet library adds functionality for discovering metadata of API-versioned controllers and actions, as well as URLs and allowed HTTP methods:

```
Microsoft.AspNetCore.Mvc.Versioning.ApiExplorer
```

2. Next, let's create two new folders inside the **Controller** directory and name them **v1** and **v2**. Then move all the following controllers inside **v1**:

**ApiController.cs**

**TourListsControllers.cs**

**TourPackagesControllers.cs**

**WeatherForecastController.cs**

Update the namespace of each controller to **namespace Travel.WebApi.Controllers.v1** and derive **WeatherForecastController** from **ApiController**.

3. Next, we update the annotations on top of the **ApiController** abstract class in **ApiController.cs**. The code should be as follows:

```
namespace Travel.WebApi.Controllers.v1
{
    [ApiVersion("1.0")]
    [ApiController]
    [Route("api/v{version:apiVersion}/{controller}")]
    public abstract class ApiController :
        ControllerBase
    {
        private IMediator _mediator;

        protected IMediator Mediator => _mediator ??=
            HttpContext.RequestServices.GetService
                <IMediator>();
    }
}
```

We added the **apiVersion** annotation and specified the version. We also updated the **Route** annotation with a dynamic **api** version between **api** and **controller**. These two annotations will also be automatically applied to any controllers derived from the **ApiController** class.

4. Now let's deprecate the **WeatherForecast** endpoint by updating **WeatherForecastController** inside the **Controller** directory:

```
namespace Travel.WebApi.Controllers.v1
{
    [ApiVersion("1.0", Deprecated = true)]
    public class WeatherForecastController :
        ApiController
    {
        ...
    }
}
```

The annotation means we are explicitly tagging **WeatherForecastController** as deprecated.

5. After deprecating, let's create a new version of the **WeatherForecast** endpoint by creating a new C# file inside the **v2** directory of the controller and name it **WeatherForecast.cs**.

After creating a new **WeatherForecast controller** file inside the **v2** folder, update the code with this code:

```
namespace Travel.WebApi.Controllers.v2
{
    [ApiVersion("2.0")]
    [ApiController]
    [Route("api/v{version:apiVersion}/{controller}")]
    public class WeatherForecastController :
        ControllerBase
    {
        ...

        [HttpPost]
        public IEnumerable<WeatherForecast>
            Post(string city)
        {
            var rng = new Random();
            return Enumerable
                .Range(1, 5)
```

```

        .Select(index => new WeatherForecast
        {
            ...
            City = city
        })
        .ToArray();
    }
}

```

If you noticed, the difference between **v1 WeatherForecast** and **v2 WeatherForecast** is the **HTTP** method. You have to send a *GET* request to get the date and temperature data in version 1, while you have to send a *POST* request with a query parameter of **city** in version 2. This is an excellent example of a breaking change in the API. Consequently, the API must have versioning to avoid breaking the applications that are consuming the first version of the **WeatherForecast** API.

The *POST* request with **query** is not a best practice because it is non-idempotent and is more appropriate to use in a *POST* request, while *GET*, *PUT*, and *DELETE* are for idempotent requests. However, the *POST* request with a **query** parameter is the easiest way to break the **WeatherForecast** endpoint. So just bear with me now.

Now, for the OpenAPI, let's create a new folder in the root of **Travel.WebApi** and name it **Helpers**. After creating the folder, create two C# files, **ConfigureSwaggerOptions.cs** and **SwaggerDefaultValues.cs**, inside of that folder:

```

// ConfigureSwaggerOptions.cs

using System;

using Microsoft.AspNetCore.Mvc.ApiExplorer;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Options;

using Microsoft.OpenApi.Models;

using Swashbuckle.AspNetCore.SwaggerGen;

namespace Travel.WebApi.OpenApi
{
    public class ConfigureSwaggerOptions :
        IConfigurationOptions<SwaggerGenOptions>
    {

```

```

...

    public void Configure(SwaggerGenOptions
        options)
    {
        ...
    }

    private static OpenApiInfo
        CreateInfoForApiVersion
            (ApiVersionDescription description)
        {
            ...
        }
    }
}

```

We need to write two methods here. Let's name the first method **Configure()** and the second one **OpenApiInfo()**. The following is the code block of the **Configure()** method:

```

public void Configure(SwaggerGenOptions options)
{
    foreach (var description in
        _provider.ApiVersionDescriptions)
    {
        options.SwaggerDoc
            (description.GroupName,
                CreateInfoForApiVersion
                    (description));
    }
}

```

What the **Configure()** method does is that it adds a Swagger document for every discovered API version. And here is the code block for the **OpenApiInfo()** method:

```

private static OpenApiInfo
    CreateInfoForApiVersion
        (ApiVersionDescription description)
    {
        var info = new OpenApiInfo
        {

```

```

        Title = "Travel Tour",
        Version =
            description.ApiVersion.ToString(),
        Description = "Web Service for Travel
            Tour.",
        Contact = new OpenApiContact
        {
            Name = "IT Department",
            Email =
                "developer@traveltour.xyz",
            Url = new Uri
                ("https://traveltour.xyz/support")
        }
    };
    if (description.IsDeprecated)
        info.Description += " <strong>This API
            version of Travel Tour has
                been deprecated.</strong>";
    return info;
}

```

Basically, this code is for the Swagger information such as **title**, **version**, **description**, **contact name**, **contact email**, and **URL** of the application:

```

// SwaggerDefaultValues.cs
using System.Linq;
using Microsoft.AspNetCore.Mvc.ApiExplorer;
using Microsoft.OpenApi.Any;
using Microsoft.OpenApi.Models;
using Swashbuckle.AspNetCore.SwaggerGen;
namespace Travel.WebApi.OpenApi
{
    public class SwaggerDefaultValues :
        IOperationFilter
    {

```

```

        public void Apply(OpenApiOperation operation,
            OperationFilterContext context)
        {
            ...
        }
    }
}

```

We're going to use this code to replace the configuration we wrote inside the **services.AddSwaggerGen()** method inside **Startup.cs**. And here is the block of code for the **Apply()** method:

```

public void Apply(OpenApiOperation operation, OperationFilterContext context)
{
    var apiDescription =
        context.ApiDescription;
    operation.Deprecated |=
        apiDescription.IsDeprecated();
    if (operation.Parameters == null)
        return;
    foreach (var parameter in
        operation.Parameters)
    {
        var description = apiDescription.
            ParameterDescriptions.First(
                pd => pd.Name == parameter.Name);
        parameter.Description ??=
            description.ModelMetadata.Description;
        if (parameter.Schema.Default == null
            &&
            description.DefaultValue != null)
            parameter.Schema.Default = new
                OpenApiString
                    (description.DefaultValue.ToString());
        parameter.Required |=
            description.IsRequired;
    }
}

```



```

    }
}

```

The **Apply()** method allows the Swagger generator to add all the relevant metadata of the API explorer.

- Now let's update the **Startup.cs** file. Find the **AddSwaggerGen()** method inside **ConfigureServices** and update it with the following code:

```

services.AddSwaggerGen(c =>
{
    c.OperationFilter<SwaggerDefaultValues>();
});

```

The preceding code allows the Swagger generator to modify the operations right after the operations are initially generated using a filter, which is **SwaggerDefaultValues** we created earlier.

Next, add a service lifetime for the **ConfigureSwaggerOptions()** method that we created earlier:

```

services.AddTransient<IConfigureOptions<SwaggerGenOptions>, ConfigureSwaggerOptions>();

```

Place the **AddTransient()** method under the **AddSwaggerGen()** method block.

We can now add our **ApiVersioning** configuration from **Microsoft.AspNetCore.Mvc.Versioning** that we installed. Place this under the **AddTransient()** method:

```

services.AddApiVersioning(config =>
{
    config.DefaultApiVersion = new
        ApiVersion(1, 0);
    config.
        AssumeDefaultVersionWhenUnspecified
        = true;
    config.ReportApiVersions = true;
});

```

The preceding code adds **ApiVersioning** in the service collection. **ApiVersioning** declares the default API version and *api-supported-versions* in the header of the API's response.

Next, we add **API Explorer** from the **Microsoft.AspNetCore.Mvc.Versioning.ApiExplorer** NuGet package that we installed. Place this under the **AddApiVersioning()** method:

```

services.AddVersionedApiExplorer(options =>

```

```
{
    options.GroupNameFormat = "'v'VVV";
    options.SubstituteApiVersionInUrl = true;
});
```

The code adds an API explorer that understands the API versions in the application. It adds a format like this: "'v'**major**[.**minor**][**-status**]'".

Now add a parameter in the **Configure()** method. Name it **provider** and the type is **IApiVersionDescriptionProvider**, like so:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
IApiVersionDescriptionProvider provider)
{
    ...
}
```

The preceding code discovers and describes the information about the API versions within the application.

Let's also update the **UseSwaggerUI()** method inside the **Configure()** method with this code:

```
app.UseSwaggerUI(c =>
{
    foreach (var description in provider.ApiVersionDescriptions)
    {
        c.SwaggerEndpoint($"/swagger/
        {description.GroupName}/swagger.json",
        description.GroupName.ToUpperInvariant
        ());
    }
});
```

This code will build a Swagger endpoint for every discovered API version by looping the provider.

- Now let's run the application and see the results of the code we've added to our application. Let's look at Swagger UI and test the version 1 **WeatherForecast** API and the version 2 **WeatherForecast** API to see if they are working correctly if we send a request.

You can see in the following screenshot, *Figure 8.1*, that you can pick which version of your APIs you want to examine:

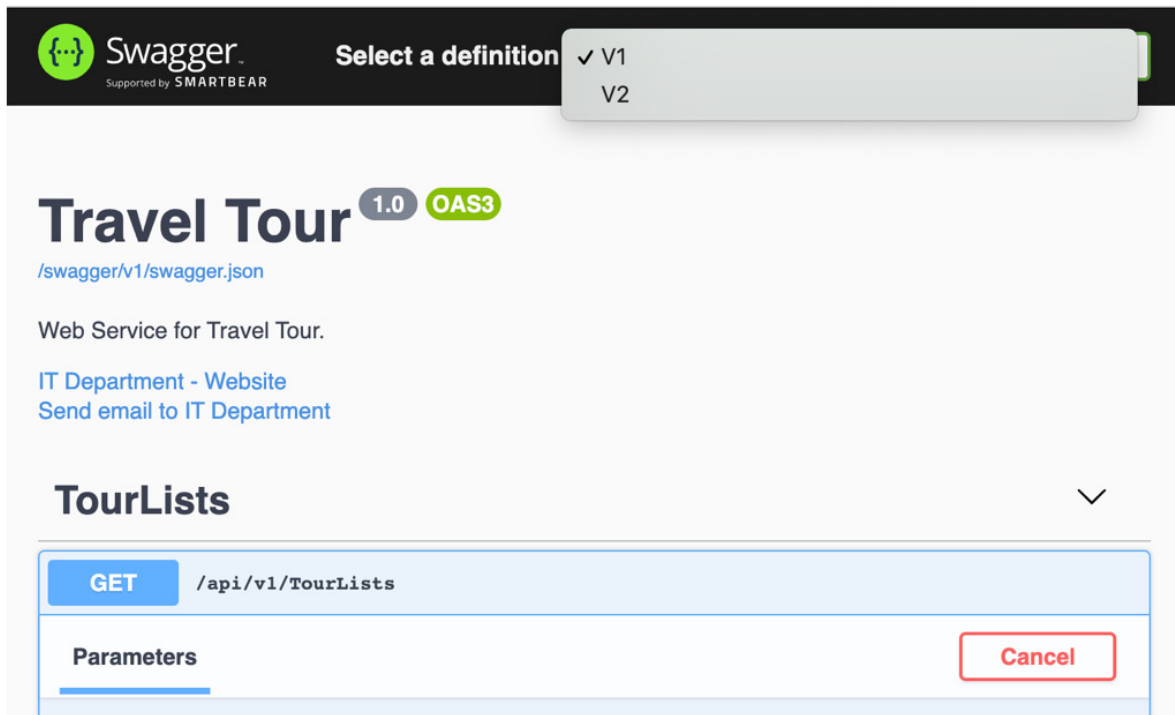


Figure 8.1 – Version picker

Figure 8.1 shows a dropdown menu that lets you choose which version of APIs to see.

Now let's check out the **WeatherForecast** controller in version 1. You will notice in the next image, Figure 8.2, that version 1 **WeatherForecast** is different from version 2 with regard to how Swagger UI presents the details of the endpoints:

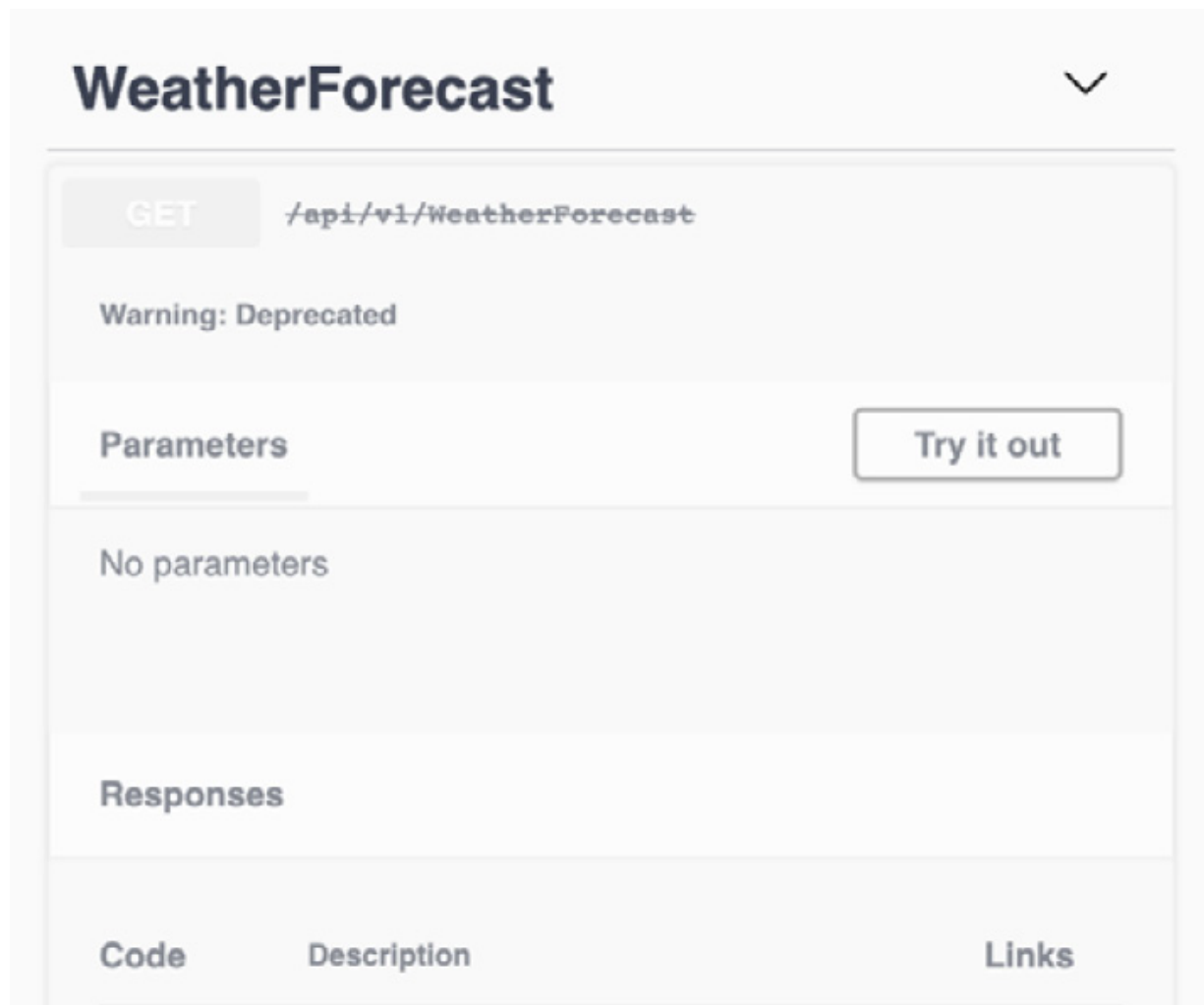


Figure 8.2 – Deprecated API

The default styling in *Figure 8.2*, with washed-out color and a struck-through endpoint name tells the developers that this endpoint has been marked as deprecated.

Now let's try version 2 of the **WeatherForecast** endpoint, which is a POST request that requires you to add a **query** parameter with a key of **city** and the value as the name of the city you want to use for your requested data:

# WeatherForecast

POST

/api/v2/WeatherForecast

Parameters

Cancel

Name	Description
city string (query)	<input type="text" value="Oslo"/>

Execute

Clear

Responses

Curl

```
curl -X POST "https://localhost:5001/api/v2/WeatherForecast?city=Oslo" -H "accept: text/plain" -d ""
```

Request URL

```
https://localhost:5001/api/v2/WeatherForecast?city=Oslo
```

Server response

Figure 8.3 – Version 2 of WeatherForecast

With the help of Swagger UI, we can easily insert query parameter in the input box. Let's type **Oslo** in the textbox and execute it.

Let's see the response of the endpoint after hitting the **Execute** button:

Code

Details

200

Response body

```
[
  {
    "date": "2020-11-17T12:49:37.540733+01:00",
    "temperatureC": 11,
    "temperatureF": 51,
    "summary": "Balmy",
    "city": "Oslo"
  },
  {
    "date": "2020-11-18T12:49:37.541086+01:00",
    "temperatureC": 5,
    "temperatureF": 40,
    "summary": "Chilly",
    "city": "Oslo"
  },
  {
    "date": "2020-11-19T12:49:37.541087+01:00",
    "temperatureC": 27,
    "temperatureF": 80,
    "summary": "Hot",
    "city": "Oslo"
  },
  {
    "date": "2020-11-20T12:49:37.541088+01:00",
    "temperatureC": 51
  }
]
```

Download

Response headers

```
api-deprecated-versions: 1.0
api-supported-versions: 2.0
content-type: application/json; charset=utf-8
date: Mon16 Nov 2020 11:49:37 GMT
server: Kestrel
transfer-encoding: chunked
```

Figure 8.4 – WeatherForecast version 2 response

You will notice that the response now includes your input as the **city** property of the **WeatherForecast** object in the array. We now can see that the new endpoint is working properly.

That's it for API versioning, and now we can move on to logging in an application, which is also an exciting topic.

## Logging in ASP.NET Core

Sometimes we assume an ideal world where our backend application runs successfully. However, in the real world, there are always events or errors to be expected. For instance, our connection to SQL Server may drop for whatever reason. To do our part, as a best practice, we should expect errors and handle them properly.

**Logging** is where you print lines of output as your application runs that give you some info about the usage, performance, errors, and diagnostics of your application. In short, the output tells us the story of what is going on internally in the application.

So how do we do this in ASP.NET Core?

## Logging in ASP.NET Core

If you have ever done any work with ASP.NET Core, you have probably seen the **ILogger** interface. Perhaps, you even know that ASP.NET Core has logging built in by default. But do you know that we can configure it or replace it? Yes, we can use third-party libraries such as **Serilog** or **NLog**, which you will see shortly. But let's discuss the basics of logging, which are the levels of logging.

Here are the defined logging severity levels of the **ILogger** interfaces:

- **Trace**: Logs that have the most detailed messages such as sensitive application data. However, messages are disabled by default and must not be enabled in a production environment. The output target is a trace listener.
- **Debug**: Logs that are used for interactive investigation during development. In almost all cases, this should be your log level.
- **Information**: Logs that monitor the general flow of the application, which are needed in production to see the running state of your application.
- **Warning**: Logs that highlight an anomaly, unexpected, or abnormal event in the application. These events don't stop the application.
- **Error**: Logs that highlight when the flow of execution is halted due to a failure. Which errors occur the most?
- **Critical**: Logs that show a system crash or unrecoverable application, or a catastrophe.

So now we know the levels of logging. Is that it? No, not yet. Here is an example of some logging output:

```
Microsoft.Hosting.Lifetime: Information: Now listening on: https://localhost:5001  
info: Microsoft.Hosting.Lifetime[0]
```

The preceding example of logging tells us which port the computer is running on, but piles of these text logs will give us a hard time figuring out what is going on in our app. Not to mention that it is laborious and time-consuming to get the info that you want out of the humungous logs of data.

So, what if there's an easy way we can query specific data from the logs? That's structured logging.

## What is structured logging?

**Structured logs** are logs that are fully structured JSON objects. For example, we can use tools such as Kibana, Elmah, Seq, Stackify, and so on to filter and analyze logs without writing regex. This type

of logging is excellent because suddenly it becomes trivial to find all sorts of info about trends and how people use the product we are selling.

So, the question now is how do we do structured logging in ASP.NET Core?

## Serilog versus NLog

**Serilog** and **NLog** are two popular logging frameworks in .NET that you can use, and both of them have a lot of documentation. Hence, you can easily find on the internet what you need to do in your application using Serilog or NLog.

The two loggers have similarities such as C# based configuration and structured logging. However, based on my experience, I have found that structured logging in Serilog is easier to setup than in NLog. Similarly, Serilog uses Fluent API to do C# based configuration.

Serilog and NLog have some similarities in log levels. Log levels in **Serilog** are **Fatal**, **Error**, **Warning**, **Information**, **Debug**, and **Verbose**. While **NLog** has **Fatal**, **Error**, **Warn**, **Info**, **Debug**, and **Trace**.

These are the log levels that you will also usually see in other programming languages and frameworks.

So, we will be using Serilog in our application. In my opinion, Serilog has better-structured logging support that's easy to configure. The interfaces in Serilog are developer-friendly, and more developers use Serilog than NLog, which increases the chances that a developer will have experience of Serilog.

Just to give you the origin of the Serilog name, it came from the term serialized log. Okay, so let's start adding and configuring Serilog to our application.

## Configuring Serilog

Now we need to add some NuGet packages in order to use and configure Serilog correctly. We will add the packages to **Travel.WebApi** and update the **Program.cs** file:

1. Here are the NuGet packages that we are adding. The following package allows you to use Serilog in ASP.NET Core:

`Serilog.AspNetCore`

The following package allows you to use **appsettings.json** to configure Serilog:

`Serilog.Settings.Configuration`

The following package allows you to log exception details:

`Serilog.Exceptions`

The following package allows you to log events in a simple and compact JSON-based format:



`Serilog.Formatting.Compact`

Let me introduce enrichers in Serilog. **Enrichers** can modify, add, and remove the properties in Serilog. The enrichers do it by implementing **ILogEventEnricher**, which is applied during logging to provide additional information to log events.

2. Here are some of the enrichers that we need. Let's also install them in the **Travel.WebApi** project.

The following package enriches Serilog log events with properties emitted from **System.Environment**:

`Serilog.Enrichers.Environment`

The following package adds a process enricher to Serilog:

`Serilog.Enrichers.Process`

The following package enriches Serilog log events with properties emitted from the current thread:

`Serilog.Enrichers.Thread`

3. Now let me also introduce you to sinks in Serilog. **Sinks** allow you to direct your logs to storage in various formats. You can target different storages to log events. There are too many sink choices to fit on a single page so please go here to see all of them: <https://github.com/serilog/serilog/wiki/Provided-Sinks>.

The following package allows you to write your Serilog events in a text file. It can be in plain text or JSON format:

`Serilog.Sinks.File`

The following package allows you to write your Serilog events in an SQLite database:

`Serilog.Sinks.SQLite`

Now we can move to the next section, where we are going to write some code.

4. After adding the packages, let's update the **Program.cs** file of the **Travel.WebApi** project.

Update the **Main()** method with this code:

```
public static int Main(string[] args)
{
    var name = Assembly.GetExecutingAssembly().GetName();
    Log.Logger = new LoggerConfiguration()
        .MinimumLevel.Debug()
        .CreateLogger();

    // Wrap creating and running the host in a
    try-catch block

    /* CreateHostBuilder inside a try catch */
}
```

```
}
```

The updated code block is an implementation of Serilog in a simple way. You can see here the **LoggerConfiguration** construct from the **Serilog** namespace. You will also see here the **MinimumLevel** of log events that will be passed to the sinks. Then we write the **CreateLogger()** method, which creates a logger using the minimum level, enrichers, and sinks.

5. Now let's add our enrichers and sinks by updating **LoggerConfiguration**. Let's insert the enrichers and the sinks between the **MinimumLevel()** and **CreateLogger()** methods like so:

```
Log.Logger = new LoggerConfiguration()
    .MinimumLevel.Debug()
    .Enrich.FromLogContext()
    .Enrich.WithExceptionDetails()
    .Enrich.WithMachineName()
    .Enrich.WithProperty("Assembly",
        $"{name.Name}")
    .Enrich.WithProperty("Assembly",
        $"{name.Version}")
    .WriteTo.SQLite(
        Environment.CurrentDirectory +
            @"/Logs/log.db",
            // For Mac and Linux users
            //
        Environment.CurrentDirectory + @"\\Logs\\log.db",
        // For Windows users

        restrictedToMinimumLevel:
            LogEventLevel.Information,
            storeTimestampInUtc: true)
    .WriteTo.File(
        new CompactJsonFormatter(),
        Environment.CurrentDirectory +
            @"/Logs/log.json",
            // For Mac and Linux users
            //
        Environment.CurrentDirectory + @"\\Logs\\log.json", // For Windows users

        rollingInterval:
            RollingInterval.Day,
```

```

        restrictedToMinimumLevel:
            LogEventLevel.Information)
        .WriteTo.Console()
        .CreateLogger();

```

And now we are using all the enrichers and sinks that we installed from the NuGet package manager. Replace the path to SQLite with your respective operating system because getting the path of a directory or a file differs depending on the operating system you are using.

6. Let's now write **CreateHostBuilder** wrapped with a **try catch**, like so:

```

try
{
    Log.Information("Starting host");
    CreateHostBuilder(args).Build().Run();
    return 0;
}
catch (Exception ex)
{
    Log.Fatal(ex, "Host terminated
        unexpectedly");
    return 1;
}
finally
{
    Log.CloseAndFlush();
}

```

A lot of Serilog syncs will send those events to different destinations asynchronously, and when your app exits for whatever reason, you really want those flashed. Returning **0** means the program succeeded; otherwise, the program exited with an error or an anomaly.

7. The last thing to do before we test our Serilog is to include the **CreateHostBuilder** definition, like so:

```

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .UseSerilog()
        .ConfigureWebHostDefaults(webBuilder
            =>
            {

```

```
webBuilder.UseStartup<Startup>();

});
```

We are setting Serilog here as the default logging provider. This is possible because the **ILogger** interface allows it to be overridden.

8. Now let's run the application again to see the log events in the JSON file and SQLite database file; both files will be generated automatically.

Once the application is running again, use your Swagger UI to send a request to the **WeatherForecast v2** endpoint or to the **TourList v1** endpoint. In *Figure 8.5*, you will see the autogenerated SQLite file and a JSON file that appears inside a new **Logs** directory. The files contain the log events that Serilog produced:

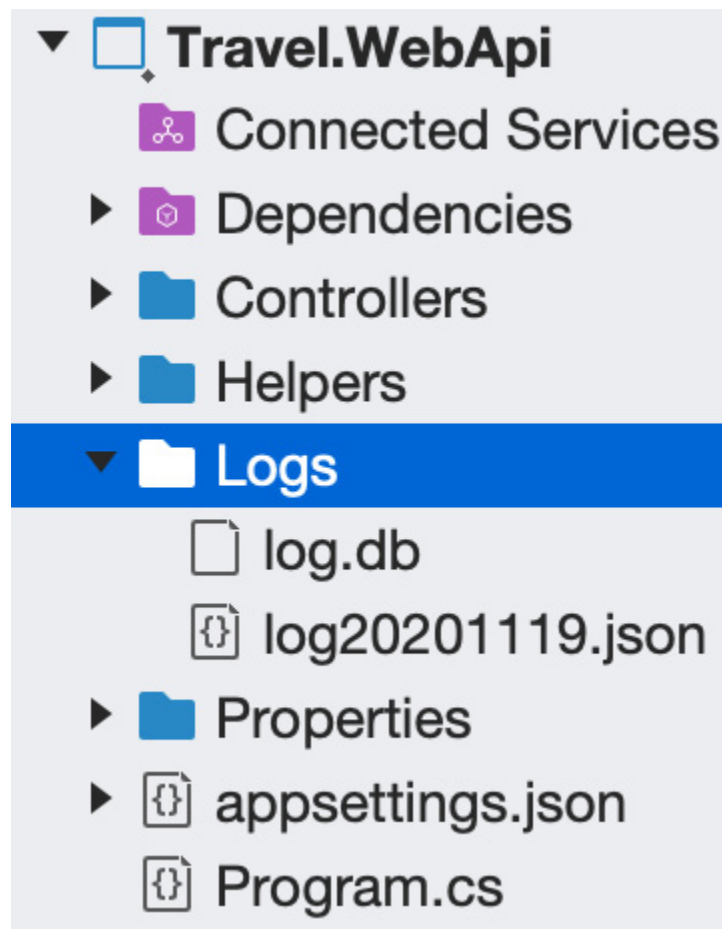


Figure 8.5 – SQLite file and JSON file in the Logs directory

Let's check out the SQLite file using SQLite Browser or SQLiteStudio. In *Figure 8.6*, you will notice that **RenderedMessage** and **Properties** are in JSON format:

	id	Timestamp	Level	Exception	RenderedMessage	Properties
	F...	Filter	Filter	Filter	Filter	Filter
1	1	2020-11-16T09:05:30	Information		Starting host	{"MachineName":"Devlins-MacBook-...
2	2	2020-11-16T09:05:32	Information		Now listening on: {address}	{"address":"https://localhost:...
3	3	2020-11-16T09:05:32	Information		Application started. Press Ctrl+C to shut down.	{"SourceContext":"Microsoft.Hosting.Lifetime","MachineName":...
4	4	2020-11-16T09:05:32	Information		Hosting environment: {envName}	{"envName":"Development","SourceContext":"Microsoft.Hostin...
5	5	2020-11-16T09:05:32	Information		Content root path: {contentRoot}	{"contentRoot":"/Users/inmetadeveloper/Dropbox/A Books/Pac...
6	6	2020-11-16T09:05:32	Information		{HostingRequestStartingLog:1}	{"Protocol":"HTTP/...
7	7	2020-11-16T09:05:32	Information		{HostingRequestFinishedLog:1}	{"ElapsedMilliseconds":39.5107,"StatusCode":...
8	8	2020-11-16T09:05:32	Information		{HostingRequestStartingLog:1}	{"Protocol":"HTTP/...
9	9	2020-11-16T09:05:32	Information		{HostingRequestFinishedLog:1}	{"ElapsedMilliseconds":123.6684,"StatusCode":...
10	10	2020-11-16T09:05:33	Information		{HostingRequestStartingLog:1}	{"Protocol":"HTTP/...
11	11	2020-11-16T09:05:33	Information		{HostingRequestFinishedLog:1}	{"ElapsedMilliseconds":1.5042,"StatusCode":...
12	12	2020-11-16T09:05:33	Information		{HostingRequestStartingLog:1}	{"Protocol":"HTTP/...
13	13	2020-11-16T09:05:34	Information		{HostingRequestFinishedLog:1}	{"ElapsedMilliseconds":99.2169,"StatusCode":...

Figure 8.6 – Log events in the SQLite file

The format will be able to help any data visualization tools to render the data in the UI graph. It also helps any data visualization tools querying a specific detail that you want to pull from the log events.

Now let's check out the JSON file. You will also see in *Figure 8.7* that the log events were stored in JSON format:

```
presentation > Travel.WebApi > Logs > {} log20201116.json > ...
1  {"@t":"2020-11-16T09:05:30.2621170Z","@mt":"Starting host","MachineName":"Devlins-MacBook-Pro","Assembl
2  {"@t":"2020-11-16T09:05:32.0709870Z","@mt":"Now listening on: {address}","address":"https://localhost:5
3  {"@t":"2020-11-16T09:05:32.0714480Z","@mt":"Application started. Press Ctrl+C to shut down.,"SourceCon
4  {"@t":"2020-11-16T09:05:32.0715180Z","@mt":"Hosting environment: {envName}","envName":"Development","So
5  {"@t":"2020-11-16T09:05:32.0715760Z","@mt":"Content root path: {contentRoot}","contentRoot":"/Users/inm
6  {"@t":"2020-11-16T09:05:32.2833920Z","@mt":"{HostingRequestStartingLog:1}","@r":["Request starting HTTP
7  {"@t":"2020-11-16T09:05:32.3210280Z","@mt":"{HostingRequestFinishedLog:1}","@r":["Request finished HTTP
8  {"@t":"2020-11-16T09:05:32.3347040Z","@mt":"{HostingRequestStartingLog:1}","@r":["Request starting HTTP
9  {"@t":"2020-11-16T09:05:32.4584650Z","@mt":"{HostingRequestFinishedLog:1}","@r":["Request finished HTTP
10 {"@t":"2020-11-16T09:05:33.5712330Z","@mt":"{HostingRequestStartingLog:1}","@r":["Request starting HTTP
11 {"@t":"2020-11-16T09:05:33.5727490Z","@mt":"{HostingRequestFinishedLog:1}","@r":["Request finished HTTP
12 {"@t":"2020-11-16T09:05:33.9183730Z","@mt":"{HostingRequestStartingLog:1}","@r":["Request starting HTTP
13 {"@t":"2020-11-16T09:05:34.0175930Z","@mt":"{HostingRequestFinishedLog:1}","@r":["Request finished HTTP
14
```

Figure 8.7 – Log events in JSON format

You will notice that there are **@ signs** in the log events. The @ sign prefix is an operator telling Serilog to serialize the object passed in instead of using **ToString()**.

## NOTE

*The database URL or URL path must vary depending on the application environment, such as testing, acceptance, and production. I used C# in the tutorial so that we could hover our mouse and read the details of the NuGet packages'*

*interfaces. It also provides us IntelliSense while writing C# code. However, you should configure Serilog in **appsettings.json**.*

Okay. So let's have a quick recap of what you have learned here.

## Summary

With that, you have learned that API versioning is essential at the beginning of API development to make it flexible for any future changes.

You have learned how to use API versioning strategies, namely URI path versioning, header versioning, and query string versioning. And the URI path is the most common among all the schemes. Part of API versioning is deprecating an API and integrating it with OpenAPI, which helps developers to see the details of all APIs in Swagger UI.

You have also learned the importance of logging in an application and which library to use in ASP.NET Core. You also now know what structured logging is and how to save it in a database so that you can easily query what you want to investigate in an application.

In the next chapter, we will add security to our ASP.NET Core application using JWT, or JSON web tokens.

## Chapter 9: Securing ASP.NET Core

Correctly securing your APIs is essential. In this chapter, we will learn how to secure an ASP.NET Core 5 Web API application because it is also the responsibility of the developer, who is building the backend application to protect the APIs. There are two available frameworks in .NET that you can use to secure your application, and there are also cloud identity providers.

We will tackle the following topics:

- Understanding ASP.NET Core Identity
- Introducing IdentityServer4
- Getting to know **customer identity and access management (CIAM)**
- Authenticating using JWT

## Technical requirements

Here is what you need to complete this chapter:

- Visual Studio 2019, Visual Studio for Mac, or Rider
- Postman
- JWT Debugger: <https://jwt.io>

Here is the link to the finished repo of this chapter: <https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter09>.

## Understanding ASP.NET Core Identity

**ASP.NET Core Identity** is an open source identity framework in .NET. It provides the .NET application with a way to implement authentication, to determine somebody's identity and authorization, which details the limits to the user's access. It also allows a .NET application to manage user accounts. Using Identity in an application is recommended so your application complies with security best practices and precautions. ASP.NET Core also saves development time, so you can just focus on your business logic.

## ASP.NET Core Identity features

The major features that ASP.NET Core Identity provides are as follows:

- User accounts and password management.
- **Multi-factor authentication (MFA)** and **two-factor authentication (2FA)**.

- Login and logout features.
- External authentication providers, such as Facebook, Google, Twitter, and Microsoft account.
- Customize the Identity framework based on your needs.

Developers need to understand how authentication flows in ASP.NET Core Identity and how it works. ASP.NET Core Identity uses a claims-based security scheme that was introduced back in .NET version 4.5.

A **claim** is an identity's property, and it includes a key-value pair. An identity can have one or more claims associated with it. For example, a student user in a school application can have multiple claims defining who they are. These claims can include **names**, **schoolEmails**, or **dateOfBirths**. Every claim must have a value that's unique to that student.

The **ClaimsIdentity** class represents an identity and all of its claims. A user in the ASP.NET Core application can have more than one identity through the help of the **ClaimsPrincipal** class.

**ClaimsPrincipal** comprises one or more **ClaimsIdentity** classes; hence, it can have one or more claims.

To give a better overview of **ClaimsIdentity** and **ClaimsPrincipal**, let me show you a diagram, *Figure 9.1*, as follows:



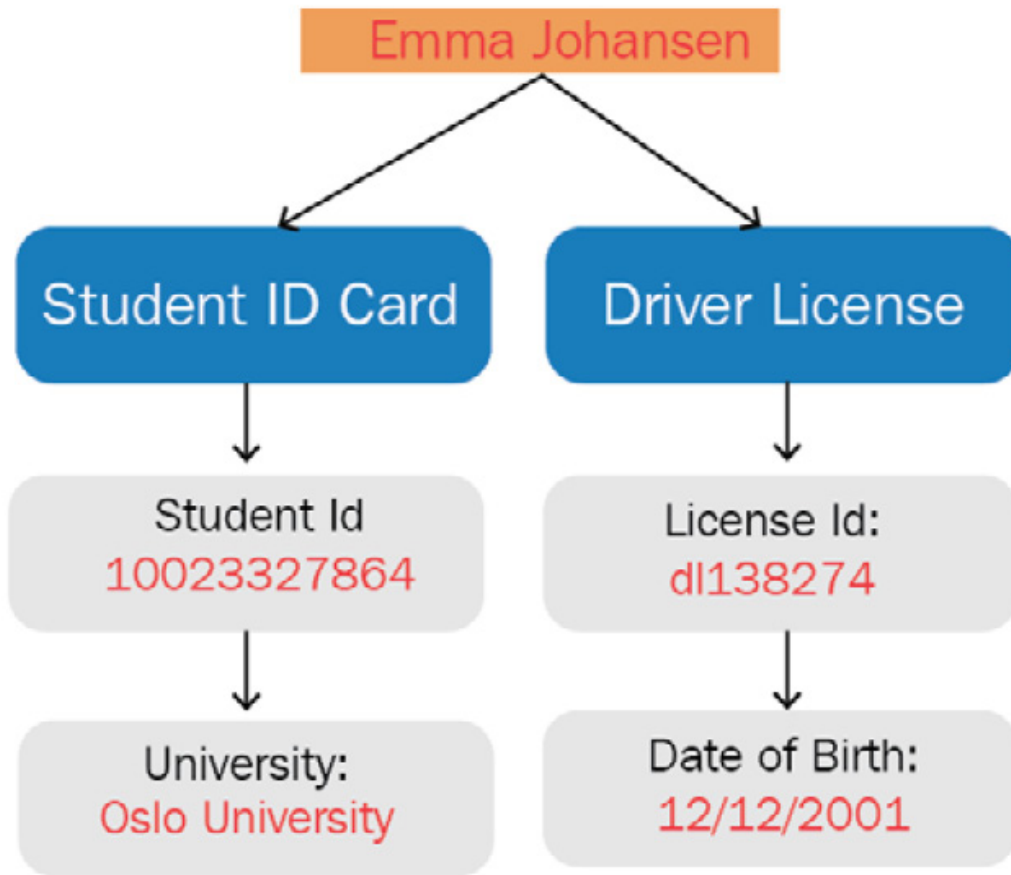


Figure 9.1 – Two forms of identification

To illustrate, see the preceding *Figure 9.1*, where a student may have two forms of identification: their student ID card and their driver's license card. The student ID number and name of the university or school are visible on the student ID card. Their driver's license card has the license ID and their date of birth. Each identity is flexible enough to allow access to different resources.

**ClaimsPrincipal** inherits the list of claims from the associated identities. In this scenario, the driver's license claims and student ID card claims identify the student user.

**Cookie authentication** is the most common authentication in ASP.NET Core applications. It's good to know that ASP.NET Core has built-in middleware that enables the application to utilize cookie-based authentication.

Also, to give a better overview of cookie authentication, let me show you a diagram, *Figure 9.2*, as follows:

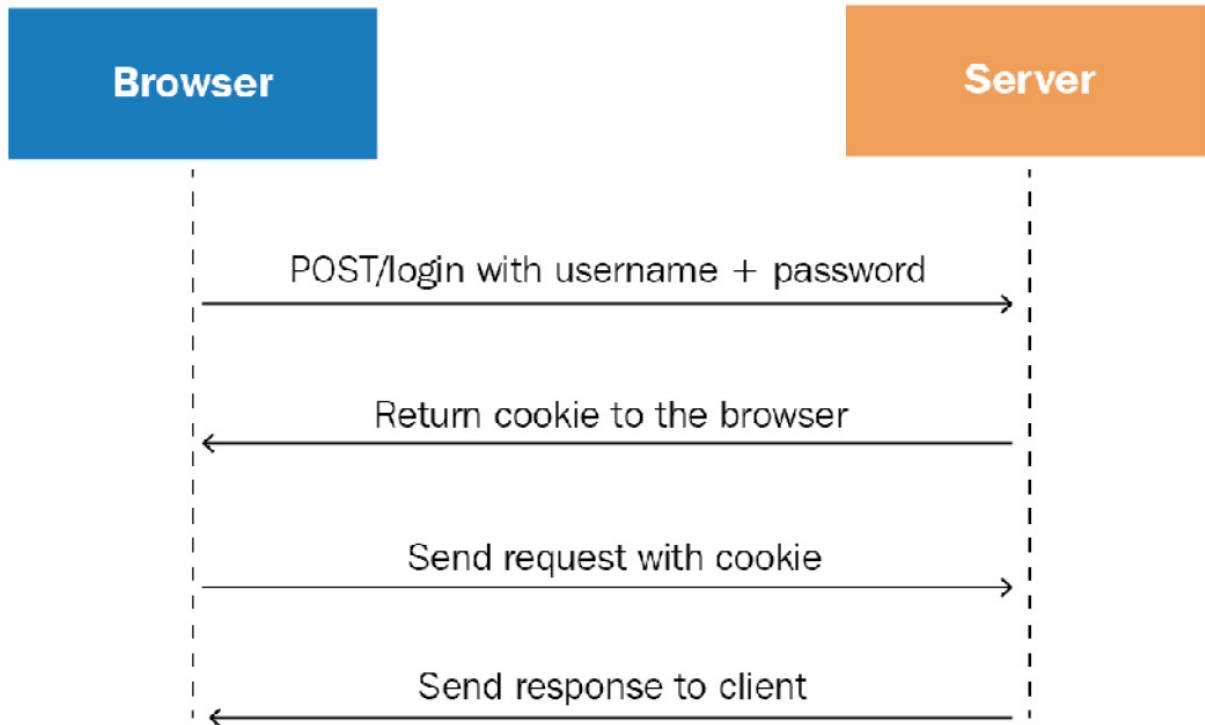


Figure 9.2 – ASP.NET Core cookie middleware

The preceding diagram is an overview of how a simple ASP.NET Core cookie middleware works. The following steps are what happens under the hood:

1. **ClaimsPrincipal** and all its inherited claims are serialized into a cookie after a successful login of the user.
2. The browser saves an encrypted cookie from the server.
3. The cookie is sent via the request header and validated by the server on every next request.
4. The cookie is then used to recreate **ClaimsPrincipal** after successful verification of it. The principal then gets attached to the HTTP context user property.
5. Finally, all the user information and claims linked to the principal can be accessed by the application code.

That's a quick introduction to ASP.NET Core Identity; now, let's go on to IdentityServer4.

## Introducing IdentityServer4

**Security Token Service (STS)** is a critical component of token-based authentication. Other terms that you might encounter are identity provider or authorization server. It's a piece of software that generates and issues security tokens to client-side applications. Client-side applications redirect user verification to STS to handle it. The token is encrypted and signed to ensure that the token is protected from any tampering.

The encryption in the application uses a private key kept by the token service, while the public key for decrypting the token is shared with client-side applications. In this way, the client applications

trust that the token came from the right token service. Standards for providing identity information, such as OpenID Connect, are used by token services.

When it comes to consuming token services, there are different options. You can use a pre-built service such as **Active Directory Federation Services (ADFS)** to build your custom service using the IdentityServer framework.

**IdentityServer4** is an open source framework for token authentication. At the time of writing, IdentityServer4 is the latest recommended version for ASP.NET Core 5 applications:

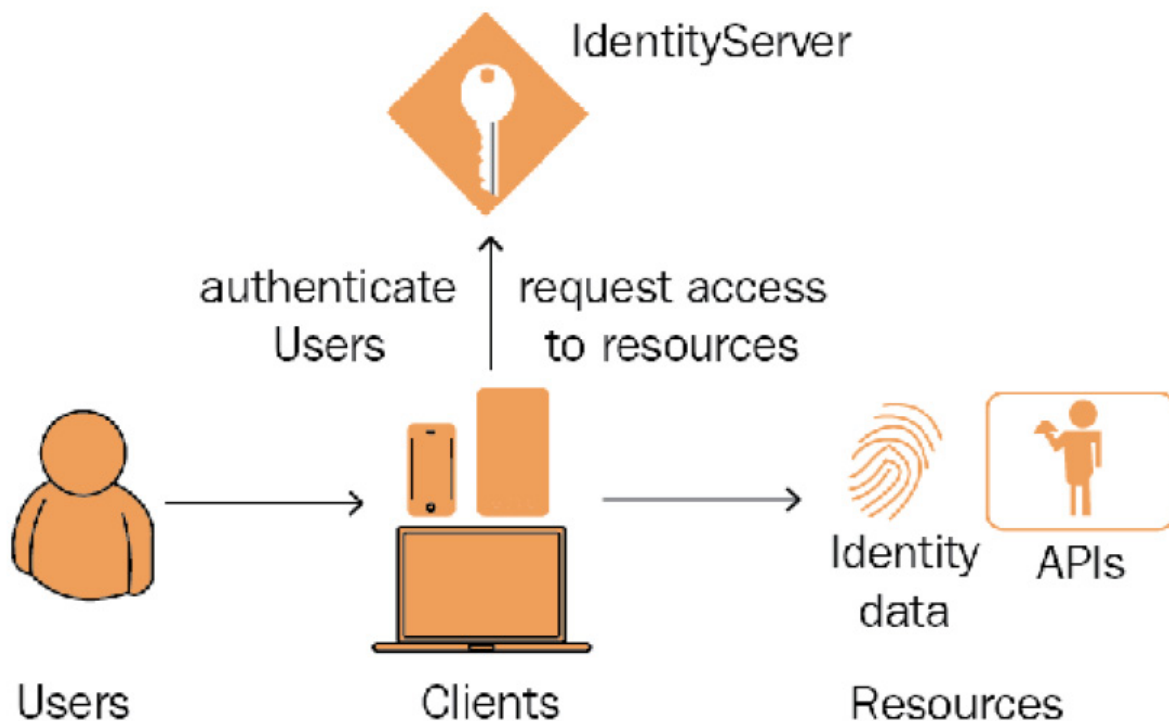


Figure 9.3 – Basic structure of a security system

The preceding diagram explains the fundamental structures of a security system. A client is an application that requests tokens from IdentityServer4. It can be a mobile app or web app, for example. IdentityServer4 has to know the client applications that are permitted to use it by registering or including them in a collection of entities or stores. This collection can be persisted or saved in a database or memory.

Next are the end users who use or interact with client-side applications. The users also have to be registered through IdentityServer. A unique identifier, along with username, and password credentials, are samples of identity data that define users. Users can also have one or multiple associated claims. The identity resources and the API resources are the stuff that needs protection. Identity resources are details about identity, such as user claims. API resources, on the other hand, represent protected functionality, such as Web APIs. Resources also have to be registered; they need

to be registered in an identity server store. Two types of tokens are generated along with the two resource types.

Information is sent with the identity token when IdentityServer4 authenticates the user. The client sends forward the access token to the API, which allows or grants access to protected functionality or data when access is requested for the API resource, and the access token is successfully issued.

IdentityServer4 is a middleware; it uses industry-standard **JSON Web Tokens (JWTs)**, and it implements two standard protocols, OAuth 2 and OpenID Connect. **OAuth 2** is an open standard for authorization; it makes sure that a user has the right permission to access a protected resource. On the other hand, **OpenID Connect** is an authentication protocol and an extension built on top of the OAuth 2 protocol. Using OAuth 2, a client can request access tokens from an STS and then communicate with APIs.

IdentityServer4 can manage or handle authentication and authorization requirements through its framework and implementation of OAuth 2 and OpenID Connect.

That's a quick introduction to IdentityServer4 now; let's proceed to **CIAM**.

## Customer identity and access management (CIAM)

Security is complex, and you must get it right. So today, developers trust authentication providers to help secure their applications. **Auth0**, **AAD B2C**, **Okta**, **AWS Incognito**, and **GCP Identity Platform** are popular auth providers, also known as **Identity as a Service (IDaaS)**. So, why should you consider using an auth provider?

### **NOTE:**

*I'm currently an Auth0 ambassador. No, I'm not an employee of the company, nor do I get any monetary compensation. Although occasionally I do get some swag and other cool perks whenever I speak at a conference and mention them. But I became an Auth0 ambassador precisely because I've been using it and recommending it.*

First, security is complicated, and it's not hard to get wrong. It can feel like you're in a giant puzzle getting all of the options right. By choosing a cloud identity provider that specializes in security, you can trust that the security engineers of an auth provider know what they're doing and you can use your time to focus on developing your application. Similarly, you can customize providers to your needs. Hence, they integrate nicely with your application. So, what about pricing? Well, pricing varies, but some auth providers offer a free tier that works for many types of applications.

I recommend that you use Auth0 because it's easy to use, is the most popular, and has excellent online documentation. It also offers excellent libraries and SDKs for creating single-page apps, and the price is reasonable because it provides a generous free tier. Auth0's free-tier pricing supports up to

7k active users. If you're wondering what features are available in the paid tiers, check out their website, but the free tier is already enough for most applications.

The terminologies *IDaaS* and *cloud identity providers* are being replaced by the acronym CIAM.

OK, so what is **CIAM**, or **customer identity and access management**?

CIAM sits at the intersection of customer experience, security, and analytics. Providing a frictionless way for users to onboard and log in is essential for building customer loyalty and driving conversions. Taking steps to prevent data breaches and protect sensitive data from malicious intrusion is the focus of CIAM for compliance with data privacy laws and security policies. Not to mention that compiling user data into one or a single source of truth is crucial to understanding your customers.

Many companies choose to engage a third-party IDaaS rather than building a solution in-house from scratch.

Companies need **identity and access management (IAM)** solutions for several classes of end users: customers, employees, and enterprise customers. But each type of user requires a different balance of **user experience (UX)** and security. Hence, CIAM solutions provide a superb set of features distinct from workforce identity solutions or B2B.

Here are four features that comprise modern CIAM solutions. No two CIAM solutions offer exactly the same features, but if you're signing up or purchasing a CIAM platform, it should have the following criteria:

- **Scalability:** CIAM has to scale from thousands to millions and even billions of users.
- **Single sign-on:** Allows users to log in or sign in to an app and automatically be signed in to a set of different applications.
- **Multi-factor authentication:** MFA is a more secure means of authenticating a user's identity rather than the common username and password combination.
- **Centralized user management:** Your insights into your end users or customers can be an excellent competitive advantage, but only if the data is accessible, organized, and accurate.

So that's the intro to CIAM. Now, let's move on to the security implementation in our application in the next section.

## Authentication implementation using JWT

The **JWT** or **JSON Web Token** is a type of token for carrying identity data between machines. It is supported by different programming languages, an industry standard, and can be easily passed around. A JWT is self-contained, and it holds the needed identity information within itself, as shown in the following figure:

## HEADER: ALGORITHM & TOKEN TYPE

---

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

## PAYLOAD: DATA

---

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "iat": 1516239022  
}
```

## VERIFY SIGNATURE

---

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
    
) ☐ secret base64 encoded
```

Figure 9.4 – Parts of a JWT

The preceding *Figure 9.4* shows the three parts of the JWT: the header, payload, and signature. The header has two properties. One is **alg**, which is short for algorithm, which determines the algorithm used for encoding this token. **typ** is **JWT**. We don't have to worry about this header because this is just a standard.

What matters to us is the second part, which is the payload. So here, we have a JSON object with three properties: **sub**, usually a user ID, **name**, and **iat**, which is when the token was generated. What you need to know here is that the payload includes public properties about the user, like how on your passport you have some properties about yourself, such as name, date of birth, and birthplace. So, we have the same concept on a JWT. We can include a few basic public properties about the user. Hence, every time we send a token from the client to the server, we can easily extract the user ID from the payload. If you need to know the user's name, you can simply extract that from the payload as well, meaning you don't have to query the database, send the user ID to get a user object, and then extract the name property.

The third part of the JWT is the digital signature. The digital signature is created based on the JWT's header and payload, along with the secret or private key. Like I mentioned earlier, the secret or private key is only available on the server.

Please also refer to the source code of this chapter in GitHub for better readability. If a malicious user gets the JWT and modifies the property, the digital signature will be invalid because the JWT content was modified. Now the hacker would need a new digital signature. Still, the hacker cannot generate a digital signature because they will need the private key available only on the server. So, if they don't have access to the server, they cannot create a valid digital signature. When they send this new tampered JWT to the server, the server will decline it. The server will say "not a valid JWT."

So this is how JWT works.

## Implementing token-based authentication

Token-based authentication using JWT is what we are going to implement in our application. This implementation generates a JWT, but what I need you to take from this chapter is knowing what kind of architecture you need and planning what type of security implementation is appropriate in your application in the future. You might need ASP.NET Core Identity, IdentityServer4, or CIAM.

So, let's start adding security to our application:

1. Create a new .NET 5 class library inside the infrastructure directory and name it **Travel.Identity**. After creating the library, delete the default **Class1.cs** file that you will find there.

We also need to create references. Create a reference from this project, **Travel.Identity**, to the **Travel.Application** project and from the **Travel.WebApi** project to this project.

Then install the following two NuGet packages inside the **Travel.Identity** project:

```
Microsoft.AspNetCore.Authentication.JwtBearer
```

This NuGet package is middleware that allows a .NET application to receive an OpenID Connect token. The following NuGet package is middleware that allows a .NET application to support the OpenID authentication workflow:

```
Microsoft.AspNetCore.Authentication.OpenIdConnect
```

The preceding NuGet packages are currently installed in the **Travel.WebApi** project, but we need them in the **Travel.Identity** project and not in the **Travel.WebApi** project. That being said, delete the **JwtBearer** and **OpenIdConnect** packages that are installed in the **Travel.WebApi** project.

2. Next, we go to the **Travel.Domain** project and create a new C# file with the name **User.cs** inside the **Entities** folder:

```
namespace Travel.Domain.Entities
{
    public class User
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Username { get; set; }
        public string Password { get; set; }
    }
}
```

The preceding code block is a simple **User** entity with **Id**, **FirstName**, **LastName**, **Username**, and **Password**.

3. Now, let's go back to the **Travel.Identity** project and create a folder in its root directory. Name the folder **Helpers**. Create three C# files inside the **Helpers** folder and name them **AuthSettings.cs**, **JwtMiddleware.cs**, and **AuthorizeAttribute.cs**, like so:

```
namespace Travel.Identity.Helpers
{
    public class AuthSettings
    {
        public string Secret { get; set; }
    }
}
```



The preceding block of code is for the secret configuration that we will write later inside the **appsettings.json** file.

The following code block from **JwtMiddleware.cs** is custom middleware that detects and extracts the authorization header from the HTTP request:

```
...

using System.IdentityModel.Tokens.Jwt;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Options;
using Microsoft.IdentityModel.Tokens;
using Travel.Application.Common.Interfaces;
namespace Travel.Identity.Helpers
{
    public class JwtMiddleware
    {
        private readonly RequestDelegate _next;
        private readonly AuthSettings _authSettings;
        public JwtMiddleware(RequestDelegate next,
            IOptions<AuthSettings> appSettings)
        {
            ...
        }
        public async Task Invoke(HttpContext context,
            IUserService userService)
        {
            ...
        }
        private void AttachUserToContext(HttpContext
            context, IUserService userService,
            string token)
        {
            ...
        }
        catch
```

```

        {
        }
    }
}
}

```

The preceding block of code also validates the token extracted from the header of the sender application, which I truncated and moved here:

```

private void AttachUserToContext(HttpContext context, IUserService userService, string
token)

```

```

    {
        try
        {
            var tokenHandler = new
                JwtSecurityTokenHandler();
            byte[] key = Encoding.ASCII.GetBytes
                (_authSettings.Secret);
            tokenHandler.ValidateToken(token, new
                TokenValidationParameters
            {
                ValidateIssuerSigningKey = true,
                IssuerSigningKey = new
                    SymmetricSecurityKey(key),
                ValidateIssuer = false,
                ValidateAudience = false,
                ClockSkew = TimeSpan.Zero
            }, out var validatedToken);
            var jwtToken =
                (JwtSecurityToken)validatedToken;
            var userId = int.Parse
                (jwtToken.Claims.First(c =>
                    c.Type == "id").Value);

            context.Items["User"] =
                userService.GetById(userId);

```

```

        } catch { }
    }
}

```

Suppose validation is correct; **userService.GetById** is invoked to get the data of the user.

The following code block from **AuthorizeAttribute.cs** is a custom attribute that lets you annotate a class or a method with **Authorize**:

...

```

namespace Travel.Identity.Helpers
{
    [AttributeUsage(AttributeTargets.Class
        | AttributeTargets.Method)]
    public class AuthorizeAttribute : Attribute,
        IAuthorizationFilter
    {
        public void OnAuthorization
            (AuthorizationFilterContext context)
        {
            var user =
                (User)context.HttpContext.Items["User"];
            if (user == null)
            {
                context.Result = new JsonResult(new {
                    message = "Unauthorized" })
                { StatusCode =
                    StatusCodes.Status401Unauthorized };
            }
        }
    }
}

```

If the request that hits an annotated route or controller is not authenticated, the controller will send a JSON object with an **Unauthorized** message.

- Now let's go to the **Travel.Application** project. Find the **Dtos** directory, create a folder inside of that directory, and name it **User**. Then create two C# files inside of the **User** folder and name them **AuthenticateResponse.cs** and **AuthenticateRequest.cs**, like so:

```

namespace Travel.Application.Dtos.User

```

```

{
    public class AuthenticateResponse
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Username { get; set; }
        public string Token { get; set; }
        public AuthenticateResponse
            (Domain.Entities.User user, string token)
        {
            Id = user.Id;
            FirstName = user.FirstName;
            LastName = user.LastName;
            Username = user.Username;
            Token = token;
        }
    }
}

```

This **User** model is the shape of the response for an authenticated user after successful login:

```

using System.ComponentModel.DataAnnotations;
namespace Travel.Application.Dtos.User
{
    public class AuthenticateRequest
    {
        [Required]
        public string Username { get; set; }
        [Required]
        public string Password { get; set; }
    }
}

```

The preceding block of code requires the login request to have **Username** and **Password** properties or keys in the request body.

5. Now let's create an interface for the user service we will create later. Look for the **Common** directory inside of the **Travel.Application** project. Now there's the **Interfaces** directory inside the **Common** directory; create an interface inside the **Interfaces** folder and name it **IUserService.cs** like so:

```
using Travel.Application.Dtos.User;
using Travel.Domain.Entities;

namespace Travel.Application.Common.Interfaces
{
    public interface IUserService
    {
        AuthenticateResponse Authenticate
            (AuthenticateRequest model);

        User GetById(int id);
    }
}
```

After creating the **IUserService.cs** file, let's create an implementation of that interface. Go back to the **Travel.Identity** project and create a **Services** folder inside the project root directory. Then create a new C# file and name it **UserService.cs**:

```
...

using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using Microsoft.Extensions.Options;
using Microsoft.IdentityModel.Tokens;
namespace Travel.Identity.Services
{
    public class UserService : IUserService
    {
        private readonly List<User> _users = new
            List<User>
        {
            new User {...}
        };
        ...

        public AuthenticateResponse Authenticate
            (AuthenticateRequest model)
```

```

    {
        ...
    }

    public User GetById(int id) =>
        _users.FirstOrDefault(u => u.Id == id);

    private string GenerateJwtToken(User user)
    {
        ...
    }
}

```

The preceding block of code is the implementation of **IUserService.cs** we created earlier. I've truncated the business logic of the **Authenticate** and **GenerateJwtToken** methods for readability purposes. Here is the complete code of the **Authenticate** method as well as the **GenerateJwtToken** method:

```

public AuthenticateResponse Authenticate
    (AuthenticateRequest model)
{
    var user = _users.SingleOrDefault(u =>
        u.Username == model.Username
        && u.Password == model.Password);
    if (user == null)
        return null;
    var token = GenerateJwtToken(user);
    return new AuthenticateResponse(user,
        token);
}

```

This implementation authenticates **User** by checking whether the user exists in the database. However, we hardcoded the **user** object here to simplify the implementation:

```

private string GenerateJwtToken(User user)
{
    byte[] key = Encoding.ASCII.GetBytes
        (_authSettings.Secret);
    var tokenDescriptor = new

```

```

        SecurityTokenDescriptor
    {
        Subject = new ClaimsIdentity(new[] {
            new Claim("id",
                user.Id.ToString()) },
            Expires = DateTime.UtcNow.AddDays(1),
            SigningCredentials = new
                SigningCredentials(new
                    SymmetricSecurityKey(key),
                    SecurityAlgorithms.HmacSha256Signature
                )
        });
        var tokenHandler = new
            JwtSecurityTokenHandler();
        var token = tokenHandler.CreateToken
            (tokenDescriptor);
        return tokenHandler.WriteToken(token);
    }

```

The **GenerateJwtToken** method generates a JWT by using the value of **Secret** in the **appsettings.json** file. It also uses **JwtSecurityTokenHandler** and **SecurityTokenDescriptor**, which create the claims or payload of the token. You will notice here that **SecurityTokenDescriptor** invokes **SigningCredentials** with key and algorithm arguments to sign the token.

6. Now let's create a **DependencyInjection** class that uses a static **IServiceCollection** for the **Travel.Identity** project. Create a C# class in the root directory of the **Travel.Identity** project like so:

```

using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Travel.Application.Common.Interfaces;
using Travel.Identity.Helpers;
using Travel.Identity.Services;
namespace Travel.Identity
{
    public static class DependencyInjection
    {

```

```

public static IServiceCollection
    AddInfrastructureIdentity
        (this IServiceCollection services,
         IConfiguration config)
{
    services.Configure<AuthSettings>
        (config.GetSection(nameof(AuthSettings)));
    services.AddScoped<IUserService, UserService>();

    return services;
}
}
}

```

Here is the registered service in our **dependency injection (DI)** container that we will register in **Startup.cs** in a bit. We are getting the object of **AuthSettings** in the **appsettings.json** file and registering a scoped lifetime for **UserService**. The **Secret** key of **AuthSettings** is a sensitive string that must be stored in an environment variable or Azure Key Vault to protect it.

7. Next, we create a C# file and name it **UsersController** inside the **Travel.WebApi** project, specifically inside the **v1** folder of the **Controllers** directory, like so:

```

using Microsoft.AspNetCore.Mvc;
using Travel.Application.Common.Interfaces;
using Travel.Application.Dtos.User;
namespace Travel.WebApi.Controllers.v1
{
    [ApiVersion("1.0")]
    [ApiController]
    [Route("api/v{version:apiVersion}/{controller}")]
    public class UsersController : ControllerBase
    {
        private readonly IUserService _userService;

        public UsersController(IUserService
            userService) => _userService = userService;

        [HttpPost("auth")]
        public IActionResult Authenticate([FromBody]

```



```

        AuthenticateRequest model)
    {
        var response =
            _userService.Authenticate(model);
        if (response == null)
            return BadRequest(new { message =
                "Username or password is incorrect"
            });
        return Ok(response);
    }
}

```

This controller does not require authorization since it is intended for logging in. It also uses the **v1** API to make it consistent. The controller authenticates the user and responds with the JWT for **User**.

8. Now, let's protect the rest of the controllers by adding the custom attribute in **ApiController**. Let's update **ApiController**'s code:

```

namespace Travel.WebApi.Controllers.v1
{
    [Authorize]
    [ApiVersion("1.0")]
    [ApiController]
    [Route("api/v{version:apiVersion}/{controller}")]
    public abstract class ApiController :
        ControllerBase
    {
        private IMediator _mediator;
        protected IMediator Mediator => _mediator ??=
            HttpContext.RequestServices.GetService
            <IMediator>();
    }
}

```

We are annotating **ApiController** with the **Authorize** attribute, the custom attribute that we created earlier. The attribute protects all the children classes of **ApiController** from authenticated consumers or users.

9. Now the last file to update is **Startup.cs** with the following code:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddApplication();
    services.AddInfrastructureData();
    services.AddInfrastructureShared
(Configuration);
    services.AddInfrastructureIdentity
(Configuration);
    services.AddHttpContextAccessor();
    services.AddControllers();
    ...
}
```

Add the **services.AddInfrastructureIdentity(Configuration);** method below the **services.AddInfrastructureShared(Configuration);** method.

10. Let's also update the **SwaggerGen** configuration. The code that we are adding will allow us to enter a JWT in the Swagger HTTP requests header through a dialog box:

```
services.AddSwaggerGen(c =>
{
    c.OperationFilter
    <SwaggerDefaultValues>();
    c.AddSecurityDefinition("Bearer", new
    OpenApiSecurityScheme
    {
        Description = "JWT Authorization header
        using the Bearer scheme.",
        Name = "Authorization",
        In = ParameterLocation.Header,
        Type = SecuritySchemeType.Http,
        Scheme = "bearer"
    });
    c.AddSecurityRequirement(new
    OpenApiSecurityRequirement
    {
```

```

        {
            new OpenApiSecurityScheme
            {
                Reference = new OpenApiReference
                {
                    Type =
                        ReferenceType.SecurityScheme,
                    Id = "Bearer"
                }
            }, new List<string>()
        }
    });
});

```

**AddSecurityDefinition** describes how the API is protected, while **AddSecurityRequirement** adds a global security requirement in the APIs.

Also, update the middleware like so:

```

public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
IApiVersionDescriptionProvider provider)
{
    ...
    app.UseRouting();
    app.UseMiddleware<JwtMiddleware>();
    app.UseAuthorization();
    ...
}

```

Insert the custom **app.UseMiddleware<JwtMiddleware>()** middleware, between **app.UseRouting()** and **app.UseAuthorization()**. This code block is the custom middleware that we created earlier.

Now, the **Travel.Identity** project is integrated into the **Travel.WebApi** project.

11. Run the application to see whether the security is working correctly in the application and Swagger.

After running the application, let's check out Swagger UI.

## Checking Swagger UI

After setting up the JWT authentication and adding it to Swagger, let's try and see how we would use and test our protected routes in our Swagger UI:

1. Swagger UI has the **Authorize** button for adding credentials or JWT, as shown in the following figure, *Figure 9.5*:

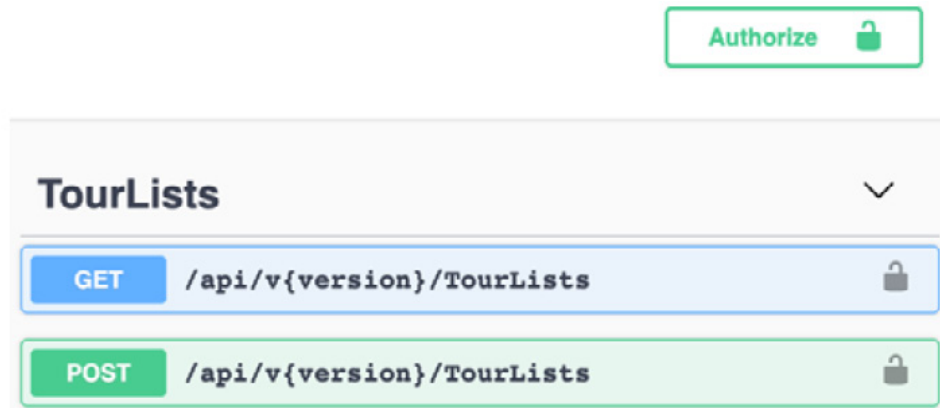


Figure 9.5 – OpenAPI with the security definition and requirement

2. Now let's check out the `/api/v1/Users/auth` route of the **Users** controller:

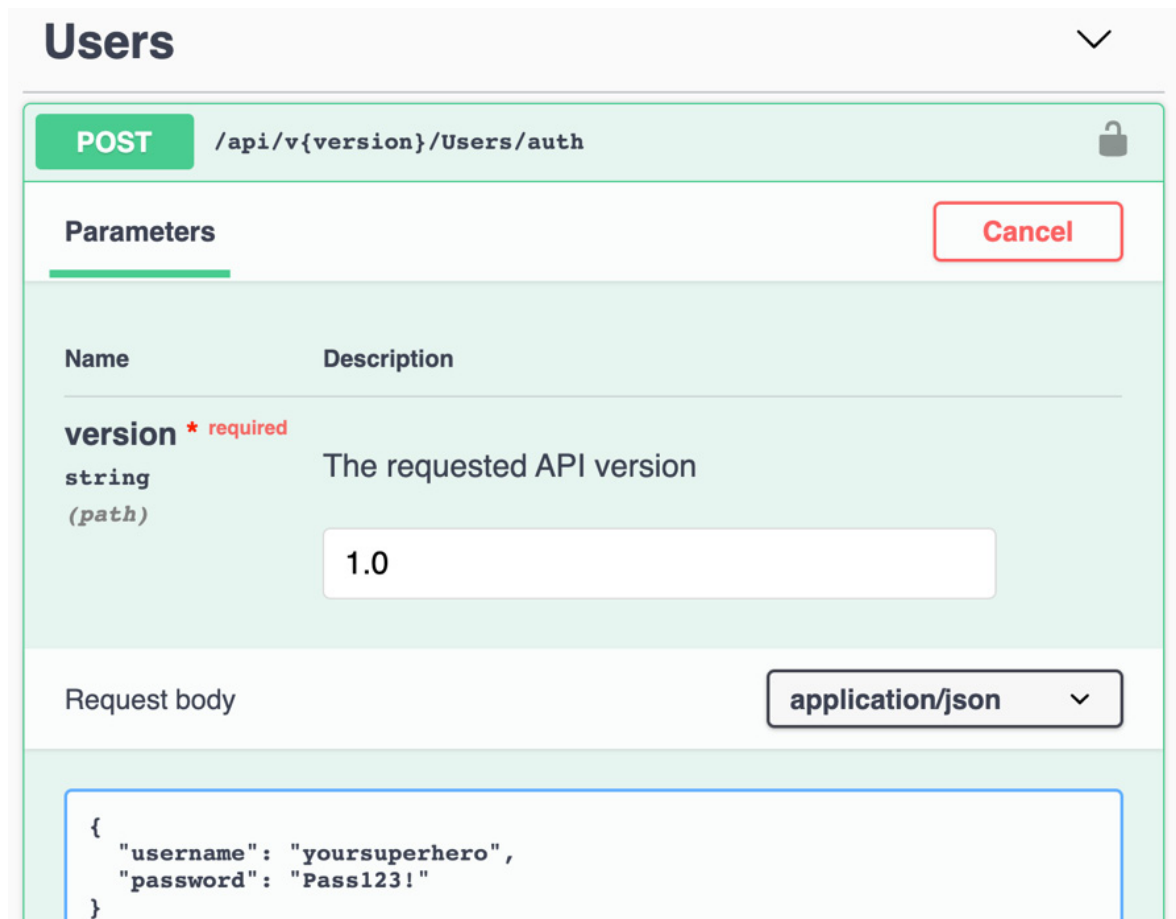
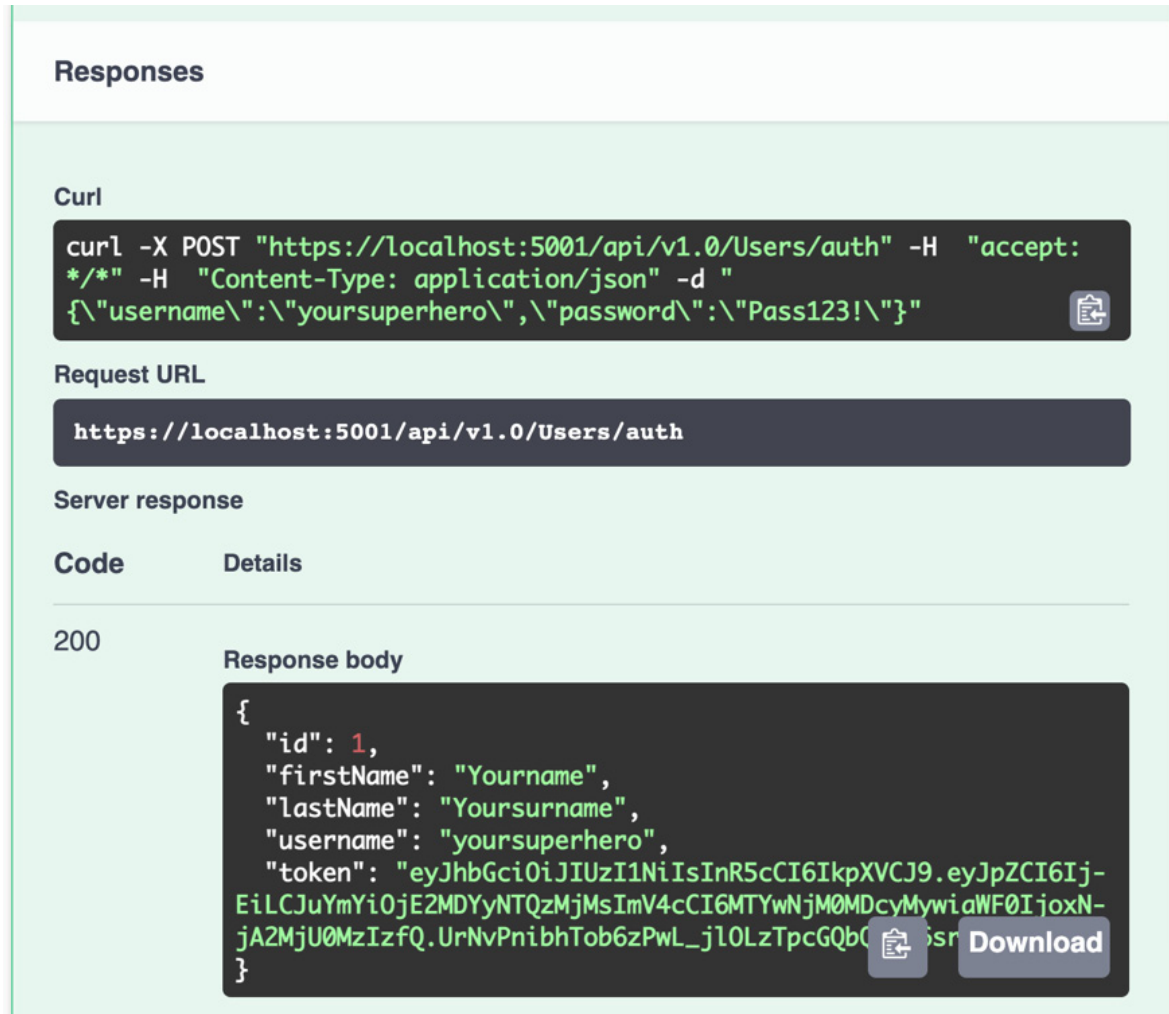


Figure 9.6 – Signing in using Swagger UI

We can test the auth API by sending a **POST** request with **username** as **yoursuperhero** and **password** as **Pass123!**, as shown in the preceding figure.

3. Now let's check out the response of the API after sending the **POST** request:



**Responses**

**Curl**

```
curl -X POST "https://localhost:5001/api/v1.0/Users/auth" -H "accept: */*" -H "Content-Type: application/json" -d '{"username":"yoursuperhero","password":"Pass123!"}'
```

**Request URL**

```
https://localhost:5001/api/v1.0/Users/auth
```

**Server response**

Code	Details
200	<p><b>Response body</b></p> <pre>{   "id": 1,   "firstName": "Yourname",   "lastName": "Yoursurname",   "username": "yoursuperhero",   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjEiLCJuYmYiOiJlMjMDYyNTQzMjMsImV4cCI6MTYwNjM0MDcyMywiaWF0IjoxNjA2MjU0MzIzfQ.UrNvPnibhTob6zPwL_jl0LzTpcGQbC" }</pre> <p>Download</p>

Figure 9.7 – JWT response after successful login

The preceding screenshot shows the JWT from the auth API's response body, which means we are authenticated after sending the username and password. Let's copy the token's value because we will use this in the **Authorize** box of Swagger UI.

4. Go to the <https://jwt.io> website and paste the JWT in the left box, as shown in the following screenshot:



6. Open up Postman and choose the bearer token under the **Auth** tab. Then paste the token in the input box. Send a **GET** request to the **TourLists** controller, **https://localhost:5001/api/v1.0/TourLists**:

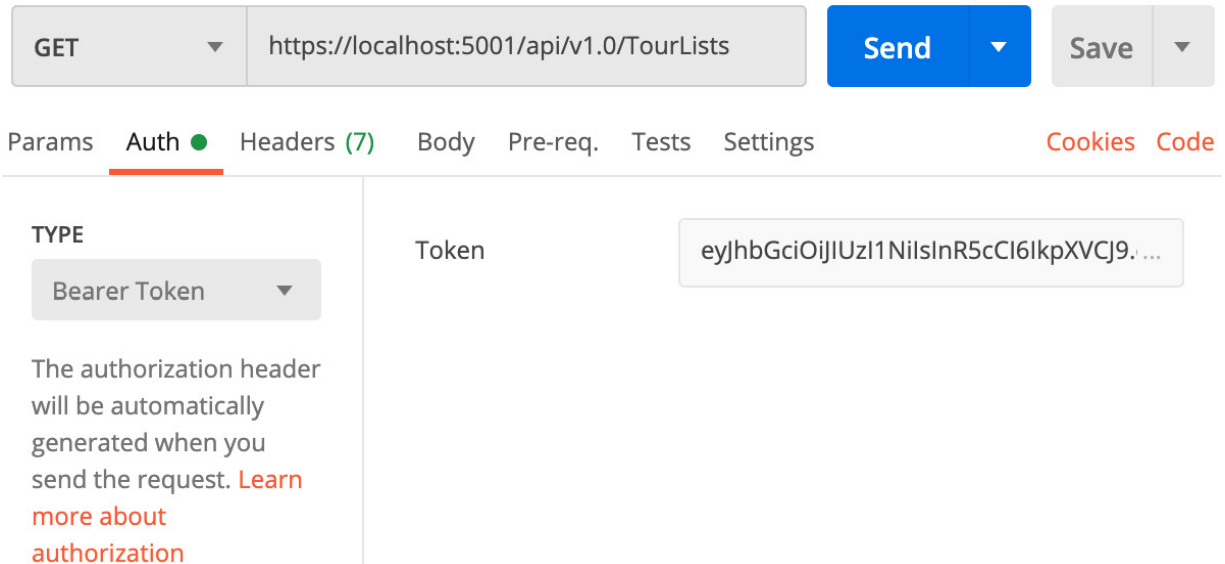


Figure 9.10 – Authorized API testing using Postman

The following screenshot shows a 200 status code with the data that we are expecting:

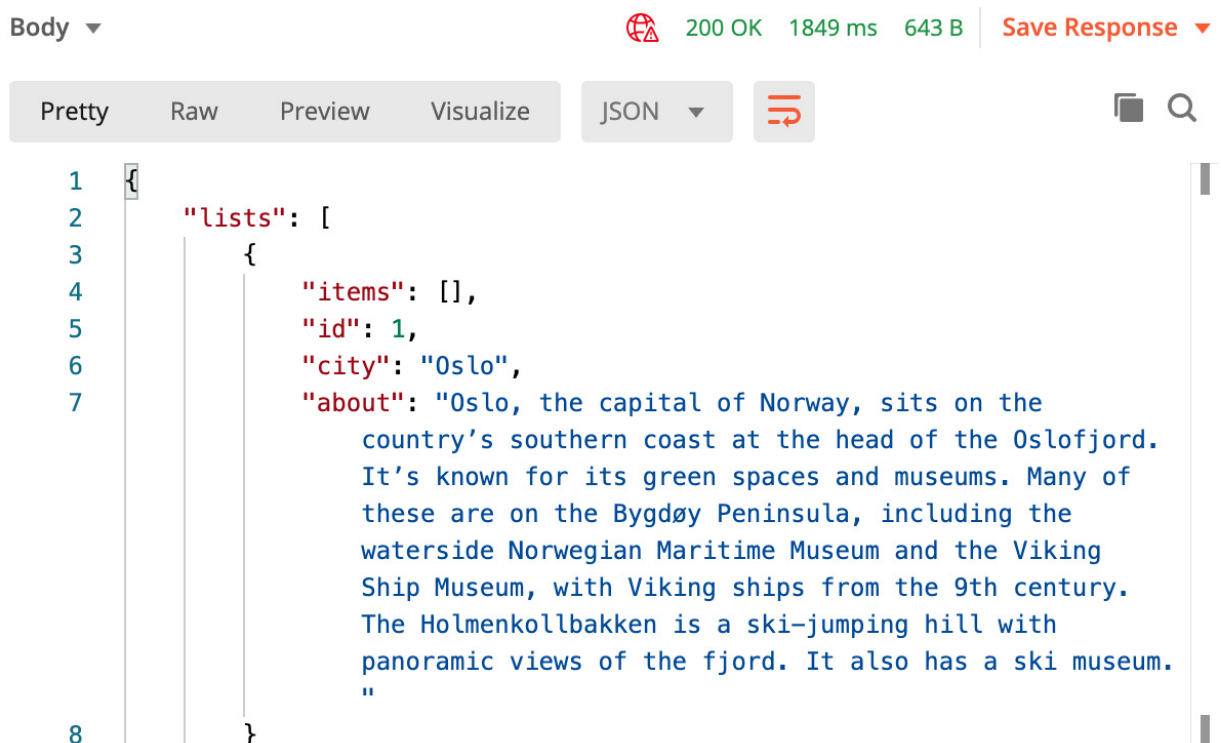


Figure 9.11 – 200 OK response from a protected API

How about if we send the wrong JWT? What do you think the application will send back? Delete some letters in the token and replace them with your name. Then hit the **Send** button to send the request:

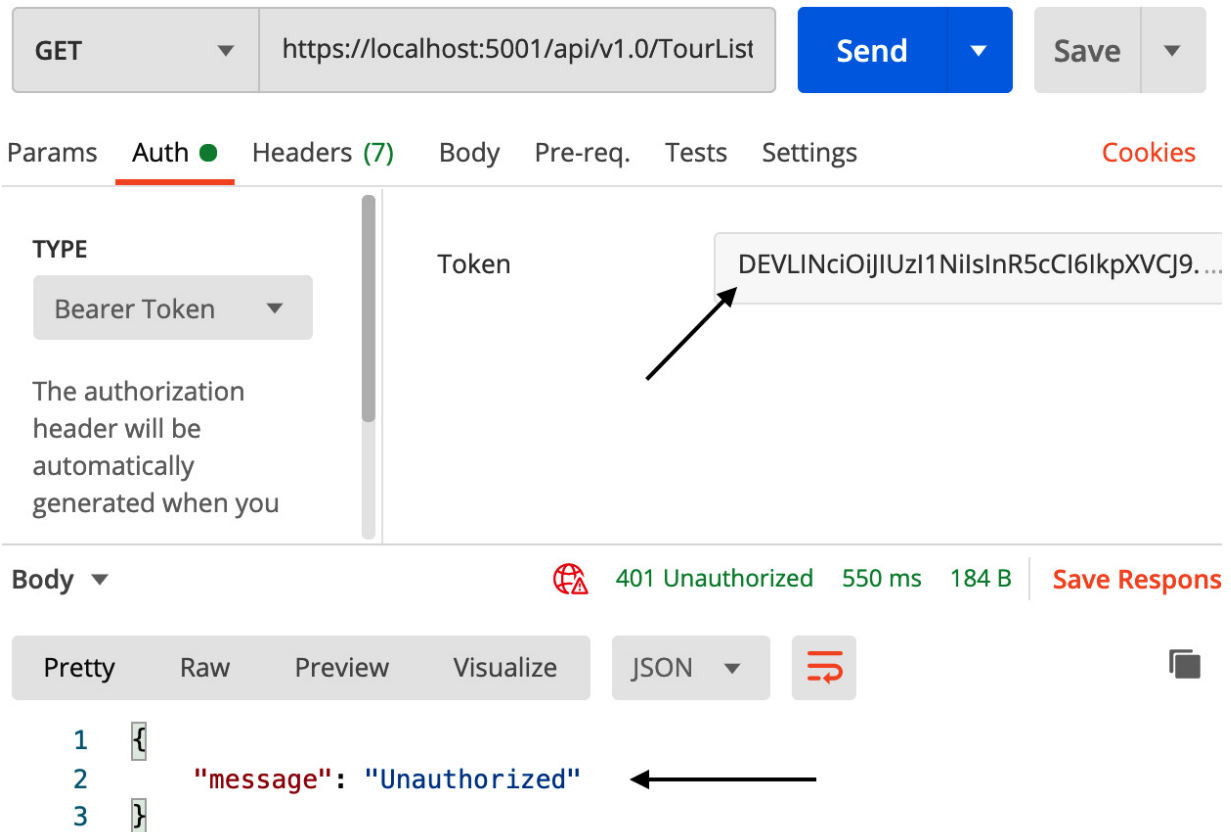


Figure 9.12 – Unauthorized request

You will see that the application responds back with a 401 status code and an **Unauthorized** message, as shown in the preceding screenshot.

That's how you would use Swagger UI with protected routes or endpoints. Now let's finish this chapter by summarizing what we have learned here.

## Summary

Here is a summary of this chapter. You've learned that ASP.NET Core Identity is an open source identity framework in .NET that provides you with the ability to manage user authentications and accounts. It has built-in middleware that enables the application to use cookie-based authentication.

You've learned that the IdentityServer4 framework takes away the heavy lifting for implementing OAuth 2 and OpenID Connect in your ASP.NET Core application. A fair use case for IdentityServer4



is when you want a centralized authentication server that authenticates requests from different services using token-based authentication.

You've also learned that CIAM is a cloud-based identity provider that provides companies with analytics, security, and a good customer experience. It's scalable, has centralized user management, and is easy to set up single sign-on and MFA.

You have learned how to secure an ASP.NET Core 5 application using the JWT. The application responds with a token, which has a payload of user information, to an authenticated request.

Lastly, you've also learned how to show protected APIs in Swagger UI and how to include an authorization header in Swagger's every request.

In the next chapter, we will talk about performance improvement using Redis caching.

## Chapter 10: Performance Enhancement with Redis

**Caching** is a common technique to boost the performance of an application. Typically, we encounter caching in **Content Delivery Network (CDN)**, HTTP caching, and database caching. Caching speeds up data retrieval performance by minimizing the need to access the underlying slower data storage layer. The caching technologies that we will learn about in this chapter are in-memory caching and distributed caching.

We will cover the following topics in this chapter:

- In-memory caching in ASP.NET Core
- Distributed caching
- Setting up and running Redis
- Implementing Redis in ASP.NET Core 5

### Technical requirements

Here is what you need to complete this chapter:

- Visual Studio 2019, Visual Studio for Mac, or Rider
- Redis
- **AnotherRedisDeskTopManager**, found at <https://www.electronjs.org/apps/anotherredisdesktopmanager>

Here is the link to the final code of this repository: <https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter10>.

### In-memory caching in ASP.NET Core

Using in-memory caching allows developers to store data in the server's resources, specifically in memory. Hence, it helps developers improve performance by removing unnecessary HTTP requests to external data sources.

Implementing in-memory caching in ASP.NET Core is dead simple. However, we will not be applying this in our application. We will opt for a more scalable way of caching, that is, distributed caching. We will just look at the parts of how to implement in-memory caching so you'll have an idea.

### Enabling in-memory caching in ASP.NET Core

I repeat, we're not going to apply the code from this section to our application. Anyway, you can enable in-memory caching in **ConfigureServices** of **Startup.cs**:

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

The **extension** method adds a non-distributed in-memory implementation in .NET. You can start using in-memory cache without installing any NuGet packages. Then inject **IMemoryCache** in a controller that needs caching:

```
[Route("api/[controller]")]
[ApiController]
public class CacheController : ControllerBase
{
    private readonly IMemoryCache _memoryCache;
    public CacheController(IMemoryCache memoryCache)
    {
        _memoryCache = memoryCache;
    }
}
```

After injecting **IMemoryCache** from the **Microsoft.Extensions.Caching.Memory** namespace, you can then start using it. The following code block simply checks whether the cake list exists in cache and returns from there if it is **true**. Otherwise, it will use the service and store the result:

```
[HttpGet("{cakeName}")]
public async Task<List<string>> Get(string
    cakeName)
{
    var cacheKey = cakeName.ToLower();
    if (!_memoryCache.TryGetValue(cacheKey, out
        List<string> cakeList))
    {
        cakeList = await Service.GetCakeList(
            cakeName);
        var cacheExpirationOptions =
```

```

        new MemoryCacheEntryOptions
        {
            AbsoluteExpiration =
                DateTime.Now.AddHours(6),
            Priority = CacheItemPriority.
                Normal,
            SlidingExpiration =
                TimeSpan.FromMinutes(5)
        };
        _memoryCache.Set(cacheKey, cakeList,
            cacheExpirationOptions);
    }
    return cakeList;
}

```

You will also notice in the code block that there's an expiration in cache. **AbsoluteExpiration** is for a definite expiration date, while **SlidingExpiration** is for monitoring the cache's inactive status or merely to put when the last time it was used was.

Although in-memory caching consumes its server's resources, in-memory is faster than distributed caching because it is physically attached to a server, but is not ideal for large and multiple web servers.

A piece of advice here. In-memory caching is not recommended when running multiple instances of one solution because data will not be consistent. There's a better caching method when working on multiple servers, which is discussed in the next section.

## Distributed caching

A **distributed cache** or **global cache** is a single instance or a group of cache servers with a dedicated network. As the applications hit the distributed cache, if cached data related to the application's request does not exist, the request redirects to the database to query the data. Otherwise, the distributed cache will simply respond with the data needed by the applications.

Here's a diagram of two servers sharing the same Redis instance for distributed caching:

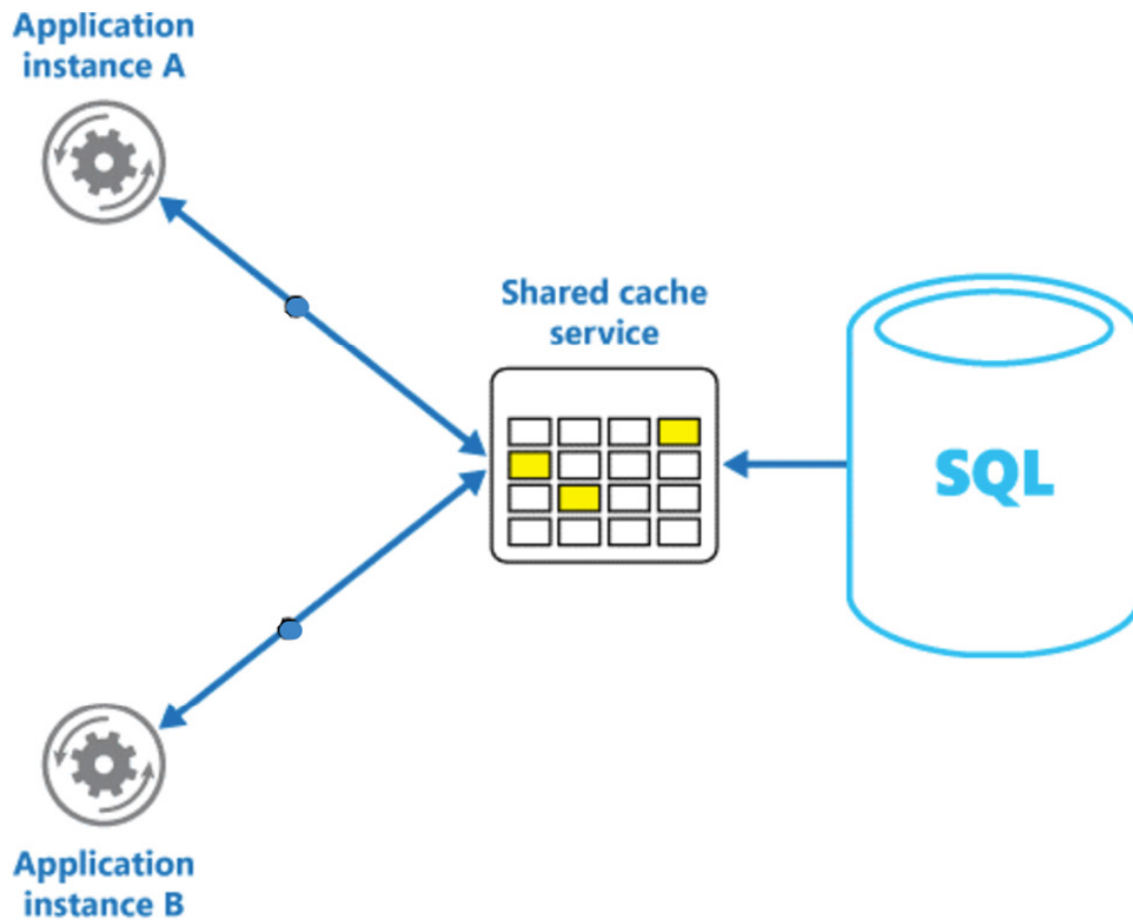


Figure 10.1 – Distributed caching

The preceding diagram shows the requests from two servers hitting a Redis cache first before deciding whether to query from the database or not.

What happens if one of your services crashes? Nothing really because everyone is going to be querying the distributed cache anyway. And because the cache is distributed, it's going to be maintaining the data consistency. We can offload all that information and all that headache to the distributed cache, which is Redis most of the time. Distributed cache is slower than in-memory, but it's more accurate.

One of the reasons why you would need a distributed cache is for higher accuracy. For instance, if a server crashes, it doesn't take its data to its grave. It is more resilient in that way.

Another reason is that you can scale the distributed cache, or your Redis cache, independently. You can scale the Redis instance independently while keeping your web services running properly without using their resources for caching.

# Setting up and running Redis

Redis officially supports Linux and macOS but not Windows because the engineers who wrote Redis use BSD Unix. The Windows port was written by some volunteer developers called the **Microsoft Open Tech group**.

Let's install Redis on Windows, macOS, and Ubuntu. The following are the steps, depending on your OS.

## For Windows users

1. Go to <https://github.com/microsoftarchive/redis/releases/tag/win-3.0.504> to download the installer of Redis for Windows:

The screenshot shows the GitHub release page for Redis 3.0.504. On the left sidebar, there is a 'Latest release' badge, the repository name 'win-3.0.504', a commit hash '10a978f', and a 'Compare' button. The main content area displays the version '3.0.504' in large blue text, followed by a release note from 'enricogior' dated July 1, 2016, mentioning 2 commits. The note states that this is a critical bug fix release for Redis on Windows 3.0, advising users to upgrade urgently if running a previous version in a cluster configuration. It also mentions that the release is based on antirez/redis 3.0.5 plus Windows-specific fixes and directs users to the release notes for details. Below the text, there is a section titled 'Assets' with a count of 4. The assets listed are: 'Redis-x64-3.0.504.msi' (6.42 MB), 'Redis-x64-3.0.504.zip' (5.6 MB), 'Source code (zip)', and 'Source code (tar.gz)'.

Asset	Size
Redis-x64-3.0.504.msi	6.42 MB
Redis-x64-3.0.504.zip	5.6 MB
Source code (zip)	
Source code (tar.gz)	

Figure 10.2 – Redis MSI installer and ZIP file

2. Download and extract the Redis ZIP. Double-click the **redis-server** file. Allow the permission dialog box that will pop up by accepting **Yes**. The Redis instance will automatically start.

To check whether the installation is complete, run the following command in your terminal:

```
redis-cli ping
```

**redis-cli** is the CLI for Redis functionalities. You should see a **pong** response from the terminal.

The following is another way to install Redis by using the **msi** file from the download link.

Download and install the **msi** file by just clicking it. Allow the permission dialog box that will pop up by accepting **Yes**. The Redis instance will automatically start.

To check whether the installation is complete, run the following command in your terminal:

```
redis-cli ping
```

**redis-cli** is the CLI for Redis functionalities. You should see a **pong** response from the terminal:

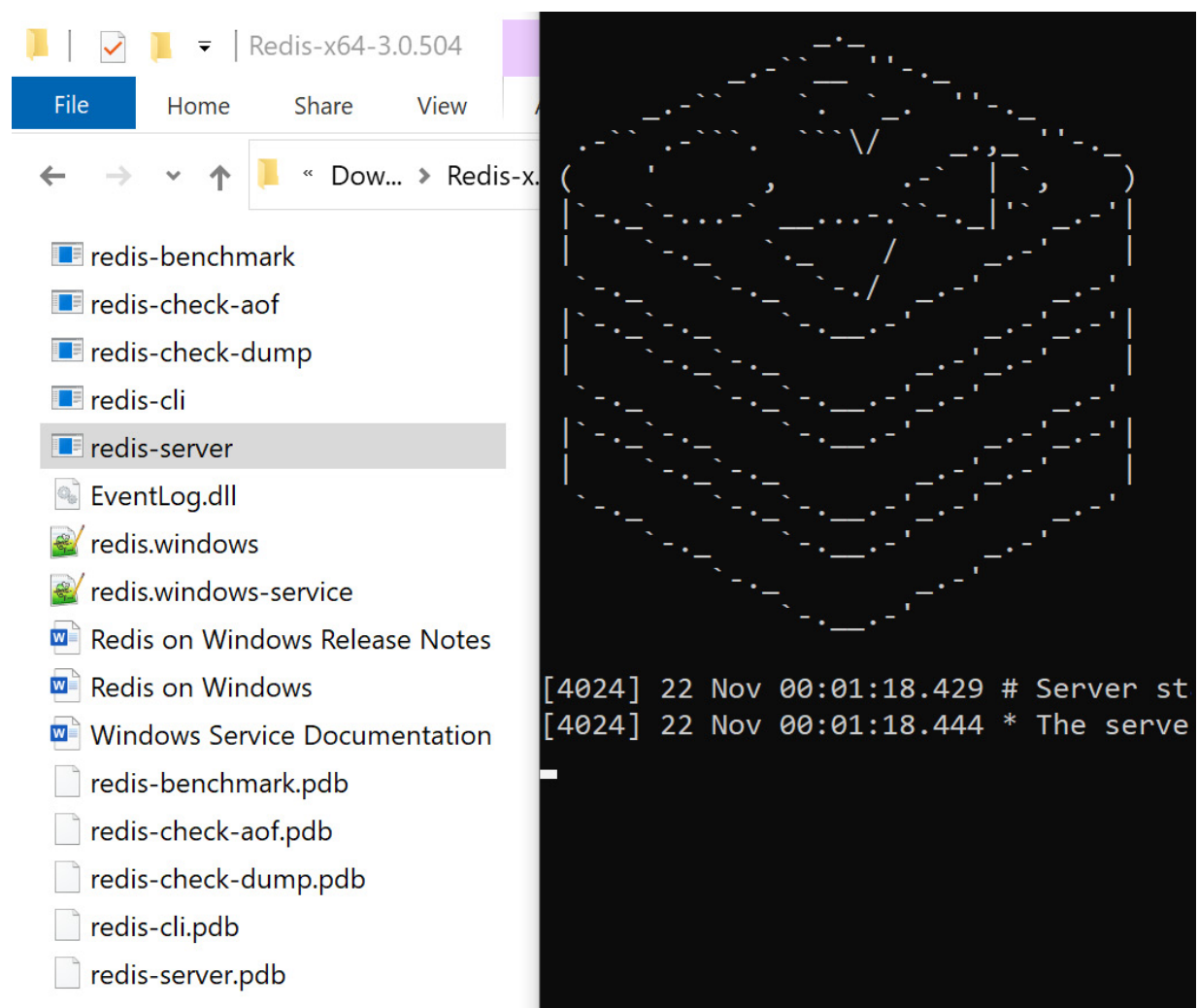


Figure 10.3 – Extracted files and the running Redis instance on the Windows terminal

Here are the extracted files from the ZIP file and CMD in Windows, which shows the Redis image after clicking the **redis-server** file.

If you are thinking of using the Chocolatey package manager for installing Redis, the URL is broken at the time of writing. I received an error saying **404 not found**.

That's it. Redis is now installed on your Windows 10 system.

## For macOS users

You can quickly install Redis on Mac using **brew**:

1. First, update **brew** by running the following command:

```
brew update
```

2. Next, we install Redis by running the following command:

```
brew install redis
```

3. Then, let's start the installed Redis by running the following command:

```
brew services start redis
```

4. Now run the following command to check whether Redis is running and reachable:

```
redis-cli ping
```

**redis-cli** is the CLI for Redis functionalities. You should see a **pong** response from Terminal.

### *NOTE:*

*The Redis installation using **brew** works in macOS Big Sur, which has had the biggest change since the original macOS.*

That's it. Redis is now installed on your macOS.

## For Linux or Ubuntu users

It's simple to install Redis in Linux:

1. Let's update our resources first by running the following command:

```
sudo apt update
```

2. Then install Redis by running the following command:

```
sudo apt install redis-server
```

3. Now run the following command to check whether Redis is running and reachable:

```
redis-cli ping
```

**redis-cli** is the CLI for Redis functionalities. You should see a **pong** response from Terminal. That's it. Redis is now installed on your Linux machine.



So, that's installing the Redis server on Windows, macOS, and Linux machines. Now let's use Redis in ASP.NET Core 5.

## Implementing Redis in ASP.NET Core

So, let's use the Redis we just installed on our machines by integrating it with our existing ASP.NET Core 5 solution. Here are the steps:

1. Go to the **Travel.Application** project and install these NuGet packages. The following NuGet package is a distributed cache implementation of the **Microsoft.Extensions.Caching.StackExchangeRedis** namespace using Redis:

```
Microsoft.Extensions.Caching.StackExchangeRedis
```

The following NuGet package helps us retrieve the configuration in **appsettings.json**:

```
Microsoft.Extensions.Configuration
```

The following NuGet package is a JSON framework for .NET:

```
Newtonsoft.Json
```

2. Next, we update the **DependencyInjection.cs** file of the **Travel.Application** project with the following code:

```
namespace Travel.Application
{
    public static class DependencyInjection
    {
        public static IServiceCollection
            AddApplication(this IServiceCollection
                services, IConfiguration config)
        {
            services.AddAutoMapper(Assembly.GetExecutingAssembly());
            services.AddValidatorsFromAssembly(Assembly.
                GetExecutingAssembly());
            services.AddMediatR(Assembly.GetExecutingAssembly());
            services.AddStackExchangeRedisCache(options =>
            {
                options.Configuration = config.GetConnectionString("RedisConnection");

                var assemblyName = Assembly.
                    GetExecutingAssembly().GetName();
                options.InstanceName = assemblyName.

```

```

        Name;

    });

    ...

    return services;

}

}

```

The dependency injection implementation of the preceding **Travel.Application** now requires an **IConfiguration** parameter. We are adding the Redis distributed caching services to the dependency injection container. The name of the connection string is **RedisConnection**, which we will set up in the next step.

3. Next is to go to the **Travel.WebApi** project and update **appsettings.json** with the following code:

```

{
  "AuthSettings": {
    "Secret": "ReplaceThsWithYourOwnSecretKeyAnd
      StoreItInAzureKeyVault!"
  },
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=
      TravelTourDatabase.sqlite3",
    "RedisConnection": "localhost:6379"
  },
  "Logging": {
    ...
  },
  "MailSettings": {
    ...
  },
  "AllowedHosts": "*"
}

```

4. We are adding connection strings for Redis and SQLite3 in this code. Consequently, we are also going to update **DependencyInjection.cs** of **Travel.Data**. So, let's update that with the following code:

```

namespace Travel.Data
{

```

```

public static class DependencyInjection
{
    public static IServiceCollection
        AddInfrastructureData(this IServiceCollection
            services, IConfiguration config)
    {
        services.AddDbContext<ApplicationDbContext>(options
=> options
            .UseSqlite(config.GetConnectionString("DefaultConnecti
on"))));
        ...
    }
}

```

The dependency injection file of **Travel.Data** now has the **DefaultConnection** configuration defined in **appsettings.json**.

5. Another thing to do here is to update the **Startup.cs** file of **Travel.WebApi**. Go to that file and update it with the following code:

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddApplication(Configuration);
    ...
    services.AddHttpContextAccessor();
    services.AddControllers();
    ...
}

```

We are now passing the **IConfiguration Configuration** in the **AddApplication** extension method. By doing so, **Travel.Application** can access **RedisConnection** in **appsettings.json**.

6. Now let's use Redis to cache the response of the **localhost:5001/api/v1.0/TourLists** endpoint to its consumers sending a **GET** request. To do this, we will update the handler of **api/v1.0/TourLists** for the **GET** request, which is **GetToursQuery**.

**GetToursQuery** can be found at

**Travel.Application/TourLists/Queries/GetTours/GetTours/GetToursQuery.cs**. Update **GetToursQuery.cs** with the following code:

```

...
using Microsoft.Extensions.Caching.Distributed;

```

```

using Newtonsoft.Json;

...

namespace Travel.Application.TourLists.Queries.GetTours
{
    public class GetToursQuery : IRequest<ToursVm> { }

    public class GetToursQueryHandler :
        IRequestHandler<GetToursQuery, ToursVm>
    {
        private readonly IApplicationDbContext
            _context;
        private readonly IMapper _mapper;
        private readonly IDistributedCache
            _distributedCache;

        public GetToursQueryHandler(
            IApplicationDbContext context, IMapper
            mapper, IDistributedCache distributedCache)
        {
            _context = context;
            _mapper = mapper;
            _distributedCache = distributedCache;
        }

        public async Task<ToursVm> Handle(
            GetToursQuery request, CancellationToken
            cancellationToken)
        {
            ...
        }
    }
}

```

We are injecting **IDistributedCache** from the **Microsoft.Extensions.Caching.Distributed** namespace into the constructor of **GetToursQueryHandler**. We will use **distributedCache** inside the **Handle** method's logic, which I truncated for readability purposes.

The following code is the updated business logic of the **Handle** method:

```

public async Task<ToursVm> Handle(GetToursQuery
    request, CancellationToken cancellationToken)
{
    const string cacheKey = "GetTours";
    ToursVm tourLists;
    string serializedTourList;
    var redisTourLists = await
        _distributedCache.GetAsync(cacheKey,
            cancellationToken);
    if (redisTourLists == null)
    {
        tourLists = new ToursVm
        {
            Lists = await _context.TourLists
                .ProjectTo<TourListDto>(_mapper.
                    ConfigurationProvider)
                .OrderBy(t => t.City).
                ToListAsync(
                    cancellationToken)
        };
        serializedTourList =
            JsonConvert.SerializeObject(tourLists);
        redisTourLists =
            Encoding.UTF8.GetBytes(serializedTourList);
        var options = new DistributedCacheEntryOptions()
            .SetAbsoluteExpiration(DateTime.Now.AddMinutes(5))
            .SetSlidingExpiration(TimeSpan.FromMinutes(1));
        await _distributedCache.SetAsync(
            cacheKey, redisTourLists, options, cancellationToken);
        return tourLists;
    }
    serializedTourList = Encoding.UTF8.GetString(
        redisTourLists);
    tourLists = JsonConvert

```

```

.DeserializeObject<ToursVm>(serializedTourList);

        return tourLists;
    }

```

The preceding block of code is the updated logic of the **GetToursQuery** handler. We have **"GetTours"** as **cacheKey**, and we are going to use that to retrieve data from the cache and save data from the cache. **cacheKey** will be used for the lookup when searching for a particular cache.

We are also checking whether there is an existing cache or not through **\_distributedCache.GetAsync**. If there is no data, we serialize the **tourLists** object and save it in the cache, **\_distributedCache.SetAsync**, before returning **tourLists**. We are caching the data in Redis, but we are putting an expiration. **SetAbsoluteExpiration** sets an absolute expiration time while **SetSlidingExpiration** sets how long the entry can be inactive.

If there's data, we return a deserialized **tourLists**.

Now, before we move on to Vue.js in the next chapter, [Chapter 11](#), *Vue.js Fundamentals in a Todo App*, let's clean up the **Startup.cs** file because it is starting to look messy.

What we are going to do is move the Swagger configuration into its directory and files, and then arrange all services and remove all unnecessary **using** statements.

7. So, go to **Travel.WebApi** and create a folder named **Extensions** in the **root** directory of the project. Create two C# files named **AppExtension.cs** and **Services.Extensions.cs**. We are moving the Swagger code from **Startup.cs** to these two files like so:

```

// AppExtension.cs

...

namespace Travel.WebApi.Extensions
{
    public static class AppExtensions
    {
        public static void UseSwaggerExtension(this
            IApplicationBuilder app,
            IApiVersionDescriptionProvider provider)
        {
            app.UseSwagger();
            app.UseSwaggerUI(c =>
            {
                ...
            });
        }
    }
}

```

```
    }  
}
```

Here, we migrated two middleware from the **Configure** method, namely **app.UseSwagger()** and **app.UseSwaggerUI()**, to the **AppExtension.cs** file.

```
// ServicesExtensions.cs
```

```
...
```

```
namespace Travel.WebApi.Extensions  
{  
    public static class ServicesExtensions  
    {  
        public static void AddApiVersioningExtension(  
            this IServiceCollection services)  
        {  
            services.AddApiVersioning(config =>  
            {  
                ...  
            });  
        }  
        public static void  
            AddVersionedApiExplorerExtension(this  
                IServiceCollection services)  
        {  
            services.AddVersionedApiExplorer(options  
                =>  
            {  
                ...  
            });  
        }  
        public static void AddSwaggerGenExtension(this  
            IServiceCollection services)  
        {  
            services.AddSwaggerGen(c =>  
            {  
                ...  
            });  
        }  
    }  
}
```

```

        ...
    });
}
}
}

```

Here, we migrated the **services.AddApiVersion()**, **services.AddVersionedApiExplorer()**, and **services.AddSwaggerGen()** services from the **ConfigureServices** method to **ServicesExtensions.cs**.

8. After moving the code to the **Extensions** directory, let's refactor **Startup.cs** by calling the **extension** methods that we created like so:

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddApplication(Configuration);
    services.AddInfrastructureData(Configuration);
    services.AddInfrastructureShared(Configuration);
    services.AddInfrastructureIdentity(Configuration);
    services.AddHttpContextAccessor();
    services.AddControllers();
    services.AddApiVersioningExtension();
    services.AddVersionedApiExplorerExtension();
    services.AddSwaggerGenExtension();
    services.AddTransient<IConfigureOptions<SwaggerGenOptions>, ConfigureSwaggerOptions>();
}

```

Now, let's see the middleware of our application:

```

public void Configure(IApplicationBuilder app,
    IWebHostEnvironment env,
    IApiVersionDescriptionProvider provider)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseSwaggerExtension(provider);
    }
}

```



```

app.UseHttpsRedirection();
app.UseRouting();
app.UseMiddleware<JwtMiddleware>();
app.UseAuthorization();
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
});
}

```

The preceding code is the refactored block of middleware of **Startup.cs**. The middleware is now cleaner than it was.

Again, remove the unused **using** statements that you'll find in the **Startup.cs** file.

Let's run the application to see whether Redis is working correctly:

9. Send a **GET** request to **/api/v1.0/TourLists** using Postman. Don't forget to include your JWT. The following screenshot shows the response time of the first request to the ASP.NET Core 5 application, which is more than 2 seconds:

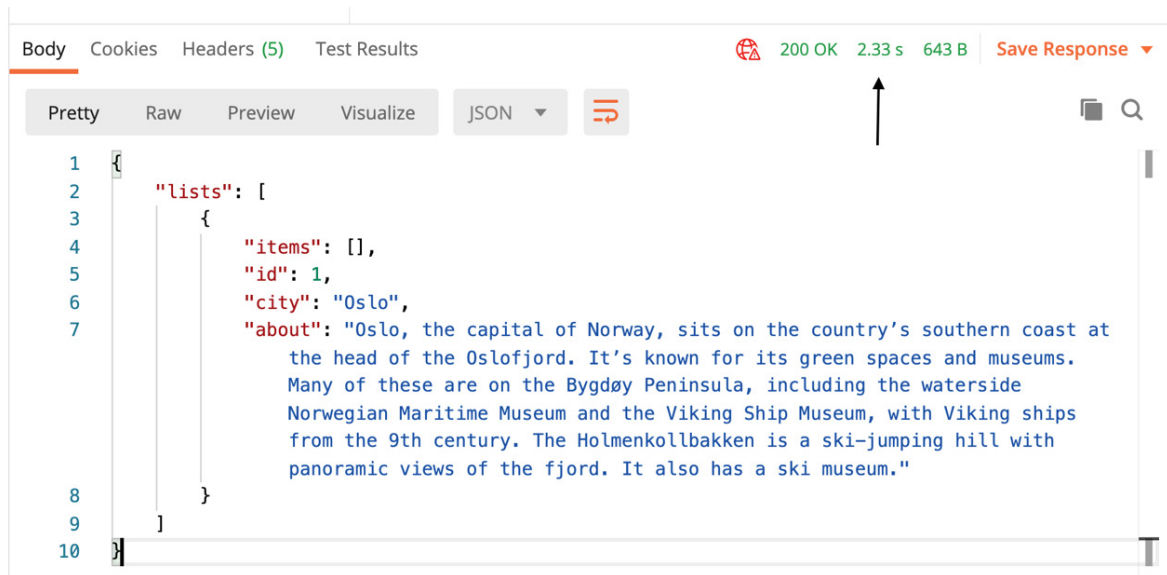


Figure 10.4 – API response without Redis cache

10. Let's send the same request to the same API to see whether the response time will be shorter:

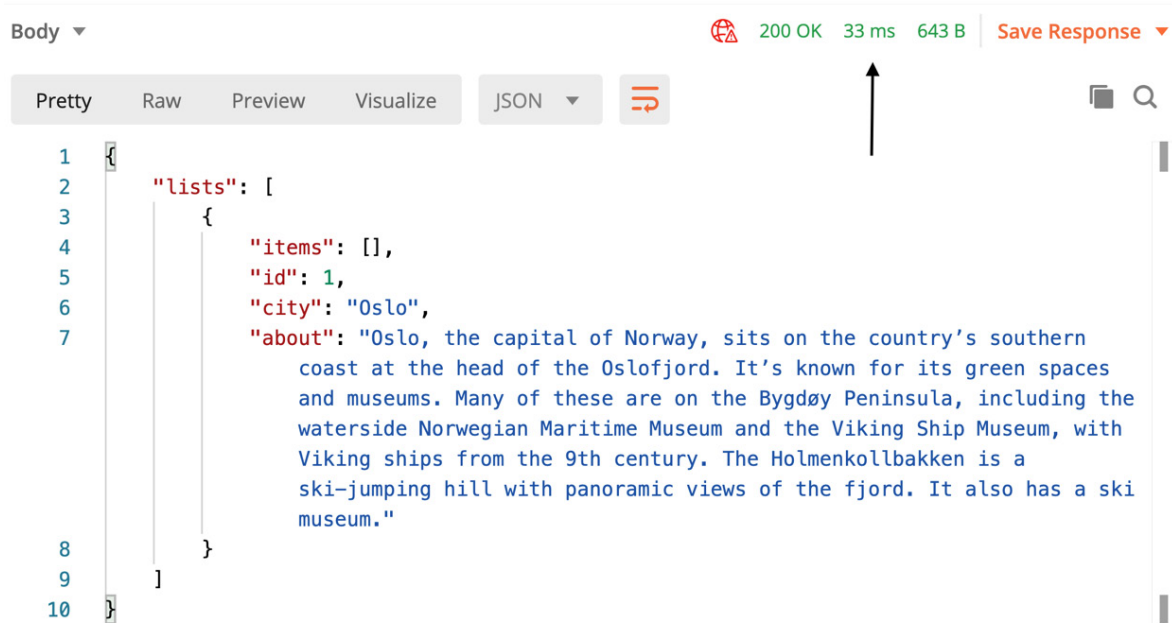


Figure 10.5 – API response with Redis cache

The preceding screenshot shows the shorter response time, 33 milliseconds, of the second **GET** request due to the cache that was stored during the first **GET** request to the same API.

11. To see the cache in Redis, you can use a Redis manager tool. Here's a free Redis manager tool that you can download and install, <https://www.electronjs.org/apps/anotherredisdesktopmanager>, and the paid version is found at <https://rdm.dev/>. RDM is a paid app for Windows and macOS users, but not for Linux users.
12. After running the Redis manager tool, send a new request to the `/api/v1.0/TourLists` API and check your Redis manager tool.

Let's check the cache in Windows 10 v20H2, macOS Pro Big Sur, and Ubuntu v20.10 Groovy Gorilla. These OSes are the latest versions at the time of writing this book.

The following screenshot shows **AnotherRedisDeskTopManager** running on Windows:

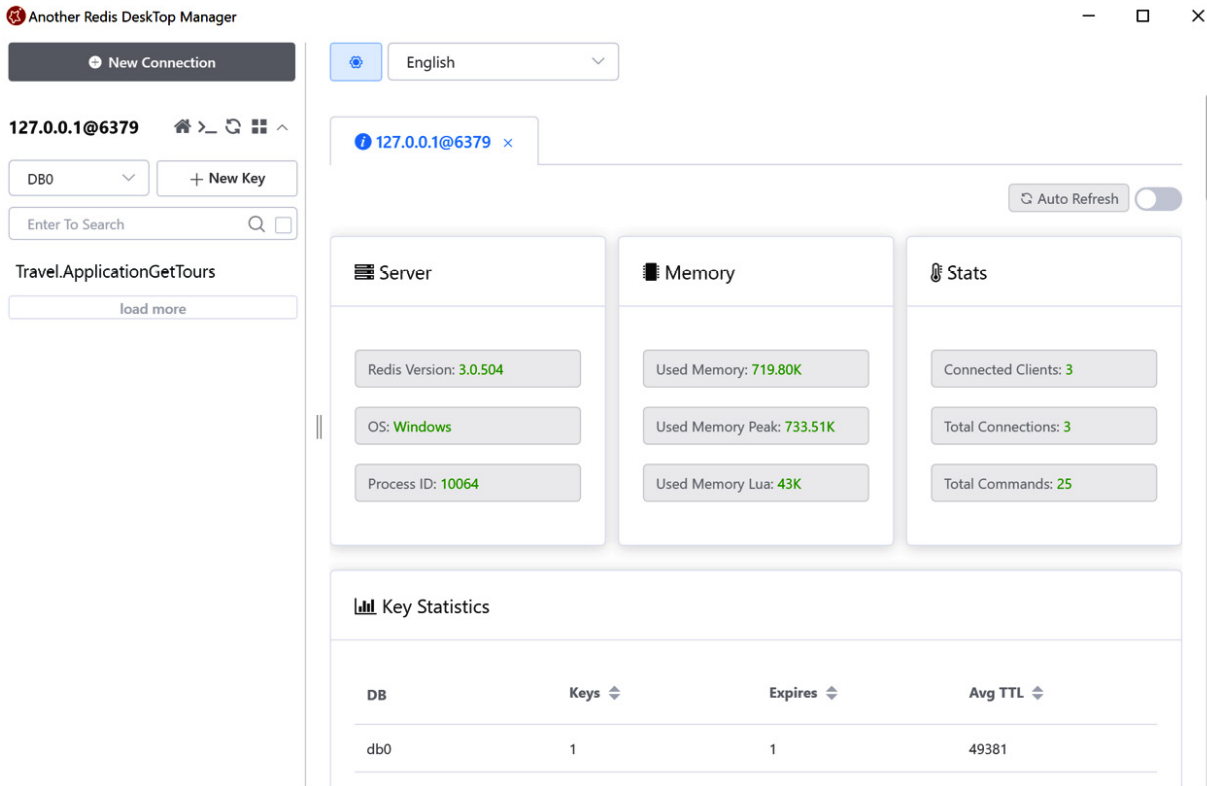


Figure 10.6 – Another Redis DeskTop Manager on Windows

The following screenshot shows **AnotherRedisDeskTopManager** running on macOS:

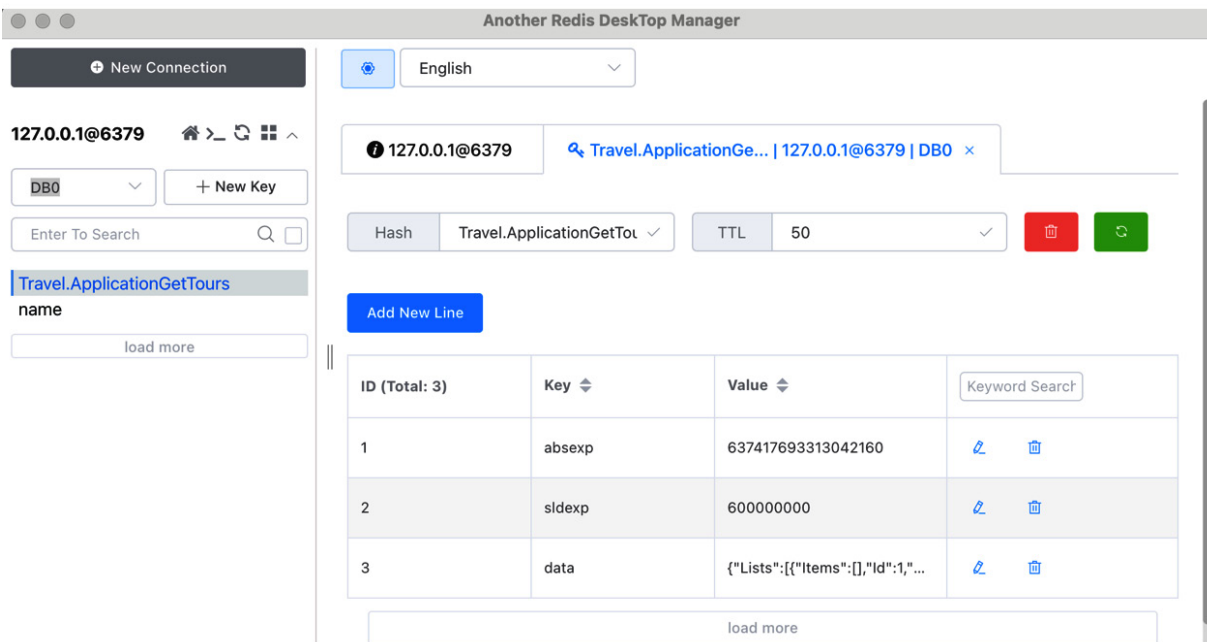


Figure 10.7 – Another Redis DeskTop Manager on macOS

The following screenshot shows the Redis GUI running on Ubuntu:

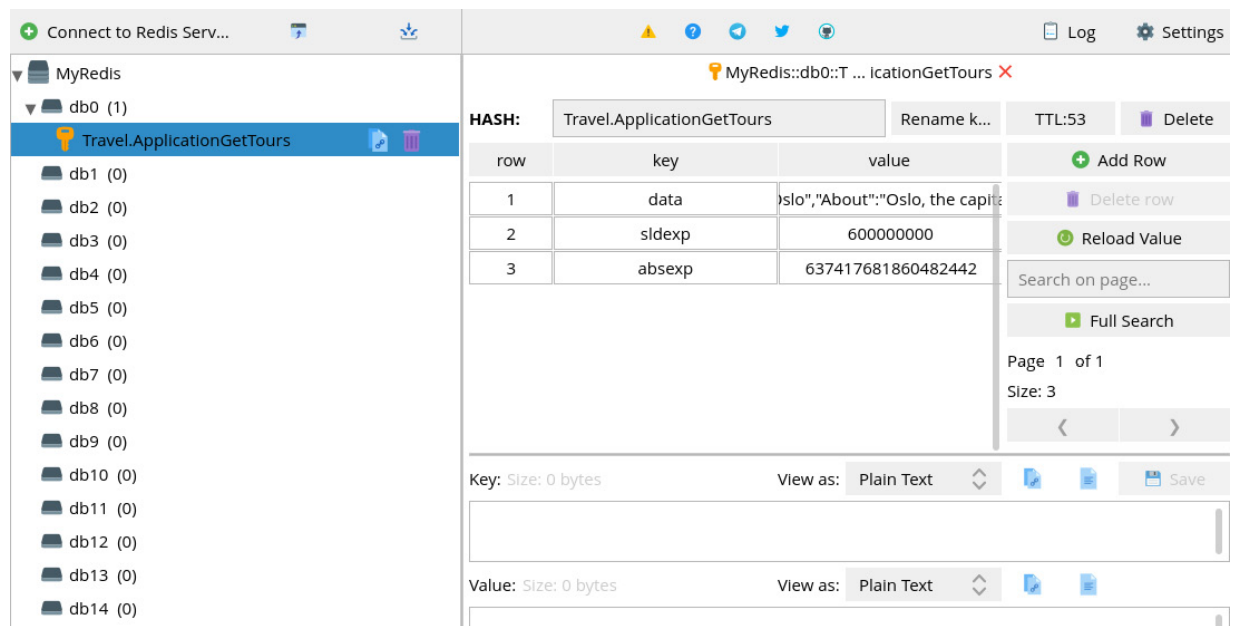


Figure 10.8 – Redis GUI on Ubuntu

If you don't like any dashboard or GUI for Redis, you can also use a CLI command to debug or monitor every command processed by your Redis server. Run the following command to enable monitoring:

**redis-cli monitor**

The following screenshot shows you how your request would look in the command line after running **redis-cli monitor**:



The preceding code is updating **Username** in the **User** class.

**Travel.Application/Dtos/Tour/TourPackageDto.cs:**

```
...

    public string WhatToExpect { get; set; }
    public float Price { get; set; }
    public string MapLocation { get; set; }
    public void Mapping(Profile profile)
    {
        profile.CreateMap<TourPackage,
            TourPackageDto>()
            .ForMember(tpDto =>
                tpDto.Currency, opt =>
                    opt.MapFrom(tp =>
                        (int)tp.Currency));
    }
}
```

The preceding code is updating the **TourPackageDto** class.

**Travel.Application/Dtos/Tour/TourListDto.cs:**

```
public TourListDto()
{
    TourPackages = new List<TourPackageDto>();
}

public IList<TourPackageDto> TourPackages { get; set; }
public string Country { get; set; }
```

The preceding code is updating the **TourListDto** class.

**Travel.Application/Dtos/User/AuthenticateRequest.cs:**

```
public string Email { get; set; }
```

The preceding code is updating **Username** in the **AuthenticateRequest** class.

**Travel.Application/Dtos/User/AuthenticateResponse.cs:**

```
public string Email { get; set; }
```

```
...
```

```
Email = user.Email;
```

The preceding code is updating **Username** in **AuthenticateResponse**.

#### **Travel.Application/TourLists/Commands/CreateTourList/CreateTourListCommand.cs:**

```
var entity = new TourList { City = request.City, Country =  
    request.Country, About = request.About };
```

The preceding code is adding **properties** in **CreateTourListCommand**.

#### **Travel.Application/TourLists/Commands/UpdateTourList/UpdateTourListCommand.cs:**

```
entity.Country = request.Country;  
entity.About = request.About;
```

The preceding code is adding properties in **UpdateTourListCommand**.

Create a new folder inside the **TourPackages** directory and name it **Queries**. Inside the Queries, create two new C# files and name them **GetTourPackagesQuery.cs** and **GetTourPackagesValidator.cs**.

#### **Travel/Application/TourPackages/Queries/GetTourPackagesQueryValidator.cs:**

```
using System.Collections.Generic;  
  
... // for brevity, please see the code in the Github  
  
using Travel.Application.Dtos.Tour;  
  
namespace Travel.Application.TourPackages.Queries  
{  
    public class GetTourPackagesQuery : IRequest  
        <List<TourPackageDto>>  
    {  
        public int ListId { get; set; }  
    }  
  
    public class GetTourPackagesQueryHandler :  
        IRequestHandler<GetTourPackagesQuery, List<  
            TourPackageDto>>  
    {  
        private readonly IApplicationDbContext _context;  
        private readonly IMapper _mapper;  
        public GetTourPackagesQueryHandler(  
            IApplicationDbContext context, IMapper mapper)  
        {  
            _context = context;  
            _mapper = mapper;  
        }  
    }  
}
```

```

    }

    public Task<List<TourPackageDto>>
        Handle(GetTourPackagesQuery request,
            CancellationToken cancellationToken)
    {
        var tourPackages = _context.TourPackages
            .Where(tp => tp.ListId == request.ListId)
            .OrderBy(tp => tp.Name)
            .ProjectTo<TourPackageDto>(_mapper.
                ConfigurationProvider)
            .ToListAsync(cancellationToken);

        return tourPackages;
    }
}

```

There is nothing new in the preceding code. The new file simply adds a query for getting a tour package.

#### **Travel.Application/TourPackages/Queries/GetTourPackagesQueryValidator.cs:**

```

using FluentValidation;

namespace Travel.Application.TourPackages.Queries
{
    public class GetTourPackagesQueryValidator :
        AbstractValidator<GetTourPackagesQuery>
    {
        public GetTourPackagesQueryValidator()
        {
            RuleFor(x => x.ListId)
                .NotNull()
                .NotEmpty().WithMessage("ListId is
                    required.");
        }
    }
}

```



The preceding lines of code add a new validator before querying a tour package.

#### **Travel.Identity/Services/UserService.cs:**

```
Email = "yoursuperhero@gmail.com",
var user = _users.SingleOrDefault(u => u.Email ==
    model.Email &&
...
u.Password == model.Password);
...
Subject = new ClaimsIdentity(new[] { new Claim("sub",
    user.Id.ToString()), new Claim("email", user.Email) }),
```

The preceding code is updating the **UserService** class.

#### **Travel.Identity/Helpers/JwtMiddleware.cs:**

```
var userId = int.Parse(jwtToken.Claims.First(c => c.Type ==
    "sub").Value);
```

The preceding code is updating the **AttachUserToContext** method of the **JwtMiddleware** class.

#### **Travel.WebApi/Controllers/v1/TourPackagesController.cs:**

```
[HttpGet]
public async Task<ActionResult<List<TourPackageDto>>>
    GetTourPackages([FromQuery] GetTourPackagesQuery query)
{
    return await Mediator.Send(query);
}
```

The preceding code is a new **Action** method of **TourPackagesController**.

Now, after the updating of the code in your repository, it's time to challenge yourself.

### **EXERCISE/PRACTICE TIME:**

*To enforce your learning here and before proceeding to the frontend part, I'd like you to create an ASP.NET Core 5 application. The application should use all the things you have learned in this book so far, such as clean architecture, CQRS, API versioning, OpenAPI, and distributed caching, without authentication or with authentication, or using an Identity as a Service such as Auth0 to save you time. An application that I can think of right now is an online store of video games. The entities could be **Developer**, **Game**, **GameReviews**, **Genre**, **Publisher**, and so on. This exercise is an easy task that you will be able to finish within a week. I know that you can do it. Good luck!*

OK, so let's summarize what you have learned in this chapter.

## Summary

You have finally reached the end of this chapter, and you have learned a lot of stuff. You have learned that in-memory caching is faster than distributed caching because it's closer to the server. However, it's not suitable for multiple instances of the same server.

You've learned that distributed caching solves in-memory caching in multiple instances because it provides a single source of truth of cached data for all the server instances.

You've learned how to install and run Redis on PCs, macOS, and Linux machines and how to integrate Redis into an ASP.NET Core Web API to improve the application's performance and bring a better user experience to the end users.

In the next chapter, you are going to build your first single-page application using Vue.js 3.

## Section 3: Frontend Development

This section deals with the real-world scenario of developing a Vue.js 3 application. The following chapters are included in this section:

- [Chapter 11](#), *Vue.js Fundamentals in a Todo App*
- [Chapter 12](#), *Using a UI Component Library and Creating Routes and Navigations*
- [Chapter 13](#), *Integrating a Vue.js Application with ASP.NET Core*
- [Chapter 14](#), *Simplifying State Management with Vuex and Sending GET HTTP Requests*
- [Chapter 15](#), *Sending POST, DELETE, and PUT HTTP Requests in Vue.js with Vuex*
- [Chapter 16](#), *Adding Authentication in Vue.js*

## Chapter 11: Vue.js Fundamentals in a Todo App

This chapter is entirely devoted to Vue.js, the Node.js npm, and the Vue CLI. These tools help developers to scaffold Vue.js projects with different configurations based on the user's options. This chapter also describes the Vue component's features and what you can do with them. Not only that, but you will also learn about the real-world structuring of a frontend web framework. We will do everything that I just mentioned in a Todo app using TypeScript.

In this chapter, we will cover the following topics:

- Starting a project using the Vue CLI
- Files and folders generated by the Vue CLI
- Getting started with a Vue component
- Common features in a Vue component

## Technical requirements

Here is what you need to complete this chapter:

- **Visual Studio Code:** <https://code.visualstudio.com/>
- **npm:** The Node Package Manager from <https://nodejs.org/en/>
- **Vue CLI:** <https://cli.vuejs.org/>

And the finished repository for this chapter can be found at the following link:

<https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter11/vue3-typescript-todo>

## Starting a project using the Vue CLI

**Vue CLI** is the standard development tool for starting a Vue.js project. The CLI lets you add different supports in a project, such as support for Babel, ESLint, TypeScript, **Progressive Web Apps (PWAs)**, PostCSS, unit testing, and end-to-end testing.

Make sure you have the npm that we installed in [Chapter 2, Setting Up a Development Environment](#). If you have forgotten to install the npm, you can go to <https://nodejs.org/en/> and install the latest **Long Term Support (LTS)** version of Node.js.

If you weren't able to install the Vue.js CLI in [Chapter 2, Setting Up a Development Environment](#), you can do so now by running the following command:

```
npm install -g @vue/cli
```

The preceding command installs the Vue CLI globally. The last part of the **npm** command is the package name, while **-g** means *globally*.

After installing the Vue CLI, let's create our first Vue.js app and build a simple Todo app to try out the common features of the Vue component:

1. Run the following command to create the directory of your Vue.js app:

```
vue create todo-app
```

The **vue create** command will also trigger the Vue CLI to open up the series of configurations for the Vue app.

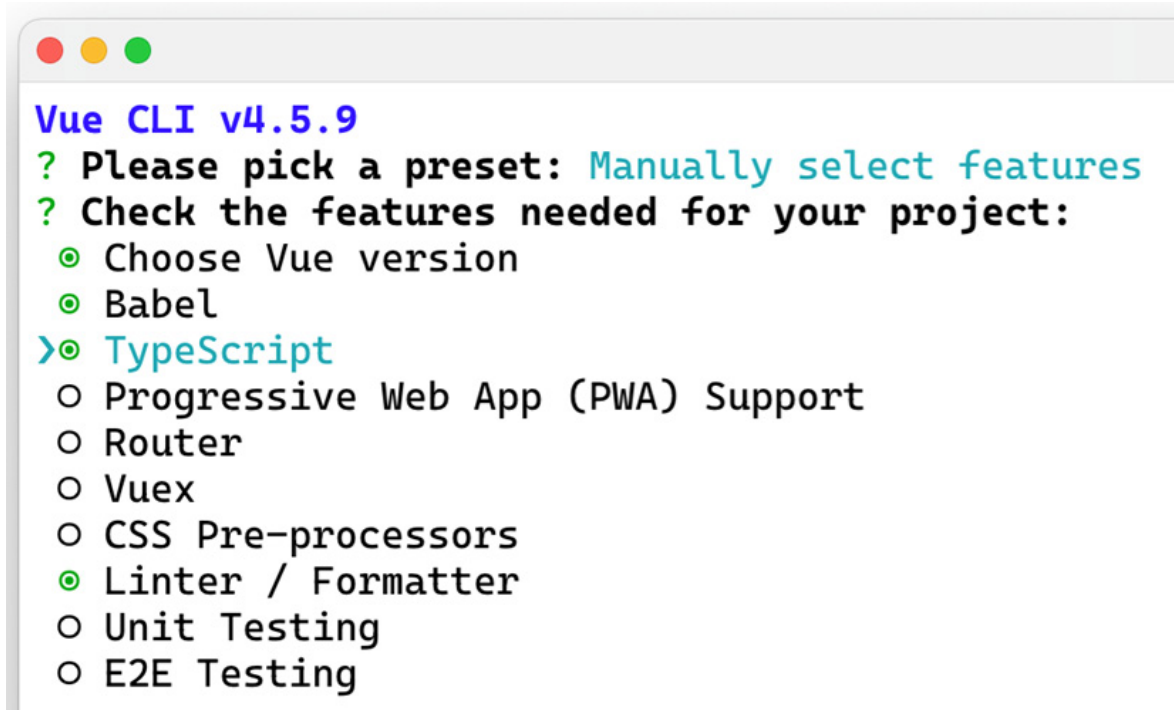
2. You should see the first list of options requesting the preset of the app, as per the following screenshot:



Figure 11.1 – Vue CLI configuration options

Choose **Manually select features** and press *Enter* to see the various options to configure.

3. The **Choose Vue version**, **Babel**, and **Linters / Formatters** options are enabled by the Vue CLI. Let's also turn on **TypeScript** by pointing the cursor to **TypeScript**, as shown, and then press the *spacebar*:

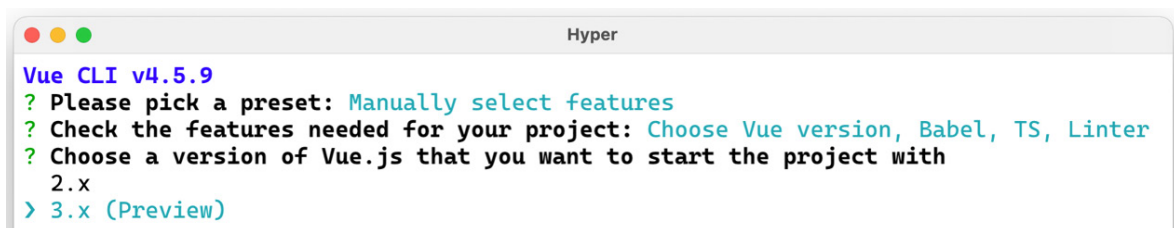


```
Vue CLI v4.5.9
? Please pick a preset: Manually select features
? Check the features needed for your project:
  ☒ Choose Vue version
  ☒ Babel
  > ☒ TypeScript
    ☐ Progressive Web App (PWA) Support
    ☐ Router
    ☐ Vuex
    ☐ CSS Pre-processors
    ☒ Linter / Formatter
    ☐ Unit Testing
    ☐ E2E Testing
```

Figure 11.2 – Adding TypeScript to the Vue.js CLI configuration

Press *Enter* to add **TypeScript** support to the Vue.js project we are creating.

4. Let's use version 3 of Vue.js in the project by selecting the **3.x (Preview)** option, as shown in the following screenshot:

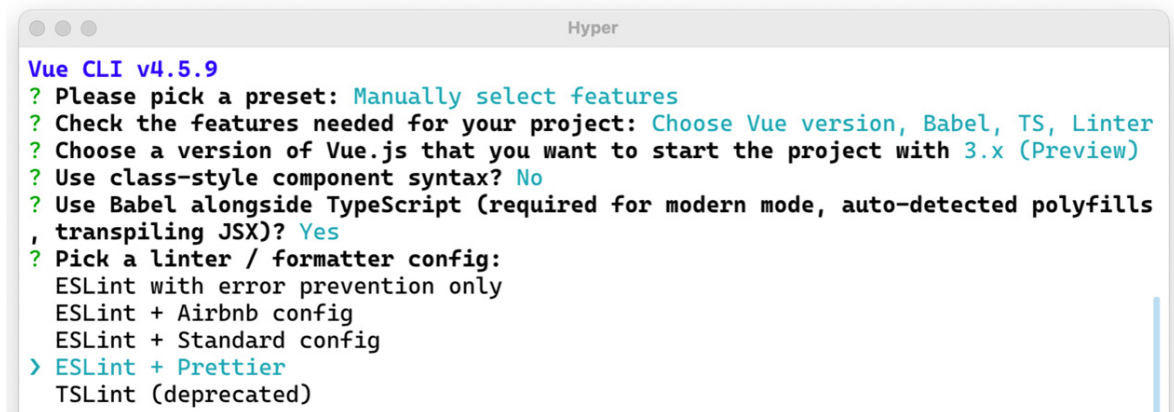


```
Vue CLI v4.5.9
? Please pick a preset: Manually select features
? Check the features needed for your project: Choose Vue version, Babel, TS, Linter
? Choose a version of Vue.js that you want to start the project with
  2.x
  > 3.x (Preview)
```

Figure 11.3 – Choosing Vue.js 3

Press *Enter* to choose Vue.js 3.

5. The next question is whether we would like to use **class-style component syntax**. Press *Enter* to refuse. The default selected value in this question is **No**, so pressing *Enter* answers no to this configuration. We will not be using **class-style component syntax**.
6. Another question is whether we would like to use **Babel alongside TypeScript**. Press *Enter* to accept. Babel is a transpiler that will help your Vue.js app autodetect polyfills, making a Vue.js app compatible with most browsers.
7. As regards **linter / formatter** while we are writing our code, let's choose **ESLint**, a static code analysis tool, plus **Prettier**, an opinionated code formatter, as shown in the following screenshot:



```
Vue CLI v4.5.9
? Please pick a preset: Manually select features
? Check the features needed for your project: Choose Vue version, Babel, TS, Linter
? Choose a version of Vue.js that you want to start the project with 3.x (Preview)
? Use class-style component syntax? No
? Use Babel alongside TypeScript (required for modern mode, auto-detected polyfills, transpiling JSX)? Yes
? Pick a linter / formatter config:
  ESLint with error prevention only
  ESLint + Airbnb config
  ESLint + Standard config
> ESLint + Prettier
  TSLint (deprecated)
```

Figure 11.4 – Formatter configuration in the Vue CLI

Press *Enter* after using your arrow keys to point to the **ESLint + Prettier** configuration.

8. For the next option, which is for an additional lint feature, select **Lint on save**.
9. Then, select dedicated configuration files for the preferred placing of configuration for Babel, ESLint, and so on.
10. Lastly, don't accept **saving the preset for future projects** for now.

The Vue CLI will start creating our project, initialize the project with Git, install the CLI plugins, and add JavaScript packages.

Now, start your VS Code text editor and install the extensions, namely, **Prettier – Code formatter**, **Vetur**, and **vscode-icons**, as you can see in the following screenshot:



Figure 11.5 – VS Code extensions

Prettier reformats code to make it more readable, Vetur is the Vue language server, while vscode-icons is designed to bring icons to your VS Code text editor's files and folders.

You can now use your Visual Studio Code to open up the project and see the files and folders that the Vue CLI has generated for you:

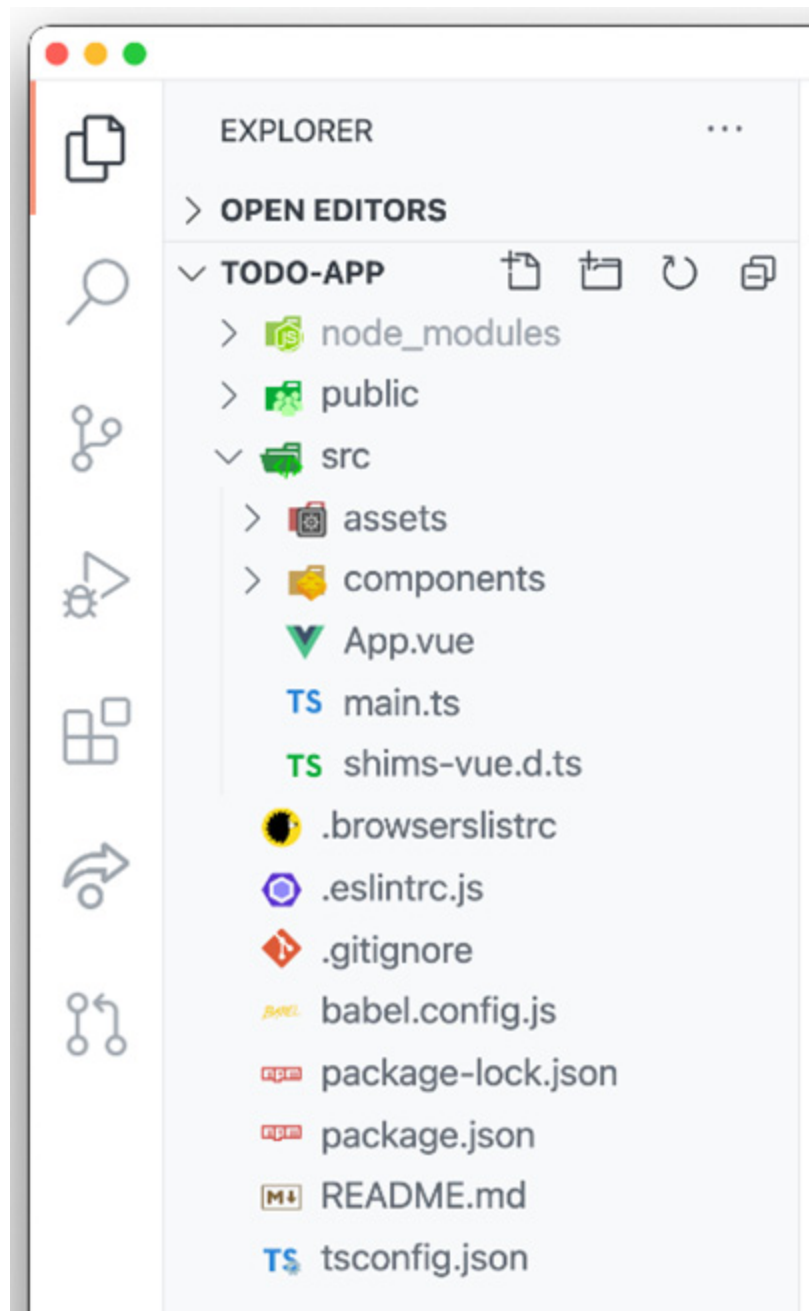


Figure 11.6 – Files and folders generated by the Vue CLI

The preceding screenshot shows the generated files and folders of the Vue CLI, and we're going to talk about these in the next section.

## Files and folders generated by the Vue CLI



The Vue CLI has generated the files and folders needed to get started with our Vue.js development. Let's look at each of these in turn:

- **node\_modules folder:** This folder contains the libraries downloaded from the npm.
- **public folder:** This folder contains the HTML file and favicon. You will only see one HTML file in the public folder, hence the term single-page application.
- **src folder:** This folder is the directory where we will write business logic, create Vue file components, and JavaScript or TypeScript files.
- **.browserlistrc:** This file is a tool for describing the app's target browsers.
- **.eslintrc.js:** This file is a configuration tool for ESLint.
- **.gitignore:** This file is used for not committing directories or files in Git. An excellent example of a directory that must be ignored is **node\_modules**. The file size of **node\_modules** is large but can always be retrieved in the project by running **npm install** in the root project directory.
- **babel.config.js:** This file is for a project-wide configuration of Babel, a JavaScript transpiler. Babel transpiles older JavaScript versions (ES6+) to lower JavaScript versions (ES5 and below) that older browsers can understand.
- **package-lock.json:** This file locks the dependencies of the project to specific versions of packages.
- **package.json:** This file holds the information of our project's dependencies and scripts, which we can use to run in the terminal.

There are three elements of **package.json** that I would like you to remember:

**scripts:** The **scripts** property is where you can declare some custom CLI commands. The Vue.js CLI has three default **npm** scripts in our project, namely, the server for running the Vue project through the **vue-cli-service serve** command, the build for building the Vue project for deployment, and the lint for running the linter of the Vue project. The idea here is that we can convert long commands into short commands by merely creating a key and value pair in the script's block.

**dependencies:** The **dependencies** property is simply a list of critical packages that our app needs in order to run. Any libraries that we install through the **npm install** command will be listed inside the dependencies.

**devDependencies:** The **devDependencies** property is a list of packages that help developers write apps. We can explicitly tell the npm that a library must be included in **devDependencies** by adding the **-D** flag. For instance, the **npm install -D prettier** command will add the prettier package to **devDependencies**. Also, **devDependencies** don't get compiled or included when you run the **npm build** command for the app.

- **tsconfig:** This file lets us declare the compiler options required whenever we compile the project. It also indicates that the project is a TypeScript project.

Now we have a basic understanding of the generated files and folders that the Vue CLI has scaffolded for us. Let's run the application to test whether it will appear on the browser by running the following

command in your VS Code's terminal:

```
npm run serve
```

Head to **localhost:8080** in your browser to see the Vue.js app running, as shown in the following screenshot:

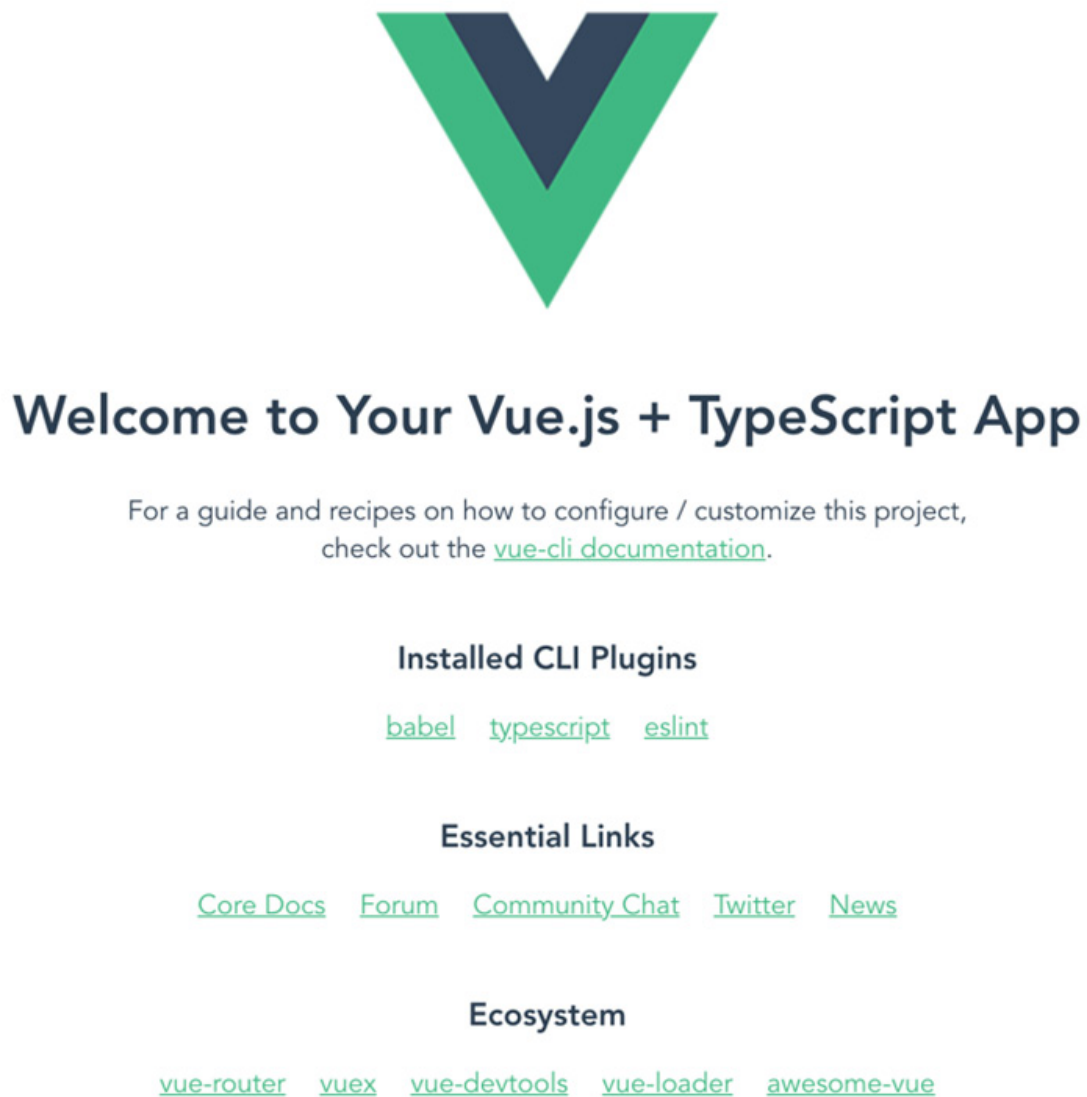


Figure 11.7 – Starting UI

You will see on the browser a welcome message with the Vue.js logo after starting the Vue.js application. The default UI has some external links concerning documentation of the features in Vue.js.

After seeing our Vue.js app running in the browser, we can now talk about the Vue component and its parts in the next section.

# Getting started with a Vue component

Vue components are the main building blocks for Vue.js applications. A component consists of **template** syntax, **script**, and **style**.

Let's check out the generated code in the **root** component, **App.vue**, of our application:

```
<template>

  <HelloWorld msg="Welcome to Your Vue.js + TypeScript App" />

</template>

<script lang="ts">
import { defineComponent } from "vue";
import HelloWorld from "../components/HelloWorld.vue";
export default defineComponent({
  name: "App",
  components: {
    HelloWorld
  }
});
</script>

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

Here are some brief and straight-to-the-point explanations of the parts of the Vue.js component:

The **template** syntax section is where we bind the state of our application to the HTML DOM of our choice. You can also see that the **HelloWorld** component is being used in the template. We will also be going to create some Vue components in the latter part of this chapter.

The **script** section is where we can write JavaScript/TypeScript code to develop some business logic in our app, import files or packages, and define our application's states. A **state** is an object where we can store property values and render them on the UI.

You can see in the script block that **HelloWorld** was imported and registered in the components property to allow the **App.vue** component to render **HelloWorld** in the **template** section.

The **style** section is where we define the styling of the component we are making. You can use CSS preprocessors here, such as SASS, LESS, Stylus, and PostCSS, but these require loader plugins in order to work.

Now, remove the **HelloWorld** component from the template and script sections, and then delete it from the components directory. You should only see the Vue.js logo on the browser after removing the **HelloWorld** component from the application.

Let's write our first Vue.js component in the next section.

## Writing a Vue component

Let's create a Vue component (the file ends in **.vue** format) inside the components directory and name it **TodoForm**. The filename should look like this – **TodoForm.vue**. Then write the following code:

```
<template>
  <h1>TodoForm Works!</h1>
</template>
```

We are not required to include any JavaScript/TypeScript code and CSS styling if we are only going to render a message in the UI. Now import it into the **App.vue** file like so:

```
<template>
  
  <TodoForm />
</template>

<script lang="ts">
import { defineComponent } from "vue";
import TodoForm from "../components/TodoForm.vue";
export default defineComponent({
  name: "App",
  components: {
    TodoForm,
  },
});
```

```
});  
</script>
```

**TodoForm** is imported in the same way that **HelloWorld** was imported and rendered in the UI. Now hit the **Save** button and see the results in the browser:



Figure 11.8 – Vue.js component

The implementation we did to create a Vue component was a quick-and-dirty way of making it, but we will add some features to it in the next section.

## Common features in a Vue component

In this section, we will use some of the common features in a Vue component while building our Todo app. There are many directives (custom HTML attributes), events, and interfaces that we can use in Vue.js, but most of them are rarely used, which you'll tend to forget over time. So, in this quick demo of the Vue.js Todo app, we will only use the most common events, directives, and APIs of a Vue component.

So let's get started.

## Writing local states in a Vue component

Since we are using TypeScript in this project, the first thing that I usually do is write the models that I will need in the app. So let's create a folder inside the **src** directory and name it **models**. Then, create a TypeScript file inside the **models** folder (**src | models | todoModel.ts**) and add the following code:

```
export type TodoType = {  
  id: string;
```

```

    done: boolean;
    content: string;
};
// OR
export interface TodoModel {
    id: string;
    done: boolean;
    content: string;
}

```

The shape or model of a state can be written using the type or interface. They are both the same; the difference is that you can't implement two or more interfaces, whereas the type allows us to do that. We will use a simple model here that has **id**, **done**, and **content** primitive type properties.

Now we are going to require some third-party packages. These are **uuid** for generating a Universally Unique Identifier for our app, and **@types/uuid** for the **Type** definitions of the UUID library. Our Vue.js TypeScript project will give us IntelliSense when using the **uuid** library, and lastly, Bootstrap 5 for styling our components.

So let's run the following command to start installing the packages:

```
npm i uuid @types/uuid bootstrap@next
```

The **i** is short for install, and the **@next** suffix of Bootstrap is explicitly indicating that we want the alpha or beta version of the library because the stable version of Bootstrap is v4 at the time of writing. However, you don't need **@next** after **bootstrap** if you see that the stable version of Bootstrap on the npm website is version 5.

Now let's use the Bootstrap package in our Vue.js project by replacing the styles generated in **App.vue** with the following code, which imports Bootstrap's CSS:

```

<style>
@import "../node_modules/bootstrap/dist/css/bootstrap.css";
</style>

```

After importing the Bootstrap in the root component, let's use the container from the Bootstrap package, one of the most basic layout element in Bootstrap. We can use it in a **div** tag and wrap the Vue logo and the **TodoForm** component like so:

```

<template>
  <div class="container">
    
    <TodoForm />
  </div>
</template>

```

```
</div>
</template>
```

Hit **Save** to save the updated file and look in your browser to see the changes in respect to the layout. Then, let's also update **TodoForm.vue** inside the components folder by changing the HTML and adding a state to the component.

Let's edit the message in the **TodoForm** component with the following code:

```
<template>

  <div class="mb-4">

    <h1>Vue 3</h1>

    <h2>TypeScript Demo {{ version }}</h2>

  </div>

</template>
```

The new message in **TodoForm.vue** is now Vue 3 TypeScript Demo, but the mustache syntax for text interpolation, **{{ version }}**, with the word version inside, will not be showing in the UI because this is a form of data binding between the UI and a state.

So let's now create a state for the **TodoForm** component:

```
<script lang="ts">

import { defineComponent, ref } from "vue";

export default defineComponent({

  name: "TodoForm",

  setup() {

    // local state

    const version = ref("v1");

    return { version }; // same as { version:version }

  },

});

</script>
```

We need **defineComponent** and **ref** from the Vue library. The **defineComponent** component is self-explanatory. **ref** is for creating and tracking a local state, the state for any changes that render it reactive, which rerenders the UI whenever the value of the state changes. We call **ref** and pass **an initial value v1** of the string type and store it in the version variable. Then we need to return it as a property of an anonymous JavaScript object.

Save and see the Vue app on the browser. You should see the **TypeScript Demo** message followed by **v1**, as shown here:



Figure 11.9 – Text interpolation data binding using a mustache

That's how easy it is to do the data binding. Now let's create another two local states, one with an explicit type and one with an implicit type.

Let's first import the **TodoType** model that we created:

```
import { TodoType } from "@models/todoModel";
```

The **@** sign in the path is a resolve alias provided by **tsconfig** to help us easily reach exported codes or libraries directly without writing too many dots and slashes. You can check out the **tsconfig** file's properties to see the value of the **baseUrl** and the paths:

```
// implicit type safe
const version = ref("v1");
const newTodo = ref(""); // ref<string> is too verbose
// explicit type safe
const todos = ref<TodoType[]>([]);
```

Update the **setup** function of your component with the new states indicated above. The **version** and **newTodo** states are written with implicit types. TypeScript knows what type you are passing, and it can infer what type you are returning from **ref**. You can hover your mouse over **version** and **newTodo**, and you'll see that they are string types.

The explicit way of telling the TypeScript compiler the initial value type is by using a generic in **ref**. You can see in the preceding code that **todos** is a **TodoType** type of array. This is a good way of



writing a state's static typing for readability purposes, and not only for IntelliSense.

You'll notice that we didn't explicitly add a type for the version and **newTodo**, both string types, because it is easy to tell what type a value is if it's a primitive type, such as a Boolean, string, and number.

OK, so we are done writing states in our **TodoForm** component and rendering it. Let's now add functionality to our **TodoForm** component.

## Adding a function in a Vue component

Writing a function in a Vue component is dead simple. The first step involves importing **uuid** below **TodoType** to generate a UUID for the ID of our model:

```
import { v4 as uuidv4 } from "uuid";
```

We are importing **v4** and renaming it to **uuidv4** for readability.

Now we can start adding a function where we can create a new **Todo** object:

```
// explicit type safe
const todos = ref<TodoType[]>([]);
function addNewTodo(): void {
  if (!newTodo.value) return;
  todos.value.push({
    id: uuidv4(),
    done: false,
    content: newTodo.value,
  });
  console.log("newTodo:", newTodo.value);
  newTodo.value = "";
}
```

The **addNewTodo** function, which returns **void**, checks whether **newTodo.value**, which we will connect to the input field, is empty. This is how we can get the value of our state, by accessing the **value** property. Then, if the value is not empty, we will push a **Todo** object to the todos. The **Todo** object will have a generated **id**, **done** set to **false**, and a **content** property whose value will come from **newTodo**. The function also logs the string value of **newTodo** and then sets it to an empty string to remove the user's input.

Then, let's include the **addNewTodo** function in the returning object of the setup, along with the **todos** and **newTodo**, like so:

```
return { version, todos, newTodo, addNewTodo };
```

As you can see in the preceding code, returning the states and function will make them available to be used in the template section of the Vue component.

Now place this HTML form below the **div** tag, which is wrapping the Vue3 TypeScript demo:

```
<form @submit.prevent="addNewTodo">
  <div class="mb-5">
    <label for="newTodo" class="form-label">New Todo
      </label>
    <input
      class="form-control"
      id="newTodo"
      placeholder="what's on your mind?"
      v-model="newTodo"
      name="newTodo"
    />
    <div class="m-2">
      <button type="submit" class="btn btn-primary">Add
        New Todo</button>
    </div>
  </div>
</form>
```

**@submit**, which is a submit listener, invokes the **addNewTodo** function we created every time a button with the **type="submit"** attribute gets hit. The **newTodo** input is binded to the input's **v-model**, directive which is a directive to create a two-way data binding between the input and the model or state. The rest of the details are basic HTML syntaxes and Bootstrap styling.

Save the file and check the browser. You should see the simple adding of the todo form under **TypeScript Demo v1**:



# Vue 3

## TypeScript Demo v1

New Todo

  
Add New Todo

Figure 11.10 – Simple form

The form is not yet ready to render what we add in the input, but we can see in the console whether it's working. Now open up your Chrome DevTools and go to the **Console** tab. Write **check all emails** in the input field and click the **Add New Todo** button of the app to see that it is being registered in the console:

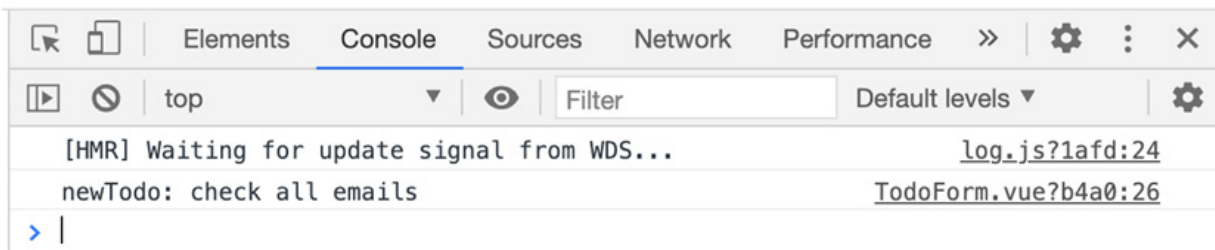


Figure 11.11 – Console log

The preceding *Figure 11.11* shows the **Console** tab and the **newTodo: check all emails** log in the DevTools. This is a quick **Proof of Concept (PoC)** that we can use to create functions in our Vue component.

We just did a PoC for a function in our Vue component. Let's now render the list of todos in our component.

## Looping in an array in a Vue component

Now we can add the UI for the array of **TodoType**, which is a collection, meaning we write a loop of the todos and render each item in the collection or list. We can do that by using the **v-for** directive.

Write the following below the **form** tag:

```
<div>
  <ul class="list-group">
    <li class="list-group-item" v-for="todo in todos"
      :key="todo.id">
      <h3>
        {{ todo.content }}
      </h3>
    </li>
  </ul>
</div>
```

The code does a **for each** loop in the todos and renders every todo's **content** property in the list. You will notice another Vue.js directive, **:key="todo.id"**, which aims to help the Vue.js runtime determine which specific object of the list alone needs to re-render whenever the **todos** state changes. In return, the rerendering of the DOM is more performant without the key.

The colon prefixed to the **key** directive is shorthand for the **v-bind** directive, which is used for one or more data binding attributes. The **key** directive (**key=" "**) needs a string value or a number value to work. A good example of what to pass in a key directive is an ID. However, the key directive does not have to be an ID, what matters is that it has to be unique.

Try the form again by writing **check all emails** in the field and hitting *Enter*. You should see **check all emails** in the UI after adding it, as shown in the following screenshot:

## New Todo

|what's on your mind?

Add New Todo

check all emails

Figure 11.12 – Rendering a list of todos

Now we can render our todos in the console logs and the screen of the browser.

What if we want to render something if the **todos** state is empty? What if we want to mark the todo as done and change the style of the todo from plain to with strikethrough? That's what we are going to write next.

## If-else conditions in a Vue component

We have two goals here in this section. Our first goal is to show a message of an empty list with a sad emoji face if the **todos** state is empty. You can get the emoji from the following link:

<https://emojipedia.org/>.

If the **todos** state is not empty, we can start showing the list of todos and hide the **Empty list** with a sad face message.

Place the code for an empty list UI between the form tag and list of the todos, like so:

```
</form>
<div v-if="todos.length === 0">
  <h3>Empty list ☹️</h3>
</div>
<div v-else>
  <ul class="list-group">
```

In this code, we use the **v-if** directive conditionally to render a block and the optional **v-else** directive as **else** in the condition.

Check out the browser and refresh your page to see the new UI, *Figure 11.13*. You should see the sad emoji face while the **todos** state is empty:

## New Todo

Add New Todo

## Empty list 🙄

Figure 11.13 – Empty list with the sad face

The empty list with the sad face immediately goes away when the **todos** state is no longer empty. Easy right?

Our second goal in this section is to create a functionality where we can update the **done** property of the **Todo** object. We will let the user know if **done** has turned into **true** by marking the todo content with a strikethrough. The strikethrough is a simple CSS style that we are also going to write in a bit.

So let's start. Write a new function named **toggleDone** that takes a **TodoType** object like so:

```
function toggleDone(todo: TodoType): void {  
    todo.done = !todo.done;  
}
```

The **toggleDone** function updates the **done** property to its opposite value every time the function gets invoked.

Now let's create a CSS class by putting it inside a **style** section like so:

```
<style>  
.mark {
```

```
    text-decoration: line-through;
  }
</style>
```

The style in the code is a line-through that will give the todos content a strike-through on it that will serve as a mark.

Now update the **h3** tag with these new attributes:

```
<h3
  :class="{ mark: todo.done }"
  style="cursor: pointer"
  @click="toggleDone(todo)"
>
```

**:class**, which is shorthand for **v-bind:class**, is a class directive that dynamically toggles the class depending on the Boolean value of **todo.done**. The **mark class** style gets activated if **todo.done** is **true**, and gets deactivated if **todo.done** is **false**. Simple yet helpful.

You will also notice here that there is **@click**. The **@** sign is shorthand for the **v-on** directive, which listens to DOM events to run or call JavaScript functions. You can see all the events in the IntelliSense of your editor when you start typing the **@** sign in any HTML tag. You will get **mouseover**, **blur**, **keyup**, and so on.

Let's check out the **toggleDone** function. The **toggleDone** function has also been placed, and we are passing the todo object that comes from the **v-for** looping directive you will find in the **<li>** element. Now we can test the new functionality of the app.

Then, enter **check all emails** and **yoga**. Then, click on **Check all emails** to mark it with strike-through, as shown in the following screenshot:

Add New Todo

~~check all emails~~

yoga

Figure 11.14 – Mark what's done with a strike-through

The strike-through means that the **done** property in the **todo** object was set to **true**.

Let's create more functionalities, such as marking all items in the list, deleting all items in the list, or just removing one item.

Write a **removeTodo** function that takes a **number** type like so:

```
function removeTodo(index: number): void {  
  todos.value.splice(index, 1);  
}
```

We are going to remove a specific **todo** object from the todos list.

Include **removeTodo** in the return object of the **setup** function of the component:

```
return {  
  ...,  
  removeTodo,  
};
```

**removeTodo** is now ready to use in the HTML.

Now let's update the **template** syntax, particularly the **li** tag:



```

<ul class="list-group">
  <li
    class="list-group-item d-flex flex-row justify-content-
      between align-items-center"
    v-for="(todo, index) in todos"
    :key="todo.id">
    <h3

```

We are updating the class and **v-for** of the **li** tag. You will notice that there's a second parameter, **index**, in **v-for**. That's the index of the todo item that we get from looping. And yes, we have access to that.

Next, we add the following code to the **h3** tag:

```

<button
  type="button"
  class="btn btn-warning"
  @click="removeTodo(index)"
>
  ✓ Done & Remove
</button>

```

We pass the index to the **removeTodo** function and name the button **Done & Remove** with a check emoji.

Now let's write two more functions and name them **markAllDone** and **removeAllTodos**, like so:

```

function markAllDone(): void {
  todos.value.forEach((todo) => (todo.done = true));
}

function removeAllTodos(): void {
  todos.value = [];
}

```

**markAllDone** sets all todos done to **true**, while **removeAllTodos** sets the todos to an empty array, removing all items in the list.

Then, include **markAllDone** and **removeAllTodos** in the **return** statement like so:

```

return {
  ...,
  markAllDone,

```

```
    removeAllTodos,  
  };
```

**markAllDone** and **removeAllTodos** are now available to use within the **TodoForm** component's **template** syntax.

Now, let's add two new buttons below the **Add New Todo** button:

```
<div class="m-2">  
  <button  
    type="button" class="btn btn-danger"  
    @click="removeAllTodos">  
    Remove All  
  </button>  
</div>  
  
<div class="m-2">  
  <button  
    type="button"  
    class="btn btn-success"  
    @click="markAllDone">  
    Mark All Done  
  </button>  
</div>
```

The two new buttons handle the **removeAlltodos** and **markAllDone** functions. Run the app and check the browser. We will get the following screenshot:

## New Todo

Add New Todo

Remove All

Mark All Done

<del>check all emails</del>	✓ Done & Remove
yoga	✓ Done & Remove
order pizza	✓ Done & Remove

Figure 11.15 – Remove All and Mark All Done

Add another todo title, **order pizza**, and click the **Mark All Done** button. You should see that all todos were struck through, as shown in *Figure 11.15*. Then, hit the **Remove All** button to remove all todos.

I am sure that you find Vue.js easy to use, and it's not that hard to add functionalities. Now, let's make it a little more complicated by passing down states from the parent to the child component in the next section.

## Creating and passing props

There are instances when a parent component has to pass a state or a function to its child component to render UI or create functionality, and we do it through props. Props is a term that is also used in other JavaScript frameworks or libraries, such as Angular, React, Svelte, and Ember.

To start, let's define props in the **TodoForm** component. Import **PropType** from Vue to get typesafe. You won't need this if your Vue project uses JavaScript, but it will be beneficial if you are using TypeScript:

```
import { defineComponent, ref, PropType } from "vue";
```

After importing **PropType**, create a model above **defineComponent** and name it **Props**, like so:

```
import { v4 as uuidv4 } from "uuid";

type Props = {
  title: string;
  subTitle: string;
};

export default defineComponent({
```

You can use a type or interface to create a model. Now, let's define **about props** of **TodoForm** and place it before the setup like so:

```
  name: "TodoForm",
  props: {
    about: {
      type: Object as PropType<Props>,
      required: true,
    },
  },
  setup() {
```

We are naming our props as **about**. **about** is a required prop, and it is an object of **PropType** with the shape of **Props**.

The next thing is to create a state name with a **title** string and subTitle string properties, like so:

```
// Vue 3
setup() {
  const about = ref({
    title: "Vue 3",
    subTitle: "TypeScript demo",
  });
  return {
    about,
  };
},
// Vue 2
/*
data:() => ({
  about: {
```

```

    title: "Vue 3",
    subTitle: "TypeScript demo"
  }
})
*/

```

Import **ref** and use it to create a state, store it in **about**, and then return **about**. You will notice here that I've put a Vue 3 label on top of the setup and a Vue 2 label on top of the data function. These are two different ways of writing a component in Vue.js that you will encounter when you start looking for a sample Vue project on the internet. I would say that 90% of what you will see at the time of this writing until the next few months will be written in Vue.js 2 because Vue.js 3 was just released in September 2020 and the majority of Vue.js communities and library authors haven't yet adopted the latest Vue.js version.

Before we can pass **about** of the **App.vue** component, we need to update the setup of **TodoForm**. So go back to the **TodoForm.vue** component and update the setup like so:

```

setup(props) {

```

There are optional parameters in the setup that we can use. The first parameter is **props**, which is the entry point of any state and functions that we will pass into a child component.

Let's map **props** to the **template** syntax of **TodoForm**, replacing the Vue 3 TypeScript demo message:

```

<template>

  <div class="mb-4">

    <h1>{{ about.title }}</h1>

    <h2>{{ about.subTitle }} {{ version }}</h2>

  </div>

```

The idea is that the **TodoForm** welcome message will come from its parent component from now on.

Now, let's update the **template** section of the **App.vue** component:

```

<TodoForm :about="about" />

```

Here we are passing the **about** state of **App.vue** to **about**, **:about**, and **props** of **TodoForm**.

You should see that the UI is still working, as shown in the following screenshot:

# Vue 3

## TypeScript demo v1

New Todo

Add New Todo

Remove All

Mark All Done

Empty list 🙄

Figure 11.16 – TodoForm with about props

The Vue3 TypeScript demo is rendering again. What if you don't see the Vue 3 TypeScript demo and you want to debug it? We can use a console log in the props, which we will do in the next section.

## Life cycle hooks in a Vue component

**Life cycle hooks** are functions or methods that get called automatically at each specific point in the life cycle of the component. We can tap into key events in the component's life cycle to write business logic in our app.

Let's use the most frequently used life cycle hooks, which are **onMounted** in Vue 3 or **mounted** in Vue 2. **onMounted** runs or triggers when the DOM is mounted:

```
import { defineComponent, ref, PropType, onMounted } from "vue";
```

We are importing **onMounted** from Vue.

Then we use **onMounted** and place it before the **return** statement of the **setup** function like so:

```
onMounted(() => console.log(props.about.title));  
return {
```

We are logging here the value of the **title** property of **about**, which you will see when you open your Chrome DevTools or Firefox DevTools, as shown in the following screenshot:

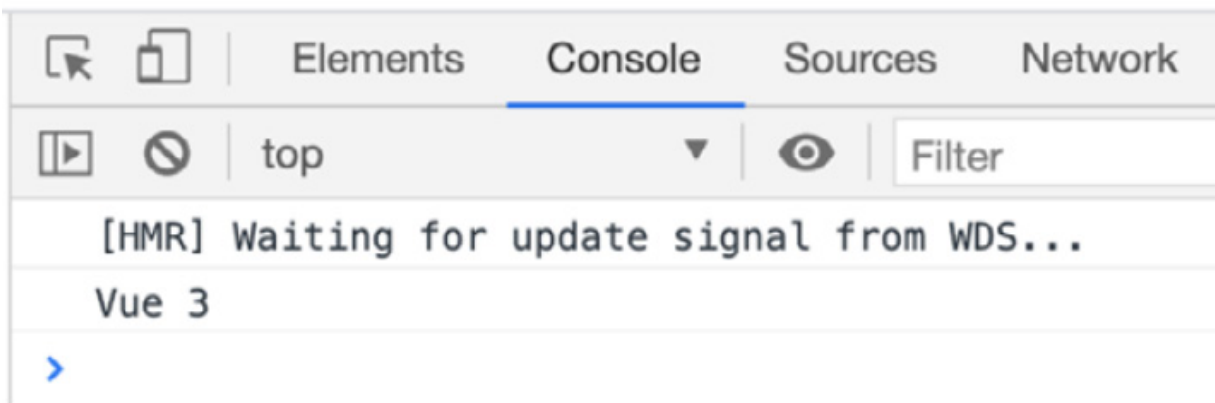


Figure 11.17 – Console logging props

You can see the Vue 3 log from the **about** props passed by **about the state** of the **App.vue** component. **onMounted** is what you will always use whenever you automatically render data from a web service on the UI for your web visitors. We will do **onMounted** with HTTP requests in our next app, the real-world Vue.js app with ASP.NET Core.

That's it. You can play around with the Todo app we just created. Now let's summarize all the things you have learned in building this simple Todo app.

## Summary

There are so many things that we've done in this chapter. You learned that the Vue CLI is a great tool for scaffolding a project that saves developers a lot of time. You learned the parts of a Vue component, namely, the template block, which is the UI part of the Vue, the script section for writing business logic, and the style block for styling the component. You also learned how to create a Vue component and how to use the common interfaces in Vue, such as **v-for** for looping, the **v-if** conditions, and the **@** sign for writing events.

You were also able to learn how to write states in a Vue component using **ref** and how to do data binding using the colon prefix or by using the double curly braces.

Lastly, you learned what props are and how to pass props between two Vue components. You also learned when to use life cycle hooks in a Vue component for triggering a function.

In the next chapter, we will start to develop our real-world enterprise Vue.js app.

## Further reading

I admit that I'm poor at remembering things. That's why, whenever I am learning a new technology in software development, I google for cheatsheets that help me to remember just the key aspects of a piece of technology. The following links are cheatsheets in Vue.js:

- **Vue Essentials cheat sheet:** [vuemastery.com/pdf/Vue-Essentials-Cheat-Sheet.pdf](https://vuemastery.com/pdf/Vue-Essentials-Cheat-Sheet.pdf)
- **Vue.js cheat sheet:** [devhints.io/vue](https://devhints.io/vue)



## Chapter 12: Using a UI Component Library and Creating Routes and Navigations

In this chapter, you will learn how to use open source UI libraries built by different Vue.js communities. You will become able to use one of the popular libraries in Vue.js, which will save you from spending countless hours building your components. Then you will set up the navigation and routing of your Vue.js 3 app with best practices in mind.

We will cover the following topics:

- Using a third-party UI component library
- Using other third-party UI libraries
- Adding navigation bars
- Writing page components
- Setting up Vue Router with lazy loading and eager loading

### Technical requirements

You need Visual Studio Code to complete this chapter. The finished project of this chapter can be found at this link: <https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter12/>.

### Using a third-party UI component library

What is a **UI component library**? It is merely a library that offers sets of ready-made UI components such as date pickers, forms, cards, navigations, buttons – you name it. Most UI components that you will need in an application are available in a mature UI component library.

What are the benefits of using a third-party UI component library, whether it is open source, free, or paid, that has premium support from the UI component library vendor? The following are the benefits that a UI component library can give you:

- **Saves time:** If creating a robust date picker with a ton of configuration, fewer bugs, and with cross-browser support will take you weeks to finish, using a third-party UI component library is a no-brainer – a UI component library that you can import and try a date picker of the UI component library in less than 4 minutes.
- **Save money:** Since a UI component library saves you countless hours, it saves your client or company money.
- **Reliable:** The top UI component libraries are highly maintained and have very large communities that improve the libraries.
- **Time to market:** If you can ship applications months earlier than your competitors, that can lead to the company's success.

The main takeaway here is not re-inventing the wheel and being conscious of your customer, client, or company's resource spending. A good developer knows how they can contribute to saving resources.

Now that we know why we should use a UI component library, let's include one in a Vue.js project.

## Setting up a Vue.js project and installing a UI component library

Let's create a new Vue.js project using the following steps:

1. Go to the **presentation** directory of the **Travel** application, and we will run the Vue CLI commands again. Inside the **presentation** folder of the **src** directory, run the following command:

```
vue create vue-app
```

The command will open up the config options for a Vue.js project. Choose **Manually** by pressing the *spacebar* and then *Enter*.

2. Unlike in the last chapter where you enabled the TypeScript, don't enable it this time but enable **Router** and **Vuex** instead. FYI, **Vuex** is for state management. Vuex version 3 and Vuex version 4 have the same interfaces.
3. Keep **Babel** and **Linter** enabled. The new Vue.js project's preset should look like *Figure 12.1*:

 Figure 12.1 – New Vue.js project's preset

Figure 12.1 – New Vue.js project's preset

4. After selecting the preset, choose **2.x** for the version of Vue.js as in *Figure 12.2*:


 Figure 12.2 – Vue.js version

Figure 12.2 – Vue.js version

5. After choosing the **2.x** version, the Vue CLI will ask you if you want to use history mode for the router. Select **Y** or yes by just pressing the *Enter* key on your keyboard.
6. Then pick **ESLint + Prettier** for the linter and formatter. Pick the configuration option with a label that says lint on the **Save** option.
7. And lastly, place the config for Babel and ESLint in dedicated config files.

The **presentation** directory should look like *Figure 12.3* after creating a project using the Vue CLI:

 Figure 12.3 – Presentation directory

Figure 12.3 – Presentation directory

The **Travel.WebApi** and the **vue-app** projects should be inside the **presentation** folder. Then run the newly created project:

```
npm run serve
```

This command will run the Vue.js project with a logo and a message that says **Welcome to your Vue.js app**. Now, why did we choose JavaScript this time and Vue.js version 2?

## Vue.js versions

We chose Vue.js version 2 because 95% of the tutorials on the internet are in Vue.js 2. The majority of the third-party libraries and plugins for Vue.js 3 are not yet compatible with that version. It's sad to say that even though Vue.js 3 was released 4 months ago, most authors of and contributors to third-party libraries and plugins are still migrating their code to the latest Vue.js version. Shout out to all contributors and authors who are working hard and spending their time on small or big commits; you guys make the Vue.js development experience so good.

So that's why we are using Vue.js 2 in this enterprise project instead of Vue.js 3. You should remember this because this is precisely what you should do if there's a new major release from your favorite library or framework. Why?

Because *A*, maybe the majority of external libraries are not yet compatible with the latest version. It's a good idea to create a small proof of concept where typical libraries will work on the newest version of the framework. If other libraries are not compatible with the latest version of the framework, you should wait a number of months for the latest version to mature.

And *B*, there's not enough documentation. Research the latest tooling by answering the following questions:

- How many questions and answers originated from the latest version?
- How many blogs have been published that demonstrate a fully functional application built using the latest version of the framework?
- How many YouTube tutorials have been published for the latest version of the framework?

If you didn't find enough articles or Q-and-As, it means that the framework is not yet mature, and that's a good sign to not use it in an enterprise application for your customers, clients, or companies.

Now let's get back to our Vue.js project and install a UI component library.

## Installing Vuetify in a Vue.js application

What is **Vuetify**? Vuetify is a Material Design framework for Vue.js. It has 250,000 weekly downloads on the npm website. If you are not familiar with Material Design, it is a design system created by Google, and they use it in their mobile and web applications. Material Design is so popular that it has also been ported to Angular, React, Svelte, Ember.js, and other JavaScript frameworks.

You can check out all the readily available components on the website, <https://vuetifyjs.com/en/>, or you can go straight to <https://vuetifyjs.com/en/components/alerts/> to see a sample component.

Using or adding the Vuetify library to a Vue application is dead simple. We will use the Vue CLI, particularly the **add** command of the Vue CLI, which will automatically set up the Vuetify library through a configuration. You will learn more about the configuration I just mentioned in a bit. Now, let's get started:

1. Stop the running Vue.js app and run the following Vue CLI command:

```
vue add vuetify
```

The command will start installing the **vuetify** plugin.

2. Choose the default preset option as in *Figure 12.4*:

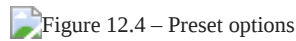


Figure 12.4 – Preset options

The default preset provides all framework services with additional baseline values.

3. After adding Vuetify in the project using the Vue CLI, you will see some changes in the project's files. See *Figure 12.5*:



Figure 12.5 – Changes in the project files

The Vue CLI created a **vue.config.js** file, a config file for transpiling Vuetify, and a **vuetify.js** file, which will add Vuetify global functionality to the app. The Vue CLI will also update the **package.json**, **index.html**, **App.vue**, **main.js**, **logo.svg**, and **HelloWorld.vue** files.

4. Rerun the Vue.js application and see that the welcome message changes and the UI, as in *Figure 12.6*:



Figure 12.6 – Welcome to Vuetify

The new welcome message means that Vuetify is fully installed and working correctly in the Vue.js application.

Before we go on to building our navigation bar, let's check out our other UI component library options in the next section.

## Other third-party UI libraries

Vuetify is not the only UI component library that you can use in Vue.js. There are other useful UI component libraries too. Here are some libraries that you should check out:

- **BootstrapVue**: This is a popular library for Vue.js that lets you design components using the Bootstrap design system. It has 250,000 weekly downloads on the npm website. You can check it out at <https://bootstrap-vue.org/>.

*Figure 12.7* shows BootstrapVue buttons that you can easily use in the Vue.js application:



Figure 12.7 – BootstrapVue's button components

- **Buefy:** This is also a popular library for Vue.js, which uses the design system of Bulma and has 45,000 weekly downloads on the npm website. You can check it out at <https://buefy.org/>.

Figure 12.8 shows Buefy buttons that you add to your Vue.js application:

 Figure 12.8 – Buefy's button components

Figure 12.8 – Buefy's button components

- **PrimeVue:** This is a UI component library based on PrimeFaces designs, which is also available in React, Angular, and Java web applications. You can check it out at <http://primefaces.org/primevue/>.

Figure 12.9 shows PrimeVue buttons that you can also easily use in the Vue.js application:

 Figure 12.9 – PrimeVue's button components

Figure 12.9 – PrimeVue's button components

- **Quasar:** This is technically not a library but a framework on top of Vue.js. It is also based on Material Design but has its own CLI and can be compiled to create desktop apps using Electron and iOS and Android apps using Cordova or Stencil. I would recommend this if you are planning to reuse your code base for cross-platform development.

The Quasar framework also offers SSR or server-side rendering, which helps search engines index the pages and content of your Vue.js app. Here's the link to the Quasar framework:

<https://quasar.dev/>.

Figure 12.10 shows Quasar button designs that you can see on the Quasar website:

 Figure 12.10 – Quasar's button components

Figure 12.10 – Quasar's button components

Another good thing that I'd like to mention here is that incoming, new Vue.js developers might already be familiar with the UI component libraries that I've listed. This will make the onboarding process faster and will make the freshly hired Vue.js developers productive in a few days.

Now let's go back to our Vue.js project and build the top-bar navigation and sidebar navigation for it.

## Adding navigation bars

In this section, we will build a top navigation bar to navigate to the different main pages of our app. We will also make a sidebar inside an admin dashboard page to allow us to navigate between other pages within the admin dashboard. Let's look at the steps:

1. Start by creating a **components** folder in the **src** directory.
2. After creating a **components** folder, create a Vue component file in the **components** folder and name it **NavigationBar**. The path should look like this:

**src | components | NavigationBar.vue.**

3. Now, in the **NavigationBar.vue** file, add the following code:

```
<template>

  <v-app-bar app color="primary" dark>

    <div class="d-flex align-center">

      // for brevity, please see the code in the
      // github repo

    </div>

    <v-spacer></v-spacer>

    <div>

      // for brevity, please see the code in the
      // github repo

    </div>

  </v-app-bar>
</template>

<script>
export default { name: "NavigationBar"};
</script>

<style scoped>
.menu {
  color: white;
  text-decoration: none;
}
</style>
```

The preceding code renders a navigation bar that we will place in our web app's top section. The design was from Vuetify's app bars; I copied the code from Vuetify's documentation and pasted it into VS Code but modified it for our needs. Here's the link to the app bar design:

<https://vuetifyjs.com/en/components/app-bars/>. I suggest taking a look at Vuetify's page to familiarize yourself with writing Vuetify components.

You will notice in the code the **v-** prefix – **<v-app-bar>**, for example. This means that the custom HTML is from Vuetify's components.

You will also find the Vue.js **router-link** component in the code. It is used for navigation and will be rendered as a hyperlink tag by default. We set the target route in **to**, **attribute**, or **props** of the **router-link** component.

Next, look at the `<v-btn>` component and check out `:to`, where we are passing an object with the `path` property to set the target route.

The last thing that you might notice here is that we are not importing `defineComponent` from `Vue` because it only works in Vue.js 3. We can simply use `export default {}` to share `NavigationBar.vue` with other components.

4. Now that we have our top navigation bar, let's update the `App.vue` component with the following code:

```
<template>

  <v-app>

    <NavigationBar />

    <v-main>

      <router-view />

    </v-main>

  </v-app>
</template>

<script>
import NavigationBar from

  "@components/NavigationBar";

export default {
  name: "App",
  components: {
    NavigationBar,
  },
};
</script>
```

We are replacing the code of `App.vue` with this code, which imports and uses the `NavigationBar` component.

5. Now let's update the `About.vue` component with the following code:

```
<template>

  <div class="about fill-height d-flex justify-center

    align-center">

    <h1>About us <img alt="about icon" data-bbox="300 820 340 840"/></h1>

  </div>
</template>
```

This is just a message with emojis.

6. Let's also update the **Home.vue** component with the following code:

```
<template>

  <div class="home fill-height d-flex justify-center
    align-center">

    <h1>Welcome to Travel Tours 🌍 🚀</h1>

  </div>

</template>

<script>

export default {
  name: "Home",

};

</script>
```

We are updating the **Home.vue** component with a simple welcome message.

7. Run the Vue.js application and see the changes in the UI as shown in *Figure 12.11*:

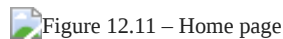


Figure 12.11 – Home page

The screenshot shows the current UI we've developed. It has a top navigation bar and a welcome message on the home page. Most of our functionalities will be on the dashboard and login forms, so let's add more pages in the next section.

## Writing page components

Now let's start adding pages to our Vue.js application particularly for admins:

1. In the **views** directory, create a new folder and name it **AdminDashboard**. Create **DefaultContent.vue**, the page for **AdminDashboard**. The **DefaultContent** page will be the default content of the application when a user goes to the **/admin-dashboard** page. Here is the code for **DefaultContent.vue**:

```
<template>

  <div>

    <div>

      <div class="text-h2 my-4">DefaultContent</div>

    </div>

  </div>

</template>
```



```
<script>
export default {
  name: "DefaultContent",
};
</script>
```

The code is simple enough to show a proof of concept that we can navigate to this page using the **/admin-dashboard** path with text on the browser's screen. We will update this in the upcoming chapters.

2. Create another Vue component and name it **TourLists.vue**. Write the following code in the **TourLists** page:

```
<template>
  <div>
    <div>
      <div class="text-h2 my-4">TourLists</div>
    </div>
  </div>
</template>
<script>
export default {
  name: "TourLists",
};
</script>
```

The **TourLists** page is for the **/admin-dashboard/tour-lists** path.

3. Create another Vue component and name it **TourPackages.vue**. The following code is for the **TourPackages** page:

```
<template>
  <div>
    <div>
      <div class="text-h2 my-4">TourPackages</div>
    </div>
  </div>
</template>
<script>
export default {
  name: "TourPackages",
```

```
};  
</script>
```

The **TourPackages** page is for the `/admin-dashboard/tour-packages` path.

4. Next, create another Vue component and name it **WeatherForecast.vue**. We are going to use the WeatherForecast API in our ASP.NET Core application here, but that will be in the next chapter. So for now, here is the code for the **WeatherForecast** page:

```
<template>  
  <div>  
    <div>  
      <div class="text-h2 my-4">WeatherForecast</div>  
    </div>  
  </div>  
</template>  
  
<script>  
export default {  
  name: "WeatherForecast",  
};  
</script>
```

The **WeatherForecast** page will have the `/admin-dashboard/weather-forecast` path.

5. Then, create an **index.vue** component for the **AdminDashboard** folder. This Vue component will contain the sidebar navigation for the dashboard. Here is the code for the **index.vue** file:

```
<template>  
  <v-sheet  
    height="100vh" class="overflow-hidden"  
    style="display: flex; flex-direction: row;  
    justify-content: flex-start"  
  >  
    <v-navigation-drawer permanent expand-on-hover>  
      // For brevity, please see the code in the  
      // github repo of this chapter. Thank you.  
    </v-navigation-drawer>  
    <v-container>  
      <router-view />  
    </v-container>  
  </v-sheet>
```

```
</template>
```

The **v-navigation-drawer** component from Vuetify that you can see in the code is a ready-made component that you can copy and paste from <https://vuetifyjs.com/en/components/navigation-drawers/>. Although I copy-pasted the code, I still had to tweak it based on the pages we've created in our application. You can see that we have a small section for the profile UI in the code. There are also links, a link to **/admin-dashboard**, a link to **/admin-dashboard/tour-lists**, a link to **/admin-dashboard/tour-packages**, a link to **/admin-dashboard/weather-forecast**, and a link for the logout functionality.

6. Next is the component name and styling of the **AdminDashboard/index.vue** component:

```
<script>
export default {
  name: "AdminDashboard",
};
</script>
<style scoped>
.link {
  text-decoration: none;
}
</style>
```

We only have a **link** class here to remove **text-decoration**, but we will update this in the upcoming chapters.

Before we finish this section, let's create a new folder inside the **views** directory and name it **Main**. Move the **About.vue** component and the **Home.vue** component to the **Main** folder. Moving the **About** page and the **Home** page to the **Main** folder gives us a better folder structure. The **Main** folder will contain all the pages outside of the dashboard, while the **AdminDashboard** folder will have all the dashboard pages.

Now we are done creating the dashboard pages, let's update the routes file in the next section.

## Setting up Vue Router with lazy loading and eager loading

To navigate to the pages we created, we have to register in routes some paths pointing to their respective components. In that case, let's go to the **router/index.js** file to update it:

1. We will update the paths of the **Home** page and **About** page because we'll move them into a new folder, which is **views/Main**:

```
import Vue from "vue";
```

```

import VueRouter from "vue-router";
import Home from "@views/Main/Home";
import TourLists from "@views/AdminDashboard/TourLists";
import TourPackages from "@views/AdminDashboard/TourPackages";
Vue.use(VueRouter);
const routes = [
  { path: "/", name: "Home", component: Home },
  /* lazy loading through dynamic import() */
  { path: "/about", name: "About",
    component: () => import("@views/Main/About") },
  { path: "/admin-dashboard",
    component: () => import("@views/AdminDashboard"),
    children: [ { path: "",
      component: () => import("@views/
        AdminDashboard/DefaultContent") },
      { path: "weather-forecast",
        component: () => import("@views/
          AdminDashboard/WeatherForecast") },
      /* eager loading through static import statement
        */
      { path: "tour-lists", component: TourLists },
      { path: "tour-packages", component: TourPackages }
    ] },
  { path: "**", redirect: "/" },
];
const router = new VueRouter({
  mode: "history", base: process.env.BASE_URL, routes,
});
export default router;

```

2. Then we add the routes for the pages we created. The **vue-router** library is responsible for creating routes in our Vue.js project. We are defining all the routes inside the array and passing the array in **VueRouter** while creating an instance of it.

Take a look at the values of the components inside the objects in the array. You will notice that I am showing different ways of importing the components. One is a **dynamic import** where we use **import()**, and the other is **static import**. Now, why do we need to do this? A dynamic import is

used here to import components only when the user is navigating to a specific page. The on-demand importing of components or pages based on URLs or the UIs that users see is called lazy loading.

**Lazy loading** is the way to split code. You will also hear the term code-splitting instead of lazy loading. We do this to not congest the network by downloading files when our web app is loading in the browser. Consequently, a network bottleneck can usually happen if a web application has many files and large files downloading simultaneously in the browser. Lazy loading improves the performance of our web apps.

On the other hand, **eager loading** is what a static import does. The resource files of any pages are downloaded right away when a user visits a web application. If it takes more than 5 seconds for your application to load because of a network bottleneck, that's a bad user experience for web visitors.

You can check out all the APIs of Vue Router at <https://router.vuejs.org/>.

Now let's once again run our application and navigate to the different pages inside the dashboard:

1. Go to the dashboard by clicking the **DASHBOARD** menu. Open up your browser's DevTools, go to the **Network** tab, click **All** for the types, and clear out the logs as shown in *Figure 12.12*:



Figure 12.12 – Dashboard default content and DevTools

Make sure to remove the logs to quickly see if a file or files appeared on the screen. We are preparing our web app for a quick demo of how lazy loading works.

2. Hover your mouse on the sidebar so you can see the navigation menus of the sidebar, like in *Figure 12.13*:



Figure 12.13 – Navigation menus of the sidebar

3. Now go to the **Tour Lists** page and **Tour Packages** while looking at the DevTools. There are no downloaded files that you can see in the **Network** tab of the DevTools because those pages are imported using static imports.
4. Go to the **Weather Forecast** page and watch out for the script and stylesheet files that will land on DevTools as shown in *Figure 12.14*:



Figure 12.14 – Lazy loaded script file and stylesheet file of the Weather Forecast page

The two files were added as soon as we navigated to the **Weather Forecast** page, using a dynamic import to lazy load the file. Imagine you have more than 50 pages in your Vue.js application. Your application will gain improvements in the loading speed if you lazy load all 50 pages of your application.

And a piece of advice: also apply code-splitting in React, Angular, Svelte, Ember.js, Preact, Solid.js, or any JavaScript framework that you'll face in the future.

Now, before we go to the list of what we have learned and wrap up this chapter, I'd like you to remember the process we followed in starting a project. The following are the steps in building an application that I came up after years of developing web and mobile apps:

1. Installing a UI component library or using a ready-made boilerplate from popular GitHub repositories.
2. Studying the required pages in the app, knowing how many pages there are, and whether a dashboard admin is needed. You would generally get this information from the UI design team in your company.
3. Building the navigations, writing lazy loaded routes, and creating pages with simple welcome messages. Creating these elements right away gives you the overall skeleton of the application.

So that's usually the pattern I use in the early stages of any web or mobile development I do.

Now let's summarize what you have learned in this chapter.

## Summary

Great work! We've done so many things in this chapter. You've learned that UI component libraries can help us develop applications faster by not building components from scratch. You have learned that Vuetify, BootstrapVue, Buefy, PrimeVue, and Quasar are some of the UI component libraries with excellent visual designs and optimized for user experience.

You've also learned how to create pages in Vue.js by adding the components in the routes, and how to get from one point of your app to another using the **router-link** component.

And lastly, you have learned how to lazy load pages to improve the loading speed of the web application.

In the next chapter, we will integrate a Vue.js application with ASP.NET Core.

## Chapter 13: Integrating a Vue.js Application with ASP.NET Core

Now, this is the time to make the Vue.js application talk to the REST API. In this chapter, you will integrate the Vue.js frontend application with the ASP.NET Core 5 backend application by installing a NuGet package and adding some code. You will also add a CORS policy in the backend to allow web apps from other domains to send requests to your backend.

We will cover the following topics:

- Putting the ASP.NET Core Web API and a Vue.js app together as a single unit
- Introducing **Cross-Origin Resource Sharing** or **CORS**
- Enabling a CORS policy in ASP.NET Core

### Technical requirements

Here is what you need to complete this chapter:

- **Visual Studio 2019 IDE:** <https://visualstudio.microsoft.com/vs/>
- **Visual Studio for Mac IDE:** <https://visualstudio.microsoft.com/vs/mac/>
- **Rider IDE:** <https://www.jetbrains.com/rider/>
- **Visual Studio Code:** <https://code.visualstudio.com/>
- **Postman:** <https://www.postman.com/>

The finished code repo can be found here: <https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter13/>.

### Putting ASP.NET Core Web API and a Vue.js app together as a single unit

The integration that we will do in this section is not required for Vue.js to send a request to the ASP.NET Core but will give us some advantages. Our goal here is to host the backend and the frontend projects in a single app project. In return, the application can be published or built as a single unit.

There are official web application project templates. You may call them boilerplates in .NET 5 that scaffold an Angular application with an ASP.NET Core Web API and a React application with an ASP.NET Core Web API. You can check out the two project templates at the following links:

- **Use the Angular project template with ASP.NET Core:** <https://docs.microsoft.com/en-us/aspnet/core/client-side/spa/angular?view=aspnetcore-5.0&tabs=visual-studio>

- Use the React project template with ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/client-side/spa/react?view=aspnetcore-5.0&tabs=visual-studio>

The boilerplates above are a convenient starting point for ASP.NET Core apps using React or Angular. The project can be quickly shipped as a single app to Azure App Service from your local machine from a Git repository such as GitHub or Azure DevOps.

Unfortunately, there is no official Vue.js with ASP.NET Core Web API boilerplate that exists. However, we can convert our ASP.NET Core API and Vue.js app into one project by adding some extra code in **Startup.cs**, updating **Travel.WebApi.csproj**, and installing a NuGet project. Are you excited? Let's start:

1. The first step is to install **VueCliMiddleware** in the **Travel.WebApi** project. The NuGet package is only needed in the **src | presentation | Travel.WebApi** project, so there's no need to install it in other projects.

**VueCliMiddleware** will allow us to build a Vue.js SPA on ASP.NET MVC Core using Quasar CLI or Vue CLI. Check out the nuGet package at this link and its GitHub repository: <https://www.nuget.org/packages/VueCliMiddleware> and <https://github.com/EEParker/aspnetcore-vueclimiddleware>.

2. The next step is to update the **Travel.WebApi.csproj** file. We are going to add some code below the last **ItemGroup** that in the file, which is this code:

```
<ItemGroup>

  <Folder Include="Extensions\" />

</ItemGroup>
```

3. We are going to add the following code to the **</ItemGroup>** closing tag. So make some space, hitting the *Enter* key several times, and then write the following code:

```
<!-- Below is for Single Page Application. -->

<!-- Please wait for compilation to finish before
going to https://localhost:5001 -->

<PropertyGroup>

  <!-- Typescript/Javascript Client Configuration
  -->

  <SpaRoot>..\vue-app\</SpaRoot>

  <DefaultItemExcludes>$(DefaultItemExcludes);
  $(SpaRoot)node_modules\**</DefaultItemExcludes>

  <!-- Set this to true if you enable server-side
  prerendering -->

  <BuildServerSideRenderer>false

  </BuildServerSideRenderer>
```



```
<IsPackable>false</IsPackable>

</PropertyGroup>
```

The newly added code tells the application that the directory of the SPA is in **vue-app**.

Also add the following code:

```
<ItemGroup>

  <!-- Don't publish the SPA source files, but do
        show them in the project files list -->
  <Content Remove="$(SpaRoot)**" />
  <None Remove="$(SpaRoot)**" />
  <None Include="$(SpaRoot)**"
        Exclude="$(SpaRoot)node_modules\**" />
</ItemGroup>

<Target Name="DebugEnsureNodeEnv"
  BeforeTargets="Build" Condition="
    '$(Configuration)' == 'Debug' And !Exists(
      '$(SpaRoot)node_modules') ">
  <!-- Ensure Node.js is installed -->
  <Exec Command="node --version"
    ContinueOnError="true">
    <Output TaskParameter="ExitCode"
      PropertyName="ErrorCode" />
  </Exec>
  <Error Condition="'$(ErrorCode)' != '0'"
    Text="Node.js is required to build and
    run this project. To continue, please install
    Node.js from https://nodejs.org/, and then
    restart your command prompt or IDE." />
  <Message Importance="high" Text="Restoring
    dependencies using 'npm'. This may take several
    minutes..." />
  <Exec WorkingDirectory="$(SpaRoot)" Command=
    "npm install" />
</Target>
```

The preceding code adds the **Spa Root** folder, which is **vue-app**, to the project.

4. Then add the final block of code we need in **Travel.WebApi.csproj**, like so:

```
<Target Name="PublishRunWebpack"

  AfterTargets="ComputeFilesToPublish">

    <!-- As part of publishing, ensure the JS
           resources are freshly built in production mode
    -->

    <Exec WorkingDirectory="$(SpaRoot)" Command=
      "npm install" />

    <Exec WorkingDirectory="$(SpaRoot)" Command=
      "npm run build" />

    <!-- Include the newly-built files in the publish
           output -->

    <ItemGroup>

      <DistFiles Include="$(SpaRoot)dist\**;"
        $(SpaRoot)dist-server\**" />

      <DistFiles Include="$(SpaRoot)
        node_modules\**"
        Condition="'$(BuildServerSideRenderer)' ==
          'true'" />

      <ResolvedFileToPublish Include="@{(DistFiles
        ->'%(FullPath)')}" Exclude="
        @(ResolvedFileToPublish)">
        <RelativePath>%(DistFiles.Identity)
        </RelativePath>
        <CopyToPublishDirectory>PreserveNewest
        </CopyToPublishDirectory>
        <ExcludeFromSingleFile>true
        </ExcludeFromSingleFile>
      </ResolvedFileToPublish>
    </ItemGroup>

  </Target>
```

The added code is for running **npm run build** in the Vue.js **root** directory, which includes the newly-built files in the publish output.

That's all for **csproj** file. Now let's go to the next step.

5. Now go to **Startup.cs** and update the **ConfigureServices** method. Add the following code at the end of the services:

```
services.AddSpaStaticFiles(configuration =>
{
    configuration.RootPath = "../vue-app/dist";
});
```

The preceding code should be below **services.AddTransient IConfigurationOptions**. The code adds a SPA static files service in the application.

6. Next, we update the middleware block, which is the **Configure** method. Add the following code under the **if (env.IsDeveloper())** block:

```
app.UseStaticFiles();
if (!env.IsDevelopment())
{
    app.UseSpaStaticFiles();
}
```

We are adding **UseSpaStaticFiles** in the middleware pipeline.

7. Now let's add **UseSpa**, another middleware, like so:

```
app.UseSpa(spa =>
{
    spa.Options.SourcePath = "../vue-app";
    if (env.IsDevelopment())
    {
        spa.UseVueCli(npmScript: "serve");
    }
});
```

The code adds the source path to **UseSpa** and also runs **Vue CLI serve** in production.

You might need to import **VueCliMiddleware** and **SpaServices** manually if your IDEs are not helping you:

```
using Microsoft.AspNetCore.SpaServices;
using VueCliMiddleware;
```

The preceding code is the namespaces that you will need to create a Vue CLI proxy mapping. **Travel.WebApi** and **vue-app** are inside the **presentation** directory, as shown in the following screenshot:


 Figure 13.1 – The presentation directory

Figure 13.1 – The presentation directory

The two folders that you see in *Figure 13.1* are also what you will see in Visual Studio Code:

 Figure 13.2 – VS Code folder arrangement

Figure 13.2 – VS Code folder arrangement

The VS Code folder arrangement in *Figure 13.2* also shows that **Travel.WebApi** and **vue-app** are on the same level in the directories. However, it is different in IDEs such as JetBrains' Rider, Visual Studio for Mac, and Visual Studio 2019, as shown in the following screenshot:

 Figure 13.3 – Vue.js files and folders in the Travel.WebApi directory

Figure 13.3 – Vue.js files and folders in the Travel.WebApi directory

IDEs are smart enough to know the SPA folders and files and expose them in the Solution Explorer's interface.

8. Now let's run the application by hitting your **IDE debug** button or running the **dotnet run** command inside the **Travel.WebApi** project. Then go to **https://localhost:5001** to see the Vue.js app as shown in the following screenshot:

 Figure 13.4 – Vue.js running on port 5001

Figure 13.4 – Vue.js running on port 5001

Accept any popup from the browser that tells you the SSL certificate of the localhost.

**localhost:5001** is proxying port **8080** of the Vue.js CLI. Start editing the **Welcome to Travel Tours** message in your code base and see that Hot Module Replacement is still darn fast. You can see the changes immediately after saving any changes.

9. Now check out the Swagger UI to see that it is also running on the same port:

 Figure 13.5 – Swagger UI on port 5001

Figure 13.5 – Swagger UI on port 5001

*Figure 13.5* shows that Swagger UI is still working on port **5001**.

Now we can ship the application as a single unit. What if you have another SPA that lives in another domain and wants to send requests to your ASP.NET Core?

It is not possible right now, and that is what we will fix in the next section of this chapter.

## Introducing Cross-Origin Resource Sharing or CORS

Before we discuss CORS policy and cross-origin resource sharing, try to send a POST request to version 2 of the **WeatherForecast** endpoint. See if you can still retrieve some of the JSON response of the **WeatherForecast** controller using Postman as shown in the following screenshot:



Figure 13.6 – Sending a POST request to WeatherForecast using Postman

Figure 13.6 – Sending a POST request to WeatherForecast using Postman

*Figure 13.6* means that the endpoint is still working correctly, but it would not work in another SPA.

I created a React application that runs on port **3000** to see if it can fetch the JSON objects from the **WeatherForecast** controller. No auth is required in the request, but the React application logs errors in the console; see *Figure 13.7*:



Figure 13.7 – Blocked by CORS policy

Figure 13.7 – Blocked by CORS policy

The error in *Figure 13.7* says that access to **XMLHttpRequest** at the endpoint from **localhost:3000** has been blocked by CORS policy. The **No Access-Control-Allow-Origin** header is present on the requested resource. If you want to see it for yourself, you can run the React application you can get from the GitHub repo's [Chapter 13](#). Use the **react-app** folder by running **npm install** then **npm run start**, which will open your browser on **localhost 3000**.

So what is happening here? How can we solve this?

**CORS** stands for **Cross-Origin Resource Sharing**. It's a pretty good security concept. If both a client app and the APIs come from the same server, like a traditional web, sending HTTP requests will just succeed. The server processes a request because the request is trying to access resources on the same server.

However, sometimes even the localhosts are considered as a different origin. If origins are not the same, in React using port **3000** as an example, React from port **3000** sends requests to the server to get a resource, but the browser will fail the requests of React.

Since we are building RESTful APIs, we want to allow this access because Web APIs are meant to be consumed by different browser applications. I explicitly said browser applications because CORS restrictions are not applied in mobile apps, so mobile apps don't need to send a pre-flight request. A preflight request is a quick, small request that the browser sends to the resource server. The pre-flight request checks whether the CORS protocol is understood and the server has policies for processing methods and headers.

Now we know what CORS is, let's update our backend to enable CORS policy configuration in the next section.

## Enabling a CORS policy in ASP.NET Core

Our goal in this section is to use an interface with CORS policy and configure it to allow any SPAs to send requests to our ASP.NET Core Web APIs.

So let's go to the **Startup.cs** file of the **Travel.WebApi** project and update the **ConfigureServices** method by adding the following code under the **AddSpaStaticFiles** method:

```
services.AddCors();
```

The **services.AddCors** method will be the last method we are calling inside **ConfigureServices**. This service is an extension method for setting up CORS services.

We are not yet done here; we still need to update our middleware. So, go to the **Configure** method and add the following code under **app.UseSpaStaticFiles**:

```
app.UseCors(b =>
{
    b.AllowAnyOrigin();
    b.AllowAnyHeader();
    b.AllowAnyMethod();
});
```

The **UseCors** method adds a CORS middleware to our web application's pipeline. This middleware will allow cross-domain requests in our application from now on.

Now let's give it a go and see if we can directly send a request coming from a different port.

You can run the React application by running **npm run start** in its root folder. You should see the same results as shown in the following screenshot:

 Figure 13.8 – React from port 3000 getting a 200 OK response

Figure 13.8 – React from port 3000 getting a 200 OK response

Figure 13.8 shows that our CORS policy configuration is working, and it is not blocking the request from React coming from **localhost:3000**.

### IMPORTANT

You can learn more about CORS policy at <https://auth0.com/docs/applications/set-up-cors>.

Let's now summarize what you have learned.

## Summary

You have arrived at the end of this chapter; you have learned how to include the Vue.js app in the ASP.NET Core Web API project to run and ship them together as one. You also know how **Cross-**

**Origin Resource Sharing** or **CORS** works. In a nutshell, it's a security feature of the browser that blocks requests from different domains or origins. You have learned how to enable and configure a CORS policy in ASP.NET Core to allow incoming requests from other domains or origins.

In the next chapter, we will start sending GET requests to our ASP.NET Core Web API and use Vuex for managing the state of our Vue.js app. It will be a significant and big chapter, so take a break first and come back after 40 to 60 minutes.

## Chapter 14: Simplifying State Management with Vuex and Sending GET HTTP Requests

This chapter is about sending HTTP requests and solving the most common problem in big web applications—the problem of syncing the state of a component with another component. In large and complex applications, you need a tool that centralizes your application's state and makes the data flow transparent and predictable.

In this chapter, we will cover the following topics:

- Understanding complex state management
- Sending HTTP requests
- Setting up state management using Vuex

### Technical requirements

You will need Visual Studio Code to complete this chapter.

The finished repository of this chapter can be found at <https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter14>.

### Understanding complex state management

You will encounter state management when developing big and complex Angular, React, or Vue web applications. So what does state management mean?

Application **state management** is when your app starts to grow from just being a view or a couple of views. You're probably going to start running into the problem where you want to share some of that application state between different and nested components. An example of that is when you have to create a mechanism where two deep components should always sync. See *Figure 14.1* for a diagram of this:



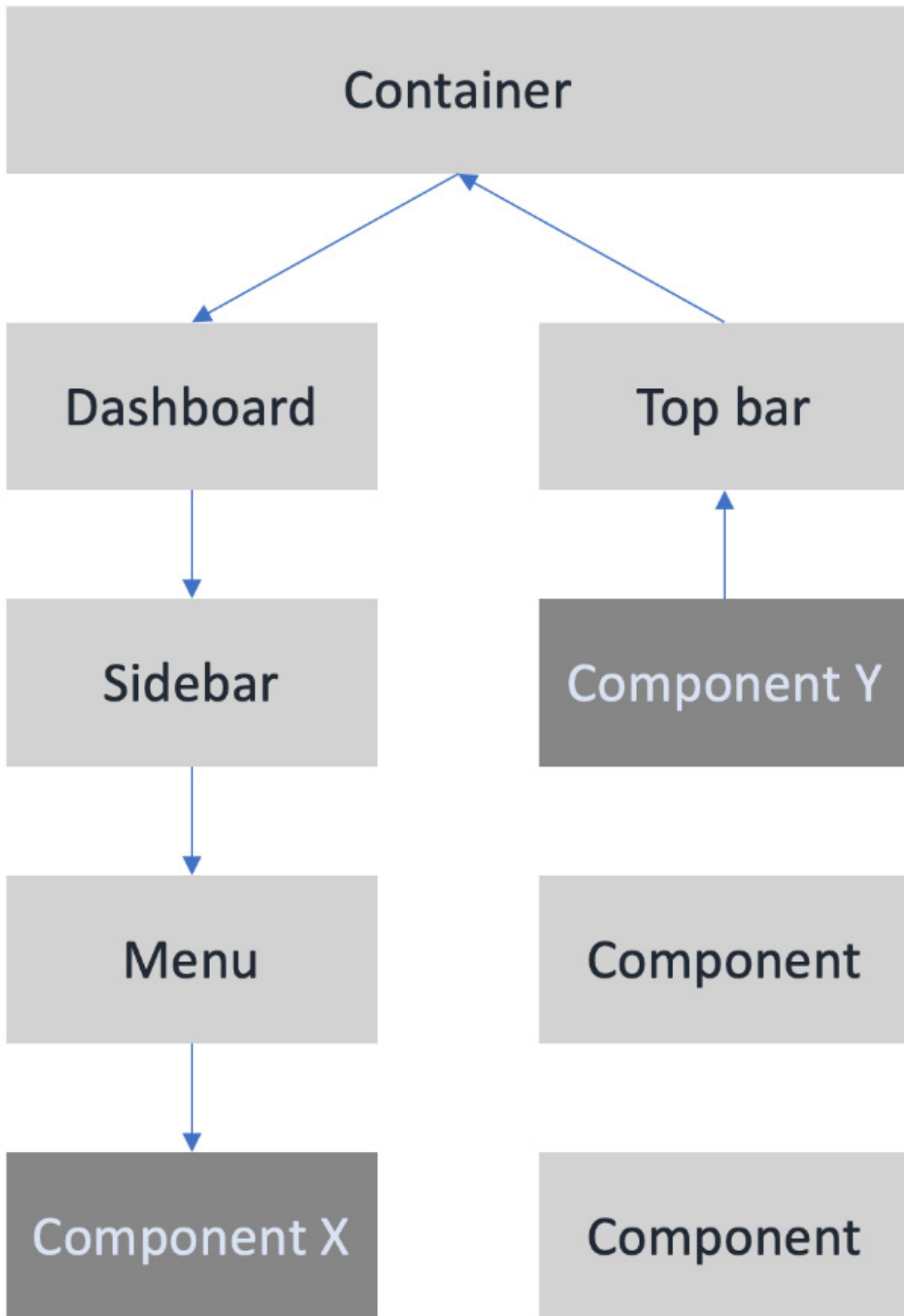


Figure 14.1 – Shared component's local state

Figure 14.1, shows that **Component Y** shares state with **Component X**. Now, there are different ways you could do that. One is bypassing events and passing in properties from a top-level component. Even though you can pass props and events, passing events and props in multiple nested components makes an application hard to maintain and makes the code hard to reason about.

## Understanding global state

So what is the solution for unmaintainable code due to the sharing of states between multiple nested components? An application-wide state or global state, which is also known as the **store**. The idea in a global state solution is that every view and every component can just be a reflection of that one central state. See Figure 14.2:

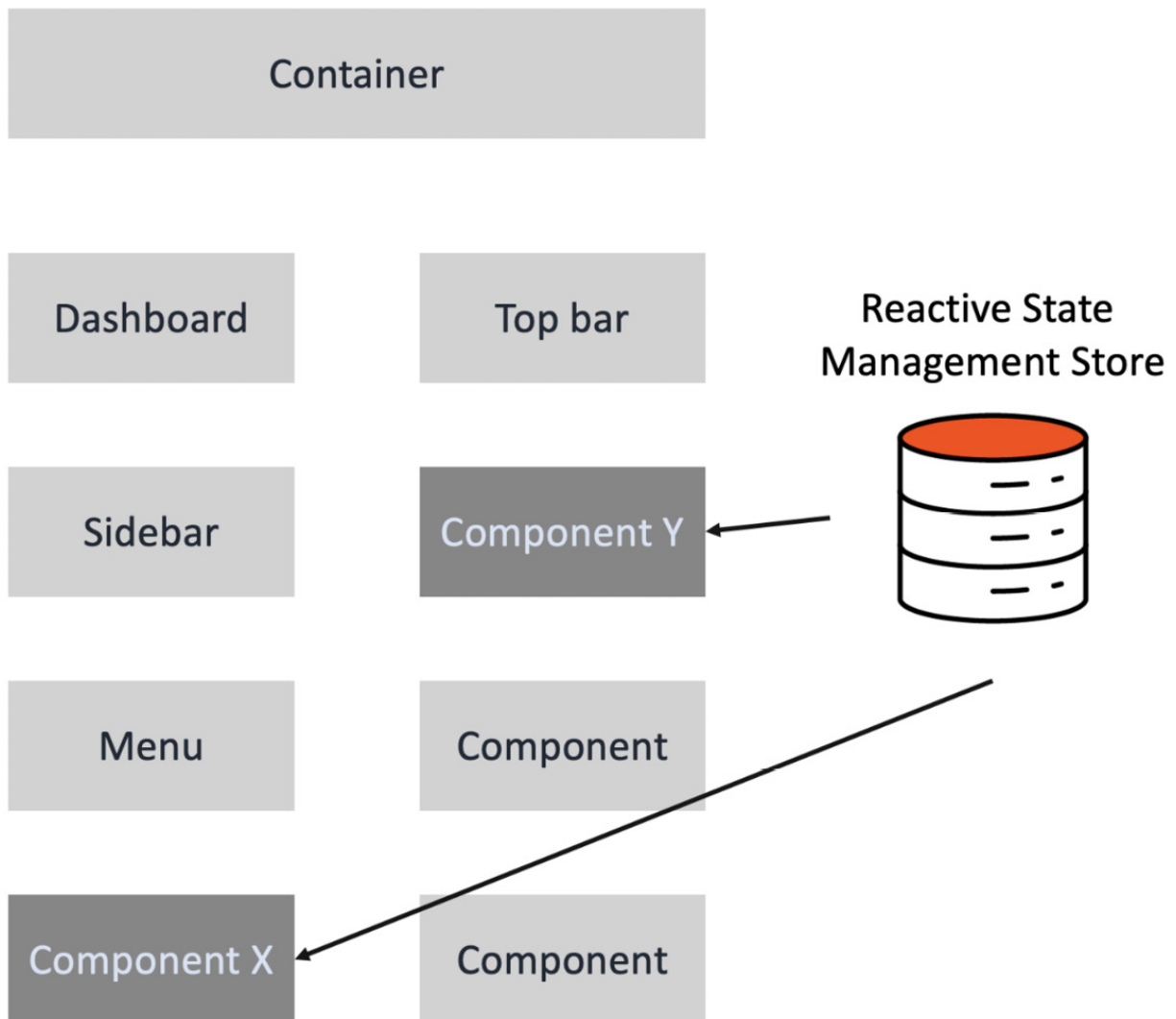


Figure 14.2 – Reactive global state

Figure 14.2 shows a reactive global state that is accessible to any components. Having a store is good. We only have one source of truth, and everything is just a reflection of that. The reactivity of the global state or store is a lovely way of making sure that everything the user sees in our application always stays in sync.

So how do you set up a store in Vue.js? We will do that later, but let's do some simple data fetching without a store in the next section. The simple data fetching will prove that we can send an HTTP GET request to our ASP.NET Core Web API.

## Sending an HTTP request in Vue.js

Sending HTTP requests to a RESTful service if you are developing a modern web application is trivial. There are HTTP client libraries such as **api-sauce**, **super-agent**, and **axios**. Do you know that JavaScript itself has a native API for sending HTTP requests? Yes, and that is the Fetch API.

The Fetch API is available in JavaScript and TypeScript. However, it only works in modern browsers and will not work when you load your application in older browsers. Not only that, but the Fetch API can also be too verbose when using its APIs to send a request, in my opinion. I would prefer an HTTP client library with an abstraction to not write extra code such as **res.json()**, **headers**, or the **property** method. In that case, we will be using Axios as our HTTP client library.

Axios is a promise-based HTTP client library for the browser and server-side Node.js app. It is simple to use and also supports older browsers. We will install Axios together with Day.js, which is a library for manipulating dates and times.

So, go to your **vue-app** root directory and run the following **npm** command:

```
npm i axios dayjs
```

The preceding **npm** command will install **axios** and **dayjs**.

Next, we create a folder named **api** in the **src** directory.

Create a JavaScript file, **api-v1-config.js**, inside the **api** folder and write the following code:

```
import axios from "axios";

const debug = process.env.NODE_ENV !== "production";

const baseUrl = debug
  ? "https://localhost:5001/api/v1.0/"
  : "https://traveltour.io/api/v1.0/";

let api = axios.create({ baseUrl });

export default api;
```

The preceding code is an Axios setup for API version one of our ASP.NET Core Web API. We are creating an instance of Axios and passing a base URL for it to use.

Let's create another JavaScript file, **api-v2-config.js**, in the **api** folder and write the following code:

```
import axios from "axios";

const debug = process.env.NODE_ENV !== "production";

const baseURL = debug
  ? "https://localhost:5001/api/v2.0/"
  : "https://traveltour.io/api/v2.0/";

let api = axios.create({ baseURL });

export default api;
```

The preceding code is an Axios configuration for API version two of our ASP.NET Core Web API. You will notice that the only thing that is different here is the version.

Create another JavaScript file, **weather-forecast-services.js**, inside the **api** folder and write the following code:

```
import api from "@/api/api-v2-config";

export async function getWeatherForecastV2Axios(city) {
  return await api.post(`/WeatherForecast/?city=${city}`);
}
```

The preceding code is a service for sending a POST request to the **WeatherForecast v2** controller. We named the default export of the Axios configuration **api** and then used it to call a **POST** method to the **WeatherForecast** endpoint. I prefer adding an **axios** suffix to the function service to help me recognize the right file to import while reading the IntelliSense of my IDE or code editor.

Now let's update the **WeatherForecast.vue** component of the **views | AdminDashboard** folder. Update the contents of the folder with the following code:

```
<script>

import { getWeatherForecastV2Axios } from "@/api/weather-forecast-services";

export default {
  name: "WeatherForecast",
  async mounted() {
    await getWeatherForecastV2Axios("Oslo");
  },
};
```

</script>

In the preceding code, we import the **getWeatherForecastV2Axios** service and use it in **mounted()** life cycle hooks so that it gets triggered in advance before the DOM gets rendered.

Run the backend application and the frontend application by running the following command inside the **Travel.WebApi** project:

```
dotnet run
```

Wait for a few seconds after running the preceding **dotnet** command, then go to your browser, type **https://localhost:5001**, and check the Chrome DevTool, as shown in the following screenshot:



Figure 14.3 – Chrome DevTool status after sending the POST request to the WeatherForecast API

Figure 14.3 shows that the **POST** request returns a **200 OK** status code, which means we get what we ask for, and that would be the following JSON results:

×	Headers	Preview	Response	Initiator	Timing
1	[{"date":"2021-01-12T18:13:11.314868+01:00","temperatureC":31,				

Figure 14.4 – JSON response after sending the POST request to the WeatherForecast API

The JSON response from the **WeatherForecast** controller can be seen in *Figure 14.4*. The response signifies that we can send a request and get a response from our backend.

Now we know that there's data in our browser that we can use in a user interface, let's build a UI for our data.

Let's update **WeatherForecast.vue** again.

Let's bring in **relativeTime** and **dayjs** from the **dayjs** library we installed:

```
import relativeTime from "dayjs/plugin/relativeTime";  
import dayjs from "dayjs";
```

**relativeTime** formats the date to relative time strings such as **an hour ago** or **in 2 days**.

Now let's update the mounted life cycle hook:

```

async mounted() {
  this.loading = true;
  dayjs.extend(relativeTime);
  await this.fetchWeatherForecast(this.selectedCity);
  this.loading = false;
},

```

We are using the library **dayjs** here to manipulate the date. You will also notice that we are using the local state **loading** by setting it to true then setting it to false after fetching the data. We are going to use the local state **loading** to show a spinner component on the screen of our app.

Now let's define our local states:

```

data() {
  return {
    weatherForecast: [],
    cities: [],
    selectedCity: "Oslo",
    loading: false,
  };
},

```

We have a **weatherForecast** array, a **cities** array, a **selectedCity** string, and a **loading** Boolean in our local states. They are all initialized with default values.

Let's add an asynchronous method inside a **methods** object like so:

```

methods: {
  async fetchWeatherForecast(city = "Oslo") {
    this.loading = true;
    try {
      const { data } = await getWeatherForecastV2Axios(
        city);
      console.log(data);
      this.weatherForecast = data?.map((w) => {
        const formattedData = { ...w };
        let date = w.date;
        formattedData.date = dayjs(date).fromNow();
        return formattedData;
      });
    } catch (error) {
      console.error(error);
    }
    this.loading = false;
  }
}

```

```

    });
  } catch (e) {
    alert("Something happened. Please try again.");
  } finally {
    this.loading = false;
  }
},
},

```

We are setting **loading** to **true** for our spinner in this function, sending the request using the **getWeatherForecastV2Axios** service. After getting the data, we are going to use the JavaScript array utility **map**, which is like a loop to format each object's date from the JSON array response.

Let's also create another method for returning a color-coded temperature. Get the complete function in the GitHub repo because it is too big to write it here:

```

getColor(summary) {
  switch (summary) {
    case "Freezing":
      return "indigo";
    // get the rest from the github
    default:
      return "grey";
  }
},

```

The method is essentially just a helper that returns a color depending on the temperature.

Now for the **template** syntax. Here it is:

```

<template>
  <v-container>
    <div class="text-h4 mb-10">
      Two-week weather forecast of different cities
    </div>
    <div class="v-picker--full-width d-flex justify-center"
      v-if="loading">
      <v-progress-circular
        :size="70"

```

```

        :width="7"
        color="purple"
        indeterminate
    ></v-progress-circular>
</div>

<!-- Insert <v-simple-table> here -->
</v-container>
</template>

```

The preceding code is the **title**, **container**, and the spinner UI of our application for the **WeatherForecast** page.

Now include the **v-simple-table** component in the container of the **WeatherForecast** UI. Insert the component in the place where you will find the **<!-- Insert <v-simple-table> here -->** comment:

```

<v-simple-table>
  <template v-slot:default>
    <thead>
      <tr>
        <th class="text-left">Dates</th>
        <th class="text-left">City</th>
        <th class="text-left">&#8451;</th>
        <th class="text-left">&#8457;</th>
        <th class="text-left">Summary</th>
      </tr>
    </thead>
    <tbody>
      <tr v-for="item in weatherForecast"
        :key="item.date">
        <td>{{ item.date }}</td>
        <td>{{ item.city }}</td>
        <td>{{ item.temperatureC }}</td>
        <td>{{ item.temperatureF }}</td>
        <td>
          <v-chip :color="getColor(item.summary)" dark>{{
            item.summary

```



```

    }}</v-chip>
  </td>
</tr>
</tbody>
</template>
</v-simple-table>

```

The **v-simple-table** component will be the table of **WeatherForecast**, which shows the temperature for a given date in Oslo:

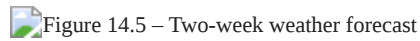


Figure 14.5 – Two-week weather forecast

The preceding forecast is a two-week forecast for **Oslo**. There you can see the dates, cities, temperatures in Celsius and Fahrenheit, and a visual summary of the weather.

In the next section, we are going to add a drop-down selector where we can select a city, and the cities we can choose will come from our ASP.NET Core Web API.

Now we have a **WeatherForecast** feature using a simple HTTP request without using a store. The feature that we are going to build in the next section will be using a Vuex state management library. We are going to reuse the functionality in another component that requires a store in the next chapter, [Chapter 15, Sending POST, DELETE, and PUT HTTP Requests in Vue.js with Vuex](#).

## Setting up state management using Vuex

**Vuex** is the official state management library for Vue.js that is widely used for managing complex components. Vuex has a reactive global store and is reasonably easy to set up. I will explain the parts of the Vuex implementation as we write the code.

But before we start building our store, let's remove the authorization from the API controller so that we don't need an auth token when sending requests to the **api/v1.0/** endpoints.

To do that, go to the **ApiController.cs** file in the namespace **Travel.WebApi.Controllers.v1** of the **Travel.WebApi** project and comment the **Authorize** attribute like so:

```
// [Authorize]
```

After commenting the **Authorize** attribute, we can now use **api/v1.0/** temporarily.

Let's start by setting up the update part of the Vuex.

## Step 1 – Writing a store

Create a folder named **tour** inside the **store** folder. It will be like this: **src | store | tour**.

## Step 2 – Writing a module

Create a JavaScript file named **services.js** inside the **tour** folder and write the following code:

```
import api from "@api/api-v1-config";
export async function getTourListsAxios() {
  return await api.get("TourLists");
}
```

We are creating a service that sends a request to the version one **TourLists** controller or endpoint.

## Step 3 – Writing a module if we are using TypeScript

Create a JavaScript file named **types.js**, still inside the **tour** folder, and then write the following code:

```
export const LOADING_TOUR = "LOADING_TOUR";
export const GET_TOUR_LISTS = "GET_TOUR_LISTS";
```

The preceding code is not a requirement in writing Vuex but became a part of the design when implementing state management. The preceding types are called **action** types, and we are going to the types to let the store know what kind of action it should take in the state after receiving an action. The **snake-casing** types are just strings, but they can prevent typo errors when writing actions, which you will see later.

## Step 4 – Writing an API service

Create a JavaScript file named **actions.js** inside the **tour** folder and then write the following code:

```
import * as types from "./types";
import { getTourListsAxios } from "@store/tour/services";
// asynchronous action using Axios
export async function getTourListsAction({ commit }) {
  commit(types.LOADING_TOUR, true);
  try {
    const { data } = await getTourListsAxios();
    commit(types.GET_TOUR_LISTS, data.lists);
  } catch (e) {
    alert(e);
  }
}
```

```
    console.log(e);  
  }  
  commit(types.LOADING_TOUR, false);  
}
```

The action carries instructions on how to update the global state of the store. Action is terminology that you can also find in the state management libraries of other JavaScript frameworks. Hence, it is valuable to remember this.

Moreover, actions can contain asynchronous operations if needed. We are defining an **action** function with a deconstructed **commit** parameter to commit to a mutation.

To use the commit, simply pass the action types as the first argument. If we need a payload in that particular commit, we can use the second optional argument to pass the payload. An example of that is **commit(types.LOADING\_TOUR, true)**. In the previous sentence, the line of code is a **commit** statement about loading **tour** with a Boolean argument as **payload** of the action.

We are not limited to writing a single commit in an **action** function. We can write one or more commits as long as we need to.

What we are doing in **getTourListAction** of **actions.js** is enabling **loading tour**. The goal is to create a spinner while we are fetching the data using the **getTourListsAxios** service. We will then use the response in **commit(types.GET\_TOUR\_LISTS, data.lists)** to keep the data in our global state essentially.

We then passed **false** to **loading tour** to stop the spinner from spinning.

You will see more about this later when we write more actions in the upcoming chapters.

## Step 5 – Writing an action type

Create a JavaScript file named **state.js** inside the **tour** folder and then write the following code:

```
const state = {  
  lists: [],  
  loading: false,  
};  
  
export default state;
```

The **state** object will be part of our app's global state, which is the store. A store is a big object with properties that have default or initial values. The initial values or initial states are something that you have to remember because they are also applicable in any other state management libraries in different JavaScript frameworks.

## Step 6 – Writing an action

Create another JavaScript file named **mutations.js** inside the **tour** folder and then write the following code:

```
import * as types from "../types";

const mutations = {

  [types.GET_TOUR_LISTS](state, lists) {
    state.lists = lists;
  },

  [types.LOADING_TOUR](state, value) {
    state.loading = value;
  },

};

export default mutations;
```

**Mutations** are functions that get triggers depending on the type of actions that get dispatched. They perform the actual state modifications. As you can see in the preceding code, **types.GET\_TOUR\_LISTS** changes the value of **state.lists**. The **state** parameter, which is the first parameter, is a state that is part of the global state. It is automatically passed here by Vuex.

The second parameter is the optional payload that comes from the action we defined earlier.

Also, the job of the mutation is to update a part of the global state directly. It depends on whether you want to delete or update an existing state.

## Step 7 – Writing a state

Create a JavaScript file named **getters.js** inside the **tour** folder, and then write the following code:

```
const getters = {

  lists: (state) => state.lists,

  loading: (state) => state.loading,

};

export default getters;
```

**Getters** are the computed properties or derived values for stores. The result or output of **getters** is cached based on its dependencies and will rerun or re-evaluate when any of its dependencies have changed. **getters** are what we will include in our component to connect the global states to our UI.

## Step 8 – Writing a mutation

Create a JavaScript file named **index.js** in the **tour** folder. This file will be in the index file of our **tour** module. Once the file is created, we write the following code:

```
import state from "../state";
import getters from "../getters";
import mutations from "../mutations";
import * as actions from "../actions";
export default {
  namespaced: true,
  getters,
  mutations,
  actions,
  state,
};
```

We import the **state** file, **getters** file, **mutations** file, and the **actions** file for the **index** file of our **tour** module or namespace. After importing the necessary files, we are exporting them and including a **namespaced** property set to **true**.

**Modules** or **namespaces** are divisions inside the store. Each module or namespace has its **actions**, **state**, **mutations**, **getters**, and even nested modules.

## Step 9 – Writing a getter

Update the **index.js** file of the store with the following code. The code is where we are going to add the **tour** module:

```
import Vue from "vue";
import Vuex from "vuex";
import createLogger from "vuex/dist/logger";
import tourModule from "../tour";
Vue.use(Vuex);
const debug = process.env.NODE_ENV !== "production";
const plugins = debug ? [createLogger({})] : [];
export default new Vuex.Store({
  modules: {
```

```

    tourModule,
  },
  plugins,
});

```

We import a logger, the tour module in the preceding file, and then remove the following properties: **state**, **mutations**, and **actions**. We then use **tourModule** as the property of the **modules** object.

The **modules** object is where all the modules of our Vue.js app are located. That means we will update the **modules** object as we add a new module or namespace in our Vue.js application.

## Step 10 – Updating the store by inserting the module

Now let's update the **WeatherForecast.vue** page. Also, let's import **mapActions** and **mapGetters** from **vuex**:

```
import { mapActions, mapGetters } from "vuex";
```

**mapGetters** is a helper that simply maps local computed properties to store **getters**. Similarly, **mapActions** is a helper that maps store dispatch to the component methods. We will be using these mappers in the **methods** block and **computed** block.

Insert **mapActions** inside the first line of the **methods** block. Use the **spread** operator, the three dots prefixing **mapActions** like so:

```

methods: {
  ...mapActions("tourModule", ["getTourListsAction"]),
  ...
},

```

**mapActions** maps **this.getTourListsAction**, a **local** method that is automatically created, to **this.\$store.dispatch("getTourListsAction")**. We can now use **this.getTourListsAction** to trigger an action for mutation.

Before using **getTourListsAction**, we would also want to access the **lists** state of our store. To do that, create **computed** block and insert **mapGetters** using a spread operator like so:

```

computed: {
  ...mapGetters("tourModule", {
    lists: "lists",
  }),
},

```

**mapGetters** maps **this.lists**, a local state that is automatically created, to **this.\$store.getters.lists**.

## Step 11 – Updating components with mapGetters and mapActions

Let's update the **mounted** method with the following code. We are going to add two new lines of code:

```
async mounted() {  
    ...,  
    await this.getTourListsAction();  
    this.cities = this.lists.map(pl => pl.city);  
},
```

We are triggering **getTourListsAction** and then extracting the cities from the lists and storing them in the cities' local state.

Now, for the dropdown or *select* UI of the **WeatherForecast** screen, add the **v-select** component on top of the **v-simple-table** component in the **template** syntax area:

```
<v-select  
    @change="fetchWeatherForecast"  
    v-model="selectedCity"  
    :items="cities"  
    label="City"  
    persistent-hint  
    return-object  
    single-line  
    clearable  
></v-select>
```

The preceding component will give the **WeatherForecast** screen a functionality where we can choose which city to check for a 14-day weather forecast.

Before running the ASP.NET Core and the Vue.js app, I want you to use the SQLite database of the **Travel.WebApi** project. I have updated the data inside it to help us see what we are working on in the frontend in terms of design. You will see the JSON equivalent of the SQLite data in the GitHub repo <https://github.com/PacktPublishing/ASP.NET-Core-5-and-Vue.js-3/tree/master/Chapter-14>, to let you know what data is currently in the database and the shape of it.

Delete the **TravelTourDatabase.sqlite3** database file you have right now in your project and replace it with the one from the GitHub repository.

Rerun the application by running the following command inside the **Travel.WebApi** project:

```
dotnet run
```

Wait for a couple of seconds, then check your browser. You should see the drop-down menu selector as shown in the following screenshot:

The screenshot shows a web application with a blue header bar containing a logo and navigation links: HOME, ABOUT, DASHBOARD, and LOGIN. Below the header, there is a user profile section with a circular profile picture and the text "mail her". To the left of the main content area is a sidebar with several icons: a grid, a list, a person, a cloud, and a document. The main content area features a title "Two-week weather forecast of different cities" and a search bar with the text "Oslo" and a close button (X). Below the search bar is a table displaying the weather forecast for Oslo over 14 days. The table has five columns: Dates, City, °C, °F, and Summary. The data is as follows:

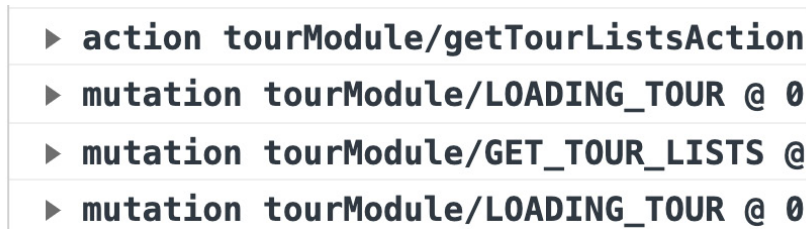
Dates	City	°C	°F	Summary
in a day	Oslo	1	33	Bracing
in 2 days	Oslo	0	32	Bracing
in 3 days	Oslo	-2	29	Freezing
in 4 days	Oslo	39	102	Sweltering
in 5 days	Oslo	22	71	Warm
in 6 days	Oslo	4	39	Bracing
in 7 days	Oslo	28	82	Balmy
in 8 days	Oslo	16	60	Mild
in 9 days	Oslo	32	89	Hot
in 10 days	Oslo	5	40	Chilly
in 11 days	Oslo	25	76	Balmy
in 12 days	Oslo	4	39	Bracing
in 13 days	Oslo	1	33	Bracing
in 14 days	Oslo	19	66	Mild

Figure 14.6 – Vuetify select component



Figure 14.6 shows the selector with the default **Oslo** as the selected city. Try clicking it to see the other available cities.

Next, check out the console logs on your browser's DevTools. You should also see the Vuex logger plugin that we included in the store setup. It will look something like what's shown in Figure 14.7:



```
► action tourModule/getTourListsAction
► mutation tourModule/LOADING_TOUR @ 0
► mutation tourModule/GET_TOUR_LISTS @
► mutation tourModule/LOADING_TOUR @ 0
```

Figure 14.7 – Vuex logger

The Vuex logger in the preceding screenshot shows the logs of what's happening inside the Vuex state management of the Vue.js application. We can use this to debug our application if there are unexpected behaviors in our application while using the Vuex store's parts.

Using Vuex and other state management libraries in other frameworks takes a lot of code to set them up. But once you've finished setting up all the moving parts and your store's configuration, adding new actions and using them becomes easy because you just need to update the existing code. You are only going to write the setup once, and the rest is seamless.

Even though the setup takes time to finish, the benefits where you don't have to pass down props and events in and out of nested components are so good and will make your life easier when developing complex synchronization of states between nested components.

I suggest remembering the preceding flow of topics to have a checklist of what to do whenever you are implementing Vuex in your project. Now let's recap what you have learned in this chapter.

## Summary

This chapter might be one of the most challenging chapters so far because of the state management concept. Nevertheless, learning how to do state management is invaluable. You have learned how to send HTTP requests using Axios. You have discovered the idea of state management in big applications.

You have also learned how to use Vuex and Vuex's parts, such as the store, modules, actions, mutations, and getters.

In the next chapter, we will build functionalities for sending *POST*, *DELETE*, and *PUT HTTP* requests in Vue.js with Vuex.

# ***Chapter 15: Sending POST, DELETE, and PUT HTTP Requests in Vue.js with Vuex***

In the last chapter, we discussed state management and set up Vuex in our Vue.js app. We managed to write our first action and tried it in our application. However, the CRUD operations using Vuex in our application are not yet complete. In this chapter, we will write the remainder of the CRUD operations with HTTP requests and Vuex involvement.

With that, we will cover the following topics in this chapter:

- Removing a tour list using Axios and Vuex
- Adding a tour list using Axios and Vuex
- Using a non-async action in Vuex
- Removing a tour package using Axios and Vuex
- Adding a tour package using Axios and Vuex
- Updating a tour package using Axios and Vuex

## **Technical requirements**

You will need Visual Studio Code to complete this chapter.

The finished repository of this chapter can be found at <https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter15/>.

## **Removing a tour list using Axios and Vuex**

If you are still hungry to do some writing with Axios and Vuex, this is perfect for you. We will send a request to our ASP.NET Core Web API to retrieve the **TourList** collection, render them on the UI using Vuetify components, and then be able to delete any of the **TourList** objects. But before we start, let's update our ASP.NET Core Web API project.

Update the **TourPackageDto.cs** file in **namespace Travel.Application.Dtos.Tour** with the following code:

```
public float Price { get; set; }

public string MapLocation { get; set; }
```

We are adding the **Price** and **MapLocation** properties.

Next, we update our backend service. Go to the **TourPackagesController.cs** file, which has a **namespace of Travel.WebApi.Controllers.v1**, and add a new async action method using the following code:

```
[HttpPut("{id}")]

public async Task<ActionResult> Update(int id,
UpdateTourPackageCommand command)

{

    if (id != command.Id)

        return BadRequest();

    await Mediator.Send(command);

    return NoContent();

}
```

The preceding C# code is an additional controller for our Web API to edit an existing tour package in our application.

Now that we've updated our ASP.NET Core Web API project, let's focus on our Vue.js app and complete our Travel Tour web app's functionalities.

In my opinion, creating a **delete** functionality is the second easiest thing to implement next to the **fetch** functionality.

So, let's update the **DefaultContent.vue** file of the **AdminDashboard** directory. The first thing is to import **mapActions** into the **DefaultContent** component, like so:

```
import { mapActions } from "vuex";
```

We will also use **mapActions** in the **DefaultContent** component, just like in the **WeatherForecast** component in the previous chapter.

Next, we create the **methods** object and use **mapActions** like so:

```
methods: {  
  ...mapActions("tourModule", ["getTourListsAction"]),  
},
```

The preceding code gives us a **getTourListsAction** method.

Now create the **mounted** method, and let's call **getTourListsAction** inside it:

```
mounted() { this.getTourListsAction(); },
```

The preceding code will trigger **getTourListsAction** upon rendering the DOM.

Next, we write a localized style section and write a default-content class:

```
<style scoped>
```

```
.default-content {
```

```
display: flex;

flex-direction: row;

flex-wrap: wrap;

justify-content: flex-start;

}

</style>
```

The preceding CSS will provide styling for a **div** tag that we will write in the next code.

Now let's update the **template** syntax section:

```
<template>

  <div>

    <div class="text-h2 my-4">Welcome to Admin

    Dashboard</div>

    <div class="default-content"></div>

  </div>

</template>
```

The preceding code is a simple welcome message that says **Welcome to Admin Dashboard**. We will come back to this component later after creating a **card** component.

Go to the **components** folder and create a **Vue.js** component file, **TourListsCard.vue**.

Add the following script to the newly created Vue component:

```
<script>

import { mapActions, mapGetters } from "vuex";

export default {

  name: "TourListsCard",

  computed: {

    ...mapGetters("tourModule", {

      lists: "lists", loading: "loading",

    }),

  },

};

</script>
```

We are bringing in **mapActions** and **mapGetters** from Vuex. We are then using **mapGetters** to bring the global states' lists and loading to local states' lists and loading. Take note that any changes in the global states' state and loading will also reflect in the localized lists and loading states.

Let's add the following code to the newly created **TourListsCard** Vue component:

```
<template>

  <v-skeleton-loader

    v-if="loading" width="300" max-width="450"

    height="100%" type="card" ></v-skeleton-loader>

  <v-card v-else width="300" max-width="450" height="100%">
```

```
<v-toolbar color="light-blue" dark>

  <v-toolbar-title>Tour Lists</v-toolbar-title>

  <v-spacer></v-spacer>

</v-toolbar>

<v-list-item-group color="primary">

  <v-list-item v-for="tourList in lists"
:key="tourList.id">

    <v-list-item-content>

      <v-list-item-title v-text="tourList.city"></v-
list-item-title>

      <v-list-item-subtitle v-text="tourList.about">

</v-list-item-subtitle>

    </v-list-item-content>

    <v-list-item-action>

      <div class="mr-2">

        {{ tourList.tourPackages &&
tourList.tourPackages.length }}

      </div>

      <v-icon> mdi-delete-outline </v-icon>

    </v-list-item-action>
```

```
</v-list-item>

</v-list-item-group>

</v-card>

</template>
```

The preceding **template** syntax renders a loader and a card with **tourLists** packages. We are using the **v-for** directive to extract the objects from the lists and then render them one by one. Each **tourList** object will show the two values of two properties– the value of the **city** property and the value of the **about** property in the UI.

Now let's go back to **DefaultContent.vue** and update it.

Import the **TourListsCard** component from the **components** directory like so:

```
import TourListsCard from "@/components/TourListsCard";
```

You can use **@/** to navigate to the folders of the **src** directory easily. Now use the imported component as follows:

```
components: { TourListsCard, },
```

We've registered the **TourListsCard** component in the components of **DefaultContent**.

Now, inside the **div** element with default content class, add the following code:

```
<div style="margin-right: 4rem; margin-bottom: 4rem">

  <TourListsCard />

</div>
```



We are rendering **TourListsCard** in the preceding code. Restart your server because we have created a new file.

In the **Travel.WebApi** project, run the following command:

```
dotnet run
```

The preceding command will run the whole application, which means both the backend and the frontend.

The following *Figure 15.1* shows a simple dashboard with a welcome message:

 Figure 15.1 – A welcome message on the Admin Dashboard component

Figure 15.1 – A welcome message on the Admin Dashboard component

Now we are about to write the **delete** functionality in Axios and store it. You will find out how easy it is to write additional actions in our state management because we already set up our store in the last chapter. Now let's proceed.

Next, we update **services.js** of the **store/tour/** directory using the following code:

```
export async function deleteTourListAxios(id) {  
  return await api.delete("TourLists/" + id);  
}
```

The preceding code is a service that deletes a **tourList** object:

Let's also update **types.js** of **store/tour/**:

```
export const REMOVE_TOUR_LIST = "REMOVE_TOUR_LIST";
```

We are adding a new action type to the **types.js** file. The action type is for removing a tour list, hence the name **REMOVE\_TOUR\_LIST** in snake

casing.

Let's also update the **actions.js** file of the **store/tour/** directory:

```
import { getTourListsAxios, deleteTourListAxios } from
"@/store/tour/services";
```

We bring in the **deleteTourListAxios** service first before adding the **remove** function.

Now, let's write the **removeTourListAction** function like so:

```
// asynchronous action using Axios

export async function removeTourListAction({ commit }, payload) {

  commit(types.LOADING_TOUR, true);

  try {

    await deleteTourListAxios(payload);

    commit(types.REMOVE_TOUR_LIST, payload);

  } catch (e) {

    alert(e);

    console.log(e);

  }

  commit(types.LOADING_TOUR, false);

}
```

The preceding function, which is **removeTourListAction**, takes a payload. The parameter you need to use when triggering **removeTourListAction** is

technically just one, the payload, whereas the payload is an ID of **tourList** that we are going to remove.

Now, you will notice in **removeTourListAction** that it enables the loading by committing **types.LOADING\_TOUR** and passing **true** before sending the delete request using the **deleteTourListAxios** service. The function also creates a commit that removes **tourList** and deletes it in ASP.NET Core.

**removeTourListAction** also commits **types.LOADING\_TOUR** again with a Boolean of **false**, which will disable the **loader** component, or **spinner** component if you will, from spinning.

Now, update **mutations.js** of **store/tour/**:

```
[types.REMOVE_TOUR_LIST](state, id) {  
  state.lists = state.lists.filter((tl) => tl.id !== id);  
  
  state.packagesOfSelectedCity = [];  
  
},
```

The **types.REMOVE\_TOUR\_LIST** action type has an instruction to filter out all the objects by only getting an object with **id** that is not equal to **id** from the parameter. Then the action type also sets the **packagesOfSelectedCity** global state to an empty array.

Now, let's update the **TourListsCard.vue** component by adding a **methods** object with **mapActions** and an extra **removeTourList** function, which takes **listId** like so:

```
methods: {  
  
  ...mapActions("tourModule", ["removeTourListAction"]),  
  
  removeTourList(listId) {  
  
    const confirmed = confirm(  

```

```

        "You sure you want to delete this tour list? This
        will also delete the packages permanently"

    );

    if (!confirmed) return;

    this.removeTourListAction(listId);

  },

},

```

You will notice that there's a confirmation prompt before deleting or removing a tour list.

Now, let's update the **v-icon** component in the **template** syntax section of the **TourListsCard** Vue component like so:

```

<v-icon @click="removeTourList(tourList.id)">

  mdi-delete-outline

</v-icon>

```

In the preceding code, we are binding the **removeTourList** function to the **@click** directive, which will provide us with a click event.

If an ESLint error prevents the Vue.js app from refreshing, restart the server. Now rerun the ASP.NET Core Web API project by running the **dotnet run** command.

The command will start the whole application again. Now, open up your browser and head to the **Admin Dashboard** page of our application:

 Figure 15.2 – Card component with the Tour Lists collection

Figure 15.2 – Card component with the Tour Lists collection

The preceding *Figure 15.2* shows the card UI component after rerunning the application. You can now see the collection of **tourList** with the **delete** functionality. The numbers above the trash bin icon indicate packages available in a tour list. **0** means no package added as yet.

Try to delete **Manila** and look closely at the UI. The UI should update after removing a tour list.

We've successfully added the functionality that removes or deletes a tour list in our application. In the next section, we will create a form where we can create a new tour list.

## Adding a tour list using Axios and Vuex

The **Create** or **Add** functionality would be the third easiest thing to implement in CRUD. Fetching and deleting functionalities is easy because they don't require a **form** component and some input fields. To create a functionality that adds or makes new data, we have to build a form.

Before building a form for the UI, we will first develop functionality in Axios and Vuex that creates a tour list.

To start, let's update **services.js** in **store/tour** by adding the following code:

```
export async function postTourListAxios(tourList) {  
  return await api.post("TourLists", tourList);  
}
```

The preceding code is a service that sends a *POST* request to the backend to create a **tourList** data entry.

Now, let's update **types.js** of the **store/tour** directory.

```
export const ADD_TOUR_LIST = "ADD_TOUR_LIST";
```

We are adding a new action type in the preceding code. The latest action type is for adding new **tourList**.

Next we update **actions.js** of the **store/tour** directory:

```
import { getTourListsAxios, deleteTourListAxios,  
  postTourListAxios, } from "@store/tour/services";
```

Let's import the **postTourListAxios** service first before adding a new action.

After importing the newly created service, let's add a new action to **actions.js**:

```
// asynchronous action using Axios  
  
export async function addTourListAction({ commit }, payload) {  
  commit(types.LOADING_TOUR, true);  
  
  try {  
    const { data } = await postTourListAxios(payload);  
    payload.id = data; // storing the id from the response  
  
    // int of ASP.NET Core, which will be used in the UI.  
    payload.tourPackages = []; // initialize the  
    // tourPackages of the newly created tourList  
    commit(types.ADD_TOUR_LIST, payload);  
  } catch (e) {
```

```
    alert(e);

    console.log(e);
}

commit(types.LOADING_TOUR, false);
}
```

The preceding code is another asynchronous action function. It means that this is an action that uses **Axios** to send an HTTP request. The action takes **tourList** as the payload. It sends it to ASP.NET Core utilizing the service.

In response, the service returns **id** of the newly created tour list from the backend. We will then store the response in **payload.id** to use it in the mutation when we update our UI.

Now, let's update **mutations.js** in **store/tour** by adding the following code:

```
[types.ADD_TOUR_LIST](state, tourList) {

    state.lists.unshift(tourList);

},
```

The preceding mutation adds newly created **tourList** at the beginning of the array. The **unshift** helps us to see freshly added **tourList** in the **card** component easily.

Now we will create a **helper** function, and we will put it in a separate folder.

Let's create a new folder inside the **src** directory and name it **helpers**. Then, create a JavaScript file named **collections.js** inside the **helpers** folder.

The **collection.js** file has a **getCountryList** function that returns an array of countries, which we will need in the form's search input. We are going to create the form and bind the **getCountryList** helper function shortly.

So, get the helper function, called **getCountryList**, from the GitHub repository here: <https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter15>.

Now for the form. Create a Vue component named **AddTourListForm.vue** in the **components** folder and add the following script:

```
<script>

import { getCountryList } from "@helpers/collections";

import { mapActions } from "vuex";

export default {

  name: "AddTourListForm",

  data: () => ({

    bodyRequest: { city: "", country: "", about: "", },

    dialog: false,

    countryList: getCountryList(),

  }),

  methods: {

    ...mapActions("tourModule", ["addTourListAction"]),

  },

};

</script>
```

We are importing the **getCountryList** helper function as well as **mapActions** from Vuex. We declare a local state **bodyRequest** object, a



Boolean dialog, and **countryList**, which is an array that **getCountryList** returns.

Now, the following code is for the **template** syntax of our **AddTourListView** component:

```
<template>

  <v-row justify="center">

    <v-dialog v-model="dialog" persistent max-
      width="600px">

      <template v-slot:activator="{ on, attrs }">

        <v-btn
          style="margin-top: 1rem" rounded color="light-
            blue"
          dark v-bind="attrs" v-on="on">

          <v-icon left>mdi-plus</v-icon>

          add new tour list

        </v-btn>

      </template>

      <!-- INSERT <v-card> BELOW -->

    </v-dialog>

  </v-row>

</template>
```

**v-btn** is a button component from Vuetify, which will trigger the template with a **v-slot** directive activator to show up the dialog form later. A **v-slot** directive is used in wrapping another component to make it reusable.

You will notice that there's a comment that says **insert v-card**. We will add the following **template** syntax. Therefore, update **AddTourListForm.vue** with the following code:

```
<v-card>

  <form @submit.prevent="
    addTourListAction(bodyRequest);
    bodyRequest = {};">

    <v-card-title>

      <span class="headline">Create New Tour
        List</span>
    </v-card-title>

    <v-card-text>

      <v-container>

        <v-row>

          <v-col cols="12" sm="6">

            <v-text-field required label="City"
              v-model="bodyRequest.city"
            ></v-text-field>

          </v-col>
```

```

    <v-col cols="12" sm="6">

      <v-autocomplete required

        :items="countryList" label="Country"

        v-model="bodyRequest.country"

      ></v-autocomplete>

    </v-col>

<!-- Removed for brevity -->

<!-- Please see the full code in GitHub -->

<v-btn color="blue darken-1" text

  @click="dialog = false" type="submit" >

  Save

</v-btn>

</v-card-actions>

</form>

</v-card>

```

The preceding code is the form that will let us create new **tourList**. The form has a text field for **city**, a field with autocomplete for **country**, and a text area for **about**. We may notice that the **v-btn** component has a **submit** type. The **submit** type triggers **@submit** of the form that runs **addTourListAction**.

Now, let's include the **form** component in the **DefaultContent.vue** file by updating it:

```
import AddTourListForm from "@/components/AddTourListForm";
```

We are bringing in the **AddTourListForm** component from the components.

Next, we register **AddTourListForm** in the components object like so:

```
components: { TourListsCard, AddTourListForm, },
```

**AddTourListForm** is now ready to use in the template syntax area of the **Default** component.

Add the **AddTourListForm** component below the **TourListsCard** component:

```
<TourListsCard />
```

```
<AddTourListForm />
```

Restart the server because we have created a file. Run the **dotnet run** command to run the whole application.

Open your browser and go to the admin dashboard of the application, where you will see the following screenshot:

 Figure 15.3 – ADD NEW TOUR LIST button

Figure 15.3 – ADD NEW TOUR LIST button

The preceding *Figure 15.3* shows the **ADD NEW TOUR LIST** button from Vuetify. Click it to open the form:

 Figure 15.4 – Adding a tour list Form

Figure 15.4 – Adding a tour list Form

*Figure 15.4* shows the form for creating or adding a new tour list. Try it and see the result of creating and sending a tour list.

In the next section, we will work on getting the tour packages for a particular tour list.

## Using a non-async action in Vuex

We can see the collections of tour lists in the UI, but not their tour packages. This section will extract the tour packages from a tour list using Vuex without Axios because we are not sending an HTTP request. No HTTP request means that we are going to use a non-asynchronous action. You will learn how to use Vuex and a non-async action efficiently. So let's start.

Update the **types.js** file in the **store/tour** folder with the following code:

```
export const GET_PACKAGES_OF_SELECTED_CITY =  
"GET_PACKAGES_OF_SELECTED_CITY";
```

The preceding code is a new action type we are adding in **types.js**.

Then, update **actions.js** in **store/tour** with the following code:

```
// non-asynchronous action  
  
export function getPackagesOfSelectedCityAction({ commit }, payload) {  
  commit(types.GET_PACKAGES_OF_SELECTED_CITY, payload);  
}
```

You will notice in the preceding action that there are no **try** and **catch** blocks since we are not using Axios in here.

Next, we update **state.js** of the **store/tour** folder:

```
packagesOfSelectedCity: [],
```

Finally, we have a state for our tour packages from a specific city.

Now, update **mutations.js** of the **store/tour** folder:

```
[types.GET_PACKAGES_OF_SELECTED_CITY](state, packages) {  
  state.packagesOfSelectedCity = packages;  
},
```

We are now using the payload as packages and then storing them in the **packagesOfSelectedCity** state, which is an array.

Next, we update **getters.js** of the **store/tour** folder:

```
packagesOfSelectedCity: (state) => state.packagesOfSelectedCity,
```

We are going to use the preceding getter function for our UI later.

It's time to update the **TourListsCard** component. Go to the **TourListsCard.vue** file and update the component with the following code:

```
...mapActions("tourModule", [  
  "removeTourListAction",  
  "getPackagesOfSelectedCityAction",  
]),
```

We are now exposing the new action, which is **getPackagesOfSelectedAction**.

Let's add a new function to **methods** using the following block of code:

```
addToPackages(packages, listId) {  
  this.getPackagesOfSelectedCityAction(packages);  
  this.$emit("handleShowPackages", true, listId);
```

```
},
```

We are using **getPackagesOfSelectedCityAction** and emitting **handleShowPackages**, which will act as props for the **TourListsCard** component.

Next, we update the **v-list-item-content** component of the **TourListsCard** component:

```
<v-list-item-content  
  
  @click="addToPackages(tourList.tourPackages,  
  
  tourList.id)">
```

The click event will trigger the **addToPackages** function that we just added in the **methods** block.

Now, let's update the **DefaultContent.vue** component file with the following code:

```
methods: {  
  
  ...mapActions("tourModule", ["getTourListsAction"]),  
  
  handleShowPackages(show, listId) {  
  
    this.showPackages = show;  
  
    this.tourListId = listId;  
  
  },  
  
},
```

We are adding a **handleShowPackages** method to the **methods** block.

Now, let's also add some local states:

```
data: () => ({ showPackages: false, tourListId: 0, }),
```

We have **showPackages**, which is a Boolean type, and **tourListId**, which is a number.

Now let's use the **mounted** life cycle hook of the **DefaultContent.vue** component:

```
mounted() {  
  
  this.getTourListsAction();  
  
  this.showPackages = false;  
  
},
```

We call **getTourListsAction** and set **showPackages** to **false** whenever we go to the admin dashboard page.

Then, update **<TourListsCard />** in the **template** syntax:

```
<TourListsCard @handleShowPackages="handleShowPackages" />
```

We are using the emitted **handleShowPackages** props of **TourListsCard** and assigning the **handleShowPackages** function of **DefaultComponent.vue**.

Let's update the **template** syntax again with the following code:

```
<div style="margin-right: 4rem; margin-bottom: 4rem">  
  
  <TourListsCard @handleShowPackages  
  
    ="handleShowPackages" />  
  
  <AddTourListForm />  
  
</div>
```



```
<div v-if="showPackages">

  <h2>Tour Packages Card Here</h2>

  <h3>Add Tour Package Form with tour list Id

  Here</h3>

</div>
```

We are adding the **v-if** directive to the preceding template area code.

Now run the app and click any city.

The **TourPackagesCard** component should appear on the right side of the screen after running the app and clicking any city.

We are doing a quick proof-of-concept here that we can render the **TourPackagesCard** component, as shown in the following screenshot:

 Figure 15.5 – Show placeholders

Figure 15.5 – Show placeholders

*Figure 15.5* shows us that the placeholder messages pop up after selecting any **Tour Lists** card cities. Now we can conditionally show a UI. Let's create the card component for the packages.

Create **TourPackagesCard.vue** in the **components** folder and add the following script:

```
<script>

import { mapActions, mapGetters } from "vuex";

export default {

  name: "TourPackagesCard",
```

```

computed: {
  ...mapGetters("tourModule", {
    packages: "packagesOfSelectedCity",
  }),
},
};
</script>

```

The preceding script is importing and using **mapAction** and **mapGetters**.

Now let's add the **template** syntax section of **TourPackagesCard**. Add the following block of code:

```

<template>

  <v-container>

    <v-card width="500" max-width="600">

      <v-toolbar color="pink" dark>

        <v-toolbar-title>Packages</v-toolbar-title>

        <v-spacer></v-spacer>

      </v-toolbar>

      <v-list two-line>

        <v-list-item-group active-class="pink--text">

          <div v-if="packages && packages.length > 0">

            <template v-for="tourPackage in packages">

```

```
    <v-list-item :key="tourPackage.id">

    <!-- <template> for <v-list-item-content>
    here -->

    </v-list-item>

  </template>

  <v-divider />

</div>

<div v-else>

  <v-list-item>

    <v-list-item-content>

      <v-list-item-title

        v-text="No package added yet 😞"

      ></v-list-item-title>

    </v-list-item-content>

  </v-list-item>

</div>

</v-list-item-group>

</v-list>

</v-card>

</v-container>
```

```
</template>
```

The preceding **template** code is a card component from Vuetify that renders every package of a selected city. The **card** component's look and feel will be similar to that of the tour lists, the only difference being the color.

Now let's add the list item content of the **card** component. Replace the comment `<!-- <template> for <v-list-item-content> here -->` with the following code:

```
<template>
```

```
  <v-list-item-content>
```

```
    <v-list-item-title v-text="tourPackage.name"></v-list-
    item-title>
```

```
    <v-list-item-subtitle
```

```
      class="text--primary"
```

```
      v-text="tourPackage.whatToExpect"
```

```
    ></v-list-item-subtitle>
```

```
    <div style="margin-top: 0.5rem; display: flex; flex-
    direction: row">
```

```
      <v-list-item-subtitle
```

```
        class="text--secondary"
```

```
        v-text="`Duration: ${tourPackage.duration}hrs`"
```

```
      ></v-list-item-subtitle>
```

```

    <v-list-item-subtitle
      class="text--secondary" v-text="
        `${tourPackage.instantConfirmation ? 'Instant
        Confirmation' : ''}`"
    ></v-list-item-subtitle>

  </div>

</v-list-item-content>

<v-list-item-action>

  <v-icon> mdi-delete-outline </v-icon>

</v-list-item-action>

</template>

```

The preceding code is the UI for the row for each package. The row shows the values of the tour package's properties, such as **name**, **whatToExpect**, **duration**, and **instantConfirmation**.

Now again, update the **DefaultContent.vue** component with the following code:

```
import TourPackagesCard from "@/components/TourPackagesCard";
```

We are importing the **TourPackagesCard** component.

Let's register the **TourPackagesCard** component in our **components** block like so:

```

components: { TourListsCard, AddTourListForm,
  TourPackagesCard, },

```

Now, the **TourPackagesCard** component is available to use.

Replace the **v-if="showPackages"** and placeholder messages with the following code:

```
<div v-if="showPackages">  
  <TourPackagesCard />  
</div>
```

Now, our **TourPackagesCard** component should be visible after clicking a city.

Run the **dotnet run** command to start the whole application again.

Then, click on **Oslo**, which will show the **TourPackagesCard** component with some packages inside, as shown in the following screenshot:

 Figure 15.6 – TourPackagesCard component

Figure 15.6 – TourPackagesCard component

*Figure 15.6* shows the **TourPackagesCard** component on the right.

Now that we have finished the only non-async action in our application, let's go back to writing async actions in our Vuex.

## Removing a tour package using Axios and Vuex

In this section, we will implement functionality that deletes a tour package in the backend and deletes the tour package in the UI. So let's start.

Update **services.js** of the **store/tour** folder:

```
export async function deleteTourPackageAxios(id) {  
  return await api.delete("TourPackages/" + id);  
}
```

We are adding a new service that deletes a tour package.

Next, we update **types.js** of the **store/tour** folder:

```
export const REMOVE_TOUR_PACKAGE =  
"REMOVE_TOUR_PACKAGE";
```

In the preceding code, we are adding a new action type to remove a tour package.

Next, we update **actions.js** of the **store/tour** folder:

```
import { getTourListsAxios, deleteTourListAxios,  
  postTourListAxios, deleteTourPackageAxios,  
} from "@store/tour/services";
```

We are importing **deleteTourPackageAxios** in the preceding code.

Let's add another action to **actions.js**:

```
export async function removeTourPackageAction({ commit }, payload) {  
  commit(types.LOADING_TOUR, true);  
  try {  
    await deleteTourPackageAxios(payload);  
    commit(types.REMOVE_TOUR_PACKAGE, payload);  
  } catch (e) {
```

```

    alert(e);

    console.log(e);
  }

  commit(types.LOADING_TOUR, false);
}

```

The **removeTourPackageAction** function enables the loader to spin, sends a *DELETE* request to the backend using Axios, commits a type that will instruct the mutation to remove the UI tour package, and then disables the loader to stop it from spinning.

Then, update **mutations.js** in the **store/tour** folder:

```

[types.REMOVE_TOUR_PACKAGE](state, id) {

  state.packagesOfSelectedCity =

  state.packagesOfSelectedCity.filter(

    (tp) => tp.id !== id);

  },

```

The preceding mutation removes the UI tour package using a filter built into the JavaScript.

Now we can go to the **TourPackagesCard.vue** component and update it with the following code:

```

methods: {

  ...mapActions("tourModule", ["removeTourPackageAction"]),

  removeTourPackage(packageId) {

```



```

const confirmed = confirm(
  "You sure you want to permanently delete this tour
  package?");
if (!confirmed) return;

// might need to wait for 1 min because of the cache
this.removeTourPackageAction(packageId);
},
},

```

We are adding a **methods** block and using **mapActions** to map **removeTourPackageAction**. We are then creating another method, called **removeTourPackage**, which triggers a confirm dialog and **removeTourPackageAction**.

And lastly, update the **v-icon** component by giving it a click event that runs the **removeTourPackage** function with **tourPackage.id** as an argument:

```

<v-icon @click="removeTourPackage(tourPackage.id)">

  mdi-delete-outline

</v-icon>

```

The **v-icon** component is ready to use. We can click it to remove a tour package. Go to the Vue app and click **City**. After clicking on a city, delete any tour package. You should see that the tour package gets removed from the UI.

Now we can go to the next section, which creates a functionality where we can add a new tour package.

# Adding a tour package using Axios and Vuex

This section will build a functionality to add a new tour package to an existing tour list using Axios and Vuex.

Let's update **services.js** of the **store/tour** folder:

```
export async function postTourPackageAxios(tourPackage) {  
  return await api.post("TourPackages", tourPackage);  
}
```

In the preceding code, we add a new service that creates new **tourPackage** in the backend.

Next, we update **types.js** of the **store/tour** folder:

```
export const ADD_TOUR_PACKAGE = "ADD_TOUR_PACKAGE";
```

The code update adds a new action type for adding a tour package.

Now, update **actions.js** in **store/tour** using the following code:

```
import { getTourListsAxios, deleteTourListAxios,  
  postTourListAxios, deleteTourPackageAxios,  
  postTourPackageAxios,  
} from "@store/tour/services";
```

We import the service **postTourPackageAxios** first, as you can see in the preceding code. Now we can use imported **postTourPackageAxios** in the following code:

```

export async function addTourPackageAction({ commit }, payload) {
  commit(types.LOADING_TOUR, true);

  try {
    const { data } = await postTourPackageAxios(payload);
    payload.id = data; // storing the id from the response
    // This is the id of ASP.NET Core, which will be used in the UI.
    commit(types.ADD_TOUR_PACKAGE, payload);
  } catch (e) {
    alert(e);
    console.log(e);
  }

  commit(types.LOADING_TOUR, false);
}

```

**addTourPackageAction** turns on the loader, sends a *POST* request to the backend, adds **tourPackage** to the UI by calling a commit, and then turns off the loader.

Next, we update **mutations.js** of the **store/tour** folder:

```

[types.ADD_TOUR_PACKAGE](state, tourPackage) {
  state.packagesOfSelectedCity.unshift(tourPackage);
},

```

The preceding code is a mutation that adds the new **tourPackage** to the first index of the **packagesOfSelectedCity** array.

Then, for the form, where we can add new **tourPackage**, create a Vue component, **AddTourPackageForm.vue**, in components and the following script:

```
<script>

import { mapActions } from "vuex";

export default {

  name: "AddTourPackageForm",

  props: {

    tourListId: { type: Number, },

  },

  data: () => ({

    id: 0,

    bodyRequest: {

      listId: 0, name: "", whatToExpect: "",

      mapLocation: "https://www.google.com/maps/place/...",

      price: 10, duration: 1, instantConfirmation: true,

    },

    dialog: false,

    currencies: ["USD", "NOK"],
```

```

    currencyValues: [0, 1],
    durations: [1, 2, 3, 4, 5, 6, 7, 8],
    durationValue: 1,
  )),
  methods: {
    ...mapActions("tourModule", ["addTourPackageAction"]),
    onSubmit() {
      this.bodyRequest.listId = this.tourListId;
      this.addTourPackageAction(this.bodyRequest); //
      triggers the method of the container holding this
      this.bodyRequest = {};
    },
  },
};
</script>

```

We have **id**, **bodyRequest**, **dialog**, **currencies**, **currencyValues**, **durations**, and **durationValue** local states in the preceding code. We also have a **mapActions** spread in the methods and an **onSubmit** method that triggers **addTourPackageAction**. And lastly, the form has a **tourListId** props, which is **id** of **tourList**.

Add the following **template** syntax for the UI:

```
<template>
```

```

<v-row justify="center">

  <v-dialog v-model="dialog" persistent max-
width="600px">

    <template v-slot:activator="{ on, attrs }">

      <v-btn

        style="margin-top: 1rem" rounded color="pink"

        dark v-bind="attrs" v-on="on">

          <v-icon left> mdi-plus </v-icon>

          add new tour package

        </v-btn>

      </template>

      <!-- Insert <v-card> here -->

    </v-dialog>

  </v-row>

</template>

```

We have a **v-btn** component that opens up a dialog or modal form. This form is similar to the form of the tour lists.

Now, replace **<!-- Insert <v-card> here -->** with the following template syntax:

```

<v-card>

  <form @submit.prevent="onSubmit">

```

```
<v-card-title>

  <span class="headline">Create New Tour
  Package</span>

</v-card-title>

<v-card-text>

  <v-container>

    <v-row>

      <v-col cols="12">

        <v-text-field required label="Name"
          v-model="bodyRequest.name"
        ></v-text-field>

      </v-col>

      <!-- Removed for brevity -->

      <!-- Please see the full code in GitHub -->

    </v-row>

  </v-container>

  <small>*indicates required field</small>

</v-card-text>

<!-- Insert <v-card-actions> here -->

</form>
```

```
</v-card>
```

The preceding form lets the user add a new tour package. The form has the following fields for tour packages – **name**, **whatToExpect**, **mapLocation**, **price**, **duration**, and **instant confirmation**.

Let's also add some buttons:

```
<v-card-actions>
```

```
<v-spacer></v-spacer>
```

```
<v-btn color="blue darken-1" text @click="dialog =  
false"> Close </v-btn>
```

```
<v-btn color="blue darken-1" text @click="dialog =  
false" type="submit">
```

```
  Save
```

```
</v-btn>
```

```
</v-card-actions>
```

**Close v-btn** and **Save v-btn** both have functionalities that close the dialog, but **Save v-btn** is the **submit** type, which means it triggers **@submit** of **for**. **@submit** then triggers the **onSubmit** method from the **methods** block.

Next we update the **DefaultContent.vue** component again:

```
import AddTourPackageForm from  
"@/components/AddTourPackageForm";
```

We are bringing in **AddTourPackageForm**. Let's then register **AddTourPackageForm** in the **components** block like so:

```
components: { TourListsCard, AddTourListForm,
```



TourPackagesCard, AddTourPackageForm, },

We can now insert **AddTourPackageForm** in the **template** syntax section.

Insert the **AddTourPackageForm** component below the **TourPackagesCard** component. Don't forget to pass **tourListId** in the **tourListId** props like so:

```
<div v-if="showPackages">  
  
  <TourPackagesCard />  
  
  <AddTourPackageForm :tourListId="tourListId" />  
  
</div>
```

**AddTourPackageForm** is now in the **template** syntax and ready to use.

If an ESLint error is preventing the Vue.js app from refreshing, restart the server.

OK, so let's check out the browser to see the **ADD NEW TOUR PACKAGE** button, as shown in the following screenshot:

 Figure 15.7 – ADD NEW TOUR PACKAGE button

Figure 15.7 – ADD NEW TOUR PACKAGE button

Now, click the new button to see whether the dialog form will appear, as shown in the following screenshot:

 Figure 15.8 – Create New Tour Package form

Figure 15.8 – Create New Tour Package form

*Figure 15.8* shows the form for creating a new tour package. Try to fill out the form and hit **SAVE**.

Now we can move on to the next section.

# Updating a tour package using Axios and Vuex

We are in the last section of this chapter, and this section is about updating a tour package using a *PUT HTTP* request and Vuex. We are going to create another form for editing a tour package. So let's start.

The first task is to update **services.js** in the **store/tour** directory:

```
export async function putTourPackageAxios(tourList) {  
  return await api.put(`TourPackages/${tourList.id}`,  
    tourList);  
}
```

We are adding a new service that sends a *PUT* request to our backend server.

Next, we update **types.js** of the **store/tour** folder again:

```
export const UPDATE_TOUR_PACKAGE =  
"UPDATE_TOUR_PACKAGE";
```

The preceding code is a new action type for updating a tour package.

Next, we also update **actions.js** in the **store/tour** folder:

```
import { getTourListsAxios, deleteTourListAxios,  
  postTourListAxios, deleteTourPackageAxios,  
  postTourPackageAxios, putTourPackageAxios,  
} from "@store/tour/services";
```

In the preceding code, we are importing the **putTourPackageAxios** service.

The following code is a new asynchronous Vuex action. The action will be the last Vuex action in the **actions.js** file:

```
export async function updateTourPackageAction({ commit }, payload) {  
  commit(types.LOADING_TOUR, true);  
  
  try {  
    await putTourPackageAxios(payload);  
    commit(types.UPDATE_TOUR_PACKAGE, payload);  
  } catch (e) {  
    alert(e);  
    console.log(e);  
  }  
  
  commit(types.LOADING_TOUR, false);  
}
```

The newly added action called **updateTourPackageAction** enables the loader, sends a *PUT* request with a payload to our backend, and commits **UPDATE\_TOUR\_PACKAGE**, passing the payload to also update the UI, and then disables the load in the UI.

Then we also update **mutations.js** in **store/tour** with the following code:

```
[types.UPDATE_TOUR_PACKAGE](state, payload) {  
  const packageIndex =
```

```

state.packagesOfSelectedCity.findIndex(
  (pl) => pl.id === payload.id);
state.packagesOfSelectedCity[packageIndex] = payload;
const listIndex = state.lists.findIndex(
  (l) => l.id === state.packagesOfSelectedCity.listId);
state.lists[listIndex] = state.packagesOfSelectedCity;
},

```

We are adding a new mutation that finds the index of the specific tour package. After finding the index, we locate the particular tour package via the index and then mutate it with the payload, the updated tour package.

We also update **tourList** by again finding the index of **selectedCity** and then editing it. The mentioned logic keeps all the affected objects and arrays in sync.

Now, create a Vue component named **UpdateTourPackageForm.vue** in the **components** directory and add the following script to the new Vue component:

```

<script>

import { mapActions } from "vuex";

export default {
  name: "UpdateTourPackageForm",
  props: {
    bodyRequest: {
      type: Object,

```

```
    required: true,

    default: { listId: 0, name: "missing name",

      whatToExpect: "missing what to expect",

      mapLocation: "missing map location",

      price: 0, duration: 0, instantConfirmation: true,

    },

  },

},

methods: {

  ...mapActions("tourModule",

    ["updateTourPackageAction"]),

  onSubmit() {

    this.updateTourPackageAction(this.bodyRequest); //

    fyi, you might not see the results right away because

    of the cache.

  },

},

data: () => ({

  dialog: false,

  currencies: ["USD", "NOK"],
```

```

    currencyValues: [0, 1],
    durations: [1, 2, 3, 4, 5, 6, 7, 8],
    durationValue: 1,
  )),
};

</script>

```

The preceding script provides a **bodyRequest** props, uses **mapActions**, has the **onSubmit** method that invokes **updateTourPackageAction**, and a set of local states – namely **dialog**, **currencies**, **currencyValues**, **durations**, and **durationValue**.

Now, let's add the **template** syntax using the following code:

```

<template>

  <v-row justify="center">

    <v-dialog v-model="dialog" persistent max-
width="600px">

      <template v-slot:activator="{ on, attrs }">

        <v-icon class="mr-3" v-bind="attrs" v-on="on">

          mdi-clipboard-edit-outline

        </v-icon>

      </template>

    <v-card>

```

```
<form @submit.prevent="onSubmit">

  <v-card-title>

    <span class="headline">Update Tour
    Package</span>

  </v-card-title>

  <v-card-text>

    <!-- Insert <v-container> Here -->

    <small>*indicates required field</small>

  </v-card-text>

  <v-card-actions>

    <v-spacer></v-spacer>

    <v-btn text color="blue darken-1"
    @click="dialog = false">
      Close
    </v-btn>

    <v-btn text color="blue darken-1"
    @click="dialog = false" type="submit">
      Update
    </v-btn>

  </v-card-actions>
```

```
    </form>

  </v-card>

</v-dialog>

</v-row>

</template>
```

The preceding block of template syntax is the UI for **UpdateTourPackageForm.vue**. The template uses the **v-card** component, which looks neat, and the form that has the **@submit** directive event to invoke or trigger the **onSubmit** function.

Next, we replace **<!--Insert <v-container> Here -->** with the following template syntax:

```
<v-container>

  <v-row>

    <v-col cols="12">

      <v-text-field required label="Name"

        v-model="bodyRequest.name"

      ></v-text-field>

    </v-col>

    <v-col cols="12">

      <v-textarea

        label="What to expect"

        v-model="bodyRequest.whatToExpect" required
```



```

        ></v-textarea>

    </v-col>

    <!-- Removed for brevity -->

    <!-- Please see the full code in GitHub -->

</v-col>

</v-row>

</v-container>

```

The preceding template is responsible for providing the fields needed in the form. These tour package fields are as follows: **name**, **whatToExpect**, **mapLocation**, **price**, **duration**, and **instantConfirmation**. You will notice that the fields are using the **bodyRequest** props because those props will be **tourPackage** the user will be editing.

Now, let's update **TourPackagesCard.vue** with the following code:

```

import UpdateTourPackageForm from
"@/components/UpdateTourPackageForm";

```

We are importing **UpdateTourPackageForm** into **TourPackagesCard.vue**. We will then register **UpdateTourPackageForm** in the **components** block like so:

```

components: { UpdateTourPackageForm },

```

We can now use **UpdateTourPackageForm** in the template syntax area.

Update the **<v-list-item-action>** component in the **template** section with the new following code:

```

<v-list-item-action>

    <UpdateTourPackageForm :bodyRequest="tourPackage" />

```

```
<v-icon @click="removeTourPackage(tourPackage.id)">
```

```
mdi-delete-outline
```

```
</v-icon>
```

```
</v-list-item-action>
```

The preceding block of template code includes the UI for updating a tour package.

Now, before testing out the new form, let's update our ASP.NET Core Web API first. Go to **UpdateTourPackageCommandValidator.cs** of the **Travel.Application.TourPackages.Commands.UpdateTourPackage** namespace. Then, edit the maximum length rule for **Name** like so:

```
public UpdateTourPackageCommandValidator(IApplicationDbContext  
context)
```

```
{
```

```
    _context = context;
```

```
    RuleFor(v => v.Name)
```

```
        .NotEmpty().WithMessage("Name is required.")
```

```
        .MaxLength(200).WithMessage("Name must not
```

```
        exceed 200 characters.");
```

```
}
```

The name should only be **200** characters. However, this is not enough. We will also create a validation in the frontend, but this will be covered in [Chapter 17](#), *Input Validations in Forms*.

Now rerun the app to see whether the application is still running without any problems:


 Figure 15.9 – Pencil pad icon

Figure 15.9 – Pencil pad icon

*Figure 15.9* shows a piece of a tour package. Try to update a tour package by clicking the pencil pad icon, which will open up the **UpdateTourPackageForm** component.

Now, let's summarize what you have learned in this long chapter about HTTP requests, Vuex state management, and forms from Vuetify.

## Summary

And you have finished. Good work! Now let's break down what you've gained from this chapter. You have learned how to do CRUD using Axios, an HTTP client library. You have learned how to write asynchronous and non-asynchronous actions in Vuex. You have also learned how to use **mapActions** and **mapGetters** in components. And lastly, you have learned how to use Vuetify to build attractive forms and buttons.

In the next chapter, we will build the login and sign-up forms for our users. We will also enable authorization in our backend so that only authenticated users can send CRUD operations to our ASP.NET Core Web API.

## Chapter 16: Adding Authentication in Vue.js

In this chapter, you will learn how to use the Vue.js app's authentication process by building a registration form and login form. You will be creating a service for auth and router auth guards to protect any routers or pages in your Vue.js app. We will also be using a **JSON Web Token (JWT)** when requesting a protected resource. This chapter is an exciting chapter, so get let's started!

We will be covering the following topics:

- Setting up Vuex for authentication
- Writing an auth guard
- HTTP interceptors
- Auto login

### Technical requirements

You will need Visual Studio Code to complete this chapter.

The following is a link to the GitHub repository for this chapter:

<https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter16/>.

### Setting up Vuex for authentication

The first section of this chapter is all about creating another module in our Vuex state management, which will be for our app's auth module. This section will also help you learn how to add a new module to our Vuex store. This module will require new action types, new actions, a new state, and new mutations.

Before we start implementing this new module for our auth, we will need help from the the **jsonwebtoken** npm library. So, let's download the **jsonwebtoken** library.

Run the following **npm** command:

```
npm i jsonwebtoken
```

The preceding command will install the **jsonwebtoken** package in our application.

Then, create a new folder in the **src** directory and name it **auth**.

Now, create a JavaScript file called **auth.service.js** inside the **auth** directory. Add the following code to the **auth.service.js** file:

```
import api from "@/api/api-v1-config";  
  
export async function loginUserAxios(login) {
```

```
    return await api.post("Users/auth", login);
}
```

In the preceding code, we created a service that takes a login object and sends a POST request to the **/api/v1/Users/auth** endpoint.

Next, we must create a folder in the **store** folder and name it **auth**.

Now, create a JavaScript file called **types.js** in the **store/auth** folder and add the following code:

```
export const LOGIN_USER = "LOGIN_USER";
```

The preceding code is an action type for logging in.

Next, we must create another JavaScript file called **actions.js** in the **store/auth** folder and add the following code:

```
import * as types from "../types";
import { loginUserAxios } from "@auth/auth.service";
export async function loginUserAction({ commit }, payload) {
  try {
    const { data } = await loginUserAxios(payload);
    commit(types.LOGIN_USER, data.token);
  } catch (e) {}
}
```

Here, we imported the **auth** action types and **loginUserAxios** service and used them inside a newly created action called **loginUserAction**. The **loginUserAction** action requires a payload, and that payload will contain the username and email of the user trying to sign in.

Next, we must create another JavaScript file called **state.js** in the **store/auth** folder and add the following code to it:

```
const state = {
  signInState: {
    email: "",
    exp: Date.now(),
    sub: "",
    token: null,
  },
};
export default state;
```

The state of our **auth** module or namespace will have a **signInState** model with **email**, **exp**, **sub**, and **token** properties.

Now, create another JavaScript file called **mutations.js** in the **store/auth** folder and add the following code:

```
import * as types from "../types";
import * as jwt from "jsonwebtoken";
const mutations = {
  [types.LOGIN_USER](state, token) {
    state.signInState.token = token;

    const loginClaim = jwt.decode(token);
    claimToState(state, loginClaim);
    localStorage.setItem("token", token);
  },
};
export default mutations;

function claimToState(state, claim) {
  state.signInState.sub = claim.sub;
  state.signInState.email = claim.email;
  state.signInState.exp = claim.exp;
}
```

Here, we are importing the types and **jsonwebtoken** into the **mutations.js** file. We are then using the imported types and **jsonwebtoken** inside the **mutation** function. The added mutation takes a token, and then we store it in **signInState.token**. After this, we decode the token and store the decoded token in **loginClaim**.

You will notice that there's a function named **claimToState**. This function maps the claim's **sub**, **mail**, and **exp** to the **signInState**'s **sub**, **mail**, and **exp**.

After storing the **sub**, **email**, and **exp** properties of the claim in **signInState**, we must save the token in the browser's local storage. The keyname for the stored token value is **token**.

Next, we must create another JavaScript file called **getters.js** in the **store/auth** folder and then add the following code to it:

```
const getters = {
  email: (state) => {
    return state.signInState.email;
  }
}
```

```

    },
    isAuthenticated: (state) => {
        return state.signInState.token;
    },
};
export default getters;

```

In the preceding code, we are creating a getter for **signInState.email**, and we are simply going to name it **email**.

Now, create an index file called **index.js** for the **store/auth** module:

```

import state from "../state";
import getters from "../getters";
import mutations from "../mutations";
import * as actions from "../actions";
export default {
    namespaced: true,
    getters,
    mutations,
    actions,
    state,
};

```

If the preceding code looks familiar to you, this is because we also wrote it in the **tour** module. Importing and exporting the state, getters, mutations, actions, and **namespaced** property set to **true** is what you will repeatedly be writing whenever you are creating a new module in the store.

The **namespaced** property set to **true** requires you to write the module's name whenever you are spreading **mapActions** or **mapGetters**. Take a look at the following code example, which is using a **foo** module:

```

...mapActions("fooModule", ["selectAction"]),

```

You will notice that the first argument is the namespace, and then the second argument is for the actions. The purpose of the namespace in the preceding code is to avoid name collisions or clashes if you have a large application and are using many modules that have the same names for actions.

Now, we must update the **index.js** file of the **store** folder:

```

import authModule from "../auth";

```

We are bringing **authModule** into the **index.js** file of the store and then using **authModule** in the **modules** object:

```
export default new Vuex.Store({
  modules: {
    tourModule,
    authModule,
  },
  plugins,
});
```

Now, **authModule** is integrated into our Vuex state management implementation.

Good! In the next section, we will learn how to protect pages of the Vue app.

## Writing an auth guard

An **auth guard** is a piece of middleware in our router. We can write functions and trigger them in routes. An excellent example of a function that we can put in a router is a function that helps with authentication. The function would then check whether the user of the application is authenticated or not. We must then show specific pages to users that are authenticated. So, let's start writing our auth guard.

Create a JavaScript file called **auth.guard.js** in the **auth** folder and add the following code:

```
import store from "@/store";

export const authGuard = (to, from, next) => {
  console.log("authGuard");

  const authRequired = to.matched.some((record) =>
    record.meta.requiresAuth);

  if (authRequired) {
    if (store.getters["authModule/isAuthenticated"]) {
      next();
      return;
    }
    next("/login");
  }
  next();
};
```



In the preceding code, we are defining our **authGuard**. Here, **authGuard** takes **to**, **from**, and **next** properties. We are also checking if a route has been tagged with **true**. We can do this by using the **authRequired** meta property from the **route** object.

If **auth** is required, we must check if the **isAuthenticated** getter returns **true**. If it does, we must go to the next pipe of the middleware. If it doesn't, we must go to the login page.

Now, let's update our router in the **router/index.js** file so that we can use the **authGuard** middleware we just wrote:

```
import { authGuard } from "@auth/auth.guard";
```

Let's bring in our **authGuard**.

Now, write the following code and put it before the export default router line of code inside the **index.js** file of the router folder, like so:

```
router.beforeEach((to, from, next) => {  
  console.log("router.beforeEach");  
  authGuard(to, from, next);  
});  
export default router;
```

**router.beforeEach** is a global before guard and will trigger whenever we navigate inside our app. We included a **console.log** so that you can see that **beforeEach** runs every time we navigate.

Now, create a new Vue component called **Login.vue** inside the **auth/views** folder and add the following script:

```
<script>  
import { mapActions } from "vuex";  
import router from "@router";  
export default {  
  name: "Login",  
  data: () => ({  
    dialog: true,  
    tab: null,  
    login: {  
      email: "",  
      password: "",  
    },  
  }),  
}
```

```

methods: {
  ...mapActions("authModule", ["loginUserAction"]),
  onSubmit() {
    this.loginUserAction(this.login).then(() => {
      this.$router.push({ path: "/admin-dashboard" });
    });
  },
  navigateHome() {
    router.push("/");
  },
},
};
</script>

```

Here, we are declaring some states in the script. We have a **dialog**, a **tab**, and a **login** object. We are then importing **mapActions** and **router** and using them in the **methods** block.

Next, we created two methods, namely **onSubmit** and **navigateHome**. **onSubmit** invokes **loginUserAction**, and if **loginUserAction** is successful, we must navigate to the **admin-dashboard** path. Meanwhile, **navigateHome** will simply send us to the home page.

Now, let's add the **template** syntax for the **Login.vue** component. Then, write the following code:

```

<template>
  <!-- https://vuetifyjs.com/en/styles/colors/#material-
  colors-->
  <v-app>
    <v-dialog
      v-model="dialog" persistent
      max-width="600px" min-width="360px"
      @click:outside="navigateHome"
    >
      <div>
        <v-tabs
          show-arrows background-color="pink accent-4"
          icons-and-text dark grow
        >

```

```

        <v-tabs-slider color="pink darken-4"></v-tabs-
        slider>
        <v-tab>
            <v-icon large>mdi-login</v-icon>
            <div>Login</div>
        </v-tab>
        <v-tab>
            <v-icon large>mdi-account-box-outline</v-icon>
            <div>Register</div>
        </v-tab>
        <!-- Login <v-tab-item> here -->
        <!-- Register <v-tab-item> here -->
    </v-tabs>
</div>
</v-dialog>
</v-app>
</template>

```

The **template** syntax here is a **dialog** component that renders two tabs – a tab for logging in and another tab for registering.

Now, let's replace the comment stating `<!-- Login <v-tab-item> here -->` with the following template syntax:

```
<v-tab-item>

<v-card class="px-4">

  <v-card-text>

    <form @submit.prevent="onSubmit">

      <v-row>

        <v-col cols="12">

          <v-text-field label="E-mail" v-
            model="login.email"></v-text-field>

        </v-col>

        <v-col cols="12">

          <v-text-field

            type="password" label="Password"
```

```

        hint="At least 8 characters"
        v-model="login.password" counter
    ></v-text-field>
</v-col>
<v-col class="d-flex" cols="12" sm="6" xsm="12">
</v-col>
<v-spacer></v-spacer>
<v-col class="d-flex" cols="12" sm="3" xsm="12"
align-end>
    <v-btn :disabled="false" color="primary"
    type="submit"
    >Login</v-btn>
</v-col>
</v-row>
</form>
</v-card-text>
</v-card>
</v-tab-item>

```

The preceding **login** tab provides a **text** field for the email and a **text** field for the password. The fields are wrapped with a form that triggers the **onSubmit** method.

Now, let's replace the comment stating **<!-- Register <v-tab-item> here -->** with the following template syntax:

```

<v-tab-item>
// ... please go to the repo for the complete code
<v-row>
    <v-col cols="12" sm="6" md="6">
        <v-text-field
            label="First Name"
            maxLength="20" required
        ></v-text-field>
    </v-col>
    <v-col cols="12" sm="6" md="6">
        <v-text-field

```

```

        label="Last Name"
        maxLength="20" required
    ></v-text-field>
</v-col>
<v-col cols="12">
    <v-text-field label="E-mail" required></v-text-
    field>
</v-col>
<v-col cols="12">
    <v-text-field
        counter label="Password"
        hint="At least 8 characters"
    ></v-text-field>
</v-col>
<v-col cols="12">
    <v-text-field
        block counter
        label="Confirm Password"
    ></v-text-field>
</v-col>
<v-spacer></v-spacer>
<v-col class="d-flex ml-auto" cols="12" sm="3"
xsm="12">
    <v-btn :disabled="false"
        color="primary">Register</v-btn>
</v-col>
</v-row>

```

...

</v-tab-item>

The **register** tab is supposed to provide text fields for **firstName**, **lastName**, **password**, and **confirmPassword**. As you may recall, we don't have an API for creating a new user in ASP.NET Core. We hardcoded our user object in the **UserService.cs** file of our ASP.NET Core Web API project:

```
new User
```

```

{
  Id = 1,
  FirstName = "Yourname",
  LastName = "Yoursurname",
  Email = "yoursuperhero@gmail.com",
  Password = "Pass123!"
}

```

So, the login details we can use are the values of the email and password we provided in the preceding code.

Now, let's update the route and a meta information of the **login** path and the **about** path, like so:

```

{
  path: "/login",
  component: () => import("@auth/views/Login"),
  meta: {
    requiresAuth: false,
  },
},

```

The preceding code adds the meta information to the route of **login**. Now, let's do the same thing to the route of **about**:

```

{
  path: "/about",
  name: "About",
  component: () => import("@views/Main/About"),
  meta: {
    requiresAuth: false,
  },
},

```

The preceding code adds the meta information to the route of **about**. **requiresAuth** will be different for the path of **admin-dashboard**, though:

```

{
  path: "/admin-dashboard",
  component: () => import("@views/AdminDashboard"),
  meta: {

```

```

      requiresAuth: true,
    },
  }

```

Here, we are setting **requiresAuth** to **true**. This means that **admin-dashboard** and the rest of its pages can only be accessed by authenticated web users.

Next, we must update the **NavigationBar.vue** file of the **components** directory:

```
import { mapGetters } from "vuex";
```

Let's import **mapGetter** from Vuex.

Now, let's use **mapGetters** with the **authModule** key as our namespace in **mapGetters**, like so:

```

computed: {
  ...mapGetters("authModule", {
    isAuthenticated: "isAuthenticated",
    email: "email",
  }),
},

```

Now, we must map **isAuthenticated** and **email** to the local states that automatically get created for us.

Now, write a **v-btn** component from Vuetify, like so:

```

<v-btn
  v-if="isAuthenticated"
  color="primary"
  outlined
  :to="{ path: '/admin-dashboard' }"
>
  <span class="menu">Dashboard</span>
</v-btn>
<v-btn v-else color="primary" outlined :to="{ path:
'/login' }">
  <span class="menu">Login</span>
</v-btn>

```

The **Dashboard** menu button is only visible to authenticated users, while the **Login** menu button is visible to unauthenticated users.

Before we start sending requests to the backend, let's put the **[Authorize]** attribute back inside **ApiController.cs** with a namespace of **Travel.WebApi.Controllers.v1**:

```
[Authorize]
```

We are commenting out the **Authorize** attribute here so that the resources of the ASP.NET Core Web API are protected.

Let's rerun the server and select the **LOGIN** menu at the top of the screen:

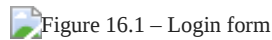


Figure 16.1 – Login form

The preceding screenshot shows the login form after clicking the **Login** menu. Use the following login credentials for authentication:

- **Email:** yoursuperhero@gmail.com
- **Password:** Pass123!

You should be able to log in, but an error alert prompt will appear. Open your browser's devtools and go to the **Network | Preview** tab:

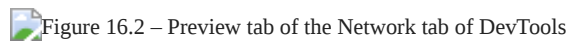


Figure 16.2 – Preview tab of the Network tab of DevTools

The preceding screenshot shows that the login was successful, but that the method that sends a GET request to fetch the tour lists collection failed. The fetch request failed because the ASP.NET Core Web API didn't find any JWT token in the header of our request. Hence, we are not authorized.

We will fix the JWT token in the header in the next section. But before that, let's check out the local storage of our browser to see if the Vue app can save the JWT token in local storage.

The successful login values, along with the token response, should be stored in **Local Storage**:



Figure 16.3 – Saved token in Local Storage

The preceding screenshot shows that the token is saved in the **Local Storage** area of the browser. We will use this token in our request to tell the ASP.NET Core Web API that we are authorized to read and write in the resource. We will do this in the next section.

## HTTP interceptor

What is an HTTP interceptor? An HTTP interceptor is a piece of functionality that intercepts incoming HTTP responses and outgoing HTTP requests. We can automatically change or modify the



header of our requests before sending them to the endpoints of web services. Luckily, Axios has HTTP interceptor interfaces readily available for developers to use. So, let's get started!

The first thing we must do is update the **auth.service.js** file of the **src/auth** folder with the following code:

```
const key = "token";
```

The token's key name is simply **key**.

We'll also be adding two functions to the **auth.service.js** file:

```
export function getToken() {  
  return localStorage.getItem(key);  
}  
  
export function logOut() {  
  localStorage.clear();  
  window.location = "/login";  
}
```

The **getToken** function gets the token's value from the local storage, while the **logout** function clears out any stored values in the local storage and then redirects the users to the login page.

Now create a new JavaScript file called **interceptors.js** in the **api** folder and write the following code in it:

```
import { getToken } from "@auth/auth.service";  
  
export function interceptorsInit(axiosInstance) {  
  axiosInstance.interceptors.request.use(function(options) {  
    const jwtToken = getToken();  
    if (jwtToken) {  
      options.headers["Authorization"] = `Bearer ${jwtToken}`;  
    }  
    return options;  
  });  
  
  axiosInstance.interceptors.response.use(  
    (response) => response,  
    (error) =>  
      Promise.reject(  
        (error.response && error.response.data) || "Something  
        went wrong"  
      )  
  );  
}
```

```

    )
  );
  return axiosInstance;
}

```

Here, we are bringing in the **getToken** service and creating a function called **interceptorsInit**. The function takes an instance of Axios and then uses it to trigger **getToken**. If the token is present in local storage, we must add an authorization key to the header with the **Bearer** space token value. Pay attention that the value of the token is an interpolation of the **Bearer** string in the token.

We are also adding an interceptor for any response. You can add any modification to the response if you wish.

Now, let's update the **api-v1-config.js** file of the **api** folder by importing **interceptorsInit**:

```
import { interceptorsInit } from "@api/interceptors";
```

We can now use **interceptorsInit**. Let's use **interceptorsInit** by passing the instance of Axios, like so:

```
api = interceptorsInit(api);
```

The **return** value of **interceptorsInit** is then stored in the same instance, which modifies Axios:

Now rerun the application. Log out and log in again to see if we can fetch the Tour Lists right after logging in.

 Figure 16.4 – Tour Lists status code 200

Figure 16.4 – Tour Lists status code 200

The preceding screenshot shows that the **getTourListsAxios** service can get the Tour List without any errors. Here, you can see that the response code is **200** for the **auth** endpoint and **200** for the **TourLists** endpoint.

Now, let's try to log out and see if the token will be removed from the browser.

Open the dev tools of your browser and go to the application tab, like so:

 Figure 16.5 – Empty local storage

Figure 16.5 – Empty local storage

The preceding screenshot shows that logging out removes the token from the browser.

At this point, we know that authentication using Vuex is working, the auth guard is working, and that the HTTP interceptor of the app is also working. Now, what if the user closes the browser and then returns to the web app at a later date?

Here, we must create a mechanism where logging in is automatic if a token in local storage is present and valid. We will build this feature, where the user doesn't have to enter credentials, in the next section.

## Auto login

To give our web users a user-friendly web app when logging in, we will create an auto login. The idea is that whenever a token is present in local storage, we will check if it's a valid token and then use it for authentication. By doing this, our web visitors don't need to reenter their login details, which is sometimes annoying for them.

To start, let's update the **auth.service.js** file of the **src/auth** folder and then import **jsonwebtoken**, like so:

```
import * as jwt from "jsonwebtoken";
```

We will use the JWT token later to decode the token.

Now, add two new functions to the **auth.service.js** file:

```
export function isTokenFromLocalStorageValid() {  
  const token = localStorage.getItem(key);  
  if (!token) {  
    return false;  
  }  
  const decoded = jwt.decode(token);  
  const expiresAt = decoded.exp * 1000;  
  const dateNow = Date.now();  
  return dateNow <= expiresAt;  
}  
  
export function getUserEmailFromToken() {  
  const token = localStorage.getItem(key);  
  if (!token) return false;  
  const decoded = jwt.decode(token);  
  return decoded.email;  
}
```

**isTokenFromLocalStorageValid** gets the token, decodes it, and then checks its expiration, while **getUserEmailFromToken** returns the email after getting the token from local storage and decoding it.

Let's update the route of the app:

```
import { isTokenFromLocalStorageValid } from "@auth/auth.service";
```

Here, we are importing the **isTokenFromLocalStorageValid** service.

Now, we are going to add a guard hook called **beforeEnter** to the login route. **beforeEnter** guards the route where we put it. We are going to run some logic before the app reads the **login** route object.

Now, update the **login** route object, like so:

```
{
  path: "/login",
  component: () => import("@auth/views/Login"),
  meta: {
    requiresAuth: false,
  },
  beforeEnter: (to, from, next) => {
    const valid = isTokenFromLocalStorageValid();
    console.log("VALID:", valid);
    if (valid) {
      next("/continue-as");
    } else {
      next();
    }
  },
},
{
  path: "/continue-as",
  component: () => import("@auth/views/ContinueAs"),
  meta: {
    requiresAuth: false,
  },
},
```

Here, we are adding a new page called **Continue As** to our routes. We will use the **Continue As** page to give our web visitors the option to use the account from the token or to log out. We will then use the **Continue As** page to reroute a user if the token in local storage is no longer valid.

Now, we will save a decoded token in our global state if the token is still valid. Let's update the **types.js** file of the **store/auth** folder and add the following code to it:

```
export const LOCAL_STORAGE_TOKEN_LOG_IN = "LOCAL_STORAGE_TOKEN_LOG_IN";
```

The preceding code is a new action type we are adding to the file. Now, let's update the **actions.js** file of the **store/auth** folder and add the following code to it:

```
import {
  loginUserAxios,
  isTokenFromLocalStorageValid,
  getToken
} from "@auth/auth.service";
```

Here, we are bringing in the **isTokenFromLocalStorageValid** service and the **getToken** service.

Now, let's create a new Vuex action for our **auth** module:

```
export function useLocalStorageTokenToSignInAction({ commit }) {
  if (!isTokenFromLocalStorageValid()) {
    return;
  }
  const token = getToken();
  commit(types.LOCAL_STORAGE_TOKEN_LOG_IN, token);
}
```

This new action checks the validity of the token and then commits the token using the **LOCAL\_STORAGE\_TOKEN\_LOG\_IN** action type.

Now, let's update the **mutations.js** file of the **store/auth** folder:

```
[types.LOCAL_STORAGE_TOKEN_LOG_IN](state, token) {
  state.signInState.token = token;
  const loginClaim = jwt.decode(token);
  claimToState(state, loginClaim);
},
```

The new mutation stores the token in **signInState.token**, decodes the token, and then maps the information of the decoded token to the other properties of **signInState**.

Next, we must update the **auth.guard.js** file of the **src/auth** folder with the following code:

```
import { isTokenFromLocalStorageValid } from "../auth.service";
```

```
if (authRequired) {
```

```

    if (store.getters["authModule/isAuthenticated"]) {
      next();
      return;
    } else if (isTokenFromLocalStorageValid()) {
      next();
      return;
    }
    next("/login");
  }
}

```

Here, we are bringing in the **isTokenFromLocalStorageValid** service and using it to validate a token inside **authGuard**.

Now, we can create the **ContinueAs.vue** component, which is the page we recently added to our route. Write the following script for the **ContinueAs** page:

```

<script>
import { getUserEmailFromToken, logout } from "@/auth/auth.service";
export default {
  name: "ContinueAs",
  data: () => ({
    email: getUserEmailFromToken(),
  }),
  methods: {
    handleLogout() {
      logout();
    },
    onSubmit() {
      this.$router.push({ path: "/admin-dashboard" });
    },
  },
};
</script>

```

The **ContinueAs** page has an email state that gets its value from the **getUserEmailFromToken** function. The **ContinueAs** page also has a method and **onSubmit**. The **handleLogout** method logs the user out, while the **onSubmit** method redirects the user to the Admin Dashboard page.

Now, for the **template** syntax of the **ContinueAs** page, copy the following code:

```
<template>
  <div class="container" v-if="email">
    <div class="text-h4 my-5">Do you like to continue as {{
      email }}? </div>
    <v-btn @click="onSubmit" color="primary" class="mr-
      4">Yes</v-btn>
    <v-btn @click="handleLogout()" outlined color="primary">
      No thanks, I'd like to log out</v-btn>
  </div>
  <div v-else class="text-h4 m-10">
    Fancy meeting you here.
    <v-btn @click="handleLogout()" outlined color="primary"
      >Go to Home Page</v-btn>
  </div>
</template>
```

This template is a simple UI the gives our web users the option to proceed with using their accounts or not.

Now, let's update the **index.vue** file of the **views/AdminDashboard** folder with the following code for the **script** section:

```
import { mapActions, mapGetters } from "vuex";
```

Here, we are importing **mapActions** and **mapGetters** again. Next, we must create a method called **localStorageLogin** that will invoke **useLocalStorageTokenToSignInAction**:

```
methods: {
  ...mapActions("authModule",
    ["useLocalStorageTokenToSignInAction"]),
  localStorageLogin() {
    this.useLocalStorageTokenToSignInAction().then();
  }
},
computed: {
  ...mapGetters("authModule", {
    email: "email",
```

```

    })
  },
  mounted() {
    this.localStorageLogin();
  },

```

We are also bringing in the **email** getter from our auth module. Let's replace the hardcoded text email here with a state from our store:

```

<v-list-item-content>
  <v-list-item-subtitle>{{ email }}</v-list-item-subtitle>
</v-list-item-content>

```

The sidebar's email will now become dynamic based on the email address of the user.

Now, rerun the app, log in, and close the browser. Revisit the Vue app's URL.

You should see a question in heading format stating *Do you like to continue as*, like so:



Figure 16.6 – Auto login with the option to proceed or not

The preceding screenshot shows the **ContinueAs** page, along with a question and the user's email address. The Vue app redirected you to the **ContinueAs** page since your browser contains the stored token that it got from the ASP.NET Core Web API response when you logged in.

Click **YES** to proceed to the Admin **Dashboard**:



Figure 16.7 – Email address on the sidebar

The preceding screenshot shows the **yoursuperhero@gmail.com** email address on the sidebar, which means that the token's details have been saved in the global state store.

With that, we have finished protecting our Vue.js app by implementing a few techniques. There are still some missing parameters that we can use to harden the protection of our app.

If you are building a complex Vue app and only have a little time to finish your project, consider using an Identity as a Service to authenticate your users. There are several options for this, such as AWS Incognito, Azure AD B2C, CGP Identity Platform, Okta, and Auth0, which I've used in several of my projects because of their superb authentication SDK and the amount of documentation they have. The Identity as a Services I've mentioned will significantly speed up your development, as well help you identify the security measures you will need in your application.

So, let's wrap things up and see what you have learned in this chapter.



## Summary

Let's quickly summarize what you have learned in this chapter. First, you learned how to set up a Vuex store for authentication and practiced how to set up a module in the store, end to end. You then learned about auth guard, which protects the route. You also learned how to intercept outgoing HTTP requests, which helps automatically modify every header of your request. Lastly, you learned how to create a simple auto login for your users to improve their user experience.

In the next chapter, we will create some validations in our forms to give our users a better user experience when they're filling out forms.

## Section 4: Testing and Deployment

This section deals with real-world and trending testing and deployment of web applications to production. The following chapters are included in this section:

- [Chapter 17](#), *Input Validations in Forms*
- [Chapter 18](#), *Writing Integration Tests Using xUnit*
- [Chapter 19](#), *Automatic Deployment Using GitHub Actions and Azure*

## Chapter 17: Input Validations in Forms

In the previous chapter, we discussed protecting your pages and creating a login form and a registration form. However, we didn't write any validations that will prevent users from entering invalid inputs.

This chapter will add validations to our forms, which will improve the user experience whenever users use our forms. So let's start.

We will cover the following topics:

- Installing an input validation library
- Using validators in forms

### Technical requirements

You need Visual Studio Code to complete this chapter.

Here is the link to the finished repository of this chapter:

<https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter17/>.

### Installing an input validation library

A user-friendly validation informs the user why the field is not valid while typing the input. We can build it from scratch by writing a validator in every field of the form and making sure that the validators are reactive. The implementation is doable, but it will require a lot of time, and the cleanliness of the code will depend on the developer.

So why not use a library that validates us? Fortunately, there are several Vue.js libraries for validation, and in this section, we will use one of the top libraries.

So let's install an input validation library:

```
npm i vuelidate
```

The **vuelidate** library is a simple lightweight model-based validation that we can use for our Vue application.

Now, let's create a JavaScript file called **vuelidate.js** in the **src/plugins** folder and apply the following code:

```
import Vue from "vue";  
  
import Vuelidate from "vuelidate";  
  
Vue.use(Vuelidate);
```

The preceding code imports **Vue** and **Vuelidate** and then passes **Vuelidate** to the **Vue use** method.

Next, we update **main.js** with the following code:

```
import "../plugins/vuelidate";
```

Let's bring in **Vuelidate** from the file we just created. That's all.

Next, we create a folder called **validators**, and a JavaScript file called **index.js** inside the folder. It should be like **validators/index.js**:

```
import { required, email, minLength } from "vuelidate/lib/validators";
export default {
  login: {
    email: { required, email },
    password: { required, minLength: minLength(8) },
  },
};
```

We define our validators in a separate file for cohesiveness. We have a login object that has an email. The email is required and must be in an email format only. The login object also has a password object, which is also required and has a minimum of eight characters.

Simple and easy, right? Now let's use some validators.

## Using validators in forms

Using validators in forms is easy since we are using an **npm** library for validating forms, and we are not re-inventing the wheel.

Now, let's update the **Login.vue** page in the **auth/views** folder:

```
import validators from "@/validators";
```

Let's import **validators**, which we've created, and then use **validators** inside the **computed** block like so:

```
computed: {
  emailErrors() {
    const errors = [];
    if (!this.$v.login.email.$dirty) return errors;
    !this.$v.login.email.email && errors.push("Must be
    valid e-mail");
    !this.$v.login.email.required && errors.push("E-mail
```

```

        is required");
        return errors;
    },
    passwordErrors() {
        const errors = [];
        if (!this.$v.login.password.$dirty) return errors;
        !this.$v.login.password.required &&
        errors.push("Password is required");
        !this.$v.login.password.minLength &&
        errors.push("Minimum characters is 8");
        return errors;
    },
},
validations: {
    login: validators.login,
},
},

```

We have a **computed** method for email errors. **emailErrors** initializes an array of errors. It checks whether the **email** field is dirty, meaning the value of the field has changed. It also checks whether the **email** field is not valid. **emailErrors** also checks whether the email is empty.

The pattern is also the same as in the **passwordErrors** computed method. The dollar sign means that the properties are reactive to any changes.

And then, we declare a **validations** block of an object with the **login** property and assign **validators.login** to it.

The next step is to update the **v-text-field** component of the login field with the following code:

```

<v-text-field
  label="E-mail"
  v-model="login.email"
  @input="$v.login.email.$touch()"
  @blur="$v.login.email.$touch()"
  :error-messages="emailErrors"
></v-text-field>

```

We add the error checkers in the preceding code. The error checkers are reactive; they run as soon as we type something and lose focus on the fields. **:error-messages** is a directive from Vuetify that we

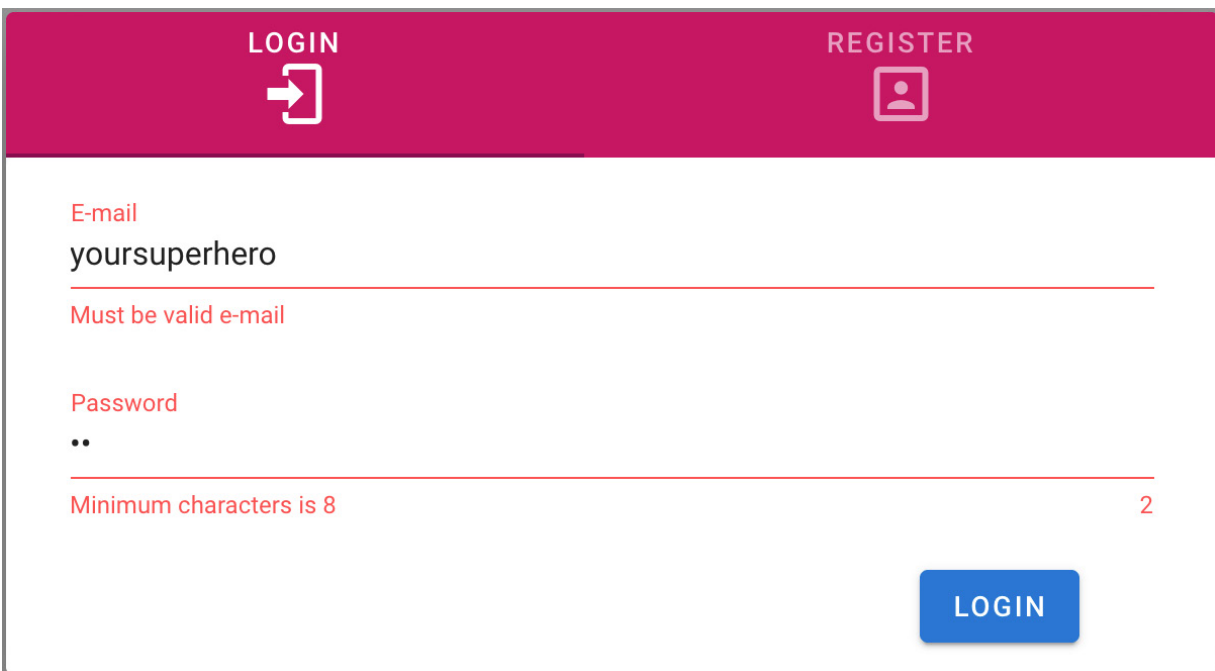
can use to assign the **emailErrors** computed method. That's neat isn't it?

Let's also update the **v-text-field** component for the login's password like so:

```
<v-text-field
  type="password"
  label="Password"
  hint="At least 8 characters"
  counter
  v-model="login.password"
  @input="$v.login.password.$touch()"
  @blur="$v.login.password.$touch()"
  :error-messages="passwordErrors"
></v-text-field>
```

The preceding code attaches the validators to **v-text-field** for the login password. We are using **:error-messages** with the value **passwordErrors**.

Now, run the application and then type **yoursuperhero** in the **Login** form without finishing the email, and then add two letters to the **Password** field:



The screenshot shows a login form with a magenta header. The header has two tabs: 'LOGIN' with a right arrow icon and 'REGISTER' with a person icon. The 'LOGIN' tab is active. Below the header, there are two input fields. The first field is labeled 'E-mail' and contains the text 'yoursuperhero'. Below this field, a red error message reads 'Must be valid e-mail'. The second field is labeled 'Password' and contains two dots '••'. Below this field, a red error message reads 'Minimum characters is 8'. To the right of this message is a red number '2'. At the bottom right of the form is a blue button with the text 'LOGIN'.

Figure 17.1 – Not valid

Figure 17.1 shows that **yoursuperhero** without **@domain.com** is not valid, as well as the fact that the two letters that are supposed to be eight letters/numbers are not valid. The field is turning from

black to red and red texts below the fields show that the validators are working.

That's good. We now have a proof of concept that the input validation library is working. Let's add more field validations to other forms.

The first step is to update the **index.js** file of the **validators** folder:

```
import {
  required,
  email,
  minLength,
  maxLength,
} from "vuelidate/lib/validators";
```

We add **maxLength** to the existing imports from **vuelidate/lib/validators**.

Then we update the exported object by adding **city**, **country**, and **about**:

```
export default {
  login: {
    email: { required, email },
    password: { required, minLength: minLength(8) },
  },
  city: {
    required,
    maxLength: maxLength(90),
  },
  country: {
    required,
  },
  about: {
    required,
  },
};
```

Most of the objects added are required fields in the following form that we are going to fix.

Now, update the **AddTourListForm.vue** component of the **components** folder with the following code:

```
import validators from "@/validators";
```

Let's import **validators** and then add a computed object with error checker methods and the **validations** object like so:

```
computed: {  
  cityErrors() {  
    const errors = [];  
    if (!this.$v.bodyRequest.city.$dirty) return errors;  
    !this.$v.bodyRequest.city.required &&  
    errors.push("City is required");  
    !this.$v.bodyRequest.city.maxLength &&  
    errors.push("Max length is 90");  
    return errors;  
  },  
  countryErrors() {  
    const errors = [];  
    if (!this.$v.bodyRequest.country.$dirty) return  
    errors;  
    !this.$v.bodyRequest.country.required &&  
    errors.push("Country is required");  
    // no need for max length because this is a dropdown  
    with options  
    return errors;  
  },  
  aboutErrors() {  
    const errors = [];  
    if (!this.$v.bodyRequest.about.$dirty) return errors;  
    !this.$v.bodyRequest.about.required &&  
    errors.push("About is required");  
    return errors;  
  },  
},  
validations: {  
  bodyRequest: {  
    // for brevity, please go to the github repo  
  },  
},
```



```
},
```

We have the **cityErrors** computed method, which adds an error if the **city** field is empty and the maximum length exceeded 90 characters. While the **country** field only needs a required validation because the maximum length wouldn't be needed in a drop-down list or autocomplete component, **aboutErrors**, on the other hand, only checks whether the field is empty.

Meanwhile, the **validations** object initializes a **bodyRequest** object with properties coming from the **validators/index.js** file.

Now, let's update the **template** syntax of the **AddTourListForm.vue** component by editing **v-text-field** of the **city** input:

```
<v-text-field
  label="City"
  v-model="bodyRequest.city"
  @input="$v.bodyRequest.city.$touch()"
  @blur="$v.bodyRequest.city.$touch()"
  :error-messages="cityErrors"
  required
></v-text-field>
```

We added **cityErrors** for the **:error-messages** directive. We updated the **@input** and **@blur** events.

Let's also edit the **v-autocomplete** component in the **template** syntax, like so:

```
<v-autocomplete
  :items="countryList"
  label="Country"
  v-model="bodyRequest.country"
  @input="$v.bodyRequest.country.$touch()"
  @blur="$v.bodyRequest.country.$touch()"
  :error-messages="countryErrors"
  required
></v-autocomplete>
```

The preceding code updates **@input**, **@blur**, and **:error-messages** with validators and error messages for country input.

Next, we update the field, actually, the text area, for the **about** input:

```
<v-textarea
  label="About"
```

```

v-model="bodyRequest.about"
@input="$v.bodyRequest.about.$touch()"
@blur="$v.bodyRequest.about.$touch()"
:error-messages="aboutErrors"
required
></v-textarea>

```

In the preceding code, let's use the computed errors and the error messages in the **v-textarea** component. The preceding code also attaches the validation to **textarea**.

And now, rerun the application.

Write a sentence with 100 characters in the **City** field, pick a country, click on the **About** text area, and then click again in the **City** field. This is to create a sample entry to our form like so:

**Create New Tour List**

**City**  
 Lorem Ipsum is simply dummy text o  
 Max length is 90

**Country**  
 Poland ▼

**About**  
 About is required

\*indicates required field

**CLOSE SAVE**

Figure 17.2 – New Tour List form validations

Figure 17.2 shows that the input validations in the **Create New Tour List** form are working. There is an error in the **City** field because the city's maximum characters are 90, while the **About** field is

empty.

## ACTIVITIES

*Let's test your skills in Vue.js by finishing up the Register Form's validation, the validation of **AddTourPackageForm.vue**, and the validation of **UpdateTourPackageForm.vue**.*

The link to Vuelidate is <https://vuelidate.js.org/>. There is another excellent input validation library that you might want to check out. The library is called **VeeValidate** (<https://vee-validate.logaretm.com/v4>). VeeValidate is what I can recommend if you are planning to use Vue.js version 3. The library is easy to use and has developer-friendly APIs. The library is also well integrated with Yup validation, which is a JavaScript schema builder for validation.

So let's summarize what you have learned.

## Summary

Let's quickly summarize what you have learned in this chapter to wrap it up. You have learned how to install Vuelidate and set it up in a Vue application. You have also learned how to use Vuelidate to create input validations in our forms, which provides better usability to our application's forms.

In the next chapter, you will see how important it is and, at the same time, how fun it is, to write tests in ASP.NET Core apps.

## Chapter 18: Writing Integration Tests Using xUnit

In the previous chapter, we wrote input field validations in our web app's forms to improve the **user experience (UX)** when filling out forms. Now, we are going to add tests to our web service.

Writing automated tests is a big topic—too big to be written in a single book and to discuss all the details you need to know. However, we are only going to scratch the surface and write practical tests. We will write a few unit tests and then write integration tests in our ASP.NET Core application.

We will cover the following topics in this chapter:

- Getting started with automated testing
- Installing **Microsoft SQL Server (MS SQL Server)** in a Docker container
- Understanding xUnit
- Understanding unit testing
- Understanding integration testing

### Technical requirements

Here is what you need to complete this chapter:

- Visual Studio 2019
- Visual Studio for Mac
- JetBrains Rider
- A Docker client
- The **Entity Framework Core (EF Core)** command-line interface (CLI)

Here's a link to the finished repository of this chapter: <https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter18/>

### Getting started with automated testing

Let's first define what automated testing is. **Automated testing** is a process of writing code to test the code in our project and then running those tests in an automated way—that's it.

Imagine this scenario: if you want to test a function manually, you have to launch your application in the browser, fill in the form to authenticate, and click some links to get to the function you want to test. You will then have to fill out all the fields in your form, hit the **Submit** button, and see the function's feedback or results in the browser. Imagine you have to repeat how you've manually tested your form and also try to test your form in different edge cases. That's crazy! Simply because of the

variations in how your user would use your form, this grows exponentially as you add more fields to your form.

As you can see, manual testing is very time-consuming. The workflow for manually testing your function may take several minutes every time and would be much worse if you had hundreds of functions in your application—imagine that.

So, what will you get out of automated testing?

## Benefits of automated testing

Here are the reasons why you should write tests in your application:

- **Test your code frequently, in less time:** You can test your application code often and in less time.
- **Catch bugs before deploying:** Catch bugs before deploying your application.
- **Deploy with confidence:** This is extremely important because it allows you to deploy your application with more confidence.
- **Code that works after refactoring:** The problem with refactoring code that has not been tested is that you have to manually test the code just to check that it is working again.

This process is excruciating if you are going to refactor hundreds of methods or functions because a) it's time-consuming, and b) as your application grows, you may forget about the parts that need to be tested. With automated tests, every time you refactor your code, you can easily run your tests to see if you broke something.

- **Focus more on the quality:** It helps if you focus more on the quality of the methods or functions you are writing, making sure that every method works with different inputs in varying circumstances.

Here's a fun fact: the mentioned benefits of automated testing help me sleep at night.

Going back to our topic, to set up our automated testing we will replace SQLite with MS SQL Server to write integration tests later. There are limitations to SQLite, which we are currently using in our ASP.NET Core application—limitations in terms of modeling, querying, and migrations. We have used SQLite in the previous chapters to avoid spending time setting up a database with a real database engine in Windows, Linux, or Mac OS X and to instead focus on the architecture of our application.

To start with, Windows users can choose between installing an MS SQL Server instance on their Windows machines or in Docker containers. In comparison, Linux and Mac OS X users will only have to use Docker to install a MS SQL Server instance.

The following screenshot shows **SQL Server Management Studio (SSMS)** for Windows machines:

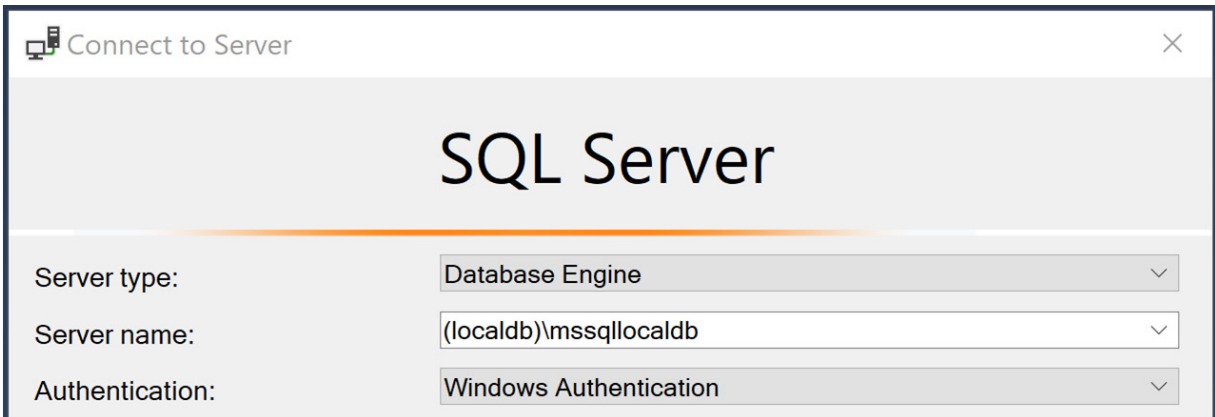


Figure 18.1 – SSMS

With that, here are two different connection strings, one for Windows and another one for Docker.

For MS SQL Server in Windows, here is the connection string:

```
"DefaultConnectionUsingWindows": "Server=(localdb)\\mssqllocaldb;Database=TravelDb;Trusted_Connection=True;MultipleActiveResultSets=true"
```

Replace the current value of the database connection string in **appsettings.json** with the preceding value.

For MS SQL Server in Docker, here is the connection string:

```
"DefaultConnectionUsingDocker": "Data Source=localhost,1433;Initial Catalog=TravelDb;User id=sa;Password=Pass123!;MultipleActiveResultSets=True"
```

Replace the current value of the database connection string in **appsettings.json** with the preceding value.

In the next section, we will install MS SQL Server in Docker.

## Installing MS SQL Server in a Docker container

Before we use **Docker** and install MS SQL Server in it, let's have a quick discussion about Docker. If you haven't tried Docker before, Docker is a **platform as a service (PaaS)** that provides **operating system (OS)**-level virtualization to deliver software installation in packages called **containers**.

I would suggest reading more about Docker and containers at

<https://www.docker.com/resources/what-container> because we will stay focused on automated testing topics in this chapter.

Here are the links for installing Docker:

- **Installing Docker Engine on Ubuntu:** <https://docs.docker.com/engine/install/ubuntu/>
- **Installing Docker Desktop on Mac:** <https://docs.docker.com/docker-for-mac/install/>

- Installing Docker Desktop on Windows: <https://docs.docker.com/docker-for-windows/install/>

After installing Docker on your Linux machine, MacBook, or Windows, you can install MS SQL Server in Docker. Here's a quick guide on how to do it: <https://devlinduldulao.pro/how-to-use-microsoft-sql-server-on-mac-for-development>.

It seems as though the guide is only for Mac users, but trust me—the installation of MS SQL Server in Docker from the guide is also applicable to Ubuntu and Windows users.

Next up is Azure Data Studio, a cross-platform database tool that is an alternative to SSMS.

The preceding link to the guide on using MS SQL Server on a Mac also provides details of how to use Azure Data Studio.

Next, after installing Docker, MS SQL Server, and Azure Data Studio, let's install the NuGet package of the EF SQL Server provider.

Install the **Microsoft.EntityFrameworkCore.SqlServer** NuGet package in the **Travel.Data** project and then update **services.AddDbContext** in the **DependencyInjection.cs** file of the **Travel.Data** project, with the following code:

```
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(config
        .GetConnectionString("DefaultConnection")));
```

The preceding code only replaces **UseSqlite** with **UseSqlServer**.

Next, we delete the **Migrations** directory of **Travel.Data** and create a new migration for the MS SQL Server database using the following **dotnet** command:

```
dotnet ef migrations add InitialCreate --startup-project ../../presentation/Travel.WebApi/
```

We are creating a new migration and syncing it with the MS SQL Server database that we will use later.

Next, we update the database by running the following command:

```
dotnet ef database update --startup-project ../../presentation/Travel.WebApi/
```

After running the preceding **dotnet ef** Core command, the MS SQL Server database should now be ready.

Let's create **DbContext** for seeding our data by creating a new C# file named **ApplicationDbContextSeed.cs** in the **Contexts** directory of the **Travel.Data** project and adding the following code:

```
namespace Travel.Data.Contexts
{
```

```

public static class ApplicationDbContextSeed
{
    public static async Task
    SeedSampleDataAsync(ApplicationDbContext context)
    {
        if (!context.TourLists.Any())
        {
            await context.TourLists.AddAsync(new TourList
            {
                City = "Oslo", About = "...",
                Country = "Norway",
                // Removed for brevity. See Github repo
            });
            await context.SaveChangesAsync();
        }
    }
}

```

You can get the sample **TourList** object's whole details with a **TourPackage** object from the finished repository of this chapter.

Now, let's create a database seeder in our **Program.cs** file by updating the code like so:

```

public static async Task<int> Main(string[] args)
{
    var name = Assembly.GetExecutingAssembly().GetName();

    Log.Logger = new LoggerConfiguration()
        .MinimumLevel.Debug()
        // Removed for brevity. See Github repo
        .WriteTo.Console().CreateLogger();

    // creating a host builder here.
}

```

The preceding code was for the configuration of our logger. We will write our database seeder with the following code:

```

try

```



```

{
    Log.Information("Starting host");
    var host = CreateHostBuilder(args).Build();
    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        try
        {
            // Removed for brevity. See Github repo
        }
        await host.RunAsync();
        return 0;
    }
    catch (Exception ex)
    {
        Log.Fatal(ex, "Host terminated unexpectedly");
        return 1;
    }
    finally
    {
        Log.CloseAndFlush();
    }
}

```

**await ApplicationDbContextSeed.SeedSampleDataAsync(context)** will seed the sample data in the MS SQL Server database once the application is started. Build and run the application to see if the application will still run without any problems.

We are now ready to move on to the next section, which is about xUnit and writing tests.

## Understanding xUnit

**xUnit** is an open source modern testing framework for .NET written by the author of **NUnit v2**, a unit-testing framework for all .NET languages. xUnit is the first testing framework that is compatible with .NET Core.

xUnit is the default testing tool in Visual Studio for .NET Core and is currently being used by Microsoft themselves in their modern projects.

# Features of xUnit

Let's check out the features of xUnit, as follows:

- **Supports multiple platforms:** You can use it for testing .NET Framework applications, Xamarin applications, .NET Core applications, and ASP.NET Core applications.
- **Supports parallel test execution:** It speeds up the execution of unit tests.
- **Supports data-driven tests:** You can write one test and then pass different data inputs to see different expected outputs.
- **It is designed to be extensible:** You can add more data types, attributes, and asserts, and also use xUnit alongside other testing frameworks.
- **Easy to install:** You can install it via NuGet packages.

Okay—after learning the basics of xUnit, let's start using it.

## Using xUnit in ASP.NET Core

Our first task is to create a folder in the root directory and name it **tests**. Go to the newly created **tests** directory and run the following command:

```
dotnet new xunit --name Application.IntegrationTests
```

The preceding command creates an **xunit** project with the name **Application.IntegrationTests** in the **tests** directory.

The command automatically adds the following packages in the **xunit** project:

- **Microsoft.NET.Test.Sdk:** This package has MSBuild targets as well as properties for building .NET test projects.
- **xunit:** This package consists of xUnit libraries for writing tests such as **xunit.core**, **xunit.assert**, and **xunit.analyzers** tests.
- **xunit.runner.visualstudio:** This package is the **Test Explorer** runner for the xUnit testing framework.

Now, go to the **Application.IntegrationTests** project folder, like so:

```
cd Application.IntegrationTests
```

Then, add a reference to the **Travel.WebApi** project, as follows:

```
dotnet add reference ../../src/presentation/Travel.WebApi/Travel.WebApi.csproj
```

Now, let's add the following NuGet packages to **Application.IntegrationTests**:

- **FluentAssertions:** An elegant and better way of writing assertions in our tests
- **Moq:** The most friendly and popular mocking library for .NET
- **Respawn:** Resets test databases to a clean state

After installing the NuGet packages, go to the **tests** directory again and run the following **dotnet** command:

```
dotnet new xunit --name Application.UnitTests
```

The preceding command creates a new xUnit project called **Application.UnitTests** in the **tests** directory.

Then, go to **Application.UnitTests** by running the following command:

```
cd Application.UnitTests
```

Next, add a reference to the **Travel.Application** project, as follows:

```
dotnet add reference ../../src/core/Travel.Application/Travel.Application.csproj
```

Now, let's add a NuGet package to **Application.UnitTests**. We are going to add a package called **FluentAssertions**.

Now, go back to the **Travel** root directory where the solution file is located and run the following **dotnet** command:

```
dotnet sln add tests/Application.IntegrationTests/Application.IntegrationTests.csproj
```

The preceding command adds the **Application.IntegrationTest** project to the solution file of our ASP.NET Core application.

Let's also add the **Application.UnitTests** project to our solution, as follows:

```
dotnet sln add tests/Application.UnitTests/Application.UnitTests.csproj
```

We are only going to have a few unit tests for **Travel.Application**—the rest are integration tests for **Travel.Application** as well.

The folder structure of the application should look like this:

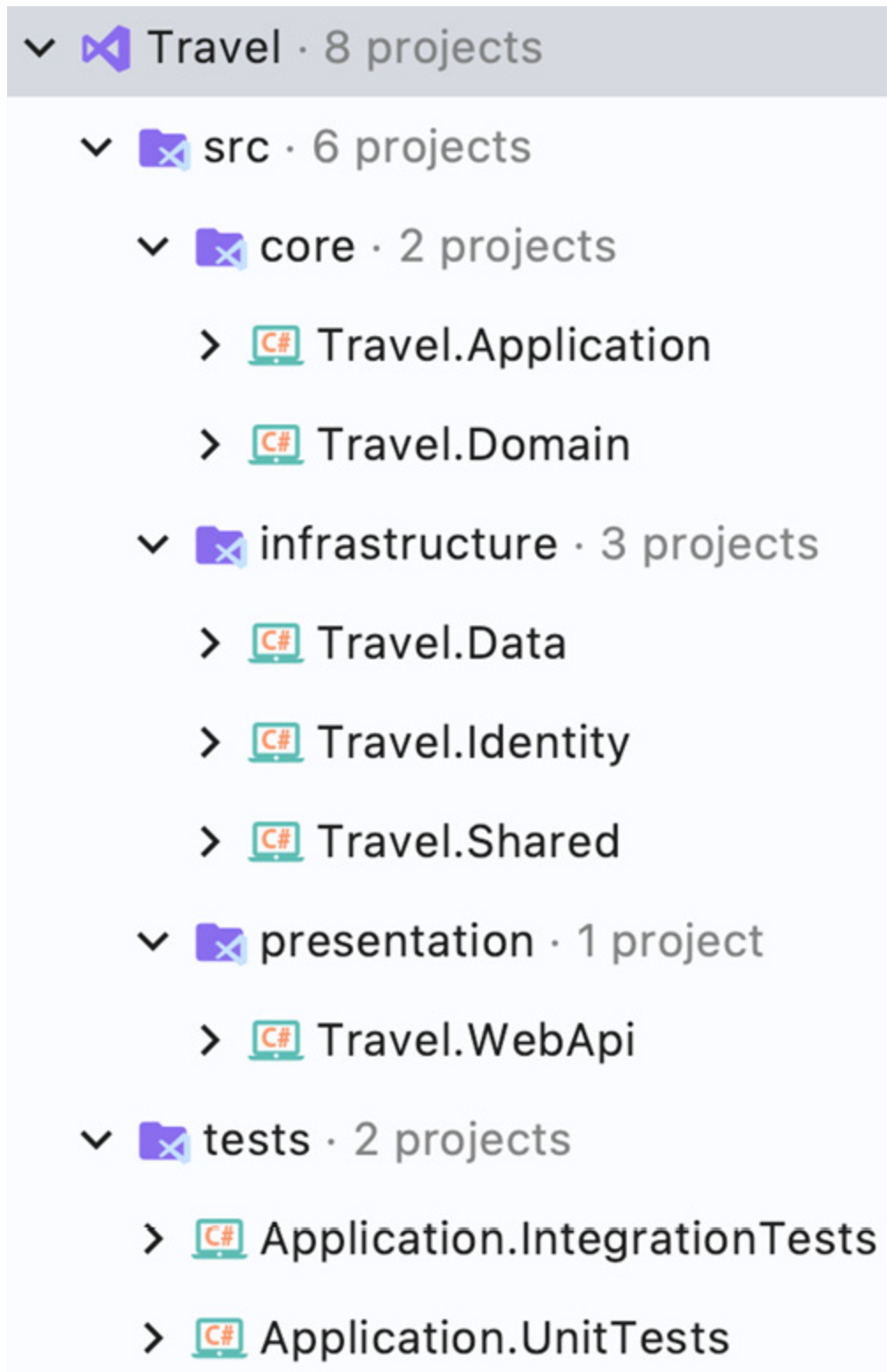


Figure 18.2 – Folder structure after adding tests projects

The preceding screenshot shows the projects in our **integrated development environment (IDE)**, where **Application.IntegrationTests** and **Application.UnitTests** are inside the **tests** directory.

Now, we are ready to go to the unit testing and integration testing sections. However, before proceeding, I highly recommend checking out the **FluentAssertions** website (<https://fluentassertions.com/>) and reading the documentation so that you have an idea of what **FluentAssertions** can do, as we will be using this to replace the built-in Assert **application programming interface (API)** of xUnit.

## Understanding unit testing

**Unit tests** test a unit of an application without external dependencies such as message queues, files, databases, web services, and so on. You can run hundreds of unit tests in just a few seconds because they are cheap to write and quick to execute, hence you can easily verify that each building block in your application is working as expected. Consequently, since you are not testing your class or components with their external dependencies, you can't gain a lot of confidence in your application's reliability. This is where integration tests come into play, which we will talk about later.

In my day-to-day work, I only write tests for complex code that is error-prone runtime code, complex code, and algorithmic logic, and then I put all my effort into integration tests. Anyway, I have other opinions about unit testing, but you will see these later. For now, let's start writing some unit tests using xUnit and **FluentAssertions**.

## Writing unit tests

Create a new folder called **Common** in the root directory of the **Application.UnitTests** project. Then, create two folders named **Exceptions** and **Mappings** inside the newly created **Common** folder.

### Tests for exception

Create a C# file named **ValidationExceptionTests.cs** inside the newly created **Exceptions** folder and add the following code:

```
using System;
using System.Collections.Generic;
using FluentAssertions;
using FluentValidation.Results;
using Travel.Application.Common.Exceptions;
using Xunit;

namespace Application.UnitTests.Common.Exceptions
{
    public class ValidationExceptionTests
    {

```

```

[Fact]
public void
DefaultConstructorCreatesAnEmptyErrorDictionary()
{
    var actual = new ValidationException().Errors;
    actual.Keys.Should()
        .BeEquivalentTo(Array.Empty<string>());
}
// another test method here.
}
}

```

What's happening in the preceding code? The **Fact** attribute is used to write unit tests that have no method arguments. The first unit test checks the default constructor of **ValidationException** and creates an empty array dictionary. The **Validation** class contains a dictionary of errors. Then, we can assert the test by using the **FluentAssertions** library.

Here is the other method:

```

[Fact]
public void SingleValidationFailure
CreatesASingleElementErrorDictionary()
{
    var failures = new List<ValidationFailure>
    {
        new ("Mobile", "Mobile is required.")
    };
    var actual = new ValidationException(failures)
        .Errors;
    actual.Keys.Should()
        .BeEquivalentTo("Mobile");
    actual["Mobile"].Should()
        .BeEquivalentTo("Mobile is required.");
}
}

```

**Should().BeEquivalentTo()** is the extension method from the **FluentAssertions** library that compares the object, which calls this method with the value passed into **BeEquivalentTo**. Then, the assertion will return **true** if both are equal.

The **FluentAssertions** library is preferred when comparing values other than strings. **Should().BeEquivalentTo()** is an extension method that can be used with different data types.

Although we can also use xUnit default methods such as **Assert.True()**, where we have to pass a condition that returns **true** or **false**, I prefer the **FluentAssertions** extension method over **Assert.True()** since it makes the comparison of the two states both easy and readable.

## Tests for mapping

Create a C# file named **MappingTests.cs** inside the newly created **Mappings** folder and then add the following code:

```
using System;

using AutoMapper;

using Travel.Application.Common.Mappings;

using Travel.Application.Dtos.Tour;

using Travel.Domain.Entities;

using Xunit;

namespace Application.UnitTests.Common.Mappings
{
    public class MappingTests
    {
        // Removed for brevity. Go to this repo's Github

        [Fact]
        public void ShouldHaveValidConfiguration()
        {
            _configuration.AssertConfigurationIsValid();
        }

        [Theory]
        [InlineData(typeof(TourList), typeof(TourListDto))]
        [InlineData(typeof(TourPackage),
            typeof(TourPackageDto))]
        public void ShouldSupportMappingFromSourceTo
            Destination(Type source, Type destination)
        {
            var instance = Activator.CreateInstance(source);
            _mapper.Map(instance, source, destination);
        }
    }
}
```

```
}  
}
```

So, what have you noticed in the preceding code? There are **Theory** and **InlineData** attributes. The **Theory** attribute is used to write unit tests whereby we pass a set of parameters to the method, while the **InlineData** attribute is used to give data to the method parameters.

We can write the **InlineData** attribute multiple times. xUnit will create multiple instances of tests and populate them with test case arguments, which we have passed in **InlineData**.

The last unit test with **InlineData** checks whether our **mapper** attribute supports the mapping of **TourList** to **TourListDto** and **TourPackage** to **TourPackageDto**.

We have a total of five unit tests in our code. The last unit test was multiplied by two because we are using two **InlineData** attributes.

Okay—so, you've now seen how to write unit tests using xUnit and **FluentAssertions**. It's not that complicated. Now, we can proceed to the next section, which is about integration testing.

## Understanding integration testing

What is an integration test? **Integration tests** test an application with its external dependencies by testing your application code's integration with concrete dependencies such as files, databases, and so on.

Integration tests take longer to execute because they often involve reading or writing to a database, but in return they give us more confidence in our application's health.

If a fast feedback loop from unit tests could send misleading results, I wouldn't mind writing a slow integration test that provides a correct outcome. A tricky feedback loop isn't worth the speed.

Enough about my opinions—let's start writing some integration tests and test all the commands and queries of our MediatR implementation.

## Writing integration tests

Our first task is to create a new **appsettings.json** file in the root folder of **Application.IntegrationTests**. Copy the content of the **Travel.WebApi**'s **appsettings.json** file and paste it into the newly created **appsettings.json** file, but change the database name from **TravelDb** to **TravelTestDb** in the connection strings.

Next, we will create a test fixture so that we can share a resource for several tests.

### Creating a test fixture



Create a C# file named **DatabaseFixture.cs** in the root folder of **Application.IntegrationTests** and add the following libraries and packages:

```
using System;
using System.IO;
using System.Threading.Tasks;
using MediatR;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Moq;
using Respawn;
using Travel.Data.Contexts;
using Travel.WebApi;
```

Now, let's define a **DatabaseFixture** class that implements **IDisposable**, like so:

```
namespace Application.IntegrationTests
{
    public class DatabaseFixture : IDisposable
    {
        private static IConfigurationRoot _configuration;
        private static IServiceScopeFactory _scopeFactory;
        private static Checkpoint _checkpoint;
        public DatabaseFixture()
        {
            var builder = new ConfigurationBuilder()
                .SetBasePath(Directory.GetCurrentDirectory())
                .AddJsonFile("appsettings.json", true, true)
                .AddEnvironmentVariables();
            _configuration = builder.Build();
            var startup = new Startup(_configuration);
            var services = new ServiceCollection();
            services.AddSingleton(
                Mock.Of<IWebHostEnvironment>(w =>
```

```

        w.EnvironmentName == "Development" &&
        w.ApplicationName == "Travel.WebApi"));
services.AddLogging();
startup.ConfigureServices(services);
_scopeFactory = services
    .BuildServiceProvider()
    .GetService<IServiceScopeFactory>();
_checkpoint = new Checkpoint
{
    TablesToIgnore = new[] {
        "__EFMigrationsHistory" }
};
EnsureDatabase();
} // static methods and IDispose are below ..
}
}

```

So, what did we write in the preceding code?

We created a configuration called **ConfigurationBuilder**, and services named **ServiceCollection**. We created a web hosting environment using **Moq** to run the application in the **Travel.WebApi** project. We also added a checkpoint for resetting the database state.

## Adding static methods

Add the static methods, as shown in the following code snippet:

```

private static void EnsureDatabase()
{
    using var scope = _scopeFactory.CreateScope();
    var context = scope.ServiceProvider
        .GetService<ApplicationDbContext>();
    context.Database.Migrate();
}

public static async Task ResetState()
{
    await _checkpoint.Reset(_configuration
        .GetConnectionString("DefaultConnection"));
}

```

```

}

public static async Task<TResponse> SendAsync<TResponse>(IRequest<TResponse> request)
{
    using var scope = _scopeFactory.CreateScope();
    var mediator = scope.ServiceProvider
        .GetService<IMediator>();
    return await mediator.Send(request);
}

```

We wrote a static method called **EnsureDatabase** that runs migrations for our database, and another static method named **ResetState** for resetting the database state before each test. Multiple database operations will not affect each other's state.

Here are the last two static methods. You know the drill, so add them as well. The code is shown in the following snippet:

```

public static async Task AddAsync<TEntity>(TEntity entity) where TEntity : class
{
    using var scope = _scopeFactory.CreateScope();
    var context = scope.ServiceProvider
        .GetService<ApplicationDbContext>();
    context.Add(entity);
    await context.SaveChangesAsync();
}

public static async Task<TEntity> FindAsync<TEntity>(int id)
    where TEntity : class
{
    using var scope = _scopeFactory.CreateScope();
    var context = scope.ServiceProvider
        .GetService<ApplicationDbContext>();
    return await context.FindAsync<TEntity>(id);
}

public void Dispose()
{
    // Code to run after all tests
}

```

The last two static methods are for adding and finding an entity. Lastly, the **Dispose** method is a global teardown that gets called after every test method.

## Using the test fixture

To use the **DatabaseFixture** class that we created, let's create a C# file named **DatabaseCollection.cs** in the root folder of **Application.IntegrationTests** again and add the following code:

```
using Xunit;

namespace Application.IntegrationTests
{
    [CollectionDefinition("DatabaseCollection")]
    public class DatabaseCollection :
        ICollectionFixture<DatabaseFixture>
    {
    }
}
```

The class has a **CollectionDefinition** attribute and implements an **ICollectionFixture** type called **DatabaseFixture**.

You will notice there is no code in the **DatabaseCollection** class because it is only used to define the collection. Moreover, the string in the **CollectionDefinition** attribute gives a unique name for the test collection identification.

Next, create a **TourLists** folder in the root directory of the **Application.IntegrationTests** project. Then, create two folders called **Commands** and **Queries** inside the newly created **TourLists** folder.

## TourLists/Queries/GetTourTests

Now, let's create a C# file named **GetToursTests.cs** inside the **Queries** folder and add the following code:

```
using System.Collections.Generic;
using System.Threading.Tasks;
using FluentAssertions;
using Travel.Application.TourLists.Queries.GetTours;
using Travel.Domain.Entities;
using Xunit;

namespace Application.IntegrationTests.TourLists.Queries
```

```

{
    using static DataFixture;
    [Collection("DatabaseCollection")]
    public class GetToursTests
    {
        public GetToursTests()
        {
            ResetState().GetAwaiter().GetResult();
        }

        // test methods here
    }
}

```

So, what's happening in the preceding code? Firstly, we are using **static DataFixture** to access all the static methods of our base class.

Then, we have a **Collection** attribute to identify the collection this test class belongs to. We are passing a **DatabaseCollection** string to make it part of that collection.

The **GetToursTests** class will be able to share the test context of the **DataFixture** class with other tests.

And in the **GetTourTests** constructor, we call the **ResetState** method to reset the database to its original state before each test runs.

Here are the test methods for the **GetToursTests** test class. Write the following code:

```

[Fact]
    public async Task ShouldReturnTourLists()
    {
        var query = new GetToursQuery();
        var result = await SendAsync(query);
        result.Lists.Should().NotBeEmpty();
    }

[Fact]
    public async Task
    ShouldReturnAllTourListsAndPackages()
    {
        await AddAsync(new TourList

```

```

    {
        City = "Manila", Country = "Philippines",
        About = "Lorem Ipsum",
        TourPackages = new List<TourPackage>
        {
            new()
            {
                Name = "Free Walking Tour Manila",
                // Removed for brevity. Go to this repo's Github.
            }
        }
    });
    var query = new GetToursQuery();
    var result = await SendAsync(query);
    result.Lists.Should().HaveCount(1);
}

```

Lastly, the **ShouldReturnTourLists** test method should not be empty, while the **ShouldReturnAllTourListsAndPackages** method should have one object.

### **TourLists/Commands/CreateTourListTests**

Next, we create a C# file named **CreateTourListTests.cs** in the **Commands** folder of the **TourLists** directory and add the following code:

```

// Removed for brevity. Go to this repo's Github.
using Travel.Application.Common.Exceptions;
namespace Application.IntegrationTests.TourLists.Commands
{
    using static DataFixture;
    [Collection("DatabaseCollection")]
    public class CreateTourListTests
    {
        public CreateTourListTests()
        {
            ResetState().GetAwaiter().GetResult();
        }
    }
}

```

```

[Fact]
public void ShouldRequireMinimumFields()
{
    var command = new CreateTourListCommand();
    FluentActions.Invoking(() =>
        SendAsync(command)).Should()
        .Throw<ValidationException>();
}
// two test methods here
}
}

```

As you can see in the preceding code, we reuse **static DataFixture**, the **Collection** attribute, and the **ResetState** function. Expect these through the last test class that we will create later. Anyhow, the **ShouldRequireMinimumFields** test method should throw a **ValidationException** error, the custom validation exception we've created in the **Travel.Application** project, because the test tried to create a **TourList** object without the required properties.

Here are two more test methods:

```

[Fact]
public void ShouldRequireAbout()
{
    var command = new CreateTourListCommand
    {
        City = "Antananarivo",
        Country = "Madagascar", About = ""
    };
    FluentActions.Invoking(() =>
        SendAsync(command)).Should()
        .Throw<ValidationException>();
}

[Fact]
public async Task ShouldCreateTourList()
{
    var command = new CreateTourListCommand
    {

```

```

        City = "Antananarivo", Country = "Madagascar",
        About = "Lorem Ipsum"
    };

    var id = await SendAsync(command);
    var list = await FindAsync<TourList>(id);
    list.Should().NotNull();
    list.City.Should().Be(command.City);
    list.Country.Should().Be(command.Country);
    list.About.Should().Be(command.About);
}

```

The **ShouldRequireAbout** test should also throw a validation error because the **About** property is empty. And lastly, the **ShouldCreateTourList** test should create a **TourList** object because all the required properties were included.

### TourLists/Commands/DeleteTourListTests

Next, we create a C# file named **DeleteTourListTests.cs** in the **Commands** folder of the **TourLists** directory and add the following code:

```

using FluentAssertions;

// Removed for brevity. Go to this repo's Github.

using Xunit;

namespace Application.IntegrationTests.TourLists.Commands
{
    using static DataFixture;

    [Collection("DatabaseCollection")]
    public class DeleteTourListTests
    {
        public DeleteTourListTests()
        {
            ResetState().GetAwaiter().GetResult();
        }

        [Fact]
        public void ShouldRequireValidTourListId()
        {
            var command = new DeleteTourListCommand

```



```

        {
            Id = 33
        };

        FluentActions.Invoking(() =>
            SendAsync(command)).Should()
            .Throw<NotFoundException>();
    }

    // another test method
}

```

**ShouldRequireValidTourListId** should throw a *not found* exception in the preceding code because the ID **33** does not exist, but the test method tried to delete it.

And here is the other test method:

```

[Fact]
public async Task ShouldDeleteTourList()
{
    var listId = await SendAsync(new
        CreateTourListCommand
    {
        City = "Beirut", Country = "Lebanon",
        About = "Lorem Ipsum"
    });

    await SendAsync(new DeleteTourListCommand
    {
        Id = listId
    });

    var list = await FindAsync<TourList>(listId);
    list.Should().BeNull();
}

```

The **ShouldDeleteTourList** list should be *null* because **listId** property was used to delete the **TourList** object.

## TourLists/Commands/UpdateTourListTests

Next, we create a C# file named **UpdateTourListTests.cs** in the **Commands** folder of the **TourLists** directory and add the following code:

```
using FluentAssertions;

// Removed for brevity. Go to this repo's Github.

using Xunit;

namespace Application.IntegrationTests.TourLists.Commands
{
    using static DataFixture;

    [Collection("DatabaseCollection")]

    public class UpdateTourListTests
    {
        public UpdateTourListTests()
        {
            ResetState().GetAwaiter().GetResult();
        }

        [Fact]

        public void ShouldRequiredValidTourListId()
        {
            var command = new UpdateTourListCommand
            {
                Id = 8, City = "Caracas",
                Country = "Venezuela", About = "Lorem Ipsum"
            };

            FluentActions.Invoking(() =>
                SendAsync(command)).Should()
                .Throw<NotFoundException>();
        }

        // another test method
    }
}
```

**ShouldRequiredValidTourListId** should throw a *not found* exception in the preceding code because it tried to update a non-existing **TourList** object.

For the next test method, **ShouldUpdateTourList**, run the following code:

[Fact]

```
public async Task ShouldUpdateTourList()
{
    var listId = await SendAsync(new
        CreateTourListCommand
    {
        City = "Al Ghanim", Country = "Qatar",
        About = "Lorem Ipsum"
    });

    var command = new UpdateTourListCommand
    {
        Id = listId, City = "Doha",
        Country = "Qatar", About = "Lorem Ipsum"
    };

    await SendAsync(command);

    var list = await FindAsync<TourList>(listId);
    list.City.Should().Be(command.City);
    list.Country.Should().Be(command.Country);
    list.About.Should().NotBeNull();
}
```

Conversely, the **ShouldUpdateTourList** test should not be *null* after calling the **FindAsync** method because a **TourList** object was created and updated.

Now, let's create a **TourPackages** folder in the root directory of the **Application.IntegrationTests** project. Then, create a folder called **Commands** inside the newly created **TourPackages** folder.

### **TourPackages/Commands/CreateTourPackageTests**

Next, we create a C# file named **CreateTourPackageTests.cs** in the **Commands** folder of the **TourPackages** directory and add the following code:

```
using FluentAssertions;

// Removed for brevity. Go to this repo's Github.

using Xunit;

namespace Application.IntegrationTests.TourPackages.Commands
{
```

```

using static DataFixture;

[Collection("DatabaseCollection")]
public class CreateTourPackageTests
{
    public CreateTourPackageTests()
    {
        ResetState().GetAwaiter().GetResult();
    }

    [Fact]
    public void ShouldRequireMinimumFields()
    {
        var command = new CreateTourPackageCommand();
        FluentActions.Invoking(() =>
            SendAsync(command)).Should()
            .Throw<ValidationException>();
    }

    // next test method here
}

```

**ShouldRequireMinimumFields** should throw a validation exception in the preceding code because it tried to create a **TourPackage** object without the required properties.

For the next test method, **ShouldCreateTourPackage**, run the following code:

```

[Fact]
public async Task ShouldCreateTourPackage()
{
    var listId = await SendAsync(new
        CreateTourListCommand
    {
        City = "New York", Country = "USA",
        About = "Lorem Ipsum"
    });

    var command = new CreateTourPackageCommand
    {

```

```

        ListId = listId,

        // Removed for brevity. Go to this repo's Github.
        WhatToExpect = "Lorem Ipsum"
    };

    var packageId = await SendAsync(command);
    var package = await
    FindAsync<TourPackage>(packageId);

    package.Should().NotNull();
    package.ListId.Should().Be(command.ListId);
    package.Name.Should().Be(command.Name);

    // Removed for brevity. Go to this repo's Github.
    package.WhatToExpect
        .Should().Be(command.WhatToExpect);
}

```

Then, **ShouldCreateTourPackage** should create a **TourPackage** object and find it because all required properties are filled.

### **TourPackages/Commands/DeleteTourPackageTests**

Next, we create a C# file named **DeleteTourPackageTests.cs** in the **Commands** folder of the **TourPackages** directory and add the following code:

```

using System.Threading.Tasks;

// Removed for brevity. Go to this repo's Github.
using Xunit;

namespace Application.IntegrationTests.TourPackages.Commands
{
    using static DataFixture;
    [Collection("DatabaseCollection")]
    public class DeleteTourPackageTests
    {
        public DeleteTourPackageTests()
        {
            ResetState().GetAwaiter().GetResult();
        }

        [Fact]
    }
}

```

```

public void ShouldRequireValidTourPackageId()
{
    var command = new DeleteTourPackageCommand
    {
        Id = 69
    };
    FluentActions.Invoking(() =>
        SendAsync(command)).Should()
        .Throw<NotFoundException>();
}
// second test method here
}
}

```

**ShouldRequireValidTourPackageId** should throw a *not found* exception in the preceding code because it tried to delete a non-existing **TourPackage** object.

For the second test method, **ShouldDeleteTourPackage**, run the following code:

[Fact]

```

public async Task ShouldDeleteTourPackage()
{
    var listId = await SendAsync(new
    CreateTourListCommand
    {
        City = "Tashkent", Country = "Uzbekistan",
        About = "Lorem Ipsum"
    });
    var packageId = await SendAsync(new
    CreateTourPackageCommand
    {
        ListId = listId, Name = "Silk Road Adventures",
        // Removed for brevity. Go to this repo's Github.
        WhatToExpect = "Lorem Ipsum"
    });
    await SendAsync(new DeleteTourPackageCommand

```

```

        {
            Id = packageId
        });
        var list = await FindAsync<TourPackage>(listId);
        list.Should().BeNotNull();
    }

```

On the flip side, **ShouldDeleteTourPackage** should create a **TourPackage** object, delete it, and receive *null* after calling the **FindAsync** method.

### **TourPackages/Commands/UpdateTourPackageDetailTests**

Next, we create a C# file named **UpdateTourPackageDetailTests.cs** in the **Commands** folder of the **TourPackages** directory and add the following code:

```

using FluentAssertions;
using Travel.Application.Common.Exceptions;
using Xunit;

// Removed for brevity. Go to this repo's Github.
namespace Application.IntegrationTests.TourPackages.Commands
{
    using static DataFixture;

    [Collection("DatabaseCollection")]
    public class UpdateTourPackageDetailTests
    {
        public UpdateTourPackageDetailTests()
        {
            ResetState().GetAwaiter().GetResult();
        }

        [Fact]
        public void ShouldRequireValidTourPackageId()
        {
            var command = new UpdateTourPackageCommand
            {
                Id = 88,
                Name = "Free Walking Tour"
            };

```

```

        FluentActions.Invoking(() => SendAsync(command))
            .Should()
            .Throw<NotFoundException>();
    }

    // another test method here
}

```

**ShouldRequireValidTourPackageId** should throw a *not found* exception in the preceding code because it tried to update a non-existing **TourPackage** object.

For the next test method, **ShouldUpdateTourPackage**, run the following code:

```

[Fact]
public async Task ShouldUpdateTourPackage()
{
    var listId = await SendAsync(new
        CreateTourListCommand
    {
        City = "Zagreb", Country = "Croatia",
        About = "Lorem Ipsum"
    });

    var packageId = await SendAsync(new
        CreateTourPackageCommand
    {
        ListId = listId,
        Name = "Free Walking Tour Zagreb",
        // Removed for brevity. Go to this repo's Github.
        WhatToExpect = "Lorem Ipsum"
    });

    var command = new UpdateTourPackageDetailCommand
    {
        Id = packageId,
        ListId = listId,
        // Removed for brevity. Go to this repo's Github.
        WhatToExpect = "Lorem Ipsum"
    }
}

```



```

};

await SendAsync(command);

var item = await

FindAsync<TourPackage>(packageId);

item.ListId.Should().Be(command.ListId);

// Removed for brevity. Go to this repo's Github.

.Be(command.WhatToExpect);

}

```

In contrast, the **ShouldUpdateTourPackage** method should create a **TourPackage** object, update it, and find it because all required properties and validations have been met.

### **TourPackages/Commands/UpdateTourPackageTests**

Next, we create a C# file named **UpdateTourPackageTests.cs** in the **Commands** folder of the **TourPackages** directory and add the following code:

```

using Travel.Application.Common.Exceptions;

// Removed for brevity. Go to this repo's Github.

using Xunit;

namespace Application.IntegrationTests.TourPackages.Commands
{
    using static DataFixture;

    [Collection("DatabaseCollection")]

    public class UpdateTourPackageTests
    {
        public UpdateTourPackageTests()
        {
            ResetState().GetAwaiter().GetResult();
        }

        [Fact]

        public void ShouldRequireValidTourPackageId()
        {
            var command = new UpdateTourPackageCommand
            {
                Id = 4, Name = "Free Walking Tour"
            };
};

```

```

        FluentActions.Invoking(() =>
            SendAsync(command)).Should()
                .Throw<NotFoundException>();
    }

    // two more test methods here..
}

```

In the preceding code, the **ShouldRequireValidTourPackageId** test method should throw a *not found* exception because it tried to update a non-existent **TourPackage** object.

For the next test method, **ShouldUpdateTourPackage**, copy the following code:

```

[Fact]
public async Task ShouldUpdateTourPackage()
{
    var listId = await SendAsync(new
        CreateTourListCommand
    {
        City = "Rabat", Country = "Morocco",
        About = "Lorem Ipsum"
    });

    var packageId = await SendAsync(new
        CreateTourPackageCommand
    {
        ListId = listId,
        Name = "Free Walking Tour Rabat",
        // Removed for brevity. Go to this repo's Github.
        WhatToExpect = "Lorem Ipsum"
    });

    var command = new UpdateTourPackageCommand
    {
        Id = packageId,
        Name = "Night Free Walking Tour Rabat."
    };

    await SendAsync(command);
}

```

```

        var item = await
        FindAsync<TourPackage>(packageId);
        item.Name.Should().Be(command.Name);
        item.WhatToExpect.Should().NotNull();
    }

```

**ShouldUpdateTourPackage** should create a **TourList** object with a **TourPackage** object as a property, update **TourPackage**, and see it after invoking the **FindAsync** method because all requirements have been met.

For the last test method, **ShouldRequireUniqueName**, copy the following code:

```

[Fact]
public async Task ShouldRequireUniqueName()
{
    var listId = await SendAsync(new
    CreateTourListCommand
    {
        City = "Bogota", Country = "Colombia",
        About = "Lorem Ipsum"
    });
    await SendAsync(new CreateTourPackageCommand
    {
        ListId = listId, Name = "Bike Tour in Bogota"
    });
    await SendAsync(new CreateTourPackageCommand
    {
        ListId = listId, Name = "Salt Cathedral Tour"
    });
    var command = new UpdateTourPackageCommand
    {
        Id = listId, Name = "Salt Cathedral Tour"
    };
    FluentActions.Invoking(() =>
        SendAsync(command)).Should()
        .Throw<ValidationException>()

```

```
        .Where(ex => ex.Errors.ContainsKey("Name"))
        .And.Errors["Name"].Should()
        .Contain("The specified name already
exists.");
    }
```

The **ShouldRequireUniqueName** method should throw a validation exception that contains a **Name** key because it tried to use a name that already exists in another **TourPackage** object. We've written all the necessary tests that we need in our application. Now, go to the root directory of our solution, rebuild the application, and run the following **dotnet** command:

```
dotnet test
```

The preceding **dotnet test** command will run the tests of the **Application.IntegrationTests** and **Application.UnitTests** projects.

The results of running all the tests are shown here:

```
Passed! - Failed:    0, Passed:    18, Skipped:    0, Total:    18, Duration:    2 s
```

There are no failing tests. It means **0** tests failed, **18** tests passed, and there were no skipped tests. Plus, all of them ran for a total of **2** seconds, which is fast. They might run for **3** to **5** seconds, depending on your computer's specifications or open apps on your computer.

You can also run all the tests using the **Test Explorer** in your IDE.

Here is the result of running the tests on Visual Studio 2019's **Test Explorer**:

## Test Explorer



Test	Duration	Traits
Application.IntegrationTests (18)	1.3 sec	
Application.IntegrationTests.TourLists.C...	714 ms	
CreateTourListTests (3)	635 ms	
DeleteTourListTests (2)	62 ms	
UpdateTourListTests (2)	17 ms	
Application.IntegrationTests.TourLists.Q...	439 ms	
GetToursTests (2)	439 ms	
Application.IntegrationTests.TourPackag...	141 ms	
CreateTourPackageTests (2)	58 ms	
DeleteTourPackageTests (2)	23 ms	
UpdateTourPackageDetailTests (2)	28 ms	
UpdateTourPackageTests (3)	32 ms	
Application.UnitTests (4)	180 ms	
Application.UnitTests.Common.Exceptio...	35 ms	
ValidationExceptionTests (2)	35 ms	
Application.UnitTests.Common.Mappings	145 ms	
MappingTests (2)	145 ms	

### Group Summary

Application.IntegrationTests

Tests in group: 18

⌚ Total Duration: 1.3 sec

Outcomes

✓ 18 Passed

Figure 18.3 – Passed tests on Visual Studio 2019's Test Explorer

The preceding screenshot shows that all the tests are passing on Visual Studio 2019.

Here is the result of running the tests in Visual Studio for Mac's **Test Explorer**:

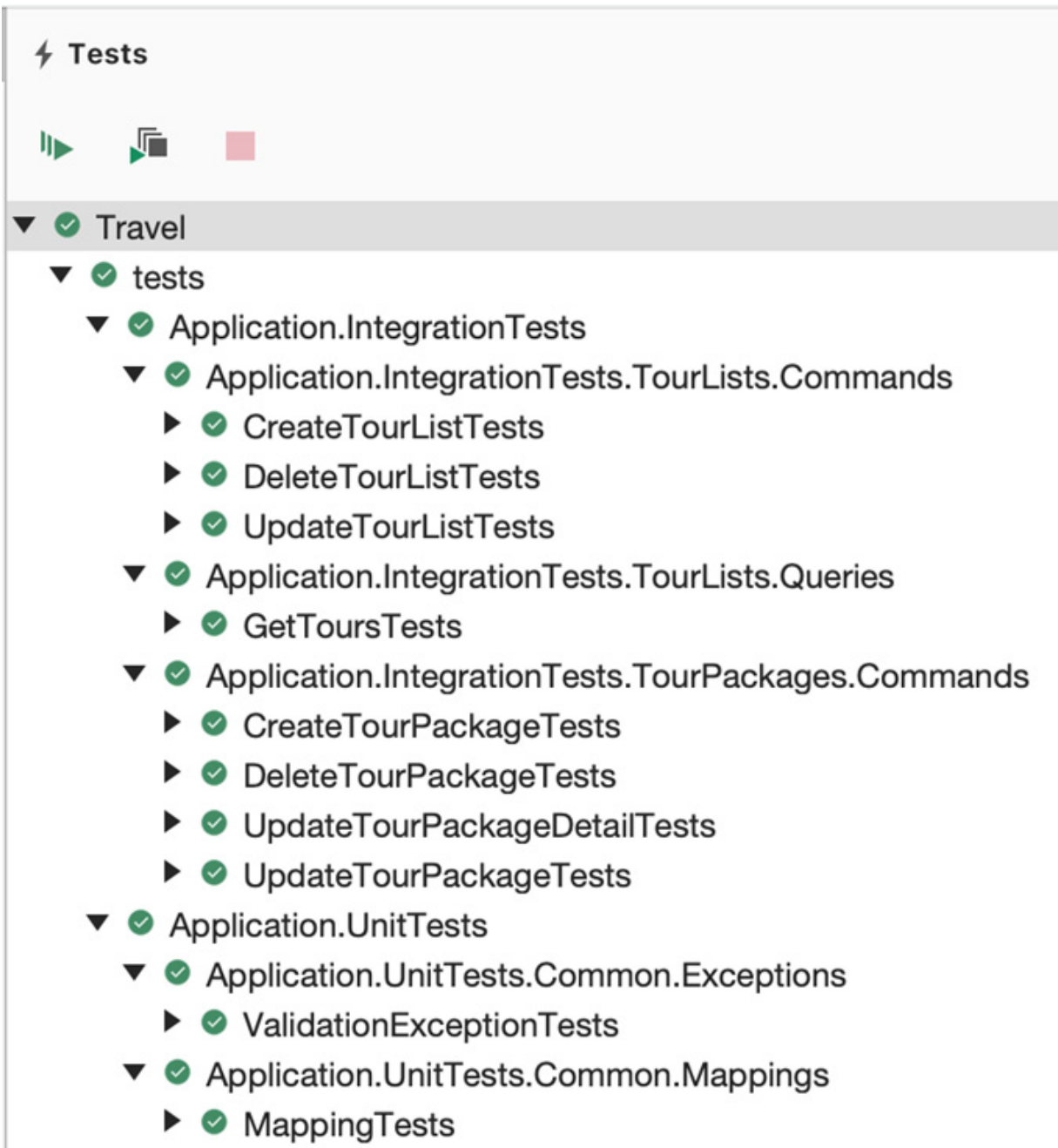


Figure 18.4 – Passed tests on Visual Studio for Mac's Test Explorer

The following screenshot shows the result of running the tests in JetBrains Rider's **Test Explorer**:

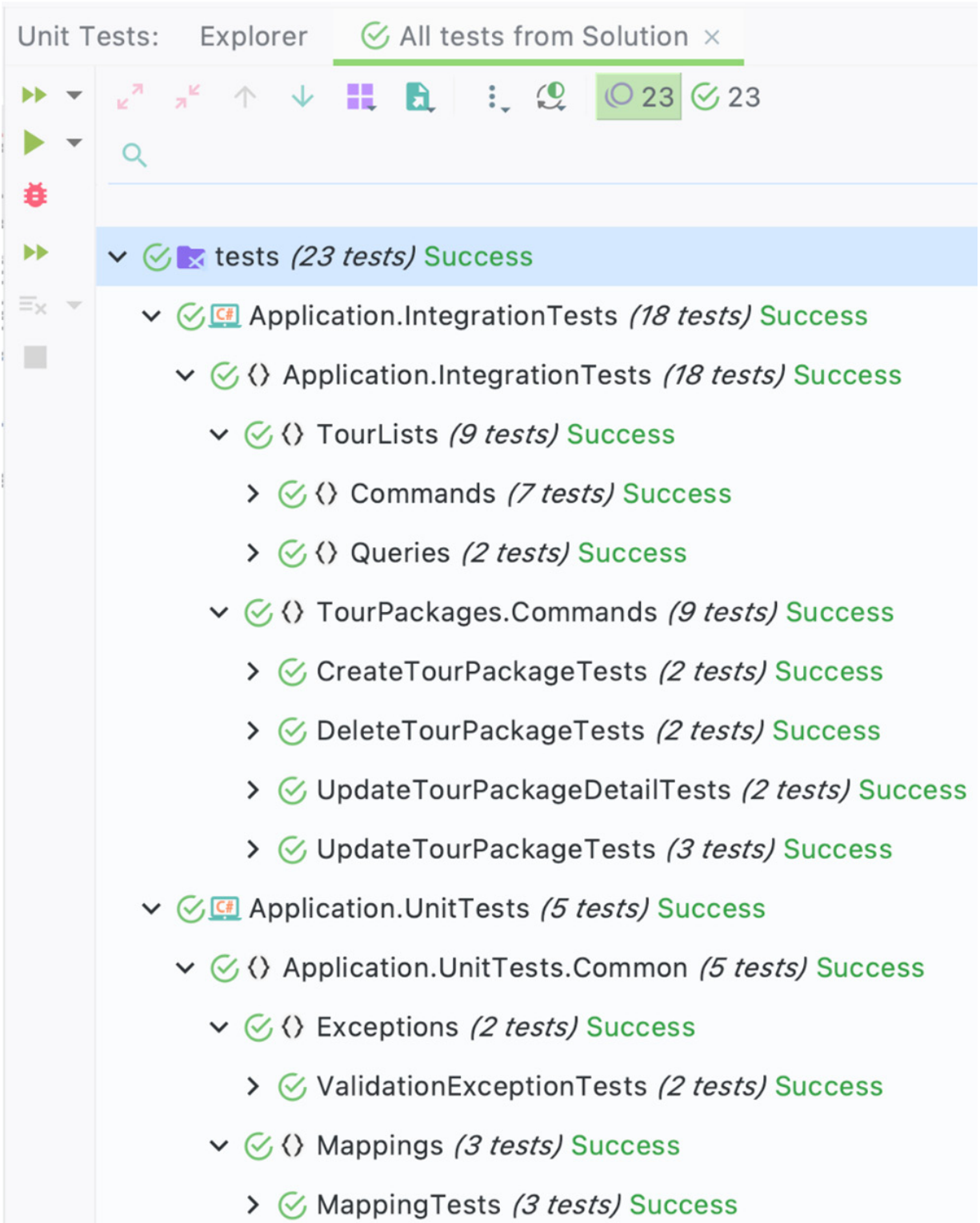


Figure 18.5 – Passed tests on JetBrains Rider's Test Explorer

And that's it! We have written automated tests using xUnit and have written these in a pragmatic way, but packed with best practices and using the latest packages/libraries in our tests.

However, I still encourage you to read more about writing tests using xUnit to solidify what you have learned here. Here are some references:

- **Testing in .NET:** <https://docs.microsoft.com/en-us/dotnet/core/testing/>
- **Using xUnit to test your C# code:** <https://auth0.com/blog/xunit-to-test-csharp-code/>
- **xUnit.net:** <https://xunit.net/>

You will see that the assertions in the tests from the preceding links are not **FluentAssertions**. To test what you have learned, you need to do some exercises.

## ACTIVITY

*Convert all the xUnit assertions that you see in the test samples of the link to **FluentAssertions**.*

This was indeed an excellent chapter for busy developers such as you. Now, let's sum up what we have tackled.

## Summary

You have learned what automated testing is and the value that it brings to software development. You've learned how to install a MS SQL Server database in a Docker container for Linux or macOS users. You've learned what xUnit is and why it is suited to building modern .NET apps, and how to use xUnit with **FluentAssertions**. You've also learned what are the pros and cons of unit and integration testing.

In the next chapter, we will do ASP.NET Core and Vue.js app deployments in containers and in the Azure Cloud.



## Chapter 19: Automatic Deployment Using GitHub Actions and Azure

In the previous chapter, we wrote integration tests in our apps to ensure that the components can interact correctly. Now we will deploy ASP.NET Core and Vue.js apps to Azure Cloud services using the new **Continuous Integration (CI)/Continuous Deployment (CD)** tools, GitHub Actions. CI/CD automates the building, testing, and deployment of applications that saves time and improves the efficiency of application development.

In this chapter, we will cover the following topics:

- Introducing GitHub Actions – a CI/CD tool
- Understanding GitHub Actions
- Understanding where to deploy
- Automated deployment to Azure App Service using GitHub Actions

### Technical requirements

Here is the link to the finished repository of this chapter:

<https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter19/>.

Here is what you need to complete this chapter:

- **An Azure account:** <https://azure.microsoft.com/en-us/free/>
- **A GitHub account:** <https://github.com/>

### Introducing GitHub Actions – a CI/CD tool

Do you still remember the methods of shipping apps into production before the arrival of CI/CD? There was copy and paste, FTP, scripting, or SSH when we deployed our apps to production. Then CI/CD came along, which helps to quickly build and publish applications without worrying about the manual configuration and deployment process. Then, a CI/CD tool called Hudson was born, which eventually became Jenkins. Finally, we started having tools such as Octopus Deploy, Visual Studio Team Services, Azure DevOps for deployment, and having better integration with our repositories. The CI/CD tools mentioned give us a single place where we can have our tools, all of our code, and the configuration needed for shipping our applications.

And then GitHub Actions came along.

### Understanding GitHub Actions

**GitHub Actions** is a platform that automates software development workflows. The CI/CD pipeline is one of several workflows that you can use to automate with GitHub Actions.

GitHub Actions is a YAML-based workflow linked to getting repositories. This workflow can be triggered by webhooks or schedules, or by manually clicking a button to start a workflow.

A runner, where workflows run, comes in two forms. You have an option to use GitHub hosted or to do self-hosting. The GitHub hosted workflow provides different operating systems such as Ubuntu, macOS, or Windows, and they have pre-installed software, including several versions of .NET Core SDK.

GitHub Actions also generates logs while running. You can see the results of your build through the logs that are available in the GitHub portal.

Also, there are custom workflows available that have been built and shared by the community. Like the Azure team, some organizations have contributed and developed actions and workflows with different scenarios for us to use as building blocks in our workflows.

## GitHub Actions for .NET apps

So how do we get our .NET Core applications wired up with GitHub Actions? There's a GitHub action for .NET that is called **setup-dotnet**. What is setup-dotnet? It provides a .NET CLI environment for a runner to specify the version of .NET to use. We can now use GitHub Actions in the **setup-dotnet** action to select a target version in our workflow.

We can also configure our environment to use private package repositories or sources. For example, if we have libraries or packages hosted as NuGet packages, GitHub packages, or Azure DevOps artifacts, we can securely leverage them by using actions.

## Understanding where to deploy

You will see workflow templates later, with which you don't have to start writing your workflow file from scratch. You can deploy to Azure, AWS, GCP, IBM Cloud, Alibaba Cloud, OpenShift, and more.

We will focus on Azure since we are using C# and .NET Core. Here are the most common services in Azure:

- Azure App Service
- Azure Functions
- Azure Static Web Apps
- Azure Kubernetes Service

Let's find out what these are and their use cases.

## When to deploy to Azure App Service?

Consider using Azure App Service if you encounter the following:

- When you have a web app from any programming languages and frameworks, or ASP.NET Razor pages, ASP.NET MVC, or an ASP.NET Web API, or even server-side Blazor
- When you want to deploy a dedicated high-performance application or a scalable application
- When you want more control over an environment and configuration without getting a whole virtual machine or an entire operating system, because App Service has rich feature sets such as slots, VNet support, and Azure Active Directory, to name a few, that are built into it

## When to deploy to Azure Functions?

**Azure Functions** is part of the Azure serverless architecture space. So, when should you consider deploying to Azure Functions?

- When you want to run small stateless tasks, you can run functions inside Azure using a serverless architecture. However, there are also stateful and durable functions, which let us run complex scenarios.
- When you want an event-driven architecture with triggers and binding; for instance, blob files being dropped in an Azure Storage container.

## When to deploy to Azure Static Web Apps?

**Azure Static Web Apps**, or **Azure SWA**, is a new service in Azure that allows you to have a static site. So, when should you consider deploying to Azure Static Web Apps?

- When you don't need a server at all to host your frontend application
- When you are using Blazor WebAssembly apps

Let's now check out Azure Kubernetes Service, or AKS.

## When to deploy to Azure Kubernetes Service?

**Azure Kubernetes Service**, or **AKS**, is a managed Kubernetes service from Azure. So, when should you consider deploying to Azure Kubernetes Service?

- When you don't have any expertise in container orchestration but you want to use containers for deployment
- When you want to scale and manage Docker containers or other container-based apps using Kubernetes

These are your options when deploying to Azure.

So, GitHub Actions is a great solution for quickly setting up your CI/CD and GitHub repositories for building and deploying .NET Core applications. GitHub provides workflows, runners inside the GitHub portal, and viewable logs to make our lives easier. We have a plethora of available custom workflows whenever we want to deploy our ASP.NET Core apps to Azure or other types of applications to other known cloud services.

Now, let's move on to the next section to deploy an ASP.NET Core Web API with the **Progressive Web App (PWA)** Vue.js TypeScript app.

## Automated deployment to Azure App Service using GitHub Actions

In this final section, we will deploy an ASP.NET Core 5 and Vue.js application to try an automated deployment to Azure App Service using GitHub Actions.

The ASP.NET Core 5 application will be a Web API project, while the Vue.js application will be built using TypeScript to show that we can add a TypeScript compiler to project's **csproj** file. We are also going to convert our Vue.js application to a PWA. If you are not yet familiar with PWAs, a PWA is a web application with mobile app features. PWAs can be installed on desktop, or mobile devices, and can be run offline.

We will use the existing ASP.NET Core and Vue app to simplify deployment to Azure using GitHub Actions. The goal here is to try deployment and focus only on GitHub Actions and the app service, which is the main topic in this chapter.

OK, so let's start:

1. Create a folder named **Travel** and then go inside the **Travel** folder using your command line and run the following **dotnet** CLI command:

```
dotnet new sln
```

2. Create a **webapi** project by running the following command:

```
dotnet new webapi --name WebUI
```

A project named **WebUI** will be created.

3. Add the project to the .NET solution:

```
dotnet sln add WebUI/WebUI.csproj
```

The WebUI project is now visible on your IDE.

4. Go to the **WebUI** project directory:

```
cd WebUI
```

While inside the **WebUI** project, we will create our Vue.js app here.

5. Run the following **vue** CLI command:

```
vue create client-app
```

The preceding command will initiate the **vue** CLI for scaffolding of the Vue.js project.

6. Choose the **Manually select features** option, and then add **TypeScript**, **PWA**, and **Router**.

7. Remove the linter, press *Enter*, and then choose version **3.x**. Then, hit the *Enter* key on your keyboard several times to choose the default settings for the remainder of the configurations.

8. We rename the **client-app** folder inside the **WebUI** project to **ClientApp**.

If you are familiar with the ASP.NET Angular and ASP.NET React templates, you will notice the templates and the **ClientApp** naming convention in the naming of the SPA folder.

9. After renaming the folder of the Vue app, let's update **WebUI.csproj** of the project with the code from the following GitHub repository: <https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter19/Travel/WebUI/WebUI.csproj>.

The code from the preceding link is similar to the **csproj** file of the **Travel.WebApi** project. The only difference is that we are adding three extra properties below the **TargetFramework** property like so:

```
<TypeScriptCompileBlocked>true</TypeScriptCompileBlocked>
<TypeScriptToolsVersion>Latest</TypeScriptToolsVersion>
<IsPackable>>false</IsPackable>
```

**TypeScriptCompileBlocked** is set to **true** to help us debug our Vue TypeScript application.

**TypeScriptToolsVersion** targets the latest TypeScript version.

**IsPackable**, which is set to **false**, will not package the project.

10. Next, we update the **Startup.cs** file for the **WebUI** project with the code from the following link:

<https://github.com/PacktPublishing/ASP.NET-Core-and-Vue.js/tree/master/Chapter19/Travel/WebUI/Startup.cs>.

The code from the preceding link is also simple to help us quickly deploy our application. It is a simple Web API project importing **VueCliMiddleware** along with the **SpaStaticFiles** service.

11. Next, we run the following command inside the **WebUI** project:

```
dotnet run
```

The preceding command will build and run the backend and frontend applications. You should see the **webpack** progress reach **100%**, as shown in the following screenshot:

```
[webpack.Progress] 100%
App running at:
- Local:    http://localhost:8080/
It seems you are running Vue CLI inside a container.
Access the dev server via http://localhost:<your container's
external mapped port>/
Note that the development build is not optimized.
To create a production build, run npm run build.
Issues checking in progress...
No issues found.
```

Figure 19.1 – Vue app build and running

*Figure 19.1* shows that the application is running without any problems. You can also check **https://localhost:5001** to see that the application is running.

12. Publish the project to your own GitHub account and put it in a private repository. After publishing your **Travel** project to GitHub, go to the **Actions** tab menu on your project's repository page.

You should see the suggested workflows as shown in *Figure 19.2*, and the other workflows for deployment, continuous integration, and other steps. These are the workflows that will make our lives easier:

## Workflows made for your C# repository Suggested

### .NET

By GitHub Actions

Build and test a .NET or ASP.NET Core project.

[Set up this workflow](#)

```
dotnet restore
dotnet build --no-restore
dotnet test --no-build --verbosity normal
```

actions/starter-workflows

C# ●

### .NET Desktop

By GitHub Actions

Build, test, sign and publish a desktop application built on .NET.

[Set up this workflow](#)

```
dotnet test
msbuild $env:Solution_Name /t:Restore
/p:Configuration=$env:Configuration
$pfxcert_byte =
[System.Convert]::FromBase64String("${{
secrets.Base64_Encoded_Pfx }}" )
```

actions/starter-workflows

C# ●

## Deploy your code with these popular services

### Deploy Node.js to Azure Web App

By Microsoft Azure

Build a Node.js project and deploy it to an Azure Web App.

[Set up this workflow](#)

actions/starter-workflows

### Deploy to Alibaba Cloud ACK

By Alibaba Cloud

Deploy a container to Alibaba Cloud Container Service for Kubernetes (ACK).

[Set up this workflow](#)

actions/starter-workflows

### Deploy to Amazon ECS

By Amazon Web Services

Deploy a container to an Amazon ECS service powered by AWS Fargate or Amazon EC2.

[Set up this workflow](#)

actions/starter-workflows

### Build and Deploy to GKE

By Google Cloud

Build a docker container, publish it to Google Container Registry, and deploy to GKE.

[Set up this workflow](#)

actions/starter-workflows

### Deploy to IBM Cloud Kubernetes Service

By IBM

Build a docker container, publish it to IBM Cloud Container Registry, and deploy to IBM Cloud Kubernetes Service.

[Set up this workflow](#)

actions/starter-workflows

### OpenShift

By Red Hat

Build a Docker-based project and deploy it to OpenShift.

[Set up this workflow](#)

actions/starter-workflows

Figure 19.2 – Cloud services in GitHub Actions

- Now, click on **Set up this workflow** under the **.NET** section. Don't click on **Commit** because we will not configure our workflow here; we will let Azure App Service do this for us:

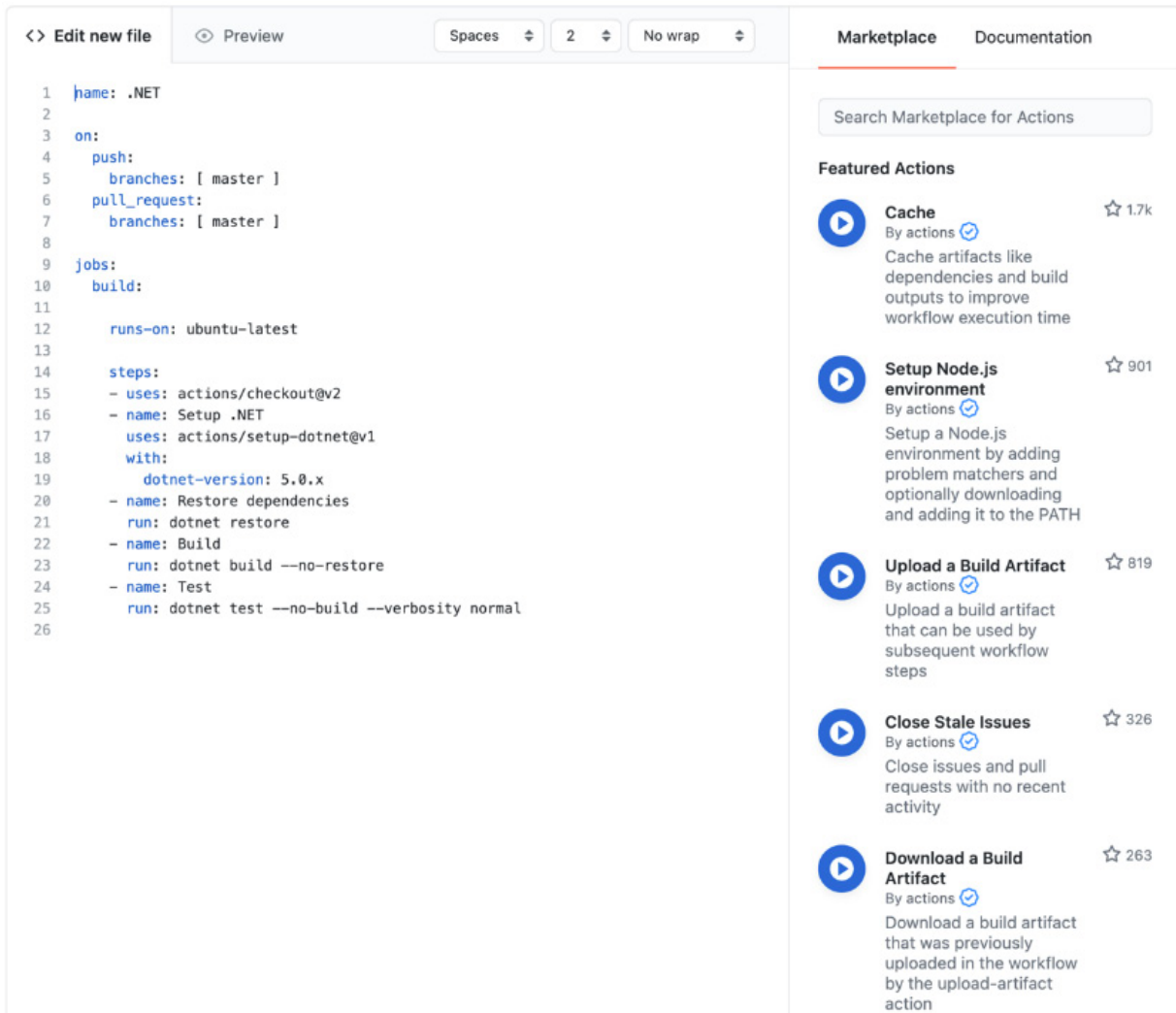


Figure 19.3 – Sample ready-made workflow

The preceding *Figure 19.3* is a pre-configured workflow that we can adjust.

Now let's check out the workflow file.

## Syntax of the workflow file

Let's go and learn the basic syntax of the workflow file to understand how to write our configuration of the workflow.

The **optional name** attribute is where you describe what the workflow is doing.

**on** is where we declare events. There are **push** and **pull\_request** events already in place here. **push** triggers the jobs every time someone pushes to the master branch, and **pull\_request** triggers the jobs again every time a **pull** request gets created with the master branch. An excellent example is to test our application to ensure that the pull request is mergeable. You can use more events by going to



<https://docs.github.com/en/actions/reference/events-that-trigger-workflows> to see the detailed explanations and usages of different events.

Now let's move on to jobs. A job contains a sequence of tasks called **steps**.

## Jobs

**Jobs** are groups of actions. There can be one or more jobs you can define in a workflow file; for instance, one for **build**, one for **test**, and another one for **deploy**. The jobs could be arbitrary, like the name of the workflow. The example runs on **ubuntu-latest**, which is the agent that runs on GitHub. The first step or task is **actions/checkout@v2** and it checks out the code before running a build or a test.

The actions path in **actions/checkout@v2** means that it is one of the predefined actions hosted in GitHub. This is amazing because we don't have to create the action ourselves. But wait, there's more. You can go to <https://github.com/actions> to see the list of different repositories containing all the actions you might need. You can also check out the documentation of actions (<https://docs.github.com/en/actions> with some code examples).

The second step uses the action called **setup-dotnet**, which is another repository in this actions list. This action essentially prepares your environment with a specific version of **dotnet** defined in the file. Unlike in Jenkins, you don't have to install or configure anything in GitHub Actions by specifying what you want in your environment.

As you notice, you use actions by defining the actions in the uses. To run commands, you can use the **run** attribute. In the sample workflow, the step for **build**, also known as the **build step**, triggers the **dotnet build --no-restore** command, and the step for **test**, also known as the **test step**, runs the **dotnet test --no-build --verbosity normal** command.

All the steps are done in the same environment in the flow where your code gets checked out, the dotnet version gets installed, and then you call **dotnet build** and **dotnet test** in the same environment.

Moving on, let's now create an Azure App Service in Azure Portal.

## Creating an Azure App Service instance in the Azure portal

We are going to create an instance of Azure App Service so that we can easily deploy our app to Azure:

1. Go to Azure App Service in <https://portal.azure.com/#home>:

Subscription \* ⓘ

Resource Group \* ⓘ

Microsoft Azure Sponsorship

(New) traveltour-rg

Create new

Instance Details

Name \*traveltour✓.azurewebsites.net

Publish \*☒ Code ☐ Docker Container

Runtime stack \*.NET 5 (Early Access)

Operating System \*☒ Linux ☐ Windows

Region \*Central US

ⓘ Not finding your App Service Plan? Try a different region.

App Service Plan

App Service plan pricing tier determines the location, features, cost and compute resources associated with your app. [Learn more](#) ⓘ

Linux Plan (Central US) \* ⓘ

(New) ASP-traveltourrg-b0c2

Create new

Sku and size \*Premium V2 P1v2  
210 total ACU, 3.5 GB memory  
[Change size](#)

Figure 19.4 – Creating an App Service instance in Azure

The preceding *Figure 19.4* shows the details of the App Service instance we are going to create. We will publish the app using code and the .NET 5 runtime. Hit the **Create** button that you will see after adding the configuration.

2. Now go to the Azure App Service instance that has been created and click on **Deployment Center**:

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Security

Events (preview)

**Deployment**

Quickstart

Deployment slots

Deployment Center (Classic)

**Deployment Center**

**Settings**

Configuration

Authentication / Authorization

Authentication (preview)

Application Insights

Identity

Backups

[Logs](#) [Settings](#) [FTPS credentials](#)

Logs will refresh in 17 seconds

Time	Commit ID	Commit A...	Status
------	-----------	-------------	--------

**CI/CD is not configured**

To start, go to Settings tab and set up CI/CD.

[Go to Settings](#)

Figure 19.5 – The deployment center of App Service

Figure 19.5 shows that **CI/CD is not configured**. Let's configure the CI/CD by going to the settings.

3. Add the private GitHub repository you created earlier:

Save

Discard

Browse

Manage publish profile

Refresh

Redeploy/Sync

LogsSettings \*FTPS credentials

You're now in the production slot, which is not recommended for setting up CI/CD. [Learn more](#)

Deploy and build code from your preferred source and build provider. [Learn more](#)

Source \*

GitHub

Building with GitHub Actions. [Change provider.](#)

GitHub

If you can't find an organization or repository, you may need to enable additional permissions on GitHub.

Signed in aswebmasterdevlin [Change Account](#)

Organization \*

webmasterdevlin

Repository \*

a-travel-tour-app

Branch \*

master

Build

Runtime stack \*

.NET

Version \*

.NET 5

Workflow Configuration

File with the workflow configuration defined by the settings above.

Preview file


Figure 19.6 – Adding the private GitHub repository

Figure 19.6 shows how we can add the private .NET project we created earlier. We are also using .NET 5 as our runtime.

- Then, click **Preview file** to see the workflow:

# Workflow Configuration

File path: .github/workflows/master\_traveltour.yml

 If an existing workflow configuration exists, it will be overwritten.



```
# Docs for the Azure Web Apps Deploy action:
https://github.com/Azure/webapps-deploy
# More GitHub Actions for Azure:
https://github.com/Azure/actions

name: Build and deploy ASP.Net Core app to Azure Web App -
traveltour

on:
  push:
    branches:
      - master
  workflow_dispatch:

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@master

      - name: Set up .NET Core
        uses: actions/setup-dotnet@v1
        with:
          dotnet-version: '5.0.x'

      - name: Build with dotnet
        run: dotnet build --configuration Release

      - name: dotnet publish
        run: dotnet publish -c Release -o ${env.DOTNET_ROOT}/myapp

      - name: Deploy to Azure Web App
        uses: azure/webapps-deploy@v2
        with:
          app-name: 'traveltour'
          slot-name: 'production'
          publish-profile: ${secrets.AzureAppService_PublishProfile_5323e3c8c4774ce0ab0cd
a8f54e1436e }
          package: ${env.DOTNET_ROOT}/myapp
```

Close

Figure 19.7 – Workflow YAML configuration file

Figure 19.7 shows five steps in the workflow YAML configuration file that App Service created for us. Since the YAML configuration file is part of the code or repository, we should not put plain text credentials in the file. We can use placeholders that will reference secrets. I'll show you later where the secrets are located in GitHub.

5. Now click **Save** and go to the project's GitHub repository.
6. Let's check out the **Actions** tab to see whether App Service could add the workflow file:

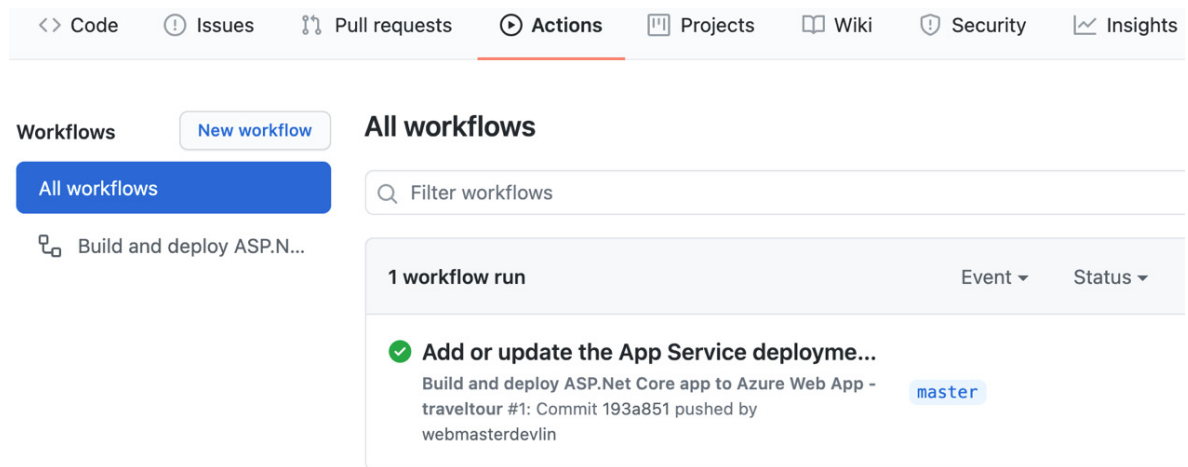


Figure 19.8 – Workflow run

Figure 19.8 shows a workflow with a green icon, which means that the application has been built successfully and deployed after waiting for a few seconds.

7. Let's click the workflow to see the jobs summary:

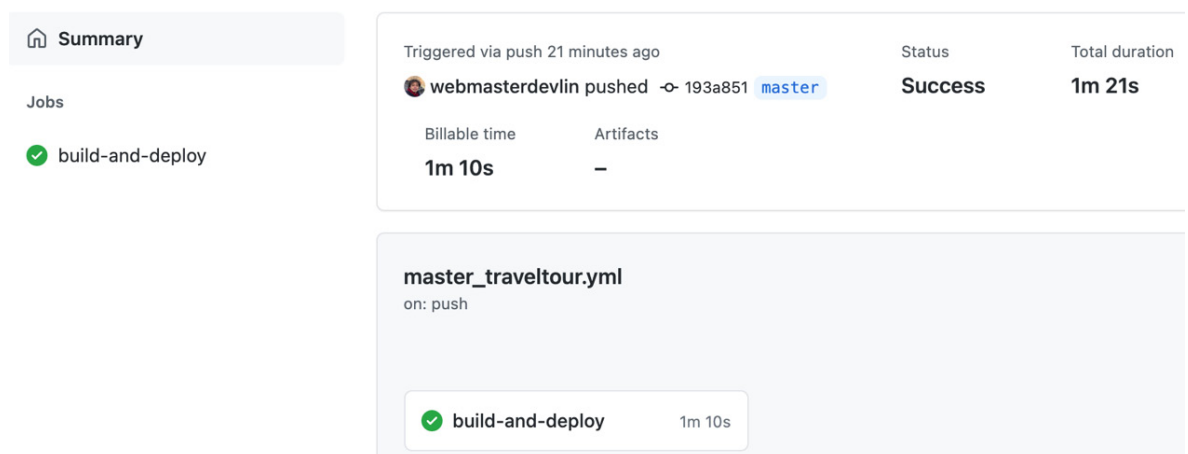


Figure 19.9 – Summary of jobs

Figure 19.9 shows the status and total duration of the workflow.

8. Now let's click on **build-and-deploy** to see the steps associated with the workflow job:

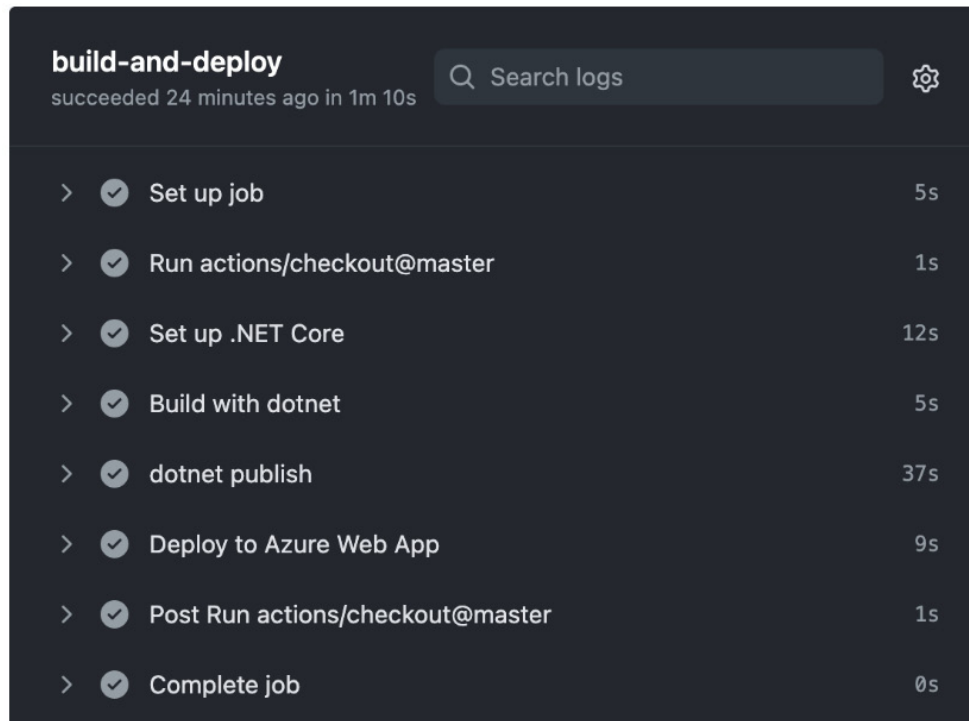


Figure 19.10 – A build-and-deploy workflow job

Figure 19.10 shows the five steps of the workflow job. The steps are as follows:

**Run actions/checkout@master**

**Set up .NET Core**

**Build with dotnet**

**dotnet publish**

**Deploy to Azure Web App**

You will also notice that there are also steps before and after running the configuration file's five steps.

9. Now go to App Service and click the **Overview** menu:

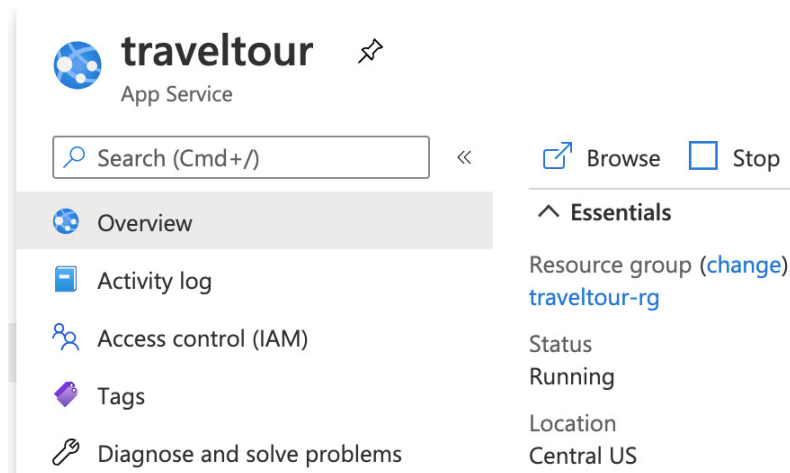


Figure 19.11 – App Service overview

Figure 19.11 shows an overview of App Service.

10. Now click the **Browse** link that will redirect us to the application's URL. You should see that Vue.js is running in the browser:

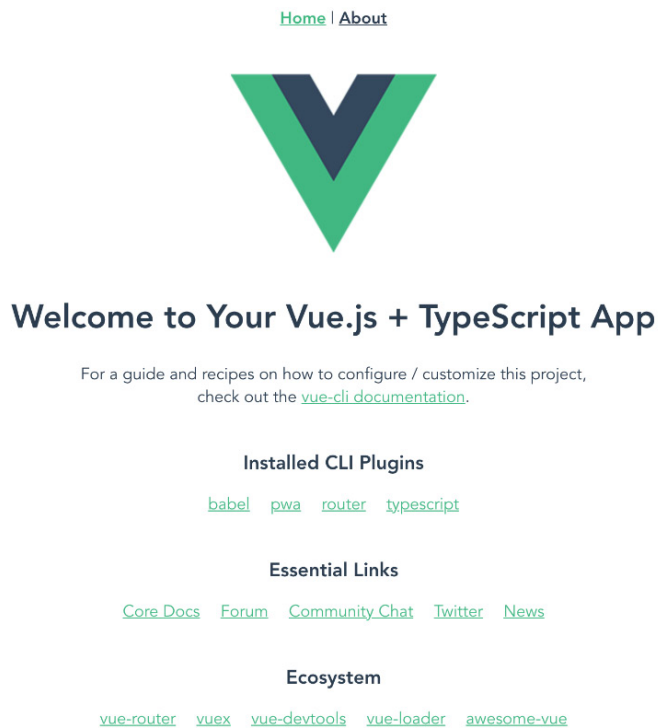


Figure 19.12 – Vue.js + TypeScript App

Figure 19.12 shows **Vue.js + TypeScript App**, which we created earlier.

11. Let's check out the end of the URL bar to see whether we can install the application:



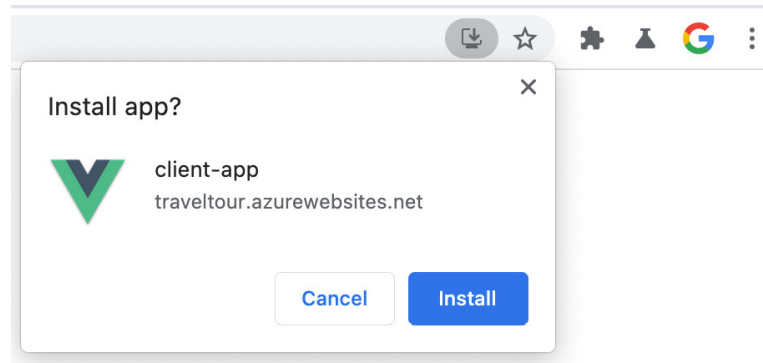


Figure 19.13 – The installable PWA Vue.js app

*Figure 19.13* shows that the Vue.js app is installable, meaning it is a PWA app and will still work in offline mode.

12. Now, go to the **Settings** tab of the GitHub repository and click the **Secrets** menu:

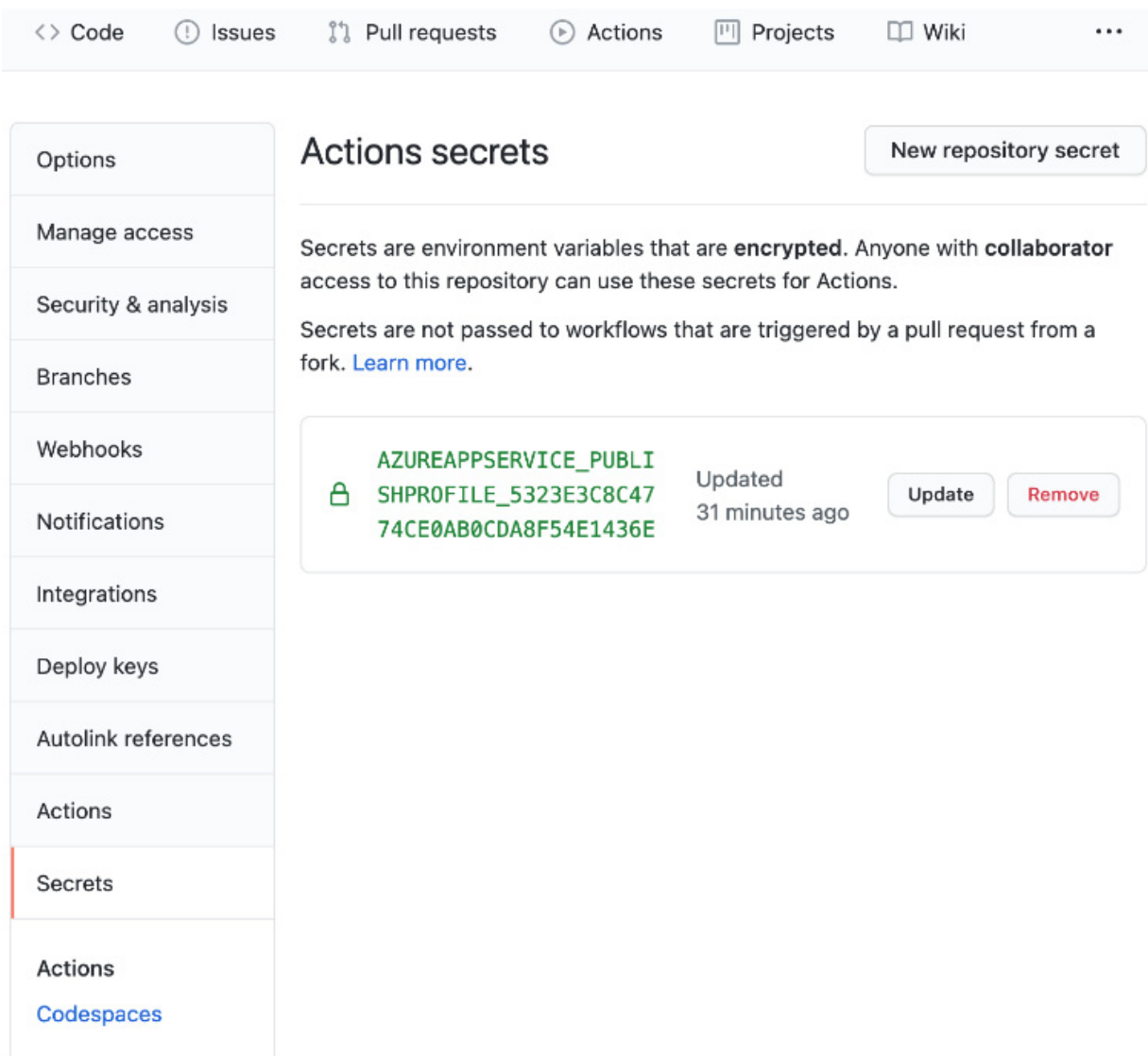


Figure 19.14 – Actions secrets

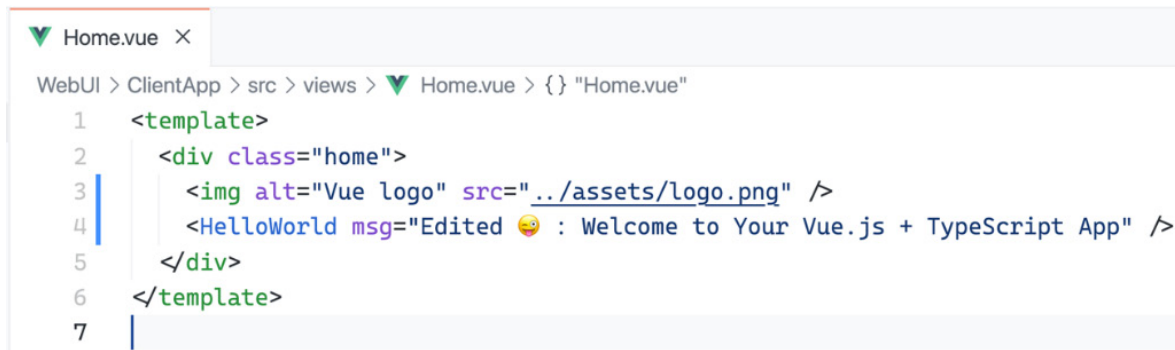
Figure 19.14 shows where we can safely keep the sensitive keys and secrets of our applications.

Now let's test the CI/CD capability of GitHub Actions and the workflow that we have set up.

## Testing the CI/CD capability of GitHub Actions and the workflow

Here, we will see whether our CI/CD configuration and set up in GitHub Actions will satisfy our needs:

1. Run the **git pull** command to bring the changes in your GitHub repository down to your local machine.
2. Update the **Welcome to Your Vue.js + TypeScript App** text or message like so:

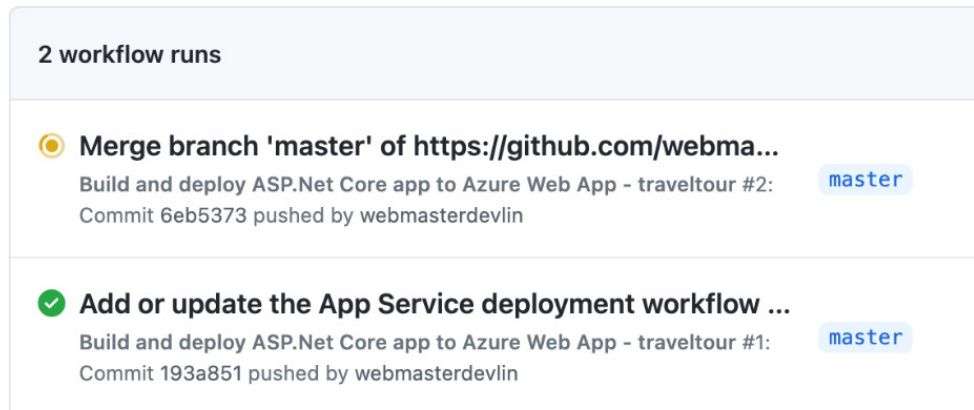


```
WebUI > ClientApp > src > views > Home.vue > {} "Home.vue"
1 <template>
2   <div class="home">
3     
4     <HelloWorld msg="Edited 😊 : Welcome to Your Vue.js + TypeScript App" />
5   </div>
6 </template>
7
```

Figure 19.15 – Edited HelloWorld msg props

Figure 19.15 shows the edited **msg** props of the **HelloWorld** component in the **Home.vue** file.

3. Next, we commit and push your changes to the master. Go to GitHub Actions to see that the workflow was triggered. Wait for the build and deploy job to finish:



2 workflow runs

- 🟡 Merge branch 'master' of https://github.com/webma...  
Build and deploy ASP.Net Core app to Azure Web App - traveltour #2: master  
Commit 6eb5373 pushed by webmasterdevlin
- ✅ Add or update the App Service deployment workflow ...  
Build and deploy ASP.Net Core app to Azure Web App - traveltour #1: master  
Commit 193a851 pushed by webmasterdevlin

Figure 19.16 – Testing CI/CD

Figure 19.16 shows that the CI functionality is working, followed by the CD functionality after a few seconds.

4. Open a new tab in the browser and go to the web app's URL to see that the changes we made in the application are now reflected on the internet:

[Home](#) | [About](#)



## Edited 🤪 : Welcome to Your Vue.js + TypeScript App

For a guide and recipes on how to configure / customize this project,  
check out the [vue-cli documentation](#).

### Installed CLI Plugins

[babel](#) [pwa](#) [router](#) [typescript](#)

### Essential Links

[Core Docs](#) [Forum](#) [Community Chat](#) [Twitter](#) [News](#)

### Ecosystem

[vue-router](#) [vuex](#) [vue-devtools](#) [vue-loader](#) [awesome-vue](#)

Figure 19.17 – Showing the updated Vue.js + TypeScript app

*Figure 19.17* shows the **Edited** word and the emoji we included in the changes.

5. Now, go to `{yourSubDomain}.azurewebsites.net/weatherforecast` to see whether the Web API's **WeatherForecast** endpoint can respond with JSON data:

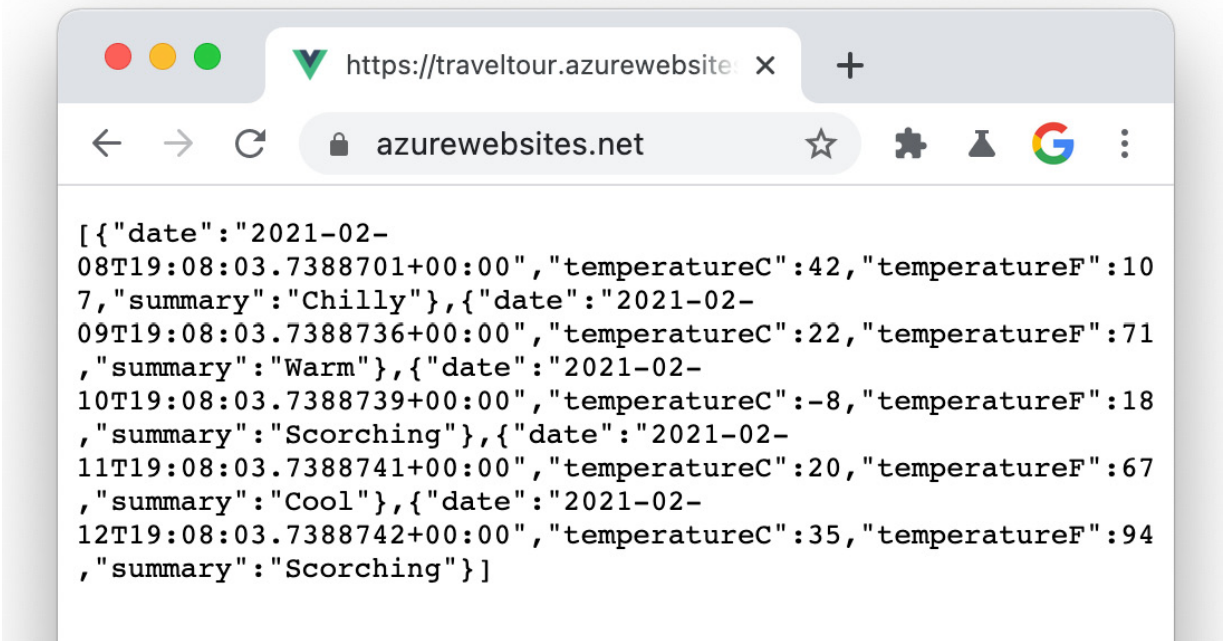


Figure 19.18 – Response from the WeatherForecast endpoint

Figure 19.18 shows that the **WeatherForecast** endpoint responds with an array in JSON format.

The Vue.js TypeScript and the controller of the ASP.NET Core Web API project are working correctly in CI/CD using GitHub Actions.

Now let's summarize what you have learned.

## Summary

That's a wrap. Here are some takeaways from this chapter. You learned what GitHub Actions is and how easy deployment can be using GitHub Actions. You learned the right time and use cases to deploy to Azure App Service, Azure Functions, Azure SWA, and Azure AKS. You also learned how to deploy to Azure App Service using GitHub Actions, including the workflow's YAML configuration, the available actions, and CI/CD.

So you've made it this far. Thank you for finishing the book, and I am proud of you and your enthusiasm for learning new tools and things. You can apply what you have learned here in a project, given the requirements of your project match the problems and solutions you have learned from the book.

The course has taught you how to architect an ASP.NET Core application and a Vue.js application like a senior developer, bringing value to your customers or clients.

The next step that I will suggest for you is to get a new Packt book about a standalone ASP.NET Core or Vue.js topic to solidify what you have learned from this book.

On behalf of the whole Packt team and editors, we wish you all the best in all stages of your career and life.

Hi!

I am Devlin Duldulao, author of ASP.NET Core and Vue.js. I really hope you enjoyed reading this book and found it useful for increasing your productivity and efficiency in ASP.NET Core and Vue.js.

It would really help me (and other potential readers!) if you could leave a review on Amazon sharing your thoughts on ASP.NET Core and Vue.js.



Your review will help me to understand what's worked well in this book, and what could be improved upon for future editions, so it really is appreciated.

Best Wishes,

Devlin Basilan Duldulao





[Packt.com](https://packt.com)

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

## Why subscribe?

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

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

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

## Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



# Vue.js 3 By Example

Blueprints to learn Vue web development, full-stack development, and cross-platform development quickly

John Au-Yeung



## Vue.js 3 By Example

John Au-Yeung

ISBN: 978-1-83882-634-5

- Get to grips with Vue architecture, components, props, directives, mixins, and other advanced features
- Understand the Vue 3 template system and use directives

- Use third-party libraries such as Vue Router for routing and Vuex for state management
- Create GraphQL APIs to power your Vue 3 web apps
- Build cross-platform Vue 3 apps with Electron and Ionic
- Make your Vue 3 apps more captivating with PrimeVue
- Build real-time communication apps with Vue 3 as the frontend and Laravel

# Blazor WebAssembly by Example

A project-based guide to building web apps with  
.NET, Blazor WebAssembly, and C#

Toi B. Wright

Foreword by Scott Hanselman, Partner Program Manager at Microsoft



## **Blazor WebAssembly by Example**

Toi B. Wright

ISBN: 978-1-80056-751-1

- Discover the power of the C# language for both server-side and client-side web development
- Use the Blazor WebAssembly App project template to build your first Blazor WebAssembly application
- Use templated components and the Razor class library to build and share a modal dialog box
- Understand how to use JavaScript with Blazor WebAssembly
- Build a progressive web app (PWA) to enable native app-like performance and speed
- Understand dependency injection (DI) in .NET to build a shopping cart app
- Get to grips with .NET Web APIs by building a task manager app

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](https://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.

## Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt.

Thank you!