

ASP.NET Core 2 and Vue.js

Full Stack Web Development with Vue, Vuex, and ASP.NET Core 2.0



Packt

www.packt.com

By Stuart Ratcliffe

www.EBooksWorld.ir

ASP.NET Core 2 and Vue.js

Full Stack Web Development with Vue, Vuex, and ASP.NET
Core 2.0

Stuart Ratcliffe



BIRMINGHAM - MUMBAI

ASP.NET Core 2 and Vue.js

Copyright © 2018 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.

Commissioning Editor: Amarabha Banerjee
Acquisition Editor: Siddharth Mandal
Content Development Editor: Aishwarya Gawankar
Technical Editor: Prajakta Mhatre
Copy Editor: Safis Editig
Project Coordinator: Sheeja Shah
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Production Coordinator: Arvindkumar Gupta

First published: July 2018

Production reference: 1270718

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

ISBN 978-1-78883-946-4

www.packtpub.com

I would like to dedicate this book to everyone who helped and supported me in writing this book: to my partner, Laura, for her unconditional support and encouragement, especially when my motivation was at its lowest; to my parents, Liz and Brian, for always being there when I needed them; to my dogs, Jack and Miska, for putting up with a severe lack of daily walks while this book was being written; to all the Packt editors and technical reviewers who made this possible in the first place; and finally, to my late grandmother, May, who passed away just a few short months before this book could be published.



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 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
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

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

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

Contributors

About the author

Stuart Ratcliffe is a professional software developer who lives and works in the East Midlands, UK. He has held positions at some of the largest IT companies in the world, working on high-profile projects for the UK government. Currently, he has been working on track-and-trace systems for medical instruments that undergo sterilization. He holds a Tech Lead position on the digital side of a healthcare company, building both web and mobile applications to support the clinical side of the business. He is a full-stack .NET developer who loves to learn new technologies.

About the reviewer

Daniel Jiménez García is a software developer with more than 12 years of experience. His journey began with C# and VB6, continued with several iterations of Microsoft technologies and web frameworks such as Backbone, and finished with Node, Vue, Docker, and ASP.NET Core.

Nowadays, he is working as a tech lead for Oliver Wyman Labs, building web and mobile applications, mentoring fellow team members, and driving their common stack, tools, and architecture for web development.

Being a regular contributor to the DotNetCurry magazine, he has written 18 articles on a range of topics including ASP.NET Core and Vue.js.

I would like to thank my friends and family for their support, love, and friendship, especially my parents, who always encouraged and inspired me to become a better person. Thanks to my colleagues for the hours of joy, frustration, debate, and learning, not always about code or near a computer! Finally, my thanks to everyone who shared their knowledge and passion, so others could learn and enjoy our profession, and be inspired.

Packt is searching for authors like you

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

Table of Contents

Preface	1
Chapter 1: Understanding the Fundamentals	10
Thinking in components	11
What is a component?	12
UI composition	14
Introduction to Vue	16
The Vue instance	16
Attaching to the DOM	16
Defining data properties	16
Rendering data into the DOM using expressions	17
Building component trees	18
Reactivity	19
Component behavior	19
State	20
Props	21
Methods	23
Computed properties	23
Watchers	25
Lifecycle hooks	26
Component presentation	28
Directives	28
Attribute binding with v-bind	29
Conditional display with v-show	31
Control flow with v-if and v-else	31
Rendering lists with v-for	33
Event handling with v-on	34
Form input binding with v-model	36
Parent-child component communication	37
ASP.NET Core – what's new?	39
Middleware pipeline	39
Application startup	41
DI is a first-class citizen	42
EF Core – what's new?	43
Configuring relationships	43
Global query filters	44
Compiled queries	45
In-memory provider for testing	45
Summary	47
Chapter 2: Setting Up the Development Environment	48

Choosing a web browser	48
Installing frontend tools and dependencies	49
Installing Node and npm	49
Installing Vue	51
npm or Yarn?	51
Installing backend tools and dependencies	52
Installing ASP.NET Core	52
Installing PostgreSQL	53
Choosing and installing an IDE	54
Productivity tools	55
Installing VS Code extensions	55
Installing the Vue.js Chrome devtools extension	57
Installing a Terminal Emulator on Windows (optional)	57
Summary	57
Chapter 3: Getting Started with the Project	58
ASP.NET Core SPA templates versus CLI tools	59
An introduction to webpack	60
What is webpack?	60
How does it work?	60
Basic webpack configuration	61
Bundle splitting	62
Production bundles	63
Scaffolding a project with the dotnet CLI	63
Refactoring the frontend setup	64
Removing TypeScript	65
Replacing the default components	66
Refactoring the backend setup	68
Refactoring to a feature folder structure	70
Setting up the database	71
Creating a database context	71
Registering the database context for DI	73
Creating the database	74
Creating an initial migration	76
Creating and seeding the database on start-up	78
Testing the completed setup	80
Summary	83
Chapter 4: Building Our First Vue.js Components	84
Displaying a list of products	84
Conditional rendering	89
Component composition	91
Client-side routing	101
Fetching data from an API	108
Summary	114

Chapter 5: Building a Product Catalog	115
Improving the existing UX	116
Choosing a UX framework	116
What is Bootstrap-Vue?	116
Installing additional required dependencies	118
Modifying the webpack configuration to support SASS	118
Updating the webpack vendor configuration	119
Rebuilding the vendor bundle	120
Adding application-wide layout elements	121
Adding application-wide styles	123
Styling the product list and product details components	123
Fetching data before navigation	127
Adding a page loading indicator	129
Adding a transition on page change	131
Extending the existing data model	133
Dropping the existing database	133
Adding new/updating existing entities	134
Updating the DbContext class	139
Creating a migration to reflect the model changes	139
Updating the application's seed data	140
Filtering on the server	143
Updating controller actions to support filtering	144
Testing our filtering logic	146
Filtering on the client	147
Installing additional dependencies	147
Installing Font Awesome	148
Installing additional npm packages	148
Building an accordion component	150
Defining the accordion template structure	150
Defining the accordion behavior	151
Styling the accordion component	153
Building the filters component	154
Scaffolding the filters component template	154
Adding a brand filter	155
Adding a price filter	156
Adding a screen size filter	157
Adding the remaining color, OS, and feature filters	158
Scaffolding the filters component behavior	159
Defining the filters component computed properties	160
Defining the filters component methods	161
Styling the filters component	167
Adding the filters component to the catalog page	167
Updating the catalog page template	167
Adding the catalog page filter behavior	168
Tidying up our existing components	172
Testing the completed filtering logic	174
Refactoring the filters component	175

Highlighting duplication in our existing implementation	175
Extracting a common multi-select filter component	176
Extracting a common range filter component	180
Rendering the new multi-select and range filter components	182
Testing that everything still works	185
Client-side sorting	185
Building a sort component	186
Adding the sort component to the catalog page	188
Creating a search bar component	190
Triggering API requests using watchers	194
Debouncing API requests to limit how often they fire	195
Summary	196
Chapter 6: Building a Shopping Cart	198
Evaluating our options	198
Persisting to the database	199
Persisting to session state	199
Persisting to local storage	200
Finishing the product details page	200
Creating the gallery component	206
Adding variants to the product details component	211
Introduction to Vuex	213
What is Vuex?	214
How does Vuex work?	215
Mutations	216
Actions	216
Getters	216
Putting it all together	217
Installing and configuring Vuex	219
Adding products to the cart	221
Creating the mutations	221
Creating an action	223
Creating a shopping cart page	229
Creating a CartItem component	229
Displaying the list of cart items	232
Creating a currency filter	234
Removing products from the cart	236
Updating cart items	238
Adding a getter to display the cart total	240
Creating a cart summary component	241
Persisting the cart to local storage	244
Improving the UX with add to cart feedback	247
Summary	248
Chapter 7: User Registration and Authentication	249
Adding JWT authentication to the API	249

Why JWTs?	250
Configuring JWT authentication	251
Issuing JWTs	254
Adding user role support	258
Testing JWT authentication	260
User registration	264
Authentication and user registration in the client app	267
Vuex state properties for authentication	267
Vuex mutations for managing authentication state	268
Vuex authentication getters	269
Vuex login, register, and logout actions	270
Authentication modal component	272
Login form component	275
Register form component	278
Auth navigation item component	280
Wiring up the new components in App.vue	283
Protecting pages with navigation guards	284
Setting the authentication state on app startup	286
Summary	288
Chapter 8: Processing Payments	290
Why use Stripe?	290
Simple PCI compliance	291
Easy integration	291
Excellent dashboard	291
Getting started with Stripe and client-side validation	292
Registering for a Stripe account	292
Including the Stripe checkout JavaScript library	292
Installing VeeValidate for client-side validation	293
Building the checkout components	293
Building a cart summary component	295
Building a checkout form component	297
First look at client-side validation	298
Finishing the delivery address form fields	300
Capturing payment information	301
Initializing Stripe elements	302
Validating form input state	303
Verifying payment details with Stripe	305
Submitting the order to the API	306
Adding basic Bootstrap styling to Stripe elements	309
Building a checkout success component	309
Building a my account page	311
Building the OrderList component	313
Formatting dates with a reusable date filter	315
Linking to the my account page	316
Fixing the register form component	316

Server-side payment processing	318
Adding orders to the data model	318
Owned entity types in EF Core 2.0	321
Why use owned entity types?	321
Defining an owned type	321
Configuring owned types	322
Creating the orders migration	323
Installing and configuring the Stripe.net NuGet package	323
Configuring Stripe	324
Processing orders and payments	325
Persisting the order object	326
Calculating the total order price	327
Processing the payment with Stripe	328
Adding an order list API endpoint	330
Summary	332
Chapter 9: Building an Admin Panel	334
Extending the authentication endpoint with user roles	335
Client-side role-based authorization	336
Adding role checks to client-side routes	337
Server-side role-based authorization	340
Hiding UI elements based on role	340
Building the admin panel components	342
Configuring nested route definitions	343
Refactoring components for reuse	345
Product list component	349
Creating a product form component	351
Creating an add variant modal component	362
Vue component inheritance	364
Defining a form input base component	366
Inheriting from a base component	368
Building custom input controls	372
Building a custom typeahead control	372
Building a multi-select control	378
Persisting new products to the database	384
Creating a slug generator	384
Creating the API endpoint	385
Remote validation with Vee-Validate	389
Making our app aware of the new custom validation rule	391
Creating the validation API endpoint	392
Tidying things up	392
Linking to the admin panel	393
Fixing a logout bug	394
Fixing a bug by selecting a product variant	394
Summary	400

Chapter 10: Deployment	401
Registering for an Azure account	401
Setting up an Azure environment	403
Understanding Azure subscriptions and resources	404
Creating a subscription and resource group	405
Creating a database	406
Creating an app service	408
Configuring environment variables	413
Preparing the application for deployment	419
Configuring multiple database providers	419
Tweaking the post-publish build steps	421
Configuring Git deployments	423
Finalizing the apps configuration	426
Enabling logging in Azure	426
Forcing HTTPS connections only	430
Summary	431
Chapter 11: Authentication and Refresh Token Flow	432
Understanding refresh tokens	432
What are refresh tokens used for?	432
What are refresh tokens?	434
Why use refresh tokens?	434
Adding refresh token support to the backend	435
Extending the AppUser model	435
Generating refresh tokens	437
Refreshing JWT access tokens	438
Finishing up	440
Adding refresh token support to the frontend	441
Extracting router configuration into separate files	442
Refreshing access tokens with axios interceptors	446
Finishing up	450
Summary	452
Chapter 12: Server-Side Rendering	453
Why use SSR in the first place?	454
Search engine optimization	455
Performance	455
How does SSR work?	456
The easy way – Nuxt.js	457
Preparing the application for SSR	458
Installing npm packages required for SSR	458
Adding Vuex actions and mutations for all API requests	459
Defining additional Vuex actions	460
Defining the additional Vuex mutations	461
Defining the additional store state properties	462

Updating existing pages to use Vuex	462
Refactoring the catalog page	463
Refactoring the product details page	465
Refactoring the account page	465
Refactoring the orders admin page	466
Refactoring the products admin page	466
Refactoring the create product admin page	467
Changing the way we persist user authentication state	468
Changing our approach of persisting state to local storage	468
Storing authentication state in cookies	470
Setting up and configuring SSR	471
Defining the shared boot logic	471
Defining the client-specific boot logic	474
Hydrating the client-side store	474
Loading shopping cart data from local storage	475
Pre-fetching component data	477
Remembering our promises	478
Defining the server-specific boot logic	479
Deleting the old boot file	481
Making webpack aware of the client/server boot files	481
Defining a shared webpack configuration object	481
Defining client- and server-specific webpack configuration objects	483
Updating the vendor webpack configuration to include SSR libraries	484
Enabling SSR	484
Conditionally rendering elements that rely on the browser	487
Fixing the range filter component	488
Fixing the checkout form component	489
Fixing page transition animations in the router	490
Fixing the store subscription to persist cart items to local storage	490
Testing our server-rendered application	491
Summary	493
Chapter 13: Continuous Integration and Continuous Deployment	494
CI/CD – why bother?	494
Continuous integration	495
Continuous deployment	495
Disabling Azure app service Git deployments	495
Getting started with VSTS	497
Creating a VSTS account	497
Setting up a team services project	498
Building a CI/CD pipeline	500
Setting up a VSTS build	500
Enabling CI	505
Setting up a VSTS release	505
Enabling CD	515
Summary	516

Table of Contents

Other Books You May Enjoy	517
Index	520

Preface

These days, it is rare to see a standard web application with little to no client-side code involved. Developers are usually burdened with writing masses and masses of JavaScript, often using libraries such as jQuery, to create rich and responsive user interfaces. As an application like this grows, it is incredibly easy for this client-side code to become unwieldy and a nightmare to maintain. This style of web application also relies on the DOM to store the application's current state, and we end up writing a lot of code to micromanage and manipulate the DOM in order to correctly display that state.

Imagine a web application using nothing but jQuery on the client, where we have a single piece of data that we need to display in multiple places within a UI. Now think about how we'd need to go about changing that piece of data, and ensuring that the DOM represents the correct value in every location where we are displaying it. Up to a handful of places is perhaps not a big deal; we could write a bunch of similar lines of jQuery that manually go and update each UI element currently displaying that piece of data, and just make sure we don't forget any.

By now, you should be able to see the problem we're trying to solve. This way of developing web applications is incredibly error-prone for anything but the most simple user interfaces. However, this is 2018 and we aren't limited to writing our client applications with jQuery any more. There are countless SPA frameworks dedicated to building UIs for our web applications that automatically react to data changes and handle the displaying of that data in the DOM for us. We no longer need to write countless lines of code with a single purpose of changing the value of a DOM element.

We don't manipulate the DOM with these frameworks. Instead, we manipulate the data. The DOM simply acts as a means of displaying that data to our users. Let's go back to the preceding example, where we have a single piece of data being displayed in 10 different UI elements. When that piece of data changes, instead of manually updating all 10 DOM elements to reflect the change, we simply store it in a JavaScript object and update the value whenever we like. The SPA framework then reacts to these changes, and handles all the heavy lifting for us by updating any DOM element that cares about that piece of data; one simple data change rather than 10 separate DOM manipulation calls.

As with choosing a technology for the frontend of a modern web app, there are also plenty of options when it comes to building a backend. Some of the most popular choices currently include Node.js, PHP, Rails, Golang, and ASP.NET. Node.js is incredibly popular for a number of reasons, most notably for being able to use JavaScript for the whole application. The Laravel framework is arguably one of the only reasons PHP is still a viable option, else it would likely be fading into the background the way Rails is. Golang is a fairly new language that is getting some very good reviews, particularly in the performance benefits it provides over Node. However, due to how new Golang is, there are far fewer packages and frameworks to assist us in building more complex applications. ASP.NET is older than both Node.js and Golang, and, as such, there is no such shortage of packages and frameworks like there is with Go. In fact, the .NET framework has a lot of functionality already built in where you'd normally be reaching for an external package in other languages and frameworks. Even when you do need an external package, there is usually one made by Microsoft themselves, which helps avoid a well-known issue with Node.js applications referred to as "dependency hell". ASP.NET is also based around strongly-typed compiled languages that can provide a number of performance and security benefits over a weakly typed language, such as JavaScript.

ASP.NET has been around since January 2002 when version 1.0 of the .NET framework was released as the successor to Microsoft Active Server Pages. Since then, there have been a multitude of major versions released, the current being version 4.7 at the time of writing. In 2016, Microsoft changed their game entirely, with the release of ASP.NET Core. For the first time in over 14 years, ASP.NET was made both open source and cross-platform in a complete rewrite from the ground up. When version 1.0 of ASP.NET Core was released, there was one potentially significant downside depending on the size and complexity of your application. If you have a requirement to host on a platform other than Windows, you can't target the full .NET framework. The issue with this was a potential lack of necessary APIs that had not yet been ported over to the core CLR. However, version 2.0 has recently been released, and with it comes a compatibility shim that enables .NET Core apps to reference any .NET framework library.

If you've been avoiding the move to ASP.NET Core, then now is a great time to change that. Version 2.0 has added a number of other improvements aside from backward compatibility with .NET framework APIs and libraries: Simplified configuration setup, simplified NuGet package references, and additional SPA project templates, to name but a few. One of Microsoft's most well-known developers, Steve Sanderson (creator of the very popular KnockoutJS framework), has made it his goal to make ASP.NET Core the best backend choice for single-page applications. To achieve that goal; his team has implemented some amazing features to seamlessly integrate frontend and backend builds using ASP.NET Core middleware. These features, along with the latest improvements released in version 2.0, really do make ASP.NET Core a fantastic choice for any web application.

With so many SPA frameworks to choose from, why should we bother with Vue? Most developers with any kind of interest in modern web application development have probably heard of React and Angular, but far fewer will have heard of Vue. Initially created by a single developer, Evan You, and currently developed by a relatively small international team, you could be forgiven for ignoring it in favor of its main competition. After all, React and Angular are developed by the tech giants that are Facebook and Google, respectively.

However, after working at both Google and the Meteor Development Group, Evan You knows a little something about SPA frameworks. Vue was created after both React and AngularJS had some time to be battle tested by thousands of developers around the world. In the eyes of its creators, Vue incorporates the best parts that either of these frameworks had to offer, while also trying to avoid the pitfalls that caused common grievances throughout the community. The result is a very lightweight and focused library dedicated to building UIs only. However, the Vue team has always intended for this library to be incrementally adoptable in any web project, and has provided several supporting libraries that make Vue a fantastic choice for building fully fledged SPAs as well.

In my opinion, Vue is far simpler to learn than most of the alternatives, which makes it a great choice for experienced backend .NET developers looking to branch out into the world of frontend frameworks and modern SPAs. If you already know enough HTML and jQuery to make a standard MVC application, then the template syntax used by Vue won't be much of a problem, and a lot of the syntax can be directly compared to that of Razor. The barrier to entry may be low, and the library itself may be incredibly lightweight, but Vue can be every bit as powerful as any other SPA framework that exists today.

Who this book is for

This book is aimed at ASP.NET developers who are looking for an entry point in learning how to build a modern client-side SPA with Vue.js, or those with a basic understanding of Vue.js who are looking to build on their knowledge and apply it to a real-world application. Knowledge of JavaScript is not necessary, but would be an advantage.

What this book covers

Chapter 1, *Understanding the Fundamentals*, starts by looking at the fundamentals of Vue.js to give readers a basic understanding of the techniques used to build the sample applications later in the book. It discusses some of the benefits of Vue.js, as well as some of the reasons why we'd bother to choose it for building our applications. Finally, it looks at how ASP.NET Core / EF Core differ from their previous counterparts, focusing on the very latest versions of the frameworks, and putting the emphasis on the newest features that you may not know about yet.

Chapter 2, *Setting Up the Development Environment*, walks you through the process of installing and configuring the tools that you'll need to build and run an ASP.NET Core and Vue.js SPA. It takes the cross-platform nature of ASP.NET Core into consideration while evaluating some of the options available to us when selecting a client-side package manager, an IDE, and an RDBMS. Finally, it shows you how to install some productivity tools that make our lives far easier while building Vue.js applications with Google Chrome.

Chapter 3, *Getting Started with the Project*, looks at the options available to you when starting and scaffolding a brand new project with ASP.NET Core and Vue.js. It introduces the basics of what webpack is and how it works, before scaffolding an application that will form the foundations that will be built on for the rest of the book. Finally, it looks at how to refactor the default application structure to meet your own needs and preferences.

Chapter 4, *Building Our First Vue.js Components*, jumps into building a basic product list component, before composing a component structure based on the standard master-details pattern to display more information about a selected product. It then introduces client-side routing by refactoring the UI into separate pages for the product list and details components, before replacing the hardcoded product data with dynamic data fetched from the backend API.

Chapter 5, *Building a Product Catalog*, expands the existing components into a fully featured product catalog, including filtering, sorting, and searching. It also improves the existing look and feel of the application by introducing the Bootstrap CSS framework, as well as adding animations and loading indicators in between page changes. The reader will learn how to identify and extract duplication into common reusable components, as well as how to import and render components from third-party libraries.

Chapter 6, *Building a Shopping Cart*, starts by evaluating the options available to us for persistent shopping cart items. It then introduces Vuex for centralizing client-side state and enabling access to it from multiple components. Readers will then learn how to consume Vuex state by building a shopping cart component, as well as a shopping cart summary component to display it. They will also learn how to create custom Vue.js filters to reduce duplication in presentation logic, as well as how to provide feedback to users by displaying toast messages. Finally, we will see how to quickly and easily persist Vuex state to local storage to make sure that it is available on subsequent visits to the application.

Chapter 7, *User Registration and Authentication*, looks at how to add access control using JWT-based authentication. You will learn how to protect API routes using ASP.NET Core middleware and action filters, as well as how to prevent access to client-side pages using Vue.js router navigation guards. You will also extend the existing Vuex store to include register and login functionality, as well as building the necessary components for consuming it.

Chapter 8, *Processing Payments*, completes the user journey of the customer by implementing a fully functioning checkout page, including payment processing with Stripe. You will learn why Stripe is the perfect library for payment processing in any type of e-commerce website, as well as how to integrate it into a Vue.js client application and ASP.NET Core API. You will also learn how to add rich client-side validation to a custom checkout form component, which provides immediate feedback to the user as they start typing in each field.

Chapter 9, *Building an Admin Panel*, adds the ability to manage the existing product catalog, and add new products to the database. Readers will learn how to reduce duplication by extracting common functionality into a base component and then using component inheritance to extend it. You will build a collection of reusable form input components and then refactor the existing forms to make use of them.

Chapter 10, *Deployment*, completes the first iteration of the application by deploying it to a production cloud environment. We start by registering for a Microsoft Azure account, before learning how to set up and configure our environment to include an app and database server. Readers will then learn how to prepare the application for deployment, including the configuration of multiple database providers to support SQL servers in production, and PostgreSQL in development. They will also learn how to enable logging within Azure, as well as how to force HTTPS connections to increase the security of the application. Finally, we will enable automated Git deployments to publish the application on every push to a specific Git repository.

Chapter 11, *Authentication and Refresh Token Flow*, builds on the existing authentication mechanism by adding refresh token support. You will learn how and why this increases the security of the application, as well as how to implement refresh token flow in an ASP.NET Core API. You will then learn how to add a client-side API request interceptor to automatically refresh users' access tokens as and when they expire, allowing them to remain logged in permanently.

Chapter 12, *Server-Side Rendering*, begins by discussing some of the reasons why you would want to initially render a client-side SPA on the server. It then provides a detailed explanation of how to refactor the application to prepare it for SSR, before showing you how to set up and configure SSR. Finally, it looks at some of the limitations of SSR and how to fix them by conditionally rendering components that are not SSR-compatible, before looking at how to test that everything is working as it should be.

Chapter 13, *Continuous Integration and Continuous Deployment*, introduces a far more robust way of automating the application build and release pipeline using VSTS rather than the existing Azure Git deployment feature. It discusses the reasons, why you would want to use a CI/CD pipeline, and very briefly why VSTS is the perfect choice when building ASP.NET Core applications hosted within Azure. It walks you through the process of setting up a VSTS account, build and release, as well as enabling triggers to automatically build and deploy the application on every push to the existing Git repository.

To get the most out of this book

It is assumed that you are already a reasonably competent ASP.NET web developer, familiar with building MVC web applications. Although not required, you will appreciate the benefits of Vue.js more if you are also familiar with incorporating a moderate amount of jQuery, or vanilla JavaScript, into your MVC applications. It is also assumed that you are familiar and comfortable with basic CSS and SCSS, with knowledge of the Bootstrap framework being desirable. It is not required to have any pre-existing knowledge of .NET Core or ASP.NET Core, but again, it would be beneficial to have explored ASP.NET Core and how it differs from previous versions of the framework. Finally, it is assumed that you are familiar with source control using Git, and will have access to a cloud-based Git repository to use for deployment in Chapter 10, *Deployment*, and Chapter 13, *Continuous Integration and Continuous Deployment*.

All software requirements necessary for completing the sample application are introduced as and when required, including links to instructions on how to install them on your native OS. It is possible to build and run the sample application on any OS supported by ASP.NET Core, including both Windows and macOS. However, the application has only been tested by myself on a Windows machine, as it is assumed that this is what the vast majority of readers will be using, based on the assumption that they are experienced ASP.NET developers.

Download the example code files

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

You can download the code files by following these steps:

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

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

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

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Hands-On-ASP.NET-Core-2-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!

Conventions used

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

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "In this case, we return a simple object with a single `name` property."

A block of code is set as follows:

```
<style lang="scss">
html,
body {
  height: 100vh;
}
div.app,
div.page {
  height: 100% !important;
}
</style>
```

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

```
vendor: [
  "event-source-polyfill",
  "isomorphic-fetch",
  "vue",
  "vue-router",
  "bootstrap/dist/css/bootstrap.min.css",
  "bootstrap-vue",
  "nprogress/nprogress.css"
]
```

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

```
webpack --config webpack.config.vendor.js
```

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: "To do so, click on the **Resource groups** link in the main menu on the left."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@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/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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

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

Reviews

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

For more information about Packt, please visit packtpub.com.

1 Understanding the Fundamentals

Many modern web applications are now built as client-rendered **Single-Page Applications (SPAs)** rather than traditional server-rendered multipage applications. An SPA is a web application that only contains a single physical HTML page. This single page then uses JavaScript within a web browser to dynamically rewrite parts of the HTML, usually based on JSON data that's retrieved from API calls to the server. By doing so, after the initial page load, the application has no need to request full HTML pages from the server, which helps make it as fast and responsive as a native desktop application.

When it comes to building SPAs in 2018, we are absolutely spoiled for choice in frameworks and technologies that we could potentially use to help us build our apps. Regardless of their chosen technology stack, most web developers will probably have extensively used jQuery for their client-side JavaScript needs, and there is no reason why it couldn't be used to build SPAs as well. However, there are far better frameworks that are specifically designed to help us build modern SPAs.

Vue.js is a JavaScript framework for building the view layer of your applications. However, it differs from other large frameworks because it's designed to be incrementally adoptable. That is, with a single CDN script reference, you can plug Vue into a small portion of an existing application in much the same way as you would with jQuery. On the other hand, you could opt to use the modern tooling and supporting libraries in Vue's ecosystem to build a fully-fledged SPA from scratch. Vue really is one of the most simple yet powerful frameworks, with very little compromise in return.

ASP.NET Core is the latest version of Microsoft's ASP.NET web development framework. It has been completely rewritten to be more lightweight and modular, as well as to offer official cross-platform support for the first time. If you are reading this book, it is reasonably safe to assume that you are most likely already an experienced ASP.NET developer, with minimal experience of frontend frameworks and technologies. However, if you already have a basic understanding of Vue.js, then you may be able to skip this chapter, as it will be aimed primarily at those with no experience at all. Regardless of your level of experience with Vue, if you are already familiar with ASP.NET Core, then at the very least you can skip the ASP.NET Core sections near the end of this chapter. That being said, I'll only be focusing on areas where ASP.NET Core differs from previous versions of ASP.NET.

In summary, we'll cover the following topics:

- What are components?
- How do we compose a UI using components?
- Client-side application state
- Fundamental Vue concepts such as props, methods, computed properties, and directives
- What's new in ASP.NET Core?
- What's new in EF Core?

Thinking in components

Building web applications using Vue or any other JavaScript SPA framework revolves around the concept of breaking the UI down into the smallest possible chunks of functionality. These chunks are referred to as *components*, and can be likened to Razor view components, tag helpers, and partial views in ASP.NET Core MVC. However, in most SPAs, you'll end up breaking the UI down into far more pieces than you would in a traditional MVC application.

What is a component?

We can think of components as the building blocks of a UI. Each one is a self-contained piece of functionality, usually combined with a host of other components in a tree-like structure to form the UI of the entire web application. These components are also often reusable, and can be simply dropped into any part of the application where required.

A component in Vue is made up of two fundamental parts: presentation and behavior. The presentation part is simply the HTML template that is used to represent the data we are trying to display in the UI. The behavior part is a JavaScript object containing only the data relevant to that specific component, and any JavaScript functions necessary to manipulate that data and interact with the browser. This interaction includes handling the events raised by the browser, as well as refreshing certain portions of the UI depending on how the data has actually changed. Vue is smart enough to only refresh the parts of the UI that need to be, and doesn't bother refreshing the parts where the data hasn't changed.

In many SPA frameworks, this results in a single component being split into at least two, potentially three, separate physical files; a HTML file for the template; a JavaScript file for the data and behavioral functions; and an optional CSS file for styling the presentation of the component. In Vue, we have the concept of a **single file component**. We can use a custom file extension that allows us to combine these three aspects into a single file that contains three root elements: `template`, `script`, and `style`. In the following code snippet, we can see an example of a single file component:

```
<template>
  <div class="product">
    {{ name }}
  </div>
</template>

<script>
export default {
  name: 'product',
  data () {
    return {
      name: 'Hands on ASP.NET Core and Vue.js'
    }
  }
}
</script>

<style>
.product {
  font-size: 12px;
}
```

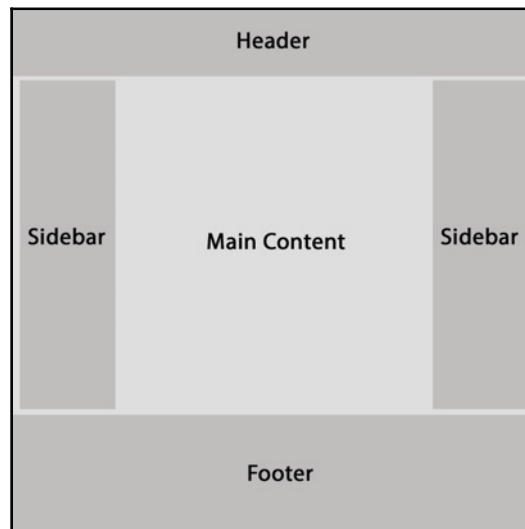
```
    font-weight: bold;
    color: teal;
  }
</style>
```

The presentation part of this component, that is, the `template` and `style` sections, are incredibly simple. All we do is render a standard `div` element with a `product` class, and set the font size, weight, and color of that specific class using CSS. Inside the `div` element, we're using Vue's standard handlebar syntax to dynamically render a `name` variable. This variable is declared within the behavior part of the component, which again means the `script` section. Standard component data properties are declared inside a plain JavaScript object, which must be returned from a function named `data`. In this case, we return a simple object with a single `name` property, initialized with the `Hands on ASP.NET Core` and `Vue.js` value. This will subsequently be the text rendered inside the `div` element of `template`.

Each component should adhere to the SOLID principles of software design, and as such should only have a single responsibility. As soon as any single component starts to become overly complicated and difficult to see at a glance what its purpose is, it's probably time to refactor and extract a new component. Components are more often than not used in parent-child relationships, and Vue provides mechanisms for allowing related components to communicate with one another. Parents can pass data down into their children, and children can notify their parents of changes to their data. These parent-child relationships are the branches of our component tree, and there are many different ways that we could choose to break a UI down into this structure.

UI composition

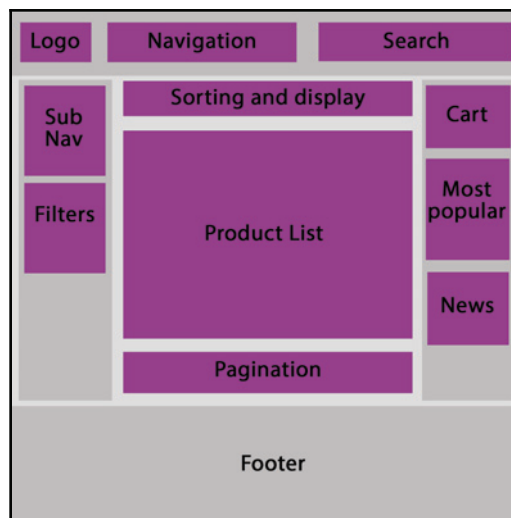
Let's look at an example that most web developers will be familiar with. Imagine a web page with the **Holy Grail** layout—that is **Header**, **Footer**, left and right **Sidebars**, and **Main Content** in the middle. When architecting this layout with components, the obvious composition is to start by creating a component for each major section of this layout:



Holy Grail web layout

This is a good start, and in a typical MVC application, we might have done something similar using layout and partial views. However, we already know that a component is the smallest possible chunk of functionality that forms part of the UI. We need to break things down further, and be far more granular with our component boundaries. How we do this will obviously be highly dependent on the type of content we actually have contained within these major page sections. Let's expand on the previous layout with some standard UI features we'd expect in a typical e-commerce product listing page.

The header section would likely contain some kind of branding or **Logo**, along with a **Navigation** menu of some sorts and potentially a **Search** bar. The left sidebar is a common place to find any secondary navigation menus and some **Filters** to control which products are visible. The main content section will contain our **Product List**, along with some UI elements above to control the display order and maybe even switch between a grid and list view; there would also usually be a standard **Pagination** control at the bottom of the page. Finally, the right sidebar can hold a widget that displays a summary of the user's shopping cart contents, a widget to display a set of featured or most popular products, and a newsletter signup form. The following diagram shows how we could start composing this UI into a component tree:



Holy Grail layout in components

We could—and probably would—still break this down further as we were actually building these components; it's almost impossible to get things right the first time round. However, this is the beauty of component-based architectures. There is no right or wrong way of composing a UI; we simply try it one way, and if it doesn't work, we refactor our component tree until we find a way that does work!

Introduction to Vue

Before we go much further, we really need to understand the basics of Vue, and how we go about using it to define and display our application's data on a web page.

The Vue instance

Every Vue application must have at least one root Vue instance, which is created when we pass an options object into the `new Vue()` constructor. There are a number of properties that we can define on this object, many of which are optional. These properties describe what data the instance has access to, and what kind of actions and manipulations it can do with that data.

Attaching to the DOM

Root Vue instances are attached directly to the DOM using the standard CSS selector syntax. In the following example, we specify an `el` property of `#app` on the `options` object, which instructs the instance to look for an element with an ID of `app` and attach to it:

```
var app = new Vue({
  el: '#app'
})
```

This forms a relationship between the Vue instance and a specific portion of the DOM, meaning we can start to display our dynamic data anywhere inside the specified element. However, if we tried to display our data outside of it, it wouldn't work as Vue is not managing the DOM at that level.

Defining data properties

In order to render data into the DOM, we first need to define the data we want to display. Every Vue instance can define a `data` object to hold as many data properties as we wish:

```
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello World'
  }
})
```


Rendering data into the DOM using expressions

To render our dynamic data properties into the DOM, Vue uses a very simple HTML template syntax which is inspired by the very popular handlebars templating library. This is why we refer to it as the handlebar syntax, which uses double curly braces, or mustaches, to render data within HTML elements:

```
<div id="app">
  {{ message }}
</div>

var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello World'
  }
})
```

Anything placed within a pair of double curly braces is treated as an expression, which means it will be interpreted using JavaScript. As such, we can reference the properties we define in the `data` object as we've done earlier, but can also make use of any standard JavaScript concept. As an example, we could easily make use of the `Date` object for rendering the current timestamp:

```
<div id="app">
  {{ message }}
  <p>
    The current date is: {{ new Date() }}
  </p>
</div>
```

We can even use logical operators to perform math-based operations or concatenate strings:

```
<div id="app">
  {{ message }}
  <p>
    {{ currentDate + ': ' + new Date() }}
  </p>
</div>

var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello World',
    currentDate: 'The current date is'
  }
})
```

Expressions allow us to perform any kind of operation on the `data` properties defined on the Vue instance, as well as built-in JavaScript objects and functions.

Building component trees

If you are only interested in plugging Vue into a handful of pages in a traditional MVC web application, you would define as many root Vue instances as required, attaching each one to a different DOM element. However, when building a full SPA with Vue, you would instead define a single root Vue instance with a nested tree of components underneath it. This single root Vue instance would then be responsible for attaching the entire application to a single DOM element.

We've already discussed what components are at a very high level, but what exactly is a component within the context of a Vue SPA? The answer is simply another Vue instance, albeit this time not a *root* Vue instance. There are a few subtle differences between a root Vue instance and a non-root Vue instance—most notably that only the root Vue instances define `el` attributes. This is because nested Vue instances, or components in a component tree, are simply rendered within their parent root Vue instances template. They do not need to be told where to attach themselves.

The other main difference between a root Vue instance and a component Vue instance is that the latter must define their `data` property as a function rather than an object. We've already seen an example of this when we first looked at the basic structure of a Vue component, but to reiterate, it looked as follows:

```
<template>
  <div class="product">
    {{ name }}
  </div>
</template>

<script>
export default {
  name: 'product',
  data () {
    return {
      name: 'Hands on ASP.NET Core and Vue.js'
    }
  }
}
</script>

<style>
```

```
.product {  
  font-size: 12px;  
  font-weight: bold;  
  color: teal;  
}  
</style>
```

Notice that the `data` property is a function that returns an object this time. This is because of the fact that every Vue component is a Vue instance. If we use a plain `data` object rather than a function returning an object, Vue cannot determine which component templates need to be refreshed when that property changes. This is a big issue when we render the same component multiple times when looping over lists of data, and then try and update a single list item's `data` property.

Reactivity

Now that we've talked about the Vue instance, it's worth talking about reactivity. One of Vue's core concepts is its reactivity system, which is also one of the main differences between building applications with Vue and sticking with jQuery or even plain old JavaScript. With jQuery or plain JavaScript, when a piece of data needs to change, we have to manually ensure that any DOM element which references that data is updated to display its new value. How big a job this is depends entirely on how big the application is, and how well-written it is. However, it's an exceptionally error-prone way of working, regardless of those factors.

In Vue, we bind the DOM to data properties defined in JavaScript, and then when those data properties change, the DOM is automatically updated for us by the reactivity system. The application quite literally becomes **reactive** to data changes, so it really doesn't matter how big the application is since every DOM element that displays a specific piece of data will automatically refresh when that value changes—no more manual DOM updates!

Component behavior

We've now seen how to define a Vue component, as well as how to declare data properties and render them into the DOM using component templates. However, this is just scratching the surface of what we can do with Vue components. We've already discussed how Vue components have two main aspects to them: presentation and behavior. Let's start to look at what else we can do with the behavior side of a component, starting by expanding on the `data` function and talking about state.

State

When talking about the **state** of an application, what do we actually mean? As soon as we introduce complex client-side logic to a web application, we also introduce multiple meanings of the word state, or rather, we introduce an additional type of state to our application. State can mean different things depending on the type of state we are interested in.

Most backend .NET developers will probably understand state to be based on a snapshot of the applications database at any point in time. In terms of our e-commerce example from earlier, this would include the current list of products and categories that make up our catalog; a list of users or customers who have registered for an account; and a list of orders and associated order items. This form of state is based on the domain of the application, and can be extended to include things that don't necessarily persist into the database, such as authentication, validation, and business rules that control how the application behaves.

The type of state that we care about at the component level is known as **UI state**. Generally speaking, UI state and domain state are separate things, but it isn't impossible for the two to cross over. For example, keeping track of the current user isn't necessarily a UI concern, but most SPAs will use some form of JWT authentication where the user's tokens, and as such their authentication state, will be tracked by the SPA. Another example is where we display paginated lists of data in an SPA—keeping track of the currently displayed list items is a UI concern, but we are still displaying a subset of the database-persisted items that belong to the domain of the application.

Other examples of UI state include keeping track of the active menu item in a navigation component; controlling the visibility of a modal window or custom drop-down menu; keeping track of which panels are open/closed in an accordion; and showing and hiding loading spinners during AJAX operations. These are fairly simple examples, and there are a lot more complex things that we can do with client-side state such as transitions and animations, but it's enough to demonstrate what we are talking about for now.

Each of our components are only responsible for their own subset of the application UI state. For example, a component that contains the filters that we've applied to a product list is only concerned with the selected values of those filter controls. It isn't—and shouldn't be—concerned with which user is currently logged in, or how many items the user has added to their shopping cart. This is all part of adhering to the single responsibility principle, which makes our components much easier to debug and maintain.

Sooner or later, we're going to come across a situation where a single component is in violation of the SRP, and we want to break that component down into a parent-child relationship instead. A common pattern is where we have a list component as a parent that contains a collection of list-item components as its children. The original component was probably already fetching the data it displays, and it makes sense to leave that responsibility up to the new parent list component; after all, the only alternative is to have each list-item component fetch its own data, which would result in multiple trips to the server instead of just one.

Props

We already know that components are self-contained, so how do the children get access to the data they need to display if the parent owns and controls it? The simple answer to that question is props. **Props** are a means of parent components passing data down into their children. The child component must explicitly declare the names of the props it expects to receive, and then these props can be referenced in much the same way that we do for any other piece of data that the component owns.

The following code demonstrates how we declare and reference a prop within a child component:

```
<template>
  <div class="product">
    {{ name }}
  </div>
</template>

<script>
export default {
  name: 'product',
  props: ['name']
}
</script>
```

We can render this child component from within its parent component template as follows:

```
<template>
  <div class="product">
    <child-component name="Hands on Vue.js and ASP.NET Core" />
  </div>
</template>

<script>
import ChildComponent from './ChildComponent.vue'
```

```
export default {
  name: 'parent',
  components: {
    ChildComponent
  }
}
</script>
```

At this stage, it is important to understand that this method of sharing data between components is strictly limited to one-way. It is impossible to send data back up the chain from a child to a parent using props. We'll look at how to communicate in the opposite direction later in this chapter.

The final point to mention about props is that Vue provides a means of validating the props being passed to a component. We can perform basic type checking; control whether props are required or optional; configure default values in the event that a prop is not provided; and even write custom validator functions in much the same way as we would with client-side validation libraries. The following code snippet shows an example of some of these validation rules and how we describe them in the component definition:

```
<script>
export default {
  name: 'validation',
  props: {
    name: {
      type: String,
      required: true
    },
    description: {
      type: String,
      required: true
    },
    price: {
      type: Number,
      required: true
    }
  }
}
</script>
```

Methods

If the data we are displaying within our components never changes, it's probably a sign that we really don't need to be using an SPA framework such as Vue. We know that the data in a component is used for things such as showing and hiding modal windows, so how do we actually change the data so that the UI can become reactive? Vue components can declare methods in order to manipulate their data. These methods are standard JavaScript functions, and automatically have their function context (that is, the value of `this`) bound to the component instance so that they can access its data, props, and computed properties. The following code shows how we can increment a simple counter using a method on a Vue component. We can trigger this method by calling it from a UI element event handler, which we'll look at later in this chapter:

```
<script>
export default {
  name: 'methods',
  data () {
    return {
      counter: 1
    }
  },
  methods: {
    increment () {
      this.counter++
    }
  }
}
</script>
```

Computed properties

As our applications grow in complexity, the chances are that sooner or later we'll need to perform some logic on one or more of our component data items and display it in a template. As a simple example, we may have data properties for a person's first and last name, but we are regularly required to concatenate them and display their full name. We could just use expressions inside the template as follows:

```
<template>
  <div>
    {{ firstName + ' ' + lastName }}
  </div>
</template>

<script>
```

```
export default {
  name: 'expressions',
  data () {
    return {
      firstName: 'Stu',
      lastName: 'Ratcliffe'
    }
  }
}
</script>
```

However, if we've duplicated this expression in multiple places throughout the template, and we then decide to change the way we display the person's full name, we have multiple places to find and update it. Even in a fairly small and simple example such as this one, this doesn't sound like much fun, and certainly isn't very maintainable.

Alternatively, we could use a *computed property* to achieve the same result. If you've ever used a computed column in SQL Server, then the concept will be familiar to you, and computed properties in Vue components behave in much the same way. In the following code snippet, we can see how we would declare a `fullName` computed property in our component declaration, and how we then render the value of that property into a template:

```
<template>
  <div>
    {{ fullName }}
  </div>
</template>

<script>
export default {
  name: 'computed-properties',
  data () {
    return {
      firstName: 'Stu',
      lastName: 'Ratcliffe'
    }
  },
  computed: {
    fullName () {
      return `${this.firstName} ${this.lastName}`
    }
  }
}
</script>
```


Although they are referred to and referenced in the same way as standard data properties, they are actually functions. As with the component data object and methods, if Vue detects that the returned value of a computed property has changed, it will automatically refresh the UI to reflect those changes.

Watchers

Watch properties are similar in functionality to computed properties, and in most cases it is actually unnecessary to use one. However, there is a limitation in that computed properties are synchronous functions and always return a value that we can bind to. This makes them impossible to use alongside asynchronous operations such as AJAX calls. So, what should we do if we want to trigger an AJAX call to our API server and react to the data that is returned?

Say we wanted to automatically display search results as a user enters their search term into a text input. We could implement a simple event listener on the input and create a method that is triggered on the keyup event, which performs an AJAX request with its current value. This would work absolutely fine, and you may never need any more control than this. However, this piece of functionality is completely coupled with the text input, and as such our component will not react and refresh the UI if we change the data value of the input directly.

Watch properties are a solution to this problem, as they provide a far more generic way of reacting to data changes. In the following example, we are performing an AJAX request that queries an API to perform some sort of search. We'll discuss what `v-model`, `v-show`, and `v-for` mean shortly, but for now, just know that the text input has its value bound to the `searchValue` property, and we display each search result in the array as a list item element:

```
<template>
  <div>
    <h1>Watchers Example</h1>
    <input v-model="searchValue" />
    <span v-show="loading">Loading...</span>
    <ul>
      <li v-for="result in searchResults">{{ result }}</li>
    </ul>
  </div>
</template>

<script>
export default {
  name: 'watchers',
```

```
data () {
  return {
    searchValue: '',
    searchResults: [],
    loading: false
  }
},
watch: {
  searchValue (newValue) {
    let vm = this
    this.loading = true

    setTimeout(function () {
      vm.searchResults.push('some', 'search', 'results')
      vm.loading = false
    }, 500)
  }
}
}
</script>
```

Note how we also have to declare a `data` property of the same name. The `watch` declaration instructs the component to quite literally *watch* the `data` property, and run the associated function each time it changes. The current value of the `data` property is passed to the function so that we can use it in any way that we please.

We also made use of this function to update a `loading` property to instruct the component that the AJAX call is in progress; this property can now be used to show a loading spinner in the UI each time the AJAX call is triggered. We could even extend this function to limit how often the AJAX call can take place by using a `debounce` or `throttle` function. None of this is possible with a `computed` property, and by using a `watch` property, it doesn't matter whether we update the data through a UI control or manually in another component method!

Lifecycle hooks

Every Vue component goes through a number of steps to initialize it when it's created. Among other things, these steps are responsible for compiling and rendering the HTML template, setting up the component's data so it becomes reactive, and mounting the component into the DOM. In order to hook into this process, we're given a number of lifecycle hooks that we can use to run our own application-specific initialization code at each stage.

These lifecycle hooks are as follows, and named appropriately enough to make themselves fairly self-explanatory:

- `beforeCreate`
- `created`
- `beforeMount`
- `mounted`
- `beforeUpdate`
- `updated`
- `beforeDestroy`
- `destroyed`

`beforeUpdate` and `updated` run in a continuous loop for the duration of the component's lifetime. Every time Vue detects a data change within the component, these steps are run before and after the virtual DOM is re-rendered.

So, how do we actually make use of these function hooks? The following code shows an example of how we declare the hooks we wish to use within a component definition:

```
<script>
export default {
  name: 'lifecycle-hooks',
  beforeCreate () {
    console.log('before create')
  },
  created () {
    console.log('created')
  },
  beforeMount () {
    console.log('before mount')
  },
  mounted () {
    console.log('mounted')
  },
  beforeUpdate () {
    console.log('before update')
  },
  updated () {
    console.log('updated')
  },
  beforeDestroy () {
    console.log('before destroy')
  },
  destroyed () {
```

```
        console.log('destroyed')
      }
    }
  }
</script>
```

We simply add a root-level function with its key matching the name of the lifecycle hook we want to use. This is all very well and good, but if you've never used an SPA framework such as Vue before, you're probably wondering why we'd ever want to bother doing this. There are many reasons to use lifecycle hooks, the most common of which is probably to fetch the data the component needs from an API somewhere. A good place to do this is in the `created` hook because we don't need access to the DOM, so there is no need to wait until the `mounted` hook is run later. The other reason to use `created` instead of `mounted` is because `created` is the only hook that is run on both client- and server-rendered versions of the component. We'll look at server-side rendering in one of the final chapters in this book!

A few other examples include triggering animations as soon as the page is displayed, and dirty checking a form to give the user a chance to complete it before navigating away. We would need to use the `mounted` and `beforeDestroy` hooks for these actions, respectively.

So far, we've been focusing on the Vue instances and the different ways they help us manage the data that our components are responsible for, but this is only half of the story. We're yet to see how Vue can help us actually display that data. Let's start focusing on the `template` section of our components!

Component presentation

We've now seen how a component can work with and manipulate the data of an application, so let's move on to look at how to actually display that data in the `template` section of our components.

Directives

Out of the box, Vue provides a number of directives that help us display our data. A directive is a special token that we attach to the HTML element markup in order to instruct Vue to do something special with the DOM element that it renders. Let's look at a simple example to make this a little easier to understand:

```
<template>
  <input v-bind:value="message" />
</template>
```

```
<script>
export default {
  name: 'data-binding',
  data () {
    return {
      message: 'Hands on Vue.js and ASP.NET Core'
    }
  }
}
</script>
```

Here, we use the `v-bind` directive to bind the value of a standard HTML input element to the `message` property of the component's `data` object. In an MVC application, we can do something similar with Razor syntax to bind DOM elements to properties on a view model. We can also use Razor to do things like loop over a list and render the same output for each item, and use conditional rendering and control flow to render different groups of elements depending on runtime model conditions.

The default directives that Vue provides can do all of these things and more, all while using a much cleaner and easier to read syntax!

Attribute binding with `v-bind`

We've already seen a simple example of `v-bind`, but there are many other uses for it. We can use `v-bind` to turn any standard HTML attribute into a reactive version that we can change the underlying data value of to cause the UI to refresh. For example, we can create a very basic image carousel by using `v-bind` on the `src` attribute of an HTML image tag, and then rotate through an array of data values to cause the image to refresh:

```
<template>
  
</template>

<script>
export default {
  name: 'v-bind',
  data () {
    return {
      imageSrc: '',
      images: [
        'path/to/image/1.jpg',
        'path/to/image/2.jpg',
        'path/to/image/3.jpg'
      ]
    }
  }
}
```

```
    },
    mounted () {
      let vm = this
      let count = 0

      setInterval(function() {
        vm.imageSrc = vm.images[count++]
        if (count == vm.images.length)
          count = 0
      }, 1500)
    }
  }
</script>
```

Notice how we're using the mounted lifecycle hook that we talked about earlier, which is a perfect example of triggering a UI transition/animation as the component is rendered and displayed in the DOM.

One of the most common usages that we'll see throughout the rest of this book is class bindings to change how an element is styled reactively. We can conditionally add or remove classes based on the component's state, and change that state in response to user interactions such as button clicks. Until now, we've been using the full `v-bind:attr` syntax, where the section after the `:` is the specific HTML attribute you wish to data bind. However, Vue also provides a shorthand syntax for a number of its directives. In the case of `v-bind`, we can simply omit the `v-bind` part and use `:attr` instead:

```
<template>
  <div :class="{ 'blue': isBlue }"></div>
</template>

<script>
export default {
  name: 'v-bind',
  computed: {
    isBlue () {
      return Math.random(0, 1) > 0.5
    }
  }
}
</script>

<style>
div {
  background: red;
}

div.blue {
```

```
    background: blue;
  }
</style>
```

Notice how we shortened the directives syntax to `:class` rather than `v-bind:class`. This is a very simple example that changes the color of a `div` element. However, we could use the same techniques to add colored borders and other indicators to show that a piece of the UI has been selected by the user, or that a form `input` element contains an error after validation.

Conditional display with `v-show`

Rich UIs usually require us to conditionally show and hide elements based on variables and user interactions. Vue gives us a couple of ways of achieving this with the `v-show` and `v-if` directives. We'll talk more about `v-if` in the next section, but ultimately, both options can be used for controlling the visibility of an element based on the component's state:

```
<template>
  <div v-show="show">
    v-show example
  </div>
</template>
```

In the preceding example, we're using `v-show` to control the visibility of a `div` element based on a simple Boolean property. This property is toggled by a simple method which is invoked by a button click.

Control flow with `v-if` and `v-else`

If...else statements will need no introduction, as we often use them in both our backend C# code and our frontend Razor views. Usage in Vue is no different; we just use `v-if` and `v-else` in place of the Razor syntax:

```
<template>
  <div v-if="show">
    v-if example
  </div>
  <div v-else>
    v-else example
  </div>
</template>
```

It is also possible to control multiple elements at once with a single `v-if` statement to prevent duplication. Once rendered, the following `template` tag will not be included, as it acts as an invisible wrapper:

```
<template>
  <div>
    <template v-if="show">
      <div class="first"></div>
      <div class="second"></div>
      <div class="third"></div>
    </template>
  </div>
</template>
```

As of version 2.1.0 of Vue, a `v-else-if` directive has also been added, and it behaves exactly as you'd expect:

```
<template>
  <div>
    <div v-if="show">
      v-if example
    </div>
    <div v-else-if="show2">
      v-else-if example
    </div>
    <div v-else>
      v-else example
    </div>
  </div>
</template>
```

As with a normal `if...else` statement, it must follow a `v-if` directive, and similarly the `v-else` statement must follow either a `v-if` or `v-else-if` statement.

There's nothing stopping us from using a `v-if` directive by itself, without any associated `v-else` or `v-else-if` directives, in which case it has a very similar behavior to `v-show`. At this point, you're probably wondering why we need both if they do the same thing. There is one very big difference between `v-if` and `v-show`, and it's important to understand how that difference should influence the decision on which to use in a given scenario.

An element with a `v-show` directive attached is *always* rendered into the DOM, and visibility is controlled via the CSS `display` property. On the other hand, elements controlled by a `v-if` directive are only rendered into the DOM if the conditional is truthy. This means that visibility is controlled by removing the element from the DOM entirely.

So, which one do we use and why? Generally speaking, `v-show` is more appropriate for use cases where we know the conditional is going to change frequently. This is because the overhead of toggling the element is much lower, as we have taken that performance hit by always rendering the element, regardless of the conditional. If the condition is unlikely to change at runtime, then we would prefer `v-if`, as there is a chance that the elements won't ever be rendered, saving us time upfront.

Rendering lists with `v-for`

Rendering lists of items is a very common requirement of most web applications, so Vue has us covered on this one with the `v-for` directive. Again, the syntax is very similar to Razor, and we can assign an alias to the current item in the loop. We can then reference the alias to give us access to any or all of its properties:

```
<template>
  <div>
    <div v-for="item in items" :key="item">
      {{ item }}
    </div>
  </div>
</template>
```

It is also possible to access the index of the item within the array by using an optional second argument:

```
<template>
  <div>
    <div v-for="(item, index) in items" :key="item">
      {{ index }} - {{ item }}
    </div>
  </div>
</template>
```

Although we've been using the standard syntax—for example, `item in items`—it is also possible to use `on` rather than `in`. This is closer to the standard JavaScript syntax, so feel free to use this version if you feel more comfortable with it.

Event handling with v-on

Vue gives us a really easy way to attach event handlers to elements using the `v-on` directive. With standard HTML elements, we can listen for any native DOM event:

```
<template>
  <div>
    <button v-on:click="clickHandler()">Click me!</button>
    <form v-on:submit="submitHandler()">
      <input type="text" />
    </form>
  </div>
</template>
```

Here, we are listening for a native click event on a standard HTML button, and invoking the `clickHandler` method each time it is clicked. We are also listening for the form's submit event to be fired, and intercepting it with the `submitHandler` method. You've probably noticed that we already saw some button click examples earlier when looking at some of the other directives in Vue. However, we didn't use the `v-on` syntax like we did here, and instead we used the `@` shorthand notation. Here's the same example again using `@click` and `@submit` instead:

```
<template>
  <div>
    <button @click="clickHandler()">Click me!</button>
    <form @submit="submitHandler()">
      <input type="text" />
    </form>
  </div>
</template>
```

When writing event handler functions in jQuery, we often find ourselves needing to prevent the default event behavior or stopping the event from propagating up through the DOM. In order to do this, we would normally receive the native DOM event as an argument to the event handler function. This is also possible with Vue, as the native event is passed to the handling function, by default, as the only argument:

```
<template>
  <div>
    <button @click="clickHandler">Click me!</button>
  </div>
</template>
<script>
export default {
  name: 'events',
  methods: {
```

```
    clickHandler (event) {
      event.preventDefault()
      event.stopPropagation()
    }
  }
}
</script>
```

If you've written a lot of jQuery in the past, then this will feel right at home for you. However, it's not the only way to achieve the same result, and Vue gives us a much nicer way through the use of modifiers:

```
<template>
  <div>
    <button @click.prevent.stop="clickHandler">Click me!</button>
  </div>
</template>
```

It should be fairly self-explanatory as to what's happening here, but adding the `.stop` modifier is a shorthand way of calling `event.stopPropagation()`, and adding the `.prevent` modifier is a shorthand way of calling `event.preventDefault()`. There are a number of other modifiers available to us, including but not limited to the following:

- `.native`: This is used for listening to the native events on the component's root element
- `.{keyCode}`: This is used for listening to specific keyboard key presses, for example, `@keyup.13`
- `.{keyAlias}`: This is also used for listening to specific keyboard key presses, for example, `@keyup.enter`
- `.left`, `.middle`, or `.right`: This is used for listening to specific mouse button clicks

Notice how we also chained the prevent and stop modifiers in the preceding example. This is perfectly valid in Vue.

Event handling in Vue is incredibly flexible and powerful. We can attach as many `v-on` directives as we wish, but we can also attach multiple handlers using a single directive by making use of the object syntax:

```
<template>
  <div v-on="{ click: clickHandler, mouseover: hoverHandler }"></div>
</template>
```

Here, we are listening for both a `click` and `hover` event on a single `div` element. We can add as many properties to this object as we wish, depending on how many events we need to listen for.

Form input binding with v-model

When rendering the form input fields, it might seem obvious to make use of the `v-bind` directive that we've already looked at. We've already seen that binding an `input` element's `value` property can be done as follows:

```
<template>
  <input type="text" :value="value" />
</template>
```

Upon rendering of the component, you'll find that the textbox is displayed seemingly correctly with the `data` property prefilled. However, if you tried to change the value of the input by typing into the box, nothing would happen. This is because `v-bind` is a one-way binding. It is not possible to update the backing property using `v-bind`, and as such, the input will always display the initial value unless we change that value programmatically via some kind of component method. Following on from the preceding example, this would look like the following:

```
<template>
  <input type="text" :value="value" @input="onInputChange" />
</template>

<script>
export default {
  name: "form-input-binding",
  data() {
    return {
      value: "Some value"
    };
  },
  methods: {
    onInputChange(event) {
      this.value = event.target.value;
    }
  }
}
</script>
```

This is quite a lot of code for something as common, and simple, as binding a text input to a `data` property. Luckily for us, Vue has a much nicer way to create a two-way binding, and this is by using the `v-model` directive. Under the hood, it does a very similar job to what we've done earlier, but helps us keep our code much leaner and focused on more complicated functionality. It can also be used on all form input elements and text area elements, and automatically uses the correct way of updating the elements' values based on their types:

```
<template>
  <input type="text" v-model="value" />
</template>
```

By using `v-model` rather than the combination of `v-on` and `v-bind`, we completely negate the need to create a component method just to update the `data` property. The property is updated automatically for us by the `v-model` directive. As with the `v-on` directive, there are some modifiers that we can use to help us out in certain circumstances. These are as follows:

- `.lazy`: This is used to sync the `input` and `data` properties after a change event rather than an `input` event
- `.number`: This is used to automatically typecast the input value as a number rather than a string
- `.trim`: This is used to automatically trim the leading and trailing whitespace from the input value

Parent-child component communication

Earlier in this chapter, we looked at composing parent-child component relationships, and without explicitly saying so, we also covered the basics of how two components in such a relationship can communicate with one another.

Firstly, for a parent component to pass data down to its children, props can be used. However, this is only suitable for a one-way flow of data from parent to child. If we need to send data in the opposite direction, that is, from child to parent, we must use event handling using the `v-on` directive that we saw earlier. More specifically, we fire events from the child component, and instruct the parent component to listen for those events using the `v-on` directive. Vue is smart enough to know that we are listening to an event on a custom HTML element, and as such, it is capable of listening for custom events as well as native DOM events:

```
<template>
  <person @name-changed="handleNameChange" />
</template>
```

In this example, we render a custom person element and listen for a custom `name-changed` event. In order to trigger this handler, the child component must manually emit the event:

```
<script>
export default {
  name: 'person',
  data () {
    return {
      name: 'Stu Ratcliffe'
    }
  },
  methods: {
    save () {
      this.$emit('name-changed', this.name)
    }
  }
}
</script>
```

This completes our overview of Vue and the main features that we'll be using throughout the rest of this book. Let's move on and take a whistle-stop tour of ASP.NET Core. Again, if you're already familiar with ASP.NET Core or, more specifically, how it differs from previous versions of ASP.NET, you can skip straight ahead to [Chapter 2, Setting up the Development Environment](#), where we'll get stuck straight into setting up our development environment.

ASP.NET Core – what's new?

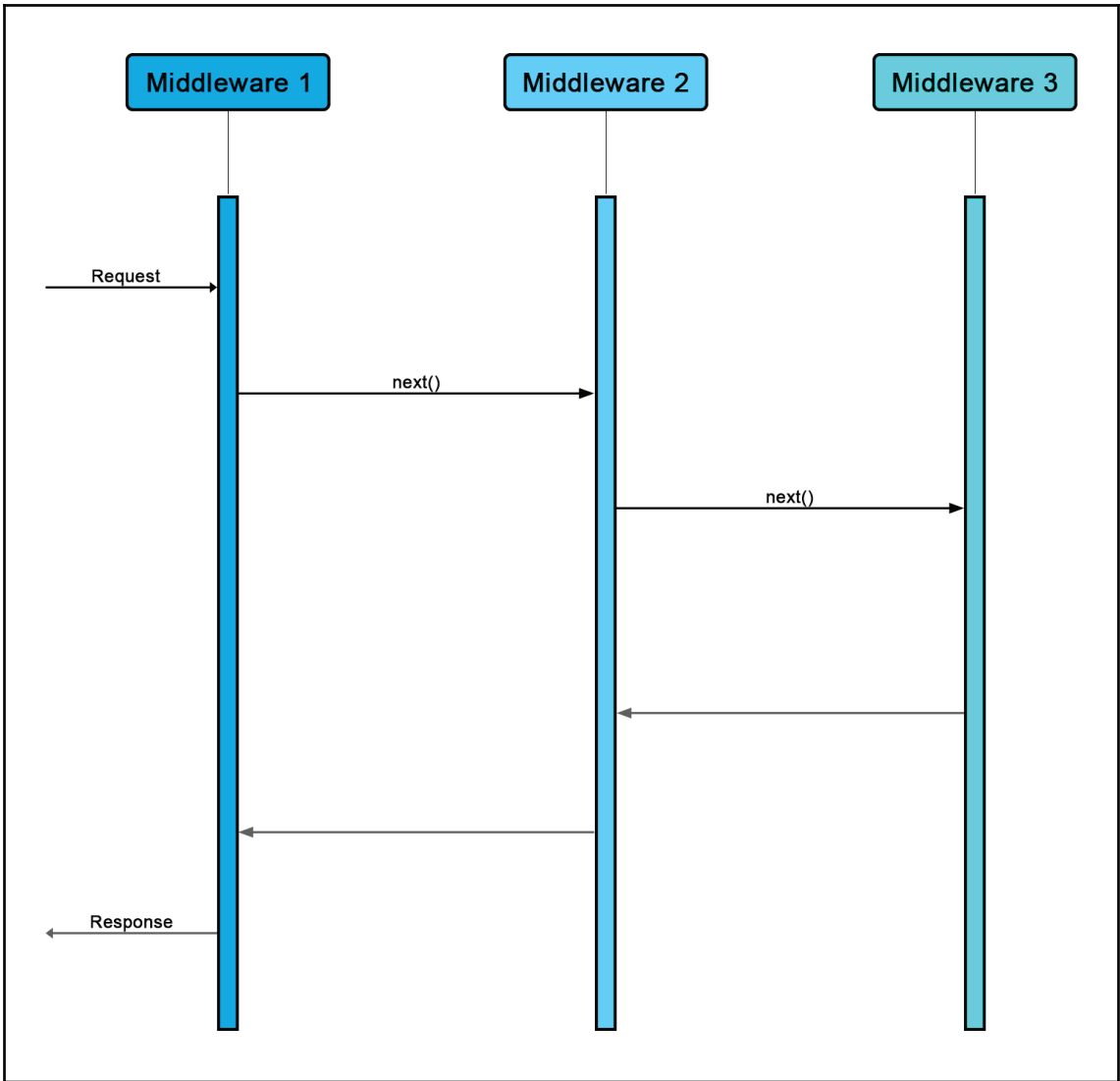
ASP.NET Core is a complete rewrite of the ASP.NET framework, rather than a new version built on top of the existing one. It is much smaller and more modular, and is the first version of ASP.NET to have official cross-platform support. Covering everything that has changed in ASP.NET Core would require a whole book in itself, so we'll focus on the main parts that we need to know a little bit about to get us started.

Middleware pipeline

ASP.NET Core apps are built on the principle of a middleware pipeline. But what exactly do we mean by this? Web applications are all about HTTP requests and responses; a middleware pipeline is simply a set of instructions for how the application should handle each HTTP request.

Ultimately, an ASP.NET Core middleware is just a function which is invoked as a result of a HTTP request. If we need multiple middleware functions, they can be chained together in a specific order to form a pipeline to process the request. After each function finishes processing, they may decide that the request should be terminated. In this instance, it sends a response back to the previous middleware, which in turn passes it back to the middleware before that, and so on until it eventually ends up back with the client that started the request. Alternatively, if everything is successful during processing, it can simply pass the request on to the next piece of middleware defined in the chain. Each middleware can essentially perform some kind of custom logic before the request, after the request, or both. As an example, you could easily create a request timing middleware that records a timestamp before and after every request, so that the total processing time of each HTTP request can be logged somewhere.

The following diagram shows how this works in an ASP.NET Core web application:



ASP.NET Core provides a number of middleware components for us to optionally make use of, and we'll see how we go about registering these with our application in the next section. The only other thing to note is that the order that we register our middleware in is extremely important. If one middleware depends on another one having processed the request before it receives it, we need to make sure that the dependent middleware is configured *after* the middleware it depends upon. Otherwise, our requests are likely to fail, or at least display some very strange results depending on what the middleware are designed to do.

Later in this book, we'll look at how we can build our own custom middleware and register them with the pipeline.

Application startup

All ASP.NET Core applications must have a `Startup.cs` file that contains a class named `Startup`. There are two methods we need to be aware of, and these are `Configure` and `ConfigureServices`.

`Configure` is a required method that is used to, as the name suggests, configure the application, specifically by setting up the request pipeline, including any optional middleware that we may need. The following sample shows how we would configure a simple ASP.NET application that uses the `StaticFiles`, `Authentication`, `RequestLocalization`, and `MVC` middleware:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseStaticFiles();
    app.UseAuthentication();
    app.UseRequestLocalization();
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

`ConfigureServices` is optional, although I've yet to find an application that doesn't implement it. It is used to configure the application's services and register them with the built-in **dependency injection (DI)** container. Most non-trivial applications will need to configure services such as **Entity Framework (EF)**, ASP.NET Identity, RequestLocalization, and MVC. The following code shows a sample configuration of these services:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseNpgsql(Configuration["ConnectionString"]));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.Configure<RequestLocalizationOptions>(options =>
    {
        options.DefaultRequestCulture = new RequestCulture("en-GB");
    });

    services.AddMvc();
}
```

You'll probably spend a fair bit of time in this class, tweaking configurations and such, so let's look into these two sections in a little more detail.

DI is a first-class citizen

ASP.NET Core has been built from the ground up with DI in mind, and in fact DI is now a pattern that is strongly encouraged for use in every application. As such, it is no longer required to use an external library for DI, as there is a simple built-in container included out of the box. On top of this, the vast majority of official supporting libraries are registered as DI services when we install them, such as EF and ASP.NET Identity. That being said, the built-in container is fairly limited in functionality, and for more advanced requirements, Microsoft is still recommending that we bring in a more fully-featured container instead.

DI is not a technique that all ASP.NET developers will be familiar with if they've only ever built fairly simple applications. As it's a core technique used in ASP.NET Core apps, if you aren't particularly comfortable with it, then I strongly recommend you go and read up on it. The Microsoft documentation for ASP.NET Core is a great place to start! We've already seen how to configure some of the built-in framework services using the extension methods provided by each package, but how do we register our own services? The answer is via a set of extension methods that provide a way of registering dependencies with different lifetimes:

```
services.AddTransient<ICartService, CartService>();
```

The `services.AddSingleton()` extension method is fairly self-explanatory, and is used for registering services that have a single instance which is shared by all dependent classes. The other two are slightly less obvious: `services.AddScoped()` is used for registering dependencies that are scoped to the lifetime of a request, that is, they are created once per request and each dependent class receives the same copy until the request terminates; `services.AddTransient()` is probably the most common, and simply means that each service is instantiated every time it is requested.

This is pretty much the limit of what we can do with the built-in container, and as previously mentioned, if you need any more complicated features, such as property injection and/or convention-based registrations, you'll need to look at a more complete container such as `StructureMap`.

EF Core – what's new?

These days, it is rare to see an ASP.NET application that doesn't make use of some kind of ORM, and even rarer to see one that uses anything other than EF. There are certainly other options, such as the much lighter Dapper and Marten, a library that takes the JSONB capabilities of PostgreSQL and uses them to turn it into a full-featured NoSQL document store. However, SQL Server is where most .NET developers' comfort lies, so we'll stick with what we know for the examples in this book.

Configuring relationships

In older versions of EF, you could get away with leaving it to do its thing without manually intervening with the way it builds out the relationships between tables in the database. It could handle one-to-one, one-to-many, and many-to-many relationships out of the box, meaning that unless you had a super complicated domain model, you didn't need to do much to get a working database.

In EF Core, only one-to-one and one-to-many relationships can be inferred without manual configuration. I don't see this as a huge problem, as it is only a few extra lines of code to tell the fluent model builder how to configure many-to-many relationships:

```
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.Entity<OrderItem>()
        .HasKey(x => new { x.OrderId, x.ProductId });

    base.OnModelCreating(builder);
}
```

Notice how we only have to instruct EF what to do with the join table. From these few lines of code, it can now go away and build the database for us without any problems.

Global query filters

One of the features that other ORMs had that EF didn't was the concept of global query filters. These queries are a means of telling EF to automatically apply a LINQ statement to every query that's executed against the type of entity in the filter. A common use case for this kind of query is when an application uses the concept of soft deletes. Rather than actually deleting the data, it is marked with a Boolean flag instead.

The following image shows how we can register a global query filter on a DbContext entity to only include records where the `IsDeleted` flag is set to `false`:

```
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.Entity<Order>()
        .HasQueryFilter(x => !x.IsDeleted);

    base.OnModelCreating(builder);
}
```

We could also use these global filters in multitenant applications, where each tenant should only be able to access the data associated with their tenancy. This is a much better solution than relying on applying these filters manually on every query, which is exceedingly error-prone, as it is too easy to forget.

Compiled queries

EF now supports the concept of explicitly compiled queries. These provide a number of benefits, most notably by increasing the query's performance, but also making it easy to run the same query in multiple places within the code.

The idea is pretty simple; if we have a query that is run many times within our application, then we can instruct EF to compile it. It is compiled only once, but we can run it as many times as we like, with different parameters each time. The following code shows an example of how we can define a compiled query and then execute it:

```
public static class CompiledQueries
{
    public static Func<ApplicationDbContext, int, Order> OrderById =
        EF.CompileQuery((ApplicationDbContext db, int id) =>
            db.Orders
                .Single(c => c.Id == id));
}
[HttpGet]
public IActionResult CompiledQuery()
{
    var order = CompiledQueries.OrderById(_context, 147);
}
```

In-memory provider for testing

It has always been exceptionally difficult to write tests around the code that was dependent on an EF DbContext. To make testing easier, developers often resorted to implementing different variations of the repository pattern so that the business layer could depend on a repository interface instead. This had the desired effect of making testing easier, but the general concept of a repository pattern over the top of EF was quite simply unnecessary, as the DbContext is already an implementation of both the repository and unit of work patterns.

EF Core has addressed this issue by providing us with an in-memory version that we can use for our tests. It is now a fairly simple task to create an in-memory database and seed it with test data before each test is run. This ensures that the database is in a known state for each test, without the complexity of attempting to mock the DbContext!

The following example shows how we can configure a test DbContext and fill it in with test data:

```
public static ApplicationDbContext GetDbContext(params object[] seedData)
{
    var connection = new SqlConnection("DataSource=:memory:");
    connection.Open();

    var options = new DbContextOptionsBuilder<ApplicationDbContext>()
        .UseSqlite(connection)
        .Options;

    var context = new ApplicationDbContext(options);
    context.Database.EnsureCreated();
    if (seedData != null && seedData.Length > 0)
    {
        context.AddRange(seedData);
        context.SaveChanges();
    }
    return context;
}
```

We can then pass this DbContext to a dependent controller within the scope of our unit tests:

```
[Fact]
public async Task Test()
{
    using (var context = GetDbContext())
    {
        //arrange
        var controller = new ProductsController(context);

        //act
        var result = await controller.GetProducts();

        //assert
        Assert.NotEmpty(result);
    }
}
```

The only thing to note when using the in-memory provider is that it isn't a full relational database, and doesn't try to mimic one. I've noticed a few weird things when using it in my applications, and found that using an in-memory SQLite provider to be far more stable and predictable. There is plenty of documentation on both options on Microsoft's own ASP.NET Core documentation pages.

Summary

In this chapter, we looked at the basic concepts that we'll need to build an application with Vue, as well as some of the most important changes to ASP.NET Core and EF Core. We also saw how Vue has a very similar syntax to that of Razor, and covers many of the same use cases where Razor and MVC views would be used in a traditional MVC-based application.

2

Setting Up the Development Environment

By this point, you should already be comfortable with the basics of Vue.js and ASP.NET Core, or have read the previous chapter. Either way, we're not going to waste any more time on theory! In my eyes, the best way to learn about a new programming language or a framework is to get stuck in and build something.

Before we can do anything else, we need to install a few things that we'll need to build and run the application, and also a few tools to make our lives easier. In summary, in this chapter, we're going to cover the following:

- Choosing and installing a browser
- Installing frontend tools and dependencies, such as Node and npm
- Evaluating and choosing a frontend package manager
- Installing backend tools and dependencies, such as ASP.NET Core and PostgreSQL
- Choosing an IDE
- Installing productivity tools to make our lives easier

Choosing a web browser

When building any web application, it is important to ensure that it is fully tested in all browsers that you need to support. At the very least, this is usually the latest version of all modern browsers. However, during the development phase, you'll most likely use a single browser, so it is important to choose the best one for our needs.

Google Chrome has been my choice of browser for many years now, but, more recently, it has been truly cemented as the best tool for the job (in my opinion) due to a number of extensions that we can install to help us while building the frontend of our application. There will be more on this later!

To install Google Chrome, navigate to the following URL and follow the instructions provided: <https://www.google.co.uk/intl/en/chrome/>.

As I'll only be using Chrome for the purposes of this book, it is the only browser that I can safely say is supported by the sample application that we're going to build. That being said, Vue is supported by all modern browsers, and we'll also be using Bootstrap for styling, which is also supported by all modern browsers. This is where we really see the benefits of using these kinds of frontend frameworks, as somebody else has already done most of the hard work of ensuring cross-browser compatibility for us.

Installing frontend tools and dependencies

For any kind of modern frontend development, we're going to need Node and **Node Package Manager (npm)** installed. Node itself is a runtime built on top of Chrome's V8 engine, enabling us to run JavaScript on the server side of our applications. So, why do we need it if we're using ASP.NET Core?

Node and npm are both requirements of the tooling used to build and run medium-large scale Vue applications. We'll also be using npm to install a lot of client-side packages that we'll use within the application itself.

Installing Node and npm

Head on over to the following URL and download the Node.js installer (<https://nodejs.org/en/>). The website should be smart enough to detect your OS and provide the right platform installer for you, which in my case is Windows (x64). You have two options to choose from, the *LTS* version or the *Current* version. I'll be sticking with the LTS version, which at the time of writing is version 8.9.3.

To test that Node has been installed properly, open up the Command Prompt or PowerShell if you are on Windows, or the Terminal if you are on Mac or Linux, and run the following commands:

```
PS C:\Users\stuar> node -v
v8.9.3
PS C:\Users\stuar> npm -v
5.5.1
```

If you receive any output other than the version number you currently have installed, then something has gone wrong. Check out the troubleshooting sections on the Node website, and if all else fails, chances are Google will have the answer!

If all is well so far, we can make sure that we have the latest version of npm installed by running the following command in our Terminal/Command Prompt of choice:

```
PS C:\Users\stuar> npm install -g npm
C:\Users\stuar\AppData\Roaming\npm\npm ->
C:\Users\stuar\AppData\Roaming\npm\node_modules\npm\bin\npm-cli.js
C:\Users\stuar\AppData\Roaming\npm\npx ->
C:\Users\stuar\AppData\Roaming\npm\node_modules\npm\bin\npx-cli.js
+ npm@5.6.0
added 27 packages, removed 11 packages and updated 38 packages in 15.399s
```

Yes, we're using npm to install/update itself, weird! If you're a Windows user building standard ASP.NET MVC applications, then you may have rarely ever needed to leave the comfort of Visual Studio. We're going to be using the command line a lot, so it's time to get used to it!

When installing npm packages, the syntax is as follows:

```
npm install <package-name>
```

This will install the package locally, and store it in a `node_modules` folder, which the tool will create if it doesn't already exist. However, when we updated npm before, we added the `-g` flag to the command. This installs packages *globally* on your machine. Global packages are usually ones which offer command-line interfaces, or CLIs. npm is an example of a CLI, as you can see by our previous usage of its `install` command.

Installing Vue

Now, technically there is no installation for Vue. In its simplest form, Vue is a single JavaScript file that we can include in a web page as we would with any other file. However, we're going to be using it to build a fully-fledged SPA. In this instance, there are a lot of moving parts to configure, as we'll be building the application in a very modular fashion, with several libraries involved and a complex code structure involving multiple files and folders. We don't want to have to reference all of these files from the web pages manually, as it would be a nightmare to ensure they are referenced in the correct order.

We also want to make use of modern JavaScript syntax, which web browsers cannot understand; this means our files will need to be *transpiled* down into a format that they can understand. We'll be using a technology called **webpack** to help us solve both of these problems. webpack is a topic that needs a book of its own to understand, so we won't be covering it in too much detail, but for now just know that it bundles all of our client-side script files into a single file that we can include on our web page. It also handles transpilation for us, so we don't need to worry about whether our users' web browsers are modern enough to understand the latest syntax.

As we don't have anything else that we need to install from a Vue perspective, let's move on to evaluating our options for managing our client-side packages.

npm or Yarn?

npm is a fantastic tool for managing JavaScript packages and dependencies, but it isn't our only option. Facebook have built their own package manager called Yarn, which offers a number of benefits over npm. If you're interested in what those benefits are, head over to the website for more details at <https://yarnpkg.com/en/>.

One of the main reasons why I chose to use Yarn was because it was so much faster than npm. Later versions of npm have definitely closed that gap, but they are still missing some of the other great features that Yarn offers, such as offline mode via local package caching. Yarn makes package installations faster the second time around because it caches every package it installs locally on your machine. The next time you install it, it doesn't need to download it again—this also means that you can work offline as long as you've installed the packages you need before hand!

I'll be using Yarn for the rest of this book, so let's get it installed! Head over to the following URL, which again should detect your OS and show you the installers that are relevant for your platform: <https://yarnpkg.com/en/docs/install>.

Once installed, we can check that everything is OK again by running the following command to detect which version is installed:

```
yarn --version
```

Be aware that if you're running PowerShell on Windows, this may well throw an error stating that *Yarn is not a recognized command*. You're not alone in this, and it seems to be an issue with the .msi installer at the time of writing this book. You can verify that Yarn was indeed installed by switching to a standard Command Prompt instead.

In order to fix this issue on PowerShell, the only way I have found that works is to install Yarn via npm. However, their website explicitly states that this is not the recommended approach, and in fact, they strongly recommend *against* installing it via npm. We're going to look at a nice Terminal Emulator for Windows later in this chapter, but if you really want to use PowerShell, then run the following command:

```
npm install -g yarn
```

Run the version detection command again; it will now work and in actual fact even if we uninstall Yarn via npm, it continues to work:

```
npm uninstall -g yarn
```

This is one of those things that I've yet to find an explanation for, but since the version detection command gets it working, this will do just fine for now. The syntax for installing packages with Yarn is slightly different to that of npm. Yarn uses the `add` function rather than `install`, as follows:

```
yarn add <package-name>
```

This covers everything we need to install for the frontend of our application, so let's move on to what we need for the backend.

Installing backend tools and dependencies

Getting things ready for the backend of our application is very simple. There are only two things that we need to install: ASP.NET Core and PostgreSQL.

Installing ASP.NET Core

To install ASP.NET Core, head over to the following URL, select your OS on the left-hand side and follow the instructions: <https://www.microsoft.com/net/learn/get-started>.



The preceding link will default to the Windows OS when you navigate to it. Make sure you select your own OS before following the download links and instructions.

As with everything else we've installed so far, a CLI tool is included, and as such, we can verify that everything is installed correctly by running the following command in our Terminal:

```
PS C:\Users\stuar> dotnet --version
2.1.3
```

If everything is OK, it should return the version number installed, which at the time of writing this book is version 2.1.3.

Installing PostgreSQL

SQL Server is the standard database choice for most ASP.NET applications and is most likely what the majority of readers are used to. However, seeing as ASP.NET Core is cross-platform now, I don't want to introduce a barrier to those readers who may be running a Mac or Linux machine. SQL Server can be run on Mac or Linux, but requires the use of Docker, which adds a significant degree of complication that is beyond the scope of this book.

PostgreSQL is a fantastic open source RDBMS that runs on any platform and is fully compatible with ASP.NET Core, or more specifically, EF Core. During the development phase of most applications, you are fairly unlikely to notice any difference to when you are working with SQL Server, other than how you connect directly to the database and view your data.

To make use of PostgreSQL over SQL Server, all that is required is a simple one-line configuration change to instruct the application which provider we wish to use. Under the hood, EF Core uses this provider in the migrations files it generates to decide how to manage the creation of, and changes to our database. All of this is hidden from you as the developer, but it is worth knowing the basics of what happens behind the scenes.

To install Postgres, head to the following URL and download the binary packages for your OS: <https://www.postgresql.org/download/>.

As of the time of writing this book, version 10.1 has just been released, so that's the one I'm installing. I've left all the default options checked, including the use of port **5432** and the inclusion of **pgAdmin4**, which we can use later to browse the database.

This is all we need to build the backend of our application; the rest of the dependencies will be installed as we go along via the NuGet package manager.

Choosing and installing an IDE

The obvious choice here is Visual Studio 2017. The community edition is free and offers all of the functionality we're likely to need. However, although Visual Studio is now available for Mac, I've not heard great things on how well it actually works. I don't own a Mac so I can't validate this for myself, but the safer bet for being truly cross-platform is to go with the much lighter weight VS Code.

VS Code is quickly becoming one of—if not the most—popular editor for frontend developers on both Mac and Windows. It is perfect for building the Vue side of our application, thanks to its excellent editing and debugging experience, but is also an exceptionally capable alternative to full-fat Visual Studio for our ASP.NET Core and C# backend. However, if you are on Windows, then you may still wish to use Visual Studio for more complex scenarios.

Now, technically VS Code is not a full IDE; it's a text editor with some IDE-like functionality baked in. For example, we can get IntelliSense, debugging, task running, and version control all builtin and ready to go. This isn't to say that VS Code is a total replacement for full-fat Visual Studio, or the two simply wouldn't need to co-exist. VS Code gives us all the functionality we need, works on any OS, and happens to be much, much faster than Visual Studio.

Let's get it installed. Head to the following URL and download the installer that's appropriate for your platform: <https://code.visualstudio.com/Download>.

Yet again, we can ensure that VS Code was installed properly by running the following command from your chosen command line or Terminal:

```
PS C:\Users\stuar> code -v
1.19.1
0759f77bb8d86658bc935a10a64f6182c5a1eeba
x64
```

At the time of writing this book, the current version is 1.19.1, and Microsoft recently released a 64-bit version, which is also confirmed by the preceding output.

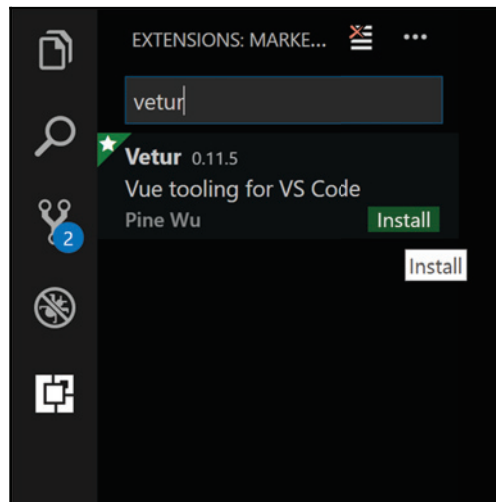
Productivity tools

VS Code already provides a number of built-in functions for navigating our code and performing basic refactoring. However, the one thing it can't do out of the box, which we'll need, is understand Vue's single-file components. These are files with a `.vue` extension, and as we've already seen, it contains HTML, JavaScript, and CSS all in a single file. This makes it difficult for text editors to know what type of file to treat them as, so we'll install a VS Code extension to help out.

Installing VS Code extensions

There are a number of VS Code extensions that we could choose, but **Vetur** is by far the most advanced. It provides some very useful features for building Vue applications, such as syntax highlighting, error checking, IntelliSense, and code snippets. Vetur is also one of the major positives of using VS Code over Visual Studio for Vue application development.

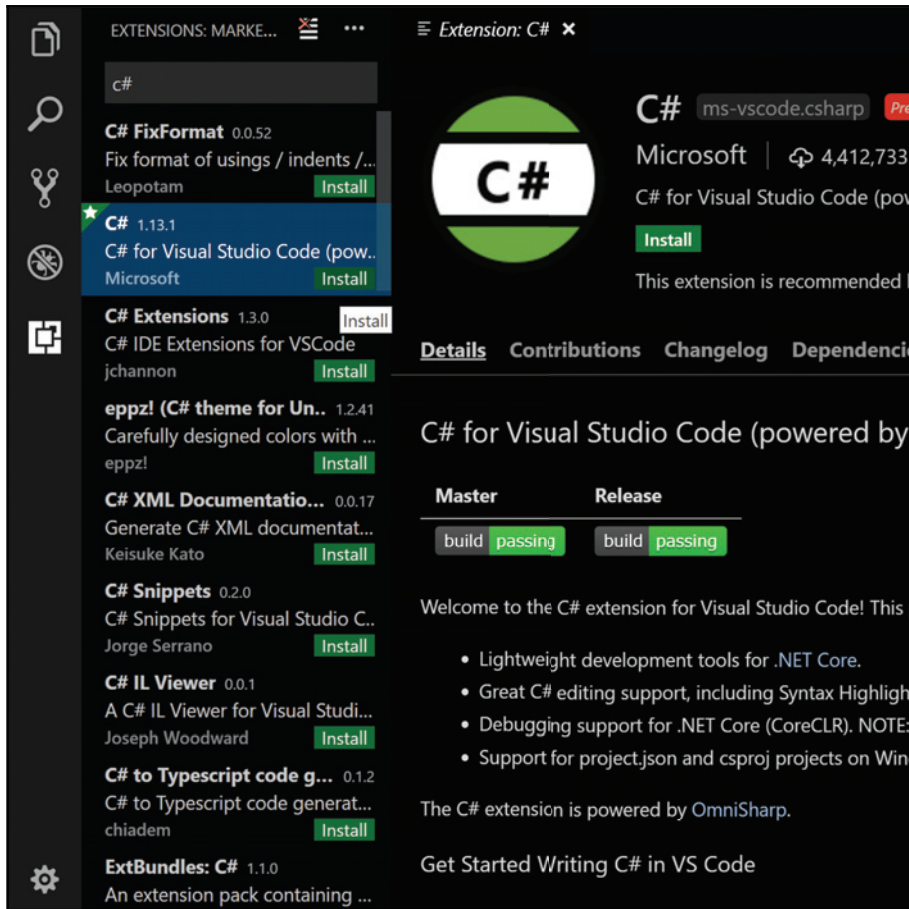
To install it, we need to open VS Code, navigate to the extensions panel from the menu on the left, then search for `vetur`:



VS Code Extensions Panel - Installing Vetur

After clicking the **Install** button and waiting for a few moments, you should see a button to reload the VS Code window. After doing so, the extension will be installed and ready to go.

While we're here, we might as well repeat these steps and install the C# extension in the same way:



VS Code Extensions Panel - Installing C#

There are a few other VS Code extensions that we'll be using, and you can install them in exactly the same way as we've just done for Vetur and C#. These are the following:

- NuGet Package Manager
- C# extensions
- Prettier: A code formatter (optional)

Prettier is an opinionated code formatter that automatically formats your code each time you hit the save button in VS Code. It is entirely optional, but all of the examples within this book have been formatted by Prettier to ensure they are consistent and legible.

Installing the Vue.js Chrome devtools extension

The Vue.js Chrome devtools extension is a plugin for Google Chrome that makes it much easier for us to debug our applications. We'll see how we can make use of it in later chapters, but for now, we just need to get it installed and ready to go. Navigate to the following URL and hit the **Add to Chrome** button near the top: <http://bit.do/eoQJ9>.

Installing a Terminal Emulator on Windows (optional)

We've already seen that some frontend CLI tools don't work as well with PowerShell as they do with Mac/Linux-based Terminals. Unfortunately, Windows just isn't quite as nice to work with when it comes to developing applications using the command line. One thing we can do to improve this is by installing a Terminal Emulator.

ConEmu is one such emulator that works very well. It provides some nice-to-have features, such as tabbed windows, and *usually* does a good job of bringing Unix-based Terminal commands to Windows, so we don't have to keep googling for the Windows equivalent.

If you want to give it a try, head over to the following URL to download and install it: <https://conemu.github.io/>.

Summary

We now have a development environment set up that includes everything we'll need to build and run both a frontend Vue.js application and a backend ASP.NET Core web API. We made the decision to use VS Code for building the application due to its versatility, speed, and the excellent Vue tooling available. Windows users have had the choice to stick with the standard Windows Command Prompt/PowerShell, or install the ConEmu Terminal Emulator.

3

Getting Started with the Project

With our development environment set up and ready to go, it's time to start scaffolding the skeleton of an application. In summary, the topics we are going to cover in this chapter are as follows:

- Evaluating our options for generating a project
- A brief introduction to webpack
- Scaffolding a project with the dotnet CLI
- Refactoring the default template to meet our needs and preferences
- Setting up the database
- Testing the completed setup

Before we do anything else, we need to make a decision on how we want to generate the barebones structure of our project. We have a number of options available to us, including CLI tools from both Microsoft and Vue that have commands dedicated to generating default project structures. On the one hand, we can use each CLI tool independently to generate two separate applications, then try and modify the configuration of each in order to integrate them into a single application. Alternatively, we can make use of a Microsoft provided project template that already integrates an ASP.NET Core backend with a Vue frontend.

ASP.NET Core SPA templates versus CLI tools

Until very recently, this decision was very simple as the SPA templates included with ASP.NET Core did not include a Vue frontend version. This forced our hand, and we had to rely on setting up the backend and frontend of our applications manually, then work out how to integrate them in a nice way. This was not an easy task, as the webpack configurations generated by the Vue CLI are very complicated and opinionated about the folder structure of our frontend code. Trying to tweak this configuration can cause a lot of headaches.

However, Microsoft has now created a Vue-based template. That being said, unfortunately, it is not quite as fully featured as its React or Angular counterparts. At the time of writing this book, it is missing a few core features, such as **server-side rendering (SSR)** and client-side state management. These features are both included in the React template or at least the React and Redux template. SSR is pretty complicated to set up and configure for ourselves, as you'll see later in this book, where we dedicate a full chapter to it.

This is not a deal breaker as, even with the Vue CLI-generated projects, we need to configure SSR ourselves anyway, and state management is very easy to add ourselves. For me, the biggest downside with the Microsoft template is the use of TypeScript. I'm by no means averse to the use of TypeScript if that's what you prefer, but the way it's been used in their template means that each component is broken down into three separate physical files: an HTML template file, a TypeScript file for the component's logic, and a CSS file for its styling. This means that we lose the benefits of Vue's **single-file components (SFCs)**. We are also forced into using the component decorator syntax and class-based API when building our components, which is something that I personally am not a fan of.

It is still very early for official TypeScript support in Vue, and my experience with it so far could only be classed as unstable at best. For example, when scaffolding a sample application using the dotnet CLI template, I am unable to simply run the application without finding at least seven or eight errors relating to TypeScript in the console. This, along with the fact that there are currently far fewer blog posts or sample projects around that make use of TypeScript, means I'll be avoiding it in this book.

Neither of our options are perfect, and require some level of manual intervention into how they are configured. The amount of manual configuration changes needed to remove TypeScript from the dotnet template is far less than trying to integrate two entirely separate applications, so we'll go with that.

An introduction to webpack

So far, we have made quite a few references to webpack without going into any kind of detail as to what it is and why we need to use it. As previously mentioned, we won't be going into too much detail here, but we'll cover enough of the basics to point you in the right direction when it comes to needing to branch away from the default configuration.

What is webpack?

Webpack is what we call a module bundler, and acts as the middleman between our client-side source code and the JavaScript files that actually get run by the browser. Essentially, webpack allows us to build the frontend of a large and complex application as we would with a backend ASP.NET application using many different files and folders. If we think of each file as a module, they can reference other modules using `import` and `export` statements. webpack, then, quite literally *bundles* these modules together so we can reference a single output file in our HTML, as we would with any other JavaScript file.

On top of this, webpack is clever enough to be able to understand all kinds of client-side assets such as JavaScript, CSS, fonts, and images; they can all be imported into modules before being output within a single JavaScript bundle file. Furthermore, webpack can also take care of *transpiling* the latest ES6 JavaScript syntax down into a format that all browsers can understand. This means we get the added benefit of being able to use ES6 syntax in our Vue application, knowing that once our code hits the browser it will be perfectly executable, even on browsers which do not yet support that syntax.

How does it work?

For webpack to work its magic, we must provide a configuration file which instructs it on where to start bundling our code. This is known as the entry point. Webpack will then traverse through our modules using the `import` and `export` statements that link them, building up a dependency graph of the entire application. This graph is how it knows what order to place our code in in the single output bundle file. The name and file location of the output bundle are also configurable using the same configuration file.

In order to work with such a wide variety of asset types, webpack uses the concept of *loaders* to evaluate and process our modules. For example, to process CSS modules, there is a corresponding `css-loader`, which we must add to the configuration before webpack will attempt to load CSS files. JavaScript files are handled by default without an explicit loader, but in order to transpile ES6 syntax, there is a loader called `babel-loader`, which does need configuring. Vue files are particularly interesting because they use a custom file extension, `.vue`, and they can contain HTML, JavaScript, and CSS all within a single file. There is no way webpack could know what to do with these files without a specific loader, which is where the `vue-loader` comes in.

Basic webpack configuration

Webpack is entirely reliant upon configuration, or more specifically, a `webpack.config.js` file residing in the root directory of your application. To begin with, webpack configuration seems incredibly simple and straightforward. Here's an example:

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

We simply point webpack at the `/src/index.js` file as the entry point and tell it to output a `main.js` file in the `/dist` directory. If all we cared about were simple JavaScript files, this would be fine. However, the amount of configuration increases drastically just by configuring the Vue, CSS, and URL loaders:

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  module: {
    rules: [
      { test: /\.vue$/, include: /ClientApp/,
        loader: 'vue-loader' },
      { test: /\.css$/, use: isDevBuild ? [ 'style-loader',
        'css-loader' ] : ExtractTextPlugin.extract
        ({ use: 'css-loader?minimize' }) },
      { test: /\.(png|jpg|jpeg|gif|svg)$/,
        use: 'url-loader?limit=25000' }
    ]
  }
};
```

```
    ]
  },
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

Add on top of that things such as additional loaders, different settings for production/development bundles, minification, and plugins for configuring bundle splitting. You can probably already see where this is going. Webpack is capable of doing some pretty amazing things, but to do so, the configuration can soon get wildly complicated compared to the basic example we started with earlier.

Bundle splitting

One such configuration complication, which is worth doing, is to split your webpack output into at least two bundles. One bundle will contain all third-party code that our own application code depends upon, and the other will just contain our own application's code. There are a few reasons we'd choose to do this, but the most prominent are as follows:

- Faster development time bundling
- Updates to the production application don't require clients to re-download the third-party bundle if they already have it cached

Later in this chapter, we're going to see how a concept called **Hot Module Replacement** works, but essentially every time we make a change to one of our files, webpack will re-bundle the application for us. By using bundle splitting and moving all third-party libraries into a separate *vendor* bundle, webpack will only bundle the code that's changed, which is the much smaller application-specific bundle. This results in faster feedback when we make changes, ultimately increasing our productivity.

Typically, the vendor bundle will have a much larger file size than that of our application bundle. Once a client's web browser has downloaded both bundles when visiting our app for the first time, as long as we've enabled caching, these files won't need to be downloaded again on subsequent visits. However, when we inevitably release a new version of the app, we can re-bundle the application-specific code, adding a new hash to the end of the filename which forces the browser to download it again. As the vendor bundle has remained unchanged, the browser will use the cached version, which saves downloading the bigger file again.

To configure bundle splitting, we must provide a second configuration file in the same location as the first, which instructs webpack on which modules it should split into the vendor bundle. We'll see more on how this works later in this chapter.

Production bundles

It is very common to separate production bundles from development bundles. When we're developing and debugging our application, in the event of an error, it is useful to know exactly which file has thrown it. Remembering that the browser is only executing one or two JavaScript files, how can we tell which of our many source files contained the code causing the bug? The answer is a concept called **source maps**, whereby webpack maintains references to the original source files so that the browser can tell us exactly which file is throwing the error. However, this isn't something we'd want to be enabled once our application is deployed into production, so we tend to have a production-specific bundle which disables it.

In addition, it is standard practice to *uglify* (or *minify*) our code in production to make it illegible to those who inspect it. We wouldn't want to do this in development or we'd find it very difficult to debug, so this is yet another difference that we tend to make between production and development bundles. There are often other differences between production and development bundles, but in my experience, these two are the most common and should be enough to demonstrate this concept. This is all we'll say about webpack for now, as you'll see some more advanced usage throughout the rest of this book. Let's move on to scaffolding our project.

Scaffolding a project with the dotnet CLI

As we've made the choice to be truly cross-platform, we can't use our typically familiar Visual Studio project templates to scaffold the project. Instead, we'll be making use of the dotnet CLI, which gives us exactly the same result anyway.

As the Vue template is a fairly new addition to the CLI, it's not actually installed with the core CLI installer. So, the first step is to download and install it so that we can use it to scaffold our application. Luckily for us, this is as easy as running a single Terminal command:

```
dotnet new --install Microsoft.AspNetCore.SpaTemplates:.*
```

Now, navigate into an empty directory with a name matching your project, and run the following command:

```
dotnet new vue
```

Note that this will not create a directory for you, and will name the application based on the folder you are inside.

If you now open this directory with VS Code, you will see a very familiar project setup, including the standard MVC application folder structure that includes controllers and views; it also includes a `ClientApp` folder that you won't be so familiar with, but we'll come to that later.



When first opening a new project in VS Code, you will be prompted to install missing required assets for building and debugging the application. Click **Yes!**

As mentioned earlier, chances are if you try and run the project now, you'll be hit by a number of TypeScript errors. As we won't be using TypeScript, we won't worry too much about this now and crack on with refactoring the frontend setup to meet our preferences and needs. Depending on whether newer versions of packages are released before you read this book, you may get lucky and it will actually work.

Refactoring the frontend setup

All of our frontend Vue application's code is stored in the `ClientApp` folder. Eventually, this will include all of our application pages, components, client-side router setup, and client-side state management. In order to remove our dependency on TypeScript, there are a couple of things we need to do—remove all references to anything TypeScript-related from our package dependencies and configuration files, and remove or update the existing components to plain old JavaScript components.

Removing TypeScript

We can start by opening up the `package.json` file and removing the following lines:

```
"@types/webpack-env": "^1.13.0",
"awesome-typescript-loader": "^3.0.0",
"bootstrap": "^3.3.6",
"jquery": "^3.1.1",
"typescript": "^2.2.1",
"vue-property-decorator": "^5.0.1"
```

This should leave the entire file looking as follows:

```
{
  "name": "ECommerce",
  "private": true,
  "version": "0.0.0",
  "devDependencies": {
    "aspnet-webpack": "^2.0.1",
    "css-loader": "^0.25.0",
    "event-source-polyfill": "^0.0.7",
    "extract-text-webpack-plugin": "^2.0.0-rc",
    "file-loader": "^0.9.0",
    "isomorphic-fetch": "^2.2.1",
    "style-loader": "^0.13.1",
    "url-loader": "^0.5.7",
    "vue": "^2.2.2",
    "vue-loader": "^11.1.4",
    "vue-router": "^2.3.0",
    "vue-template-compiler": "^2.2.2",
    "webpack": "^2.2.0",
    "webpack-hot-middleware": "^2.12.2"
  }
}
```

Next, we can delete the `tsconfig.json` file entirely, and then make some changes to the `webpack.config.js` file. This is quite tricky, as it's a fairly large and complicated file. The first thing to do is remove the following line, from right near the top of the file:

```
const CheckerPlugin = require('awesome-typescript-loader').CheckerPlugin;
```

Then, around the line 12 mark, there are two lines that look as follows:

```
resolve: { extensions: [ '.js', '.ts' ] },
entry: { 'main': './ClientApp/boot.ts' },
```

As we're not using TypeScript, we don't need to support the `.ts` extension, so we can tweak these lines as follows:

```
resolve: { extensions: [ '.js' ] },
entry: { 'main': './ClientApp/boot.js' },
```

Right beneath these two lines is a `module: { }` object section. Again, we can remove anything related to TypeScript, leaving it looking as follows:

```
module: {
  rules: [
    { test: /\.vue$/, include: /ClientApp/,
      loader: 'vue-loader' },
    { test: /\.css$/, use: isDevBuild ? [ 'style-loader',
      'css-loader' ] : ExtractTextPlugin.extract(
      { use: 'css-loader?minimize' }) },
    { test: /\.(png|jpg|jpeg|gif|svg)$/,
      use: 'url-loader?limit=25000' }
  ]
}
```

A few sections lower down is a `plugins: []` array section, where we can remove the new `CheckerPlugin()` line from the top. This completes our changes to the webpack configuration file, but as it's quite a fiddly process, if you have any problems, make sure to check out the source code for this chapter, which contains the fully completed file.

Replacing the default components

If you open up the `ClientApp/components` folder, you'll find five subfolders, each containing a component generated by the template. These components are currently making use of TypeScript and the class-based API with decorators, rather than true SFCs. We could refactor these into standard JavaScript-based SFC's, but we'd be wasting our time as we'll end up deleting these unnecessary components later anyway.

Delete everything within the `ClientApp/components` directory, as well as the `ClientApp/css` directory itself. To replace the components we just deleted, for now, we'll create a simple `ClientApp/components/App.vue` component to act as the root level component of our application. The contents will look as follows:

```
<template>
  <div>
    <h1>Welcome to Hands on Vue.js with ASP.NET Core!</h1>
    <p>
      The time is: {{ time }}
    </p>
  </div>
</template>

<script>
export default {
  name: 'app',
  data () {
    return {
      time: new Date().toString()
    }
  }
}
</script>
```

Finally, we need to rename the `ClientApp/boot.ts` file to `ClientApp/boot.js`, and change its contents to reflect the following:

```
import Vue from 'vue';
import VueRouter from 'vue-router';

Vue.use(VueRouter);

const routes = [
];

new Vue({
  el: '#app-root',
  router: new VueRouter({ mode: 'history', routes: routes }),
  render: h => h(require('./components/App.vue'))
});
```

For now, we've removed all client-side route definitions as we only have a single component, but we've left the router setup intact to save us putting it back in again later. Run the application now and navigate to `http://localhost:5000/`. If you've got everything configured correctly, we should finally have a running application with a very simple home page:



Testing the initial project setup and App component

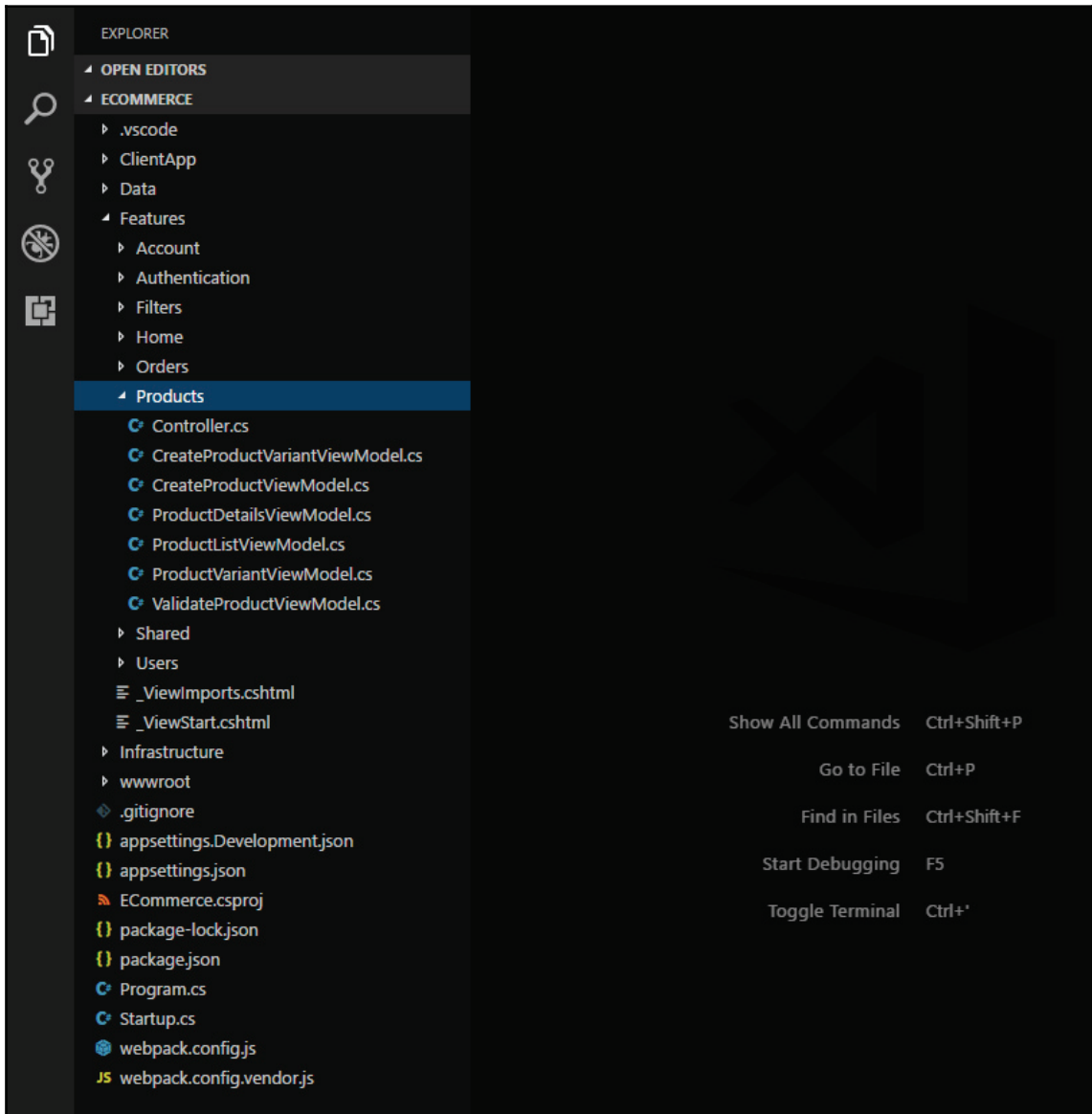
If you add some additional markup to the `App.vue` template, you'll notice that the browser updates and displays the new content immediately, without the need for a page refresh. This is thanks to a technology called Hot Module Replacement, which is configured for us using webpack-based middleware in ASP.NET Core.

This completes the changes we need to make to the frontend setup, so let's move on to the backend.

Refactoring the backend setup

I'm a huge fan of the feature folder approach to structuring ASP.NET Core applications. As you will probably already know, in the default application folder structure of a typical MVC application, we have separate directories for controllers, models, and views. As our application grows and we add more and more features, we spend a lot of time switching between these folders in our IDE or editor. It can also become quite difficult to quickly find the specific files we're looking for, particularly in the `Models` folder, as you'll usually have a handful of different view models per database entity.

An alternative approach is to group these controllers, models, and views into a folder per feature of the application. For example, the `Features/Products` directory would contain the `ProductsController`, right alongside the views and view models that are necessary to display and manage the `Product` database entity:



Feature folder structure

This creates a far more cohesive file structure, where all the files that you are likely to be changing at once are located within the same directory of the project. In my opinion, this makes us more productive, as we don't waste time switching from folder to folder hunting for the files we need to modify. The preceding screenshot shows a feature folder in a Web API project where we don't have any views or static assets such as CSS and JavaScript, but the same concept applies in an MVC application. Place all the views, CSS, JavaScript, images, and other things that belong to a feature within its feature folder. I hope you'll be pleasantly surprised as to how much easier the project is to maintain just by making a simple folder structure change.

Refactoring to a feature folder structure

There are a few steps that we need to take to achieve this desired outcome. The first thing to do is rename the `Views` folder to `Features`, then move the `HomeController.cs` file into the `Features/Home` directory. We can now delete the `Controllers` and `Models` directories entirely, including the obsolete `SampleDataController.cs` file that we're not using. If you try and run your application now, you'll see an exception page because ASP.NET can no longer find the index view that the controller is trying to render.

It doesn't matter where we put our controllers, as ASP.NET Core can locate them using the `[Feature]Controller.cs` naming convention. However, if we change the default name of the `Views` folder, we need to tell ASP.NET Core how to find them as by default, it only knows to look for a folder named `Views`. There are two steps to do this, the first of which is to create a class that instructs ASP.NET Core where we want to search for view files, which I'll name `FeatureLocationExpander.cs`:

```
namespace ECommerce.Infrastructure
{
    public class FeatureLocationExpander : IViewLocationExpander
    {
        public IEnumerable<string>
            ExpandViewLocations(ViewLocationExpanderContext
                context, IEnumerable<string> viewLocations)
        {
            return new[] { "/Features/{1}/{0}.cshtml",
                "/Features/Shared/{0}.cshtml" };
        }

        public void PopulateValues(ViewLocationExpanderContext
            context)
        {
        }
    }
}
```

```
}
```

It doesn't matter where you put this file, but I like to put configuration files like this in a folder called `Infrastructure`. Next, we need to register this class so that ASP.NET Core knows about it and can start looking for our views in the correct location. Find the `ConfigureServices` method in `Startup.cs` and modify it as follows:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.Configure<RazorViewEngineOptions>(options =>
    {
        options.ViewLocationExpanders.Add(new
            FeatureLocationExpander());
    });
}
```

Restarting the application now should have everything working properly again.

Setting up the database

If you're on Windows as I am, then your database server should already be up and running as a Windows service that starts automatically when your machine boots up. If you're on Mac/Linux, this may or may not be the case, so just ensure that the database server is running—the PostgreSQL website will have instructions on how to do this if necessary.

Creating a database context

EF Core will actually handle the creation of the database; we just need to tell it what kind of RDBMS we're using, and provide a connection string for its location and authentication credentials. Again, it doesn't really matter where we put the following files, but my preference is a `Data` folder for anything to do with EF setup/configuration, entity models, and seed data.

The first file we'll need to create is an EF database context class:

```
namespace ECommerce.Data
{
    public class EcommerceContext : IdentityDbContext<AppUser,
        AppRole, int>
    {
        public EcommerceContext(DbContextOptions<EcommerceContext>
```

```
        options) : base(options)
        { }
    }
}
```

Notice how our context inherits from `IdentityDbContext`, rather than the base `DbContext` class. We're using the ASP.NET Core Identity class because it will help us with user authentication and management later in this book. Here, we're instructing EF and Identity that we'll be using classes called `AppUser` and `AppRole` for our application's users and roles entities, respectively. We've also specified that we'll be using integers for the primary keys of all ASP.NET Core Identity-generated database tables.

These classes don't exist yet, so let's create them so that we can check if the application still builds. Again, my preference is to store my database entities in the `Data/Entities` directory, but this is entirely up to you. The `AppUser` class is, at least for now, as follows:

```
namespace ECommerce.Data.Entities
{
    public class AppUser : IdentityUser<int>
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }

        [NotMapped]
        public string FullName
        {
            get { return $"{FirstName} {LastName}"; }
        }
    }
}
```

Again, we're inheriting from an ASP.NET Core Identity class and instructing it to use an integer for the primary key. This is all pretty standard setup for ASP.NET Core applications that use EF Core and Identity Core, so I won't go into the details, but it's worth mentioning that if we don't inherit from this `IdentityUser` class, our database context class will complain.

The `AppRole` class also inherits from an Identity base class, and looks like this:

```
namespace ECommerce.Data.Entities
{
    public class AppRole : IdentityRole<int>
    {
        public AppRole() { }

        public AppRole(string name)
```



```
        {  
            Name = name;  
        }  
    }  
}
```

Our application should now build successfully, but we still have a way to go before we're finished.

Registering the database context for DI

In order to make use of the database context class in our application, we need to register it with the built-in DI container. To do this, we first need to install the `Npgsql.EntityFrameworkCore.PostgreSQL` and `Microsoft.EntityFrameworkCore.Design` NuGet packages. You can do this using the VS Code extension we installed earlier, using the dotnet CLI, or by manually adding them to the projects `.csproj` file. We then need to drop back into the `ConfigureServices` method of the `Startup.cs` class, and modify it as follows:

```
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddDbContext<EcommerceContext>(options =>  
        options.UseNpgsql(Configuration.  
            GetConnectionString("DefaultConnection")));  
  
    services.AddIdentity<AppUser, AppRole>()  
        .AddEntityFrameworkStores<EcommerceContext>()  
        .AddDefaultTokenProviders();  
  
    // rest of method as before...  
}
```

The first section registers the database context with the DI container, while instructing it to use PostgreSQL with the `options.UseNpgsql()` statement. We are also passing in a reference to a connection string, which we'll create shortly.

The second section is preempting our use of ASP.NET Core Identity later in this book, and instructing it to use the database context we've just created, along with the default EF stores for Identity-related data.



We don't really need to worry about all this ASP.NET Core Identity setup just yet, but it's easier to configure it now than to come back later and make changes!

Creating the database

There are a few different ways that we can choose to actually create the database, but regardless of which one we choose, we need to supply `ConnectionString`. We've already told the DI container to use `Configuration` to look for a connection string named `DefaultConnection`, so let's create one. Open the `appsettings.json` file in the project root, and modify it as follows:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "host=localhost;database=ecommerce;
      username=postgres;password=postgres"
  },
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

Make sure you use the appropriate username and password for your local database server. I'm keeping things simple and sticking with the defaults that the Windows installation provides. We are now ready to create the database, but there is one more NuGet package that we need to install. This is where it gets a little more complicated, as we can't rely on the VS Code extension because it places it in the wrong place within the project file.

This isn't the most user-friendly or streamlined process, but for now it will have to do. Open up the project file for your application and modify it as follows:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
```

```
<PackageReference Include="Microsoft.AspNetCore.All"
Version="2.0.3" />
<PackageReference Include="Microsoft.EntityFrameworkCore.Design"
Version="2.0.1" />
<PackageReference
Include="Npgsql.EntityFrameworkCore.PostgreSQL"
Version="2.0.0" />
</ItemGroup>
<ItemGroup>
<DotNetCliToolReference
Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools"
Version="2.0.1" />
<DotNetCliToolReference
Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
Version="2.0.0" />
</ItemGroup>
<!-- rest of file omitted for brevity... -->
</Project>
```

Notice that there is a difference between `PackageReferences` and `DotNetCliToolReferences`. If we use the VS Code extension to install packages, they all end up as package references, but the `Microsoft.EntityFrameworkCore.Tools.DotNet` package must be referenced as a CLI tool reference.

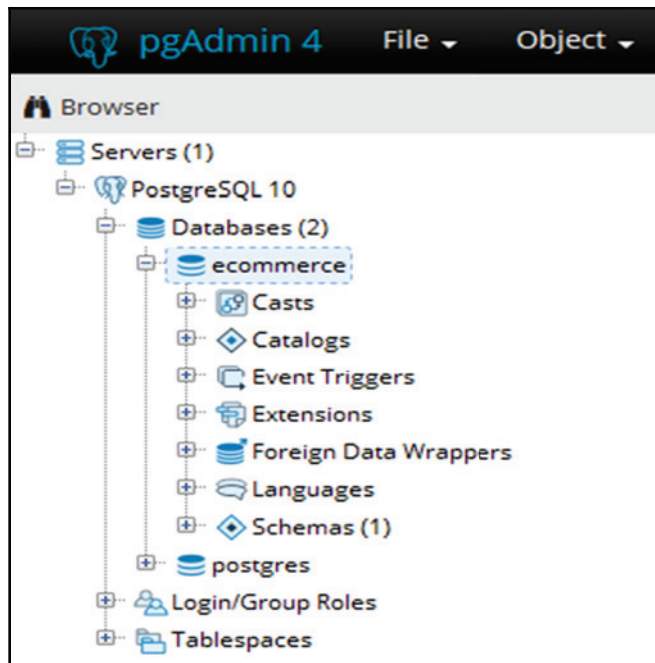
Make sure that you trigger package restoration, then with these changes we now have access to the `dotnet ef` CLI command, and can create our database by dropping into the Terminal and running the following command:

```
dotnet ef database update
```

All being well, you should see some output that ends with a statement such as the following:

```
No migrations were applied. The database is already up to date.
Done.
```

This is fine, as we have not yet told EF about any of our database tables, but if we open the pgAdmin 4 application that gets installed as part of the PostgreSQL installation, we can verify that the database has indeed been created:



pgAdmin 4 local database listing

Creating an initial migration

In order to have EF generate our tables for us, we need to create a *migration*. Migrations are a way of keeping the database schema in sync with our C# application model. When we make changes to the model, we have to add a migration. Migrations are run when we instruct EF to update the database, where a history of previously run migrations is stored to prevent running them more than once.

We can create an initial migration by running the following Terminal command:

```
dotnet ef migrations add Initial
```

If we go and look at our project in VS Code again, you'll notice a new folder in the project root labelled `Migrations`. As these are database-related, my preference is to drag this folder under the `Data` directory, and then any future migrations will automatically follow. There should be three files generated, but the one we are interested in starts with a timestamp and ends with `_Initial.cs`. Its contents should look a little like this:

```
namespace ECommerce.Data.Migrations
```

```
{
    public partial class Initial : Migration
    {
        protected override void Up(MigrationBuilder
migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "AspNetRoles",
                columns: table => new
                {
                    Id = table.Column<int>(nullable: false)
                    .Annotation("Npgsql:ValueGenerationStrategy",
NpgsqlValueGenerationStrategy.SerialColumn),
                    ConcurrencyStamp = table.Column<string>
                    (nullable: true),
                    Name = table.Column<string>(maxLength: 256,
                    nullable: true),
                    NormalizedName = table.Column<string>
                    (maxLength: 256, nullable: true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_AspNetRoles", x => x.Id);
                });

            // rest of file omitted for brevity...
        }
    }
}
```

The code is quite readable, and it's easy to see what the generated SQL will do when it's run against the database. I'd recommend having a good look at each migration file before you run them, as it's much easier to remove one now than it is to remove it after it hits the database. If there is anything wrong with it, it is better to make some changes and regenerate it before applying it to the database.

With this in place, we can now run our `update` command again, but this time we should end up with a few more tables in our database:

```
dotnet ef database update
```

You can verify this using the `pgAdmin` application again.

Creating and seeding the database on start-up

We could rely on manually creating and updating the database as and when we need it, but a better approach while we're in development is to have the application ensure that the database is created on application startup.



You may not wish to use this approach for production applications, but generally speaking, there is no issue with using this method in development/testing environments!

We'll use the built-in `Migrate` method on the database context class, as well as create a custom extension method to make sure it is populated with some seed data as it can make our lives easier if and when we need to drop and recreate the database in the future. Create the following `DbContextExtensions` class in the `Data` directory:

```
namespace ECommerce.Data
{
    public static class DbContextExtensions
    {
        public static UserManager<AppUser> UserManager
        { get; set; }

        public static void EnsureSeeded(this EcommerceContext
context)
        {
            if (UserManager.FindByEmailAsync("stu@ratcliffe.io").
GetAwaiter().GetResult() == null)
            {
                var user = new AppUser
                {
                    FirstName = "Stu",
                    LastName = "Ratcliffe",
                    UserName = "stu@ratcliffe.io",
                    Email = "stu@ratcliffe.io",
                    EmailConfirmed = true,
                    LockoutEnabled = false
                };

                UserManager.CreateAsync(user,
>Password1*").GetAwaiter().GetResult();
            }
        }
    }
}
```

As this is a static class, we cannot instantiate it, and as such cannot make use of DI for the dependencies that we need. In order to gain access to an instance of the `UserManager` class, we need to tell the application to provide one at start up. Open up the `Startup.cs` class again, and add the following statement to the bottom of the `ConfigureServices` method:

```
public void ConfigureServices(IServiceCollection services)
{
    // rest of class removed for brevity...
    DbContextExtensions.UserManager =
        services.BuildServiceProvider().
            GetService<userManager>();
}
```

Next, open up the `Program.cs` file in the root directory, locate the `Main` method, and modify it as follows:

```
public static void Main(string[] args)
{
    var host = BuildWebHost(args);

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        var dbContext =
            services.GetRequiredService<EcommerceContext>();

        dbContext.Database.Migrate();
        dbContext.EnsureSeeded();
    }

    host.Run();
}
```



Note that without registering our database context class with the DI container earlier, it wouldn't be available to us in this way!

We can confirm that everything is now working by first dropping our current database with the following Terminal command:

```
dotnet ef database drop
```

And then we can run our application from the **Debug** tab of VS Code before checking pgAdmin again to make sure that the database has been created.



We can also run the application as we do in Visual Studio, by pressing *F5*. There is also a dotnet CLI command we can use: `dotnet run`.

Testing the completed setup

Before we wrap up this chapter, let's quickly test that everything is configured correctly. A feature we'll most likely need later on is a users feature, so let's create a `Features/Users/UsersController.cs` file, and fill it in with the following contents:

```
namespace ECommerce.Features.Users
{
    [Route("api/[controller]")]
    public class UsersController : Controller
    {
        private readonly EcommerceContext _db;

        public UsersController(EcommerceContext db)
        {
            _db = db;
        }

        [HttpGet]
        public async Task<IActionResult> Get()
        {
            return Ok(await _db.Users.ToListAsync());
        }
    }
}
```


This is all we need to test the completed setup, so run the application again now and navigate to `http://localhost:5000/api/users` in your web browser. You should be presented with a JSON array containing a single object representing the user we seeded in the database earlier:

```
[
  - {
    firstName: "Stu",
    lastName: "Ratcliffe",
    fullName: "Stu Ratcliffe",
    id: 1,
    userName: "stu@ratcliffe.io",
    normalizedUserName: "STU@RATCLIFFE.IO",
    email: "stu@ratcliffe.io",
    normalizedEmail: "STU@RATCLIFFE.IO",
    emailConfirmed: true,
    passwordHash: "AQAAAAEAACcQAAAAEDwecWxIDzkBy8",
    securityStamp: "51829d68-ec88-49bf-a634-b39c8",
    concurrencyStamp: "6dfedd34-9204-4fc3-bf54-fe",
    phoneNumber: null,
    phoneNumberConfirmed: false,
    twoFactorEnabled: false,
    lockoutEnd: null,
    lockoutEnabled: true,
    accessFailedCount: 0
  }
]
```

Users API request JSON response



In order to have Chrome format all JSON responses like in the preceding screenshot, install the **JSONView** extension!

So, what exactly are we validating here? For a start, the MVC middleware is registered and working properly, or the browser wouldn't know what to do with the URL we supplied. We also know that our DI container is functioning correctly and that our database context is registered with it; otherwise, the users controller we just created would not have received an instance in the constructor parameter we defined. Finally, we know that our database context has successfully created and seeded our database, and is able to query it for data.

To really finish off testing our application end-to-end, let's display some of this data in our frontend app component. Open up `ClientApp/components/App.vue` and modify the `<template>` section as follows:

```
<template>
  <div>
    <h1>Welcome to Hands on Vue.js with ASP.NET Core!</h1>
    <p>
      The time is: {{ time }}
    </p>
    <p>
      The current users of our system are:
      <ul>
        <li v-for="user in users" :key="user.userName">
          {{ user.fullName }} - {{ user.userName }}
        </li>
      </ul>
    </p>
  </div>
</template>
```

Add a `users` property to the data object, and a mounted component hook to fetch the data from the API as follows:

```
<script>
export default {
  name: 'app',
  data () {
    return {
      time: new Date().toString(),
      users: []
    }
  },
  mounted () {
    fetch('/api/users')
      .then(response => {
        return response.json()
      })
      .then(data => {
        this.users = data
      })
  }
}
</script>
```

If the application isn't already running, start it up now! At the bottom of the homepage, you should see the user that we seeded into the database earlier.

Summary

We've covered a lot of ground in this chapter, so let's have a quick recap.

Building SPA frontends is a complicated process with a lot of moving parts. Configuring everything ourselves manually is not only a very tedious and error-prone process, but also a very complicated one. In that sense, we evaluated our options for generating our project using the dotnet and Vue CLI tools, electing to go with the newly released dotnet CLI template with a Vue.js frontend.

We decided that using TypeScript with Vue was still a little too unstable, and as such refactored the frontend to remove TypeScript and tweak the configuration to expect plain old JavaScript instead. We also made some changes to the backend folder structure by implementing a feature folder approach, rather than the stereotypical `Controllers/Models/Views` folders.

We set up a database and the associated EF Core configuration files to interact with it, and instructed ASP.NET that we'll be using the Identity library for user accounts later in this book. We looked at how to run commands to manipulate the database directly from the command line, as well as how to have our application create and seed the database as part of its startup process.

Finally, we tested the entire application by adding a feature to query the database for all users, return them to the client, and display them in our client-side app component.

4

Building Our First Vue.js Components

With our project set up and ready to go, it's time to start writing some more involved client-side code and build our first custom components. We'll start by creating a component to display a static list of products, with the option of selecting a product to view its full details. Then, we'll finish things off by creating some API endpoints to provide real data and hook our components up to them.

In summary, we'll be covering the following topics:

- Component composition
- Rendering collections of data with the `v-for` directive
- Data binding with the `v-bind` directive
- Event handling with the `v-on` directive
- Conditional rendering with the `v-show` and `v-if` directives
- Client-side routing
- Fetching data from an API

Displaying a list of products

As we'll be building a full e-commerce application, it makes sense to start things off with a list of fictional products that we are going to sell. We'll keep things simple to start with and keep this component contained within our home page before we introduce the additional complexity of multiple pages.

When first starting out with Vue, it is very easy to just keep dropping all of our components in the `ClientApp/components` folder. However, it will soon get very difficult to find what we're looking for once we have more than a handful of components to maintain. Instead, I tend to group my components by page or feature, with a folder for each inside the `components` directory. It doesn't really matter what we name these features, as long as it's obvious to us which components belong to each one.

Create a `ClientApp/components/products/List.vue` file. The `<template>` section will eventually contain two sections, one for the list of products and one for showing the full details of the currently selected product. However, the product list template should currently look as follows:

```
<template>
  <div class="products">
    <h1>Products</h1>
    <div class="list">
      <div class="item" v-for="product in products"
        :key="product.slug">
        <h3 @click="select(product)">{{ product.name }}</h3>
        
        <p @click="select(product)">{{ product.shortDescription }}
        </p>
        <p @click="select(product)">{{ product.price }}</p>
        <hr />
      </div>
    </div>
  </div>
</template>
```

The first thing to note here is that all Vue components must only contain one root element. For most of our components, it is fairly safe to simply wrap everything in a `div` tag as we have done here. However, certain components intended to render specific tags, such as list items or table rows, would be wrapped in a single `li` or `tr` tag instead.

The next most important line is the following:

```
<div class="item" v-for="product in products" :key="product.slug">
```

Here, we are using the `v-for` directive to loop over a list called `products`, and render the contents of this `div` element for each item in the list. Also notice how we use the shorthand for the `v-bind` syntax to bind a `key` property to the element. This `key` property helps Vue to manage the ordering of items in dynamic lists that may change at runtime. It isn't required, and the component will work perfectly fine without it. However, it certainly can't hurt to provide it, and it can seriously improve the performance of the application in certain circumstances, so there really isn't a reason not to.



If you're using VS Code and you followed along in [Chapter 2, Setting Up the Development Environment](#), you'll have installed the Vetur extension, in which case you'll notice that if you fail to provide a `key` property, it will underline this line of code in red until you add it!

The last thing to know about the `key` property is that it has to be unique to the list item being rendered, or it won't do much good in helping to track the items in the list. The ideal value to use is some form of ID value if we are rendering items that are stored in a database; we aren't pulling this data from the API yet, so we have no real ID values, but we will be using a `slug` property as another unique identifier for a product so that we can use nice friendly URLs for our pages later in this chapter.

The `v-for` directive also binds a variable representing the list item to the context of the element. In this case, we bound a variable called `product`. We then made use of this variable in a bunch of other elements as follows:

```
<h3 @click="select(product)">{{ product.name }}</h3>
<p @click="select(product)">{{ product.shortDescription }}</p>
<p @click="select(product)">{{ product.price }}</p>
```

With the `h3` and `p` elements, it is a simple case of data binding using string or text interpolation. Any expression placed between a pair of double curly braces, including the braces themselves, is replaced by the value of that expression. In this case, we simply bind the `name`, `shortDescription`, and `price` properties of the `product` variable to the associated HTML elements.

Where things get a little more complicated is with the `img` element that we rendered:

```

```

Again, we used the `v-bind` shorthand to data bind the `src` and `alt` attributes of the image. This is a great example of where we take a standard HTML attribute and use Vue to interpret it dynamically at runtime. If we hadn't included the colon before the attributes, they would have been treated as normal, that is, as plain text. Instead, we are able to bind the properties of our `product` variable, which includes a URL to an image that can be used as a thumbnail for the product.

Finally, on each of these elements we also attached a `click` event handler using the `@` shorthand syntax. This `click` event handler is used to trigger a method, or function, that we'll see contained in the `script` section of the component shortly. As with data binding, we have access to the `product` variable declared in the loop, and can pass it as a parameter of this method. This enables us to perform any logic or operations based on the specific product that the user clicks on, such as adding a specific product to a shopping cart, or in this instance selecting a product to view more information about it.

Let's have a look at the `script` section of this component:

```
export default {
  name: "product-list",
  data() {
    return {
      products: [
        //products omitted for brevity
      ],
      selectedProduct: null
    };
  },
  methods: {
    select(product) {
      this.selectedProduct = product;
    }
  }
};
```

There are a few things going on in here, so let's go over this line by line.

First off, we have the `export default { }` declaration. In order to use this component within another component or page of our app, we need to declare it as a child component, and to do that we need to be able to import it. This line tells webpack that this is a module that can now be imported and used within other modules, or components, in our case. Without it, when we try and display this component within our main `App.vue` component, it will not be able to find anything to import and display.

Next up, we have the `name` declaration. This isn't a requirement, and the component would work perfectly fine if we omitted it. However, there are a few benefits to being vigilant and remembering to name each of our custom components. First of all, it makes debugging our components much nicer when using the Chrome DevTools extension that we installed earlier. We can inspect the application with the dev tools to see a nice component tree—if we don't name the component, then it shows up as `<AnonymousComponent>`, which makes it difficult for us to tell what's what. Secondly, it allows a component to recursively invoke itself from within its `template` section. However, this isn't a particularly common requirement apart from in tree-like structures.

The data section is next, which as we already know must be a function that returns an object. We have two simple properties in this component: a `products` array that, as its name suggests, is a collection of product objects; and a `selectedProduct` object, which again is fairly self-explanatory and holds the product that the user selects to display further details for. The only thing missing here is the actual product objects within the `products` array, as the actual data isn't particularly important or interesting to read. Considering these object definitions only differ very slightly in their property values, I chose to omit them to save space. However, as an example, here is one of those product objects:

```
{
  name: "Samsung Galaxy S8",
  slug: "samsung-galaxy-s8",
  thumbnail: "http://placeholder.it/200x300",
  shortDescription:
    "Samsung Galaxy S8 Android smartphone with
     true edge to edge display",
  price: 499.99,
  description:
    "Lorem ipsum dolor sit amet consectetur adipisicing elit.
    Perferendis tempora ad cum laudantium, omnis fugit amet iure animi corporis
    labore repellat magnam perspiciatis explicabo maiores fuga provident a
    obcaecati tenetur nostrum, quidem quod dignissimos, voluptatem quasi? Nisi
    quaerat, fugit voluptas ducimus facilis impedit quod dicta, laborum sint
    iure nihil veniam aspernatur delectus officia culpa, at cupiditate? Totam
    minima ut deleniti laboriosam dolores cumque in, nesciunt optio! Quod
    recusandae voluptate facere pariatum soluta vel corrupti tenetur aut
    maiores, cumque mollitia fugiat laudantium error odit voluptas nobis
    laboriosam quo, rem deleniti? Iste quidem amet perferendis sed iusto
    tempora modi illo tempore quibusdam laborum? Dicta aliquam libero, facere,
    maxime corporis qui officiis explicabo aspernatur non consequatur mollitia
    iure magnam odit enim. Eligendi suscipit, optio officiis repellat eos quis
    iure? Omnis, error aliquid quibusdam iste amet nihil nisi cumque magni
    sequi enim illo autem nesciunt optio accusantium animi commodi tenetur
    neque eum vitae est."
}
```


The other five objects in the array are very similar; just make sure the `slug` property is unique or the component won't render! Each of these properties are referenced on one element or another in the `template` section, but we can only see the full description on the selected product.



We can scaffold out a UI with placeholder images from `http://placeholder.it/` by passing the size of the image in the query string as we did just now!

The final section is the `methods` object, which contains a single method for selecting a product to view further details about it. We've already seen how this method is triggered using `click` events on each of the elements within the list item, passing the product currently in the loop context to this method for processing. All it does is simply set the product that's been clicked on as the `selectedProduct` within the `data` function of the component. As our component stands currently, triggering this method will have no impact.

Conditional rendering

We need to modify the `template` section to add a section that is only displayed once a product is selected from the list:

```
<template>
  <div class="products">
    <h1>Products</h1>
    <div class="list"></div> <!-- product list omitted
    for brevity -->
    <div v-if="selectedProduct" class="details">
      <h1>{{ selectedProduct.name }}</h1>
      
      <p>{{ selectedProduct.shortDescription }}</p>
      <p>{{ selectedProduct.description }}</p>
      <p>{{ selectedProduct.price }}</p>
    </div>
  </div>
</template>
```

With the addition of this `div` element, the `select` method now has a purpose. When the component is first rendered, the `selectedProduct` property is initialized as `null` which evaluates to `false`, and as such the section is not rendered. When the user clicks on an element within the list, the `select` method is fired with the selected product as an argument. The `select` method then assigns this product to the `selectedProduct` property, which is enough to cause the Vue reactivity system to perform a UI refresh. Now that the `selectedProduct` property has a value, the `v-if` directive on the `div` element we just added evaluates to `true`, and the product details section is rendered, showing the product the user selected.

Once a product is selected, selecting a different product still causes the UI to refresh as the reactivity system detects that the UI is bound to a property that has been changed; the details of the newly selected product overwrites the previously selected one. Finally, it is worth noting at this point that, as the state of this component is stored in memory in the browser, performing a browser refresh will clear the selected product and the details section will no longer be rendered.

The final section in the component declaration is the `style` block. This isn't a book about CSS, so we won't go over this in too much detail, but the styles for this component are as follows:

```
<style>
* {
  box-sizing: border-box !important;
}
.products {
  padding: 20px;
  max-width: 1200px;
  margin: 0 auto;
}
.list,
.details {
  width: 50%;
  float: left;
}
.list .item {
  width: 50%;
  float: left;
  padding: 20px 10px 20px 0;
}
.list img,
.list h3,
.list p {
  cursor: pointer;
}
```

```
.list img {  
  width: 100px;  
}  
</style>
```

What all these styles do is add a little padding to the entire page to bring the content in off the sides of the screen; display the product list and details sections side by side with 50% of the width of the page each; and display the product list items in a two-column grid. This completes the definition of this component, but as yet we aren't actually displaying it anywhere!

Component composition

The first thing to do is import this component into the `ClientApp/components/App.vue` file, and register it as a child component of our main `App` component. Replace the entire `script` block of the `App` component with the following:

```
<script>  
import ProductList from "../products/List.vue";  
export default {  
  name: "app",  
  components: {  
    ProductList: ProductList  
  }  
};  
</script>
```

The first thing to note here is the `import` line at the top of this block. If you're new to frontend frameworks, this might not make a lot of sense to you. In the modern JavaScript world, we break the application up into chunks, usually also splitting it into separate physical files, which are known as **modules**. A module will export some piece of functionality which can then be imported into another dependent module using a line of code like this one. This can loosely be thought of as a `using` statement in a C# application, where the object or function exported from a module is its interface defining the functionality that can be used from a dependent module.

Any parent component will have a number of these lines depending on how many child components it needs to import in order to make use of them in its `template` section. These imported components are then registered as children of this parent component using a `components` object. This is an object containing a collection of key/value pairs where the key is the name we want to reference the child component by, and the value is the object we imported from the child component definition we mentioned previously.

Generally speaking, the key and value of these properties will be the same as they are in the preceding example. Therefore, as our application supports modern ES6 syntax, we can shorten this property declaration as follows:

```
components: {  
  ProductList  
}
```

When this code is transpiled by webpack into standard syntax that the browser can understand, this will be expanded into the full key/value pair we had earlier.

Another bit of syntax specific to ES6 is our actual `import` line. This syntax is fairly simple:

```
import ComponentName from “./relative/component/location”
```

The `ComponentName` part of this `import` line does not need to correspond to the name we gave the component earlier, as we are not using named exports from the child component definition. Instead, we are using `export default {...}`, which means that there is only a single export from the module, and as such we can use whatever name we wish when importing, as there is only a single object or function to import anyway. For example, we could use the following if we wanted:

```
import FooComponent from “./products/List.vue”
```

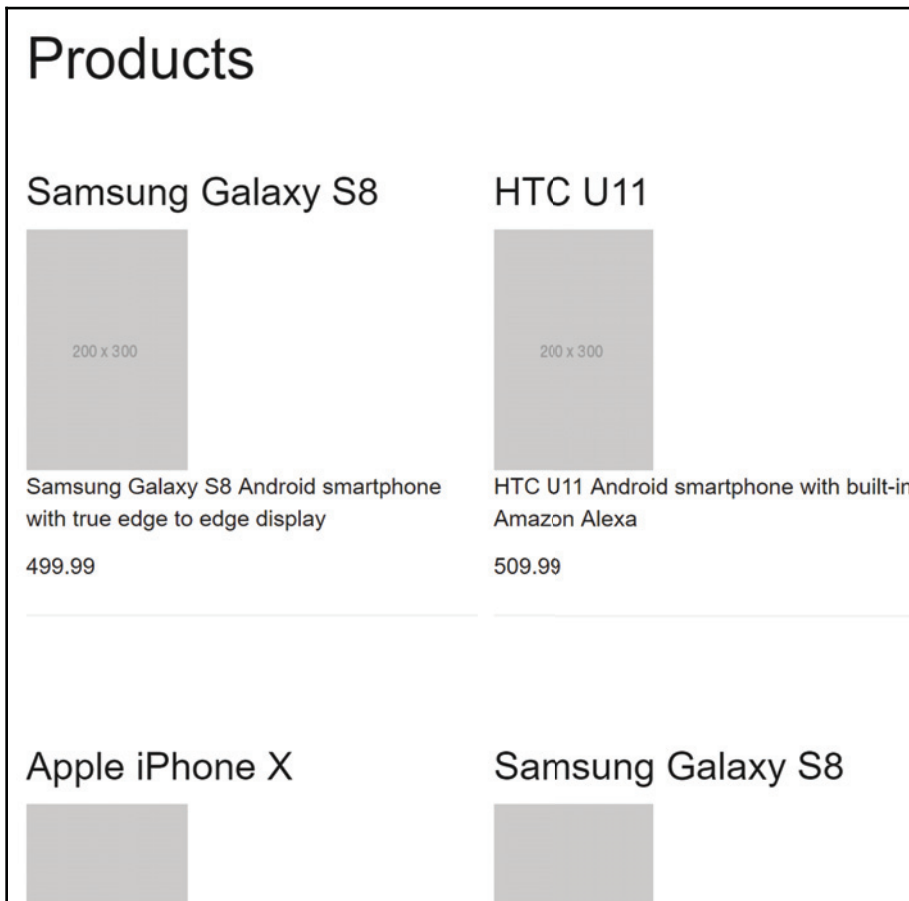
The only difference would be when we have to render the child component within the `template` section of the parent component. This is because the name we use while importing the child component is used to determine the HTML element name we use to render it. Regardless of whether we chose to use kebab case (`component-name`), camel case (`componentName`), or pascal case (`ComponentName`) for the import name, we must use the kebab case equivalent for the HTML element when we enter into the `template` section. In this instance, we used pascal case for our `ProductList` component import, and when rendering this, we do so as follows:

```
<template>  
  <div>  
    <product-list />  
  </div>  
</template>
```

On the other hand, if we had indeed named our component `FooComponent` when importing it, we would simply change the HTML element to the following:

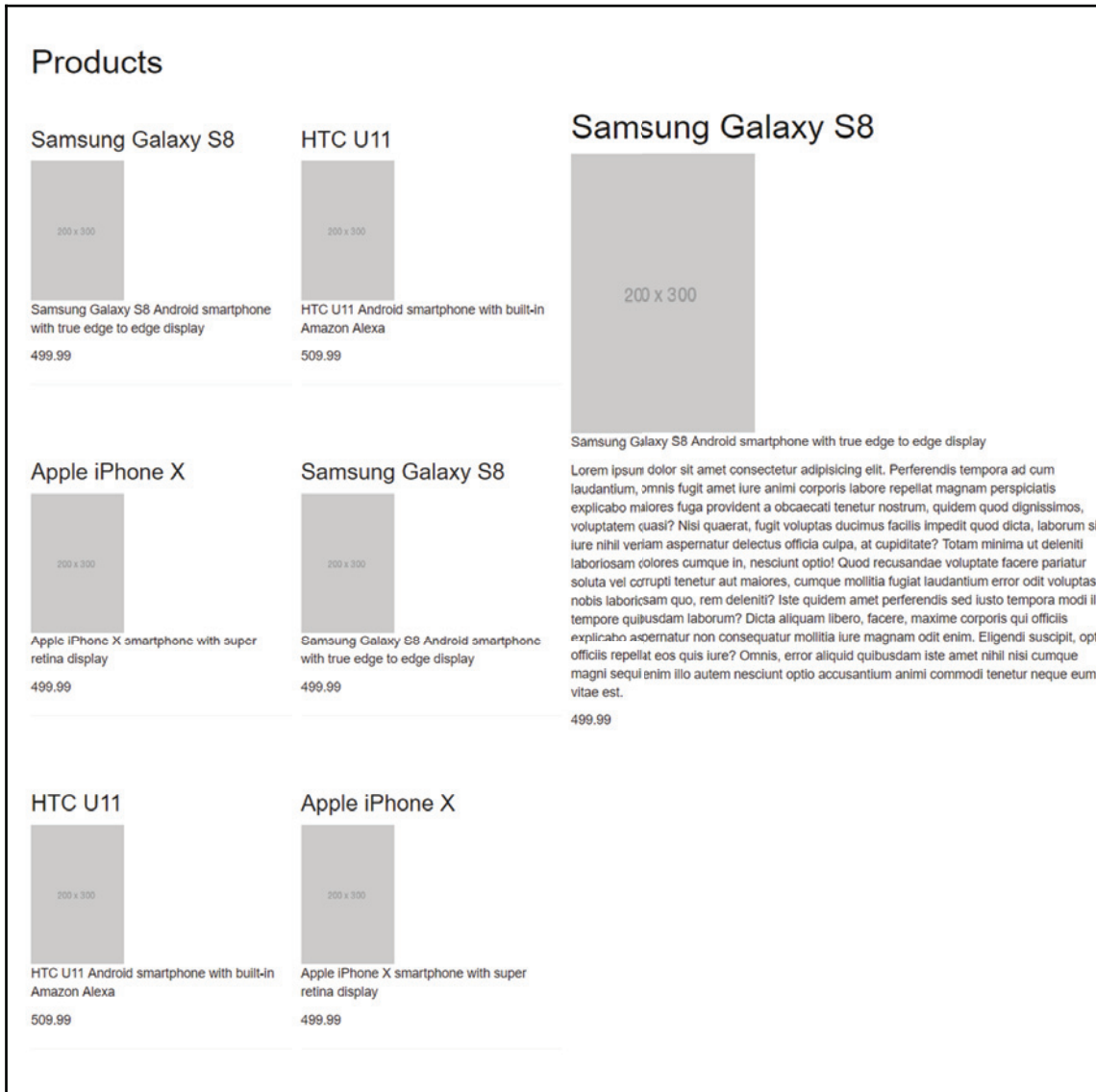
```
<template>
  <div>
    <foo-component />
  </div>
</template>
```

As you can see, the only relevance the name we use to import a component has is when rendering it in the `template` section of the parent. In this instance, `ProductList` makes more semantic sense, so that's what I've stuck with! At this point, you can run the application and should be greeted with a product listing:



Basic product list component

And when clicking any of the elements within a single product, that product should be selected and displayed on the right:



Displaying the selected product details



A really useful way of generating placeholder text is by using VS Code built-in snippets! In the `template` section of a component, type `lorem50` and press *Enter*—this will generate 50 words of lorem placeholder text, but can be adjusted to whatever length you need by changing the number 50.

As it currently stands, our `ProductList` component isn't really just a list of products, as it should be if we were sticking to SOLID principles. It is responsible for rendering the product list as well as the details of a selected product. In a very simple example like this, it isn't too much of an issue, as there is minimal HTML markup, JavaScript logic, and CSS styling involved. However, in my experience, more often than not these kinds of components will become more complicated, or certain parts of the UI will be moved or copied to another page, which is made more difficult when everything is bundled up into a single component.

Let's extract the selected product details section into its own component, and render it as a child of our existing `ProductList` component. Create a `ClientApp/components/products/Details.vue` file with a `template` section as follows:

```
<template>
  <div class="details">
    <h1>{{ product.name }}</h1>
    
    <p>{{ product.shortDescription }}</p>
    <p>{{ product.description }}</p>
    <p>{{ product.price }}</p>
  </div>
</template>
```



If you're using VS Code and followed along from earlier in this book when we installed the Vetur extension, you can quickly scaffold out an empty component definition by creating a new empty file and then typing `scaffold` at the top and pressing *Enter*.

This is a direct copy and paste from the `ClientApp/components/products/List.vue` component, but with a few tweaks to make it suitable to be its own component. Firstly, I removed the `details` class and the `v-if="selectedProduct"` directive from the enclosing `div` tag; we no longer need these as the component should not control when and if to display itself—the parent component will still control the visibility; and we have no CSS styles needing to make use of the old `details` class. I also renamed all instances of `selectedProduct` to `product`. We may use this component in other places within our UI in the future, so a more generic name for the `product` object that will be passed down via props made more sense for now.

The script section of this component looks as follows:

```
<script>
export default {
  name: "product-details",
  props: {
    product: {
      type: Object,
      required: true
    }
  }
};
</script>
```

There is nothing new here; we name the component as we usually do, then declare a single prop for the product we want to display to be passed in. The only stipulation is that this prop must be an object, and it is required—if we fail to provide one, we'll see errors in the browser console.

At this point, we are finished with the definition of this new details component. The only CSS styles specific to the display of the details section in the parent component are that it should only take up to 50% of the width of the screen. For this component to be truly reusable across multiple UI locations, we probably don't want to let it make its own decision about how much space it takes up. We'll leave that up to the parent components to decide, which means we can leave all of the current CSS styles in place in the parent list component.

In order to make use of this new component, we only have two very simple changes to make in the `ClientApp/components/products/List.vue` file. First of all, we need to import the new component and register it as a child:

```
<script>
import ProductDetails from './Details.vue'

export default {
  name: "product-list",
  components: {
    ProductDetails
  },
  // rest of script section as before...
}
</script>
```

We can then replace the following section of the component template:

```
<div v-if="selectedProduct" class="details">
  <h1>{{ selectedProduct.name }}</h1>
  
  <p>{{ selectedProduct.shortDescription }}</p>
  <p>{{ selectedProduct.description }}</p>
  <p>{{ selectedProduct.price }}</p>
</div>
```

With the following:

```
<product-details class="details" v-if="selectedProduct"
:product="selectedProduct" />
```

There's a few things going on here that may not be immediately obvious, so let's quickly go over what this line is doing.

We already know that after importing our new `ProductDetails` component, we can render it using its kebab case equivalent name, which is what we've done here. However, once Vue has finished rendering this component, what the browser receives in place of this line is the full `template` section of the product details component, in other words a standard `div` element. By adding a class to this line, Vue automatically adds that class to the root element of the nested component, which means that we can write CSS styling in a parent component that targets the root element of a child component. This is how we control the width of the product details section of the UI, and don't need to move any styles into the details component itself.

The `v-if="selectedProduct"` line has remained the same, and as before, simply controls the visibility of this component based on the user selecting a product in the list. The last part is the most interesting—we've used the shorthand `v-bind` directive syntax to bind an attribute called `product` to the `selectedProduct` property of the list component's `data` object. Remembering back to when we created the product details component, we declared a single `product` prop, with a type of object and a required validation check. This is how we pass that `product` prop into the component using the `v-bind` directive with a reference to an appropriate piece of data within the parent component.

Now, you might be wondering why this isn't throwing any errors in the browser console, seeing as the `product` prop in the details component is required, and the `selectedProduct` property that we bind onto that prop in the list component is initialized as `null` until a user clicks on a product. The simple answer is that we chose to use the `v-if` directive rather than the `v-show` directive to control the visibility of the details component. We briefly touched on the differences in these directives back in [Chapter 1, *Understanding the Fundamentals*](#), but to reiterate, using `v-if` means the section in question will not be rendered at all until the conditional statement becomes true. This means that until a user clicks on a product, and the `selectedProduct` property gets an object value assigned to it, the details component isn't even rendered in order for it to validate that its props have been passed in appropriately. This would not be the case with `v-show`, as it would be rendered from the start with its CSS visibility set to `hidden`, and prop validation would fail as the `null` value from the `selectedProduct` property would be passed in.

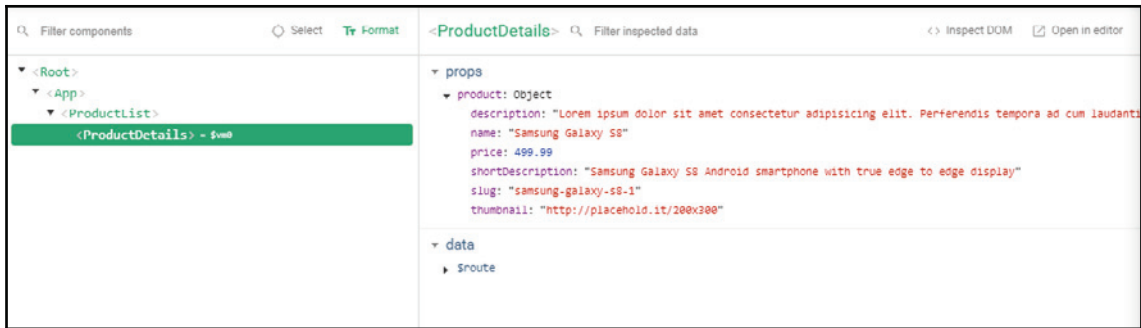
If you haven't already, check back in the browser to check everything still works as expected. There won't be any visible difference in how the app works after these changes, but we can still validate that the change has definitely taken effect. As long as you are using Chrome as your browser and have installed the Vue devtools extension as described back in [Chapter 2, Setting Up the Development Environment](#), you can press *F12* to open the Chrome developer tools window, then find the **Vue** tab at the far right of the tabs list across the top. Once inside this tab, a number of useful panes of information are displayed for us to validate what is going on under the hood of our running app. The default tab that's currently open should be the **Components** tab, and on the left-hand side of the developer tools window should be a GUI representation of our component tree, which looks something like this:



Chrome devtools extension - visual component tree

This proves that our new **<ProductDetails>** component has indeed been set up properly and is being used by the **<ProductList>** component! If you can only see the list component with no nested details component, make sure you've clicked on a product in the list and that the details component is visible.

On the right-hand side in the other half of the developer tools window should be another pane showing some detailed information about the component currently selected in the tree view that we've just seen. Make sure the **<ProductDetails>** component is highlighted in the component tree, and you should see some information as follows:



Chrome devtools extension - inspecting component data

This confirms that the product we selected in our list component is being passed down into the details component.

We've only just scratched the surface of what the Chrome dev tools extension can do to help us build and debug Vue applications, but you should already be able to see how much we can benefit from making use of it.

We now have a much better separation of concerns between the components of our UI, but it's not a very realistic example. In most online shops, there is a lot more functionality on a product details page, such as drop-down menus for customizing any potential product options, such as color or size; lists of related product suggestions based on previous user activity; comparisons with other similar products; and customer-based frequently asked question/answers and reviews. This amount of detail just wouldn't work with the master-detail pattern that we're currently using, so let's refactor again to move the details component onto a separate page, and have the user navigate to that page when clicking on one of the elements in the product list.

Client-side routing

At this point, our single-page application is quite literally that—a single page. This is where things get a little more complicated, as we can't rely on the server for handling the routing as we would with a standard ASP.NET MVC application. We need a way of routing to the *pages* of our SPA on the client. Fortunately for us, Vue has an official client-side routing library called *Vue-Router*, which is already installed and configured for us seeing as we started out by using the Microsoft application template.

So, what exactly is a page in a Vue SPA? As with most questions that I've come across while building Vue applications, there is a very simple answer; a page is nothing more than a standard component! As such, when creating a page for our application, the process is exactly the same as we've been following to create new components throughout this chapter; the only exception is that we have some additional configuration to tell *Vue-Router* how handle and navigate to it.

That being said, I do like to differentiate between page components and the UI components that build up that page by creating a `ClientApp/pages` directory. We can then have subfolders underneath the `ClientApp/components` directory that correlate to the page that will consume the files inside of it. For application-wide components that don't belong to a specific page, I'll create a `ClientApp/components/common` folder. There is absolutely no reason why you *have* to follow this convention, as it's purely my personal preference. However, I would always strongly advise against dropping all of your pages and UI components straight into the `ClientApp/components` folder as it soon gets very difficult to locate what you're looking for when coming back to make changes later!

Let's start by creating the aforementioned `ClientApp/pages` directory, with two new components inside it.

The first component is `ClientApp/pages/Catalogue.vue`:

```
<template>
<product-list />
</template>

<script>
import ProductList from "../components/catalogue/ProductList.vue";

export default {
  name: "catalogue",
  components: {
    ProductList
  }
};
```

```
</script>
```

The second component is `ClientApp/pages/Product.vue`:

```
<template>
  <product-details :product="product" />
</template>

<script>
import ProductDetails from "../components/product/Details.vue";

export default {
  name: "product",
  components: {
    ProductDetails
  },
  data() {
    return {
      product: {
        name: "Samsung Galaxy S8",
        slug: "samsung-galaxy-s8",
        thumbnail: "http://placeholder.it/200x300",
        shortDescription:
          "Samsung Galaxy S8 Android smartphone with
            true edge to edge display",
        price: 499.99,
        description:
          "Lorem ipsum dolor sit amet consectetur adipisicing elit.
            Perferendis tempora ad cum laudantium, omnis fugit amet iure animi corporis
            labore repellat magnam perspiciatis explicabo maiores fuga provident a
            obcaecati tenetur nostrum, quidem quod dignissimos, voluptatem quasi? Nisi
            quaerat, fugit voluptas ducimus facilis impedit quod dicta, laborum sint
            iure nihil veniam aspernatur delectus officia culpa, at cupiditate? Totam
            minima ut deleniti laboriosam dolores cumque in, nesciunt optio! Quod
            recusandae voluptate facere pariatur soluta vel corrupti tenetur aut
            maiores, cumque mollitia fugiat laudantium error odit voluptas nobis
            laboriosam quo, rem deleniti? Iste quidem amet perferendis sed iusto
            tempora modi illo tempore quibusdam laborum? Dicta aliquam libero, facere,
            maxime corporis qui officiis explicabo aspernatur non consequatur mollitia
            iure magnam odit enim. Eligendi suscipit, optio officiis repellat eos quis
            iure? Omnis, error aliquid quibusdam iste amet nihil nisi cumque magni
            sequi enim illo autem nesciunt optio accusantium animi commodi tenetur
            neque eum vitae est."
          }
        };
      }
    };
  }
};
</script>
```

You may have noticed that, even though we're still using the existing `ProductList` and `ProductDetails` components, the folder structure they reside in has changed slightly. As we now have our two pages, `Catalogue` and `Product`, I've added a `ClientApp/components/Catalogue` directory and moved the `ProductList` component into it, as well as renaming the `ClientApp/components/products` directory to `ClientApp/components/product`.

Now that we're no longer taking up 50% of the page with the product details component, we can make use of that space with the product list component instead. There are a few minor updates to make to the `ClientApp/components/catalogue/ProductList.vue` component, starting by removing the product details `div` element from the `template` section. After doing so, the `template` section should look as follows:

```
<template>
  <div class="products">
    <h1>Products</h1>
    <div class="list">
      <div class="item" v-for="product in products"
        :key="product.slug">
        <h3 @click="select(product)">{{ product.name }}</h3>
        
        <p @click="select(product)">{{
          product.shortDescription }}</p>
        <p @click="select(product)">{{ product.price }}</p>
        <hr />
      </div>
    </div>
  </div>
</template>
```

Next, we can update the `script` section to remove the child component import and declaration, as well as modify the `select(product)` method to navigate to our newly created product page:

```
<script>
export default {
  name: "product-list",
  data() {
    return {
      products: [
        // products array as before...
      ]
    };
  },
};
```

```
    methods: {
      select(product) {
        this.$router.push(`/products/${product.slug}`);
      }
    }
  };
</script>
```

This is the first time we've seen any code specific to client-side routing with Vue-Router, so this will probably look a little alien to you. As mentioned previously, Vue-Router is already configured for us, which gives us access to the `$router` property in every component. One of the methods on this property is the `push` method, which takes the URL path to navigate to as a parameter. We're yet to set up this route to actually go anywhere, but the general idea is that we want nice, clean, "friendly" URLs based on the product name for SEO purposes. As an example, the URL for the first product in the list would look like this: <http://localhost:5000/products/samsung-galaxy-s8>.

The only other change in this component is to tweak the styling. As we no longer have the details component on this page, we can remove the 50% width limitation on the product list, and also display four products side by side rather than two. The updated style block looks like this:

```
<style>
* {
  box-sizing: border-box !important;
}

.products {
  padding: 20px;
  max-width: 1200px;
  margin: 0 auto;
}

.list .item {
  width: 25%;
  float: left;
  padding: 20px 10px 20px 0;
}

.list img,
.list h3,
.list p {
  cursor: pointer;
}

.list img {
```



```
    width: 100px;
  }
</style>
```

The changes we need to make to the

`ClientApp/components/product/Details.vue` component are even simpler; the only thing we need to do is add a `style` block to the bottom of the component with a little padding to bring the content off the sides of the screen now that it's being displayed on its own page:

```
<style>
.details {
  padding: 20px;
}
</style>
```

To make these style changes take effect, I've added the `details` class back on to the root `div` element in the `template` section as well.

As we now have `Catalogue` and `Product` page components dedicated to displaying our list of products and product details, respectively, we can no longer rely on the `ClientApp/components/App.vue` root component to be directly tied to the `ProductList` and `ProductDetails` components. We need a way of deciding which page to display based on the route that has been matched by the router. `Vue-Router` contains a custom component called `<router-view />` that does just that. All we need to do is render it in the `template` block of our `ClientApp/components/App.vue` component:

```
<template>
  <router-view />
</template>

<script>
export default {
  name: "app"
};
</script>
```

Notice how we've also removed the `import` lines for the two previously rendered components, as these are now imported and rendered within the page components we created earlier. The `<router-view />` component renders a matched component from the route definitions we're about to define.

The final change we need to make is in the `ClientApp/boot.js` file. After the `Vue.use(VueRouter);` line, we need to add the following:

```
//import page components
import Catalogue from "../pages/Catalogue.vue";
import Product from "../pages/Product.vue";

const routes = [
  { path: "/products", component: Catalogue },
  { path: "/products/:slug", component: Product },
  { path: "*", redirect: "/products" }
];
```

The most important part of this block is the `const routes = [...]` array definition. This array will eventually contain all of the route definitions for our application, much the same way that we can configure server-side routes in the `Startup.cs` class of an ASP.NET Core application. The array is a collection of objects, each having a minimum of a string `path` that defines the browser URL to look for, and a `component` object reference to one of our app components. As with nested parent-child relationships, in order to reference the page-level components of our app, we need to import them first before they can be used in the route definitions.

The second entry in the `routes` array is for our product details page. As this page needs to be able to individually display the details for every product in our system, it needs a way of dynamically matching a unique product. As with ASP.NET MVC routes, we can define dynamic route parameters. In Vue-Router, these are prefixed with a colon, for example, `/products/:slug`. A **slug** is a URL segment that is unique to a specific product, usually generated from the product name by replacing spaces and other special characters with dashes instead.

This convention allows us to navigate to products with a friendly URL, such as: `http://localhost:5000/products/samsung-galaxy-s8`, rather than using ID parameters such as: `http://localhost:5000/products/1`. We'll use this parameter later in this chapter to fetch a single product from our API.

The last entry in the `routes` array is a catch-all, or wildcard entry, that matches any URL that's not already been matched by the previous definitions. Again, like with ASP.NET MVC route definitions, the order that we define our Vue-Router routes in matters. If a URL matches multiple routes, the first one found within the array will take precedence over any others. By including a wildcard definition at the very end of the array, we ensure that any request to a path on the domain of our application will at the very least end up being redirected to our catalog page.

The complete `ClientApp/boot.js` should look like this:

```
import Vue from "vue";
import VueRouter from "vue-router";

Vue.use(VueRouter);

//import page components
import Catalogue from "./pages/Catalogue.vue";
import Product from "./pages/Product.vue";

const routes = [
  { path: "/products", component: Catalogue },
  { path: "/products/:slug", component: Product },
  { path: "*", redirect: "/products" }
];

new Vue({
  el: "#app-root",
  router: new VueRouter({ mode: "history", routes: routes }),
  render: h => h(require("./components/App.vue"))
});
```

At this point, we have finished refactoring, and the app should be working. If you left it running while making these changes, you may need to force the browser to refresh as the Hot Module Replacement plugin may not pick up on all the changes we've made. Notice how the app will now redirect to the `/products` route by default, and clicking on an element in the product list does a client-side redirect to a product listing URL based on the name of the product you clicked on.

If you didn't notice that we were hardcoding the product being displayed on the product page earlier, you'll have undoubtedly noticed now that it doesn't matter which product you click on in the list; we always receive the same details when taken to the product page. This is because we now have multiple pages to our application, but no way of sharing data between them as we did when they were both rendered by the root app component. There are ways that we could achieve this sharing of data if we wanted, but there's really no need yet as, sooner or later, we need to hook the frontend components up to our API anyway. This will solve our issue as each page can be in control of requesting only the data it needs as we navigate between pages.

Fetching data from an API

We currently have no concept of a "product" on the server side of our application, so let's remedy that by creating a product entity and migration to start off with. Create a `Data/Entities/Product.cs` class with the following contents:

```
namespace ECommerce.Data.Entities
{
    public class Product
    {
        public int Id { get; set; }
        [Required]
        public string Name { get; set; }
        [Required]
        public string Slug { get; set; }
        [Required]
        public string Thumbnail { get; set; }
        [Required]
        public string ShortDescription { get; set; }
        [Required]
        public string Description { get; set; }
        [Required]
        public decimal Price { get; set; }
    }
}
```

The properties of this class match the properties that we've been rendering in our UI so far. We've also added `Required` attributes on all but the `Id` property to ensure that the database creates these columns with `not null` constraints. The `Id` property is required by default as it is used as the primary key for the table.

Next, we need to make EF aware of the entity by updating our `Data/EcommerceContext.cs` class:

```
namespace ECommerce.Data
{
    public class EcommerceContext : IdentityDbContext<AppUser,
        AppRole, int>
    {
        public EcommerceContext(DbContextOptions<EcommerceContext>
            options) : base(options)
        { }

        public DbSet<Product> Products { get; set; }

        protected override void OnModelCreating(ModelBuilder
```

```
        modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<Product>()
            .HasIndex(b => b.Slug)
            .IsUnique();
    }
}
}
```

We've added the `public DbSet<Product> Products { get; set; }` line as well as the `OnModelCreating(...)` method override at the bottom of the class. The `DbSet` declaration is what triggers EF to actually decide that we need a table to be created, as well as giving us a property on the context to access our product-related database queries. The overridden `OnModelCreating` method is used to add a unique index on the `Slug` property of our `Products` table, which can't be done using data annotations on the class itself.

We don't yet have a UI for saving new products, so we need to seed the database with some sample data to keep us going until that UI exists. Update the `Data/DbContextExtensions.cs` class to include the following static method:

```
private static void AddProducts(EcommerceContext context)
{
    if (context.Products.Any() == false)
    {
        var products = new List<Product>()
        {
            new Product
            {
                Name = "Samsung Galaxy S8",
                Slug = "samsung-galaxy-s8",
                Thumbnail = "http://placeholder.it/200x300",
                ShortDescription = "Samsung Galaxy S8 Android
                smartphone with true edge to edge display",
                Description = "Lorem ipsum dolor sit amet consectetur adipisicing
                elit. Perferendis tempora ad cum laudantium, omnis fugit amet iure animi
                corporis labore repellat magnam perspiciatis explicabo maiores fuga
                provident a obcaecati tenetur nostrum, quidem quod dignissimos, voluptatem
                quasi? Nisi quaerat, fugit voluptas ducimus facilis impedit quod dicta,
                laborum sint iure nihil veniam aspernatur delectus officia culpa, at
                cupiditate? Totam minima ut deleniti laboriosam dolores cumque in, nesciunt
                optio! Quod recusandae voluptate facere pariatur soluta vel corrupti
                tenetur aut maiores, cumque mollitia fugiat laudantium error odit voluptas
                nobis laboriosam quo, rem deleniti? Iste quidem amet perferendis sed iusto
```

```
tempora modi illo tempore quibusdam laborum? Dicta aliquam libero, facere,
maxime corporis qui officiis explicabo aspernatur non consequatur mollitia
iure magnam odit enim. Eligendi suscipit, optio officiis repellat eos quis
iure? Omnis, error aliquid quibusdam iste amet nihil nisi cumque magni
sequi enim illo autem nesciunt optio accusantium animi commodi tenetur
neque eum vitae est.",
    Price = 499.99M
  },
  // rest of product list omitted for brevity...
};

context.Products.AddRange(products);
context.SaveChanges();
}
}
```

Then, in the same class, update the `EnsureSeeded` method to call this newly created method, as follows:

```
public static void EnsureSeeded(this EcommerceContext context)
{
    // ...rest of file

    AddProducts(context);
}
```

With these changes in place, we can now add a migration and run the application to apply it to the database. Open your Terminal and run the following command:

```
dotnet ef migrations add ProductEntity
```

If you've followed along accurately, you shouldn't see any errors in the Terminal. However, if you receive errors stating that the application build failed, make sure you've stopped the app from running before trying to add the migration!

Running the application now should yield no visible changes to the application, which should work exactly the same as it did before. However, if you check the database in `pgAdmin 4`, there should be a newly created `Products` table containing our seed data. Inspecting further should also show a unique index on the `Slug` field.

Now that we have some data to query, we can create a controller to query and return it to our client app. Create a `Features/Products/Controller.cs` class with the following contents:

```
namespace ECommerce.Features.Products
{
    [Route("api/[controller]")]
    public class ProductsController : Controller
    {
        private readonly EcommerceContext _db;

        public ProductsController(EcommerceContext db)
        {
            _db = db;
        }
    }
}
```

There is nothing new going on here; it is very similar to the `Features/Users/Controller.cs` class that we created in the previous chapter. In terms of our current UI data needs, we will need to add two endpoints to this controller: a first HTTP get action to return a list of all of our products, and a second HTTP get action to request a single product based on its `Slug` property.

These two endpoints are defined as follows:

```
[HttpGet]
public async Task<IActionResult> Find()
{
    var products = await _db.Products.ToListAsync();
    return Ok(products);
}

[HttpGet("{slug}")]
public async Task<IActionResult> Get(string slug)
{
    var product = await _db.Products.SingleOrDefault(x =>
        x.Slug == slug);

    if (product == null)
        return NotFound();

    return Ok(product);
}
```

Rebuild and run the application again now and we should have two functioning API endpoints to fetch our data from. You can test these by testing the following URLs in your browser:

- `http://localhost:5000/api/products`
- `http://localhost:5000/api/products/apple-iphone-x`

With these in place, we can update our page components to fetch their data from the server, rather than hard coding it. The first thing we'll do is refactor our catalog page and product list component to follow the same pattern we're using on the product page and details component. The page component will be responsible for fetching its data on page load, before passing it down into the product list component as props.

In the `ClientApp/components/catalogue/ProductList.vue` file, remove the `data()` `{...}` function entirely, and replace it with the following props declaration:

```
props: {
  products: {
    type: Array,
    required: true
  }
}
```

Next, we need to actually provide this product array from the `ClientApp/pages/Catalogue.vue` component. First of all, we need to update the `<product-list />` HTML element in the template section to bind a `products` prop:

```
<template>
  <product-list :products="products" />
</template>
```

Next, in the script section, we need to add a data object that contains a `products` array matching this prop reference:

```
data() {
  return {
    products: []
  };
}
```


And finally, we need to add a mounted life cycle hook to query our API and populate the `products` array:

```
mounted() {
  fetch("/api/products")
    .then(response => {
      return response.json();
    })
    .then(products => {
      this.products = products;
    });
}
```

Here, we use the `fetch` API to perform a HTTP GET request to the endpoint we just created, parse the response as JSON, then bind our `products` array to the response JSON array.

Finally, we need to make a similar change in the `ClientApp/pages/Product.vue` component. In the `data` object, clear out the hardcoded product object and replace it with `null`:

```
data() {
  return {
    product: null
  };
}
```

Then, add a mounted life cycle hook in this component as well, performing a similar HTTP GET request, albeit to fetch a single product rather than a list:

```
mounted() {
  const slug = this.$route.params.slug;

  fetch(`/api/products/${slug}`)
    .then(response => {
      return response.json();
    })
    .then(product => {
      this.product = product;
    });
}
```

We've already seen how the route definition that renders this page has a dynamic route parameter called `slug`. In this hook, we access this parameter from the built-in `$route` object, then use it to form the URL that we query for the product we're trying to view. When using Vue-Router, all components have access to the `$route` object by default.

With these changes in place, our application should be fully functioning. Clicking on a product now not only takes us to a separate product details page, but also loads the appropriate product details from the API and renders it dynamically based on the `slug` parameter in the URL.

Summary

We've covered a lot of ground in this chapter, so let's quickly recap what we've achieved. We started out with a very simple component for displaying a static list of hard coded products. We then added an implementation of the master-detail pattern to allow the user to select a product to view additional details.

We looked at when and why to refactor our component structure, and how we can compose a UI of parent-child component relationships by breaking out a product details component. We then took this a step further by moving the product details component into its own separate page, and introducing a multipage setup using client-side routing with Vue-Router.

Finally, we created a `Products` table in our database, seeded it with sample data, and built two API endpoints to serve the data required by our UI to replace the static hard coded data we had been using. Thinking back to the earlier chapter on fundamentals of Vue.js, we have already covered a lot of topics outlined in that chapter. We've covered component composition, parent-child relationships, the `data` object/function, props, prop validation, life cycle hooks, the `v-for/v-bind/v-on` directives, and fetching data from an API.

There's still a long way to go, and our app is anything but "pretty," so in the next chapter, we'll carry on building out a functioning catalog and make things look a little nicer at the same time!

5

Building a Product Catalog

In the last chapter, we got off to a good start with building our very first custom Vue.js components. We covered a lot of the fundamentals that we'll be using throughout the rest of this book, such as data-binding, looping over, and rendering lists of data; conditional rendering, component composition with props and event handlers; client-side routing between multiple pages, and fetching data from our server-side API. These are the kinds of things that will become the bread and butter of building applications with Vue.js, so it was important to make sure we understand them before moving forward.

In this chapter, we are going to build on what we already have and start to transform it into a proper e-commerce application. We'll begin by improving the overall look and feel using **Bootstrap**, as well as adding transitions and loading indicators on page changes to improve the **User Experience (UX)**. We'll then move on to flesh out the existing product list to become a fully featured catalog with filtering and sorting. We're going to cover a lot of ground, including topics such as the following:

- Styling with Bootstrap and SASS
- Pre-fetching data with Vue-Router
- Transitions, animations, and loading indicators
- Manipulating the URL query object to store user selections
- Triggering API calls when the URL query object changes
- Server-side filtering
- Client-side sorting
- Searching for text using the `EF.Functions.Like` operator
- Triggering search API requests in real time using watchers
- Limiting expensive operations with debounce functions

Improving the existing UX

As we move into building a real-world application, it's time to start making things look a little nicer by introducing some CSS styling. However, this isn't a book about CSS or web design, so we'll rely on the very popular Bootstrap CSS framework to handle 99% of our styling needs.

Choosing a UX framework

Back in [Chapter 2, *Setting Up the Development Environment*](#), we removed the default Bootstrap installation included with the project template, and you may now be wondering why. We removed the default installation for three reasons:

- It had a dependency on jQuery, which in my opinion shouldn't be necessary when creating SPAs with a modern frontend framework such as Vue
- It was referencing Bootstrap 3, which at the time of writing has just been replaced by Bootstrap 4, which we'll be using instead
- Rather than reference Bootstrap directly, we're going to be using a Bootstrap-based Vue component library called Bootstrap-Vue

There is nothing wrong with jQuery, and I use it a lot in projects that don't utilize a frontend SPA framework, but it simply isn't necessary when we are using one. For SPA projects, my personal preference is to make use of a CSS-only framework such as **Bulma** (<http://bulma.io>), or to remove the jQuery dependency from Bootstrap and add the interactivity using Vue components. This can be pretty time-consuming, but luckily for us the creators of **Bootstrap-Vue** have already done it for us, which is why we'll use it for this project.

What is Bootstrap-Vue?

Bootstrap-Vue is a component library based entirely on the hugely popular **Bootstrap** frontend CSS framework. It provides a number of Vue.js components that ultimately render Bootstrap HTML markup decorated with the correct classes to be styled by the Bootstrap CSS files. However, they also add the necessary JavaScript behavior without taking a dependency on jQuery.

Rendering Bootstrap-Vue components is often far more simple and concise than rendering the full Bootstrap syntax for a given component. It is easy to determine the components we use from the Bootstrap-Vue library, because they all come prefixed with a `b-`. For example, to render a modal component, we'd use the following syntax:

```
<b-modal title="Bootstrap-Vue">
  <p>Modal content</p>
</b-modal>
```

Bootstrap-Vue includes the unmodified original CSS files from Bootstrap itself, so if needs be we can even fall back to using standard Bootstrap HTML markup instead, which will still be styled appropriately as long as we include the correct classes:

```
<div class="modal" role="dialog">
  <div class="modal-dialog" role="document">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title">Bootstrap-Vue</h5>
        <button type="button" class="close" data-dismiss="modal"
          aria-label="Close">
          <span aria-hidden="true">&times;</span>
        </button>
      </div>
      <div class="modal-body">
        <p>Modal content</p>
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-primary">Save
          changes</button>
        <button type="button" class="btn btn-secondary" data-
          dismiss="modal">Close</button>
      </div>
    </div>
  </div>
</div>
```

We can already see how much easier it is to render a Bootstrap-Vue component than it is to render all the HTML markup required by Bootstrap itself. However, we still have the flexibility of falling back to this markup if we need it, which we'll demonstrate in a later chapter.

Installing additional required dependencies

First things first, we need to install a handful of npm modules that we'll be making use of to improve the UX of our app. Open a Terminal and run the following two commands:

```
yarn add node-sass sass-loader -dev
```

```
yarn add bootstrap-vue nprogress
```

The first line is necessary for adding the `node-sass` and `sass-loader` modules to the `devDependencies` section of the `package.json` file. Out of the box, the project template we are using does not support SASS for our styles. We won't be writing all that much custom styling, but what we do write will use SASS due to the added benefits it provides over plain old CSS. These two modules are required to instruct webpack how to handle the style blocks of our components when we start making use of SASS.

The second line is adding the `Bootstrap-Vue` and `NProgress` libraries to our project. We've already covered what `Bootstrap-Vue` is and why we're using it, but `NProgress` may not be quite so obvious. It's a fairly simple library for adding progress indicators to our app, which we'll use for adding a loading animation on page changes while data fetching is in progress.

Modifying the webpack configuration to support SASS

With these dependencies installed, we need to update our webpack configuration to make use of them. Open up the `webpack.config.js` file and find the `module` section, which currently looks like this:

```
module: {
  rules: [
    { test: /\.vue$/, include: /ClientApp/, loader: 'vue-loader' },
    { test: /\.css$/, use: isDevBuild ? [ 'style-loader', 'css-loader' ] : ExtractTextPlugin.extract(
      { use: 'css-loader?minimize' } ) },
    { test: /\.(png|jpg|jpeg|gif|svg)$/, use: 'url-loader?
      limit=25000' }
  ]
}
```

This section needs to be updated to look like this:

```
module: {
  rules: [
    {
      test: /\.vue$/,
      include: /ClientApp/,
      loader: "vue-loader",
      options: {
        loaders: {
          scss: "vue-style-loader!css-loader!sass-loader",
          sass: "vue-style-loader!css-loader!sass-loader?
            indentedSyntax"
        }
      }
    },
    {
      test: /\.css$/,
      use: isDevBuild
        ? ["style-loader", "css-loader"]
        : ExtractTextPlugin.extract({ use: "css-loader?minimize" })
    },
    { test: /\.(png|jpg|jpeg|gif|svg)$/, use: "url-loader?limit=25000" }
  ]
}
```

Apart from expanding these lines to make them more readable, the only changes are to the first object in the `rules` array section. Specifically, we've added the `options` object, which utilizes the `sass-loader` that we installed earlier. We can now specify a `lang` attribute on the style blocks of our components in order to write our styles in SASS instead of CSS; we'll see how to do that shortly.

Updating the webpack vendor configuration

One of the perks of using the Microsoft project template is that it splits the webpack configuration, and therefore our JavaScript bundle, into two parts. One of these parts is for third-party code and one is for our own custom code. This provides a huge benefit to the speed at which our custom code is processed and bundled by webpack, because it doesn't need to process 3rd party code such as the Vue and VueRouter libraries, or any other 3rd party libraries such as those we've just installed. If it did, our on-the-fly code changes would take a lot longer to be pushed to the browser by the HMR plugin.

Open up the `webpack.config.vendor.js` file, find the `vendor` array, and update it as follows:

```
vendor: [
  "event-source-polyfill",
  "isomorphic-fetch",
  "vue",
  "vue-router",
  "bootstrap/dist/css/bootstrap.min.css",
  "bootstrap-vue",
  "nprogress/nprogress.css"
]
```

We're simply adding the **Bootstrap**, **Bootstrap-Vue**, and **NProgress** libraries to our vendor bundle.

The next thing we need to do is to register the **Bootstrap-Vue** plugin in our app entry point, the `ClientApp/boot.js` file, like so:

```
import Vue from "vue";
import VueRouter from "vue-router";
import BootstrapVue from "bootstrap-vue";

Vue.use(VueRouter);
Vue.use(BootstrapVue);

//rest of file unchanged...
```

With this done, all of the components included in the **Bootstrap-Vue** library are now globally accessible to any of our custom components, without the need to import them specifically as we've seen with our parent-child relationships so far.

Rebuilding the vendor bundle

We're now finished with the changes to our app's overall configuration, and usually we would simply need to restart the application for those changes to take effect. However, due to an issue with the project template we're using, the vendor bundle does not automatically rebuild itself when we restart the application. We have to explicitly instruct it to do so, by running the following command in a Terminal window:

```
webpack --config webpack.config.vendor.js
```


In order for this command to work, you need webpack installed globally on your machine. You can install it by running the following command:

```
npm install -g webpack
```

However, rather than relying on all of your developers having webpack installed globally, we can add an npm script that makes use of the local copy of webpack installed along with the other project dependencies, which is stored inside the `node_modules` directory. Open the `package.json` file, and add a `scripts` block like so:

```
"scripts": {
  "webpack": "webpack --config webpack.config.vendor.js && webpack"
}
```

We can now simply run `yarn webpack` from a Terminal, and both of our webpack bundles will be processed and rebuilt; better still, we don't need to remember to install webpack globally before doing so.

If you haven't already, run the `yarn webpack` command, and your vendor bundle will now contain all of the scripts and style sheets needed for the new libraries we've just added to our project.

Adding application-wide layout elements

We're now ready to start adding some global components and styles to our application. We'll start by adding a Bootstrap navbar to all of our pages, which we can easily do by adding to the `ClientApp/components/App.vue` file. You can think of this file like the `_Layout.cshtml` file from the server side of our application, which contains all of the global HTML markup included on every page of an MVC application.

Open the `ClientApp/components/App.vue` file and update the template section to look like this:

```
<template>
  <div class="app">
    <b-navbar toggleable="md" type="dark" variant="dark">
      <b-container>
        <b-navbar-toggle target="nav_collapse"></b-navbar-toggle>
        <b-navbar-brand to="/">PhoneShop</b-navbar-brand>
        <b-collapse is-nav id="nav_collapse">
          <b-navbar-nav>
            <b-nav-item to="/products">Products</b-nav-item>
          </b-navbar-nav>
        </b-collapse>
      </b-container>
    </b-navbar>
  </div>
</template>
```

```
    </b-container>
  </b-navbar>
  <router-view />
</div>
</template>
```

As we now have more than one element being rendered in this template, we've had to add a `<div class="app"></div>` wrapper element. We've then made use of the Bootstrap-Vue library by using a number of its components, namely the `b-navbar` component and its related sub-components. If you're at all familiar with Bootstrap, it should be fairly easy to see what each of these components is doing. Essentially, each one is like rendering a `div` element with appropriate class attributes to apply Bootstrap styling.

All we really need to know about these components is that they're used to display a list of menu items in a responsive way. We constrain the navbar contents to a fixed width with the `b-container` component, and the menu items are hidden on small screen sizes until the `b-navbar-toggle` component is clicked to have them animate into view. You can try this now by running the application, then reducing the width of your browser and seeing how the navbar responds to the reduced width.



If this component was likely to get much bigger, it would be a good idea to extract the navigation menu part of this template into a separate **Navbar** component. You can try this for yourself now as a learning exercise!

One of the most important lines is the `b-nav-item` component line. Under the hood, this component renders a standard HTML `a` tag, so it would be easy to use a `href` attribute to instruct the browser where to navigate to when the link is clicked. However, although it is perfectly OK to use a `href` attribute instead of the `to` attribute that we've used here, doing so would cause a full page load when loading the next page. Instead, if we're navigating to an internal page, we can use the `to` attribute, which will utilize Vue-Router and client-side navigation. The next page will then be rendered by the client, which will be much faster than a full round trip to the server.

Finally, the existing `router-view` component that was previously the only element in this template is now rendered beneath our navbar. As such, the navbar will be displayed on every page of our application, without needing to explicitly include it in each page's template.

Adding application-wide styles

The `ClientApp/components/App.vue` file is also the recommended place to put our global styles, either directly in the `style` section, or imported from an external style sheet. Styles that are only applicable in a single component should be included in said component's `style` section, and set as `scoped` to prevent them affecting any other components—we'll see how to do that shortly.

Open up the `ClientApp/components/App.vue` file and add a style section at the bottom like so:

```
<style lang="scss">
html,
body {
  height: 100vh;
}
div.app,
div.page {
  height: 100% !important;
}
</style>
```

As we're using Bootstrap, there are already a load of base styles applied by default, so for now all we need to do is force the `html` and `body` elements to stretch to the full height of the screen. We can then force our root `<div class="app"></div>` element, and each `<div class="page"></div>` element, to also stretch to 100% height. Normally this wouldn't be necessary, but we'll see why we need these styles shortly.

Styling the product list and product details components

As we did with the navbar, we're going to make use of some of the components from the Bootstrap-Vue library. Open up the `ClientApp/components/catalogue/ProductList.vue` file and modify the template section:

```
<template>
  <div class="products">
    <b-container>
      <h1 class="mt-4 mb-4">Products</h1>
      <b-row>
        <b-col class="mb-4" sm="6" v-for="product in products"
          :key="product.id">
```

```
<b-media class="product">
  
  <h2 class="mt-2" @click="view(product)">{{ product.name }}
</h2>
  <p class="mt-4 mb-4">
    {{ product.shortDescription }}
  </p>
  <p class="mt-4 mb-4">{{ product.price }}</p>
  <b-button variant="primary">Add to cart</b-button>
</b-media>
</b-col>
</b-row>
</b-container>
</div>
</template>
```

Again, the names of these components should be fairly familiar if you've used Bootstrap before, and as such it will be fairly easy to work out what's going on. However, if you aren't familiar with Bootstrap, please do check the documentation for the Bootstrap-Vue library which provides a full description of each component: <https://bootstrap-vue.js.org/docs>.

We're also making heavy use of Bootstrap utility classes to add spacing between elements. Specifically the `mt` and `mb` classes, which add margin to the top and bottom of the associated element respectively. These classes can take a numeric modifier between 0 and 5 to control how much margin is applied, for example, `mt-1` and `mt-5`.

The only functional change that's been made to this component is the name of the function called when the user clicks the product name or image. As we are no longer *selecting* a product, I renamed the function `view` instead:

```
methods: {
  view(product) {
    this.$router.push(`/products/${product.slug}`);
  }
}
```

Finally, replace the entire `style` section with the following:

```
<style lang="scss" scoped>
.product {
  border: 3px solid #eee;
  img,
  h2 {
    cursor: pointer;
  }
}
```

```
    }  
  }  
</style>
```

As previously mentioned, we will be writing as little custom CSS as possible, and relying heavily on Bootstrap to make things look nice. That being said, it is important to understand how *scoped* component styles work, and how you can utilize SASS in your own projects.

We already did all of the setup required to make use of SASS, which means the only thing we need to do at the component level is add the `lang="scss"` attribute. We can then make full use of SASS, including nested selectors as we have here, as well as variables, mixins, and other SASS goodies. Note how we also added the `scoped` attribute to this style block. By scoping the styles in this way, they will only apply to the HTML markup in the `template` section of this component. You can try this out now by hovering your mouse over a product image in the catalog page, then clicking on a product and hovering over the image on the product details page. The cursor should not indicate that the image is clickable on the product details page, but it should on the catalog page.



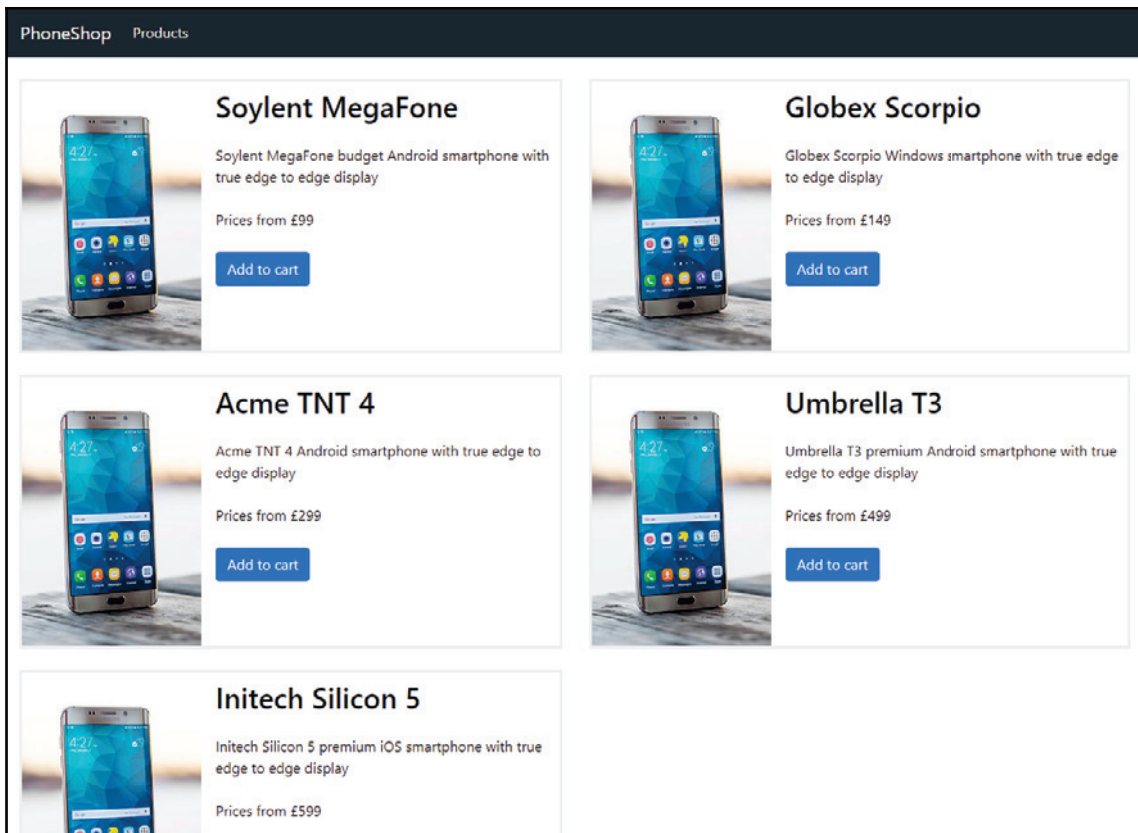
Adding the `lang="scss"` attribute may require a full page refresh before the styles take effect properly!

Next, open the `ClientApp/components/product/Details.vue` file and update the `template` section like this:

```
<template>  
  <b-container class="pt-4">  
    <b-media>  
        
      <h2>{{ product.name }}</h2>  
      <p class="mt-4 mb-4">  
        {{ product.shortDescription }}  
      </p>  
      <p class="mt-4 mb-4">{{ product.price }}</p>  
      <b-button variant="primary">Add to cart</b-button>  
    </b-media>  
    <h3 class="mt-4">Product details</h3>  
    <p class="mt-4 mb-4">  
      {{ product.description }}  
    </p>  
  </b-container>  
</template>
```

Nothing new here except the `pt-4` class, which, as you might expect, works very similarly to the `mt-4` class, except it applies padding instead of margin. There are no component-specific styles needed for this component, so you can delete the `style` section entirely.

This completes the layout and style updates, so feel free to run the application now and see how it looks, which should be something like this:



Styled product list component

Fetching data before navigation

As it stands, both of our pages are configured to fetch the data they require *after* navigating to the page, by triggering the API calls in the `mounted` lifecycle hook of the page-level component. While working locally, it is very hard to notice the side effects to this setup, because the API calls complete so quickly that we can barely tell that the page wasn't rendered instantly. However, when accessing the application over the internet, the network latency will be far more apparent, and you will notice a brief time when the page is first displayed where there is a blank screen while the API call is still in progress.

We can combat this and improve the UX of our application in one of two ways: we can continue to fetch page data after navigation, but display some kind of loading indicator while the API call is in progress; or we can configure the component to fetch its data *before* navigation, so that the page will not change until the API call completes. Either way, there is still a need for displaying progress indicators, as even if we fetch data before navigation there will be a brief period where the API call is in progress and nothing appears to be happening.

We've already seen how to fetch data after navigation, so let's now update both of our page components to fetch their data before navigation so you have a comparison. Open up `ClientApp/pages/Catalogue.vue` and remove the `mounted()` life cycle hook function entirely. In its place, we need to add the following:

```
beforeRouteEnter(to, from, next) {
  fetch("/api/products")
    .then(response => {
      return response.json();
    })
    .then(products => {
      next(vm => vm.setData(products));
    });
}
```

This is the first time we've seen this method, as it isn't a standard life cycle hook. We gain access to this hook by using `VueRouter`, and, as the name of the function suggests, it will run before we try to navigate to this component. We also have access to three arguments: a `to` object that models the route that we are navigating *to*; a `from` object that models the route we are navigating *from*; and a `next` callback function that we can invoke to allow the navigation to complete.

In this instance, we perform an API call to fetch our list of products from the server like we did in the `mounted()` hook before, but this time we call the `next` function after the API call is complete. We also pass it a lambda function to invoke once the page has been loaded. At this point, navigation hasn't happened yet, so we can't simply assign the products we receive from the server straight onto the component's products array. Instead, we instruct `VueRouter`, in this case, to call a `setData()` function on the component, passing it the list of products we just received from the API. This function does not exist yet, so let's add it by creating a `methods` object like so:

```
methods: {
  setData(products) {
    this.products = products;
  }
}
```

The last change we need to make in this page component is to update the `template` section to look like the following:

```
<template>
  <div class="page">
    <product-list :products="products" />
  </div>
</template>
```

The only thing we've done is wrap the `product-list` component with a `div` element that has a class of `page`. This class picks up the global styles we applied earlier to force the page to fill 100% of the browser's height.

This is everything we need to change on this page, but we need to make similar changes to the `ClientApp/pages/Product.vue` file. Update its `template` section like so:

```
<template>
  <div class="page">
    <product-details :product="product" />
  </div>
</template>
```


Then, add a `methods` object to the `script` section like so:

```
methods: {
  setData(product) {
    this.product = product;
  }
}
```

Finally, replace the `mounted()` life cycle hook with the following `beforeRouteEnter` hook:

```
beforeRouteEnter(to, from, next) {
  fetch(`/api/products/${to.params.slug}`)
    .then(response => {
      return response.json();
    })
    .then(product => {
      next(vm => vm.setData(product));
    });
}
```

The only difference here is that we make use of the `to` object argument, extracting the `slug` parameter from its `params` object. This is similar to how we extracted the `slug` parameter from the `$route.params` object before, as both objects represent the query string parameters on the current route.

Start the application up again now and everything should still be working as before, except that the very slight API call delay occurs before the page changes rather than after.

Adding a page loading indicator

As we already discussed, the delay while the API calls are in progress will be far more noticeable when our app is live on the internet. We need to provide some feedback to assure our users that something is happening, and the page they are expecting to view is loading—this is where the `NProgress` library we installed earlier comes into play.

There are two ways that we could achieve the desired result. Firstly, on each page on which we wish to show a loading indicator, we could utilize the same `beforeRouteEnter` hook that we've just been using to start the loading animation, then utilize the `next` function call to stop the animation after navigation completes. This would look something like this:

```
beforeRouteEnter(to, from, next) {
  NProgress.start();
  fetch(`/api/products/${to.params.slug}`)
    .then(response => {
      return response.json();
    })
    .then(product => {
      next(vm => {
        vm.setData(product)
        NProgress.done();
      });
    });
}
```

This would work perfectly fine, and achieve the desired result in a fairly clean way. However, we would potentially need to repeat this code in quite a few places, depending on how many pages we have in our application. It would be much better if we could achieve the desired result without so much repetition by adhering to DRY principles.

Fortunately, we can intercept *all* route changes and display the loading animation very easily using global route hooks. To do so, we need to open up `ClientApp/boot.js` and make a few modifications. Firstly, we need to import the `NProgress` library like so:

```
import Vue from "vue";
import VueRouter from "vue-router";
import BootstrapVue from "bootstrap-vue";
import NProgress from "nprogress";

//rest of file unchanged
```

Next, instead of initializing the `VueRouter` object inline like we do here:

```
new Vue({
  el: "#app-root",
  router: new VueRouter({ mode: "history", routes: routes } ),
  render: h => h(require("./components/App.vue"))
});
```

We need to initialize it separately so that we can add our navigation hooks to it before we attach it to the root component Vue instance:

```
const router = new VueRouter({ mode: "history", routes: routes });

router.beforeEach((to, from, next) => {
  NProgress.start();
  next();
});

router.afterEach((to, from) => {
  NProgress.done();
});

new Vue({
  el: "#app-root",
  router: router,
  render: h => h(require("./components/App.vue"))
});
```

The `router.beforeEach()` hook is used to call the `NProgress.start` function before every page change, and similarly the `router.afterEach()` hook is used to call the `done` function. That's all there is to it, and we now have a subtle loading bar and spinner displayed across the top of the page while navigation is in progress. Because we set our page data to be loaded before the page changes, this animation will occur for the full duration of the API request and subsequent page render.

Adding a transition on page change

The final step in updating the UX of our existing application is to add a transition on page change. Thankfully, Vue makes this task incredibly easy, and we can achieve what we want with just a few lines of HTML and CSS.

Open up the `ClientApp/components/App.vue` file, find the `<router-view>` component in the template section, and wrap it in a `<transition>` component like this:

```
<transition name="fade" mode="out-in">
  <router-view />
</transition>
```

This `<transition>` component is a built-in Vue component that, as its name suggests, provides a way of transitioning elements as they are hidden and displayed again. There are a number of ways we can control the type of transition that occurs, including a comprehensive set of functions that we can hook into (we'll see how to do this later). However, for now, we'll keep things simple and use CSS class-based transitions. Add the following CSS styles to the `App.vue` style section:

```
.fade-enter-active,  
.fade-leave-active {  
  transition: opacity 0.3s ease-in-out;  
}  
  
.fade-enter,  
.fade-leave-to {  
  opacity: 0;  
}
```

The names of these classes must match the names we used in the preceding `<transition>` component. As we set a `name="fade"` attribute on the transition element, we had to use "fade" as the first part of each of the preceding CSS classes. The second part of these class names, for example, `enter-active`, ties in with the hooks that fire at different stages of the transition.

In this instance, by setting the opacity to zero for `.fade-enter` and `.fade-leave-to`, we are telling the transition component that, before the child component is rendered, and at the end of its life when it is being hidden again, it has an opacity of zero. The default opacity value is one, which will be used while the component is shown, and the transition property in `.fade-enter-active` and `.fade-leave-active` will be added while the component is being displayed and hidden, respectively.

The only other thing to note on the `<transition>` component is the `mode` attribute that we set to `out-in`. If we omit this attribute, the page being hidden will fade out smoothly, but the new page that loads will simply appear without the fade. By adding this attribute, we ensure that the old page fades out, and the new page fades in.

Finally, this is where the 100% height CSS classes we added earlier take effect. If the page being transitioned out does not fill the full height of the screen, the new page appears at the bottom of the screen briefly before being displayed properly. By forcing all pages to fill the height of the screen, we no longer get this problem—we just need to remember to add the `page` class to each root page-level component.

Extending the existing data model

Our database model is currently extremely basic. Before we can implement a fully functioning catalog, we need to extend it considerably to store more information about our products. In our fictional phone shop, this includes things such as screen size, battery life, brand, operating system, features, colors, and storage options.

Dropping the existing database

As we're going to be adding quite a few new required fields to the `Product` entity, unless we specify default values for these properties, the changes will fail when they hit our existing database. At this point in the development cycle, there is absolutely no harm in dropping and recreating the database as often as is required until we get the structure exactly how we want it. This wouldn't be the case if our application was already live, as we would want to ensure that any changes we make can be successfully applied to an existing database.

Before doing anything else, open a Terminal and run the following command:

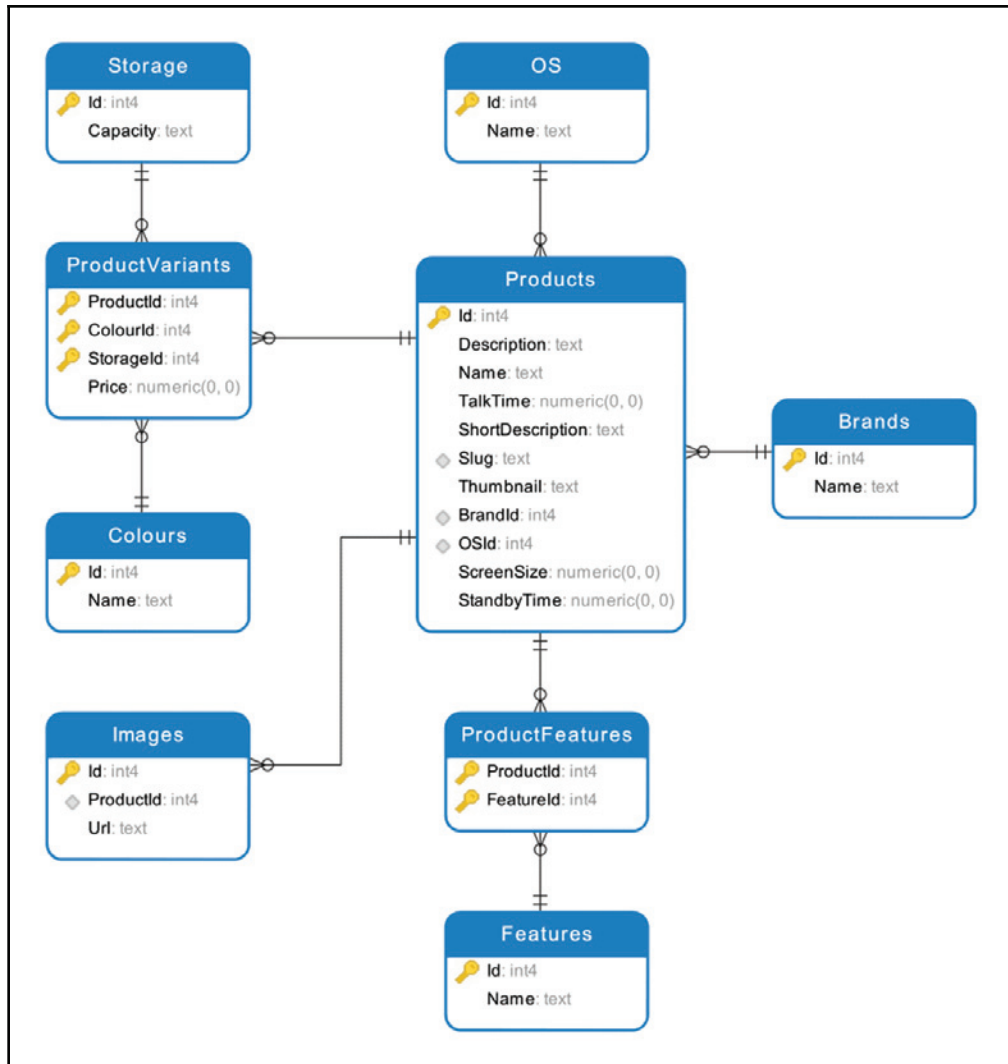
```
dotnet ef database drop
```

After a few seconds, you should see a console message asking if you really do wish to delete the database, which you can confirm by pressing `Y` followed by **Enter**. Just be aware that the webpack middleware will be the last thing to run, so it's easy to miss this message as it won't be the last thing printed to the console. On my machine, it looks like this:

```
Are you sure you want to drop the database 'ecommerce' on server
'tcp://localhost:5432'? (y/N)
infowebpack built 71bacf76d35cf420b3c3 in 1417ms
: Microsoft.AspNetCore.NodeServices[0]
  webpack built 71bacf76d35cf420b3c3 in 1417ms
```

Adding new/updating existing entities

In order to store these things, we need a number of new entities, which will have the following relationships in the database:



Product catalog database structure

The first of these new entities is the Brand entity, which looks like this:

```
namespace ECommerce.Data.Entities
{
    public class Brand
    {
        public int Id { get; set; }
        [Required]
        public string Name { get; set; }
        public List<Product> Products { get; set; } = new List<Product>();
    }
}
```

These entities are very simple and self-explanatory, and, seeing as we have a lot to cover in this chapter, we'll scoot over these pretty quickly. The next one we need is a Colour entity:

```
namespace ECommerce.Data.Entities
{
    public class Colour
    {
        public int Id { get; set; }
        [Required]
        public string Name { get; set; }
        public List<ProductVariant> ProductVariants { get; set; } =
            new List<ProductVariant>();
    }
}
```

At this point, you'll notice we're referencing an entity that does not yet exist. Don't worry, just don't try to build the app until we're finished with the remaining entities. Next up, we have the Feature entity:

```
namespace ECommerce.Data.Entities
{
    public class Feature
    {
        public int Id { get; set; }
        [Required]
        public string Name { get; set; }

        public List<ProductFeature> ProductFeatures { get; set; } =
            new List<ProductFeature>();
    }
}
```

We need to store a collection of images that we can display in a gallery, so we'll need an Image entity:

```
namespace ECommerce.Data.Entities
{
    public class Image
    {
        public int Id { get; set; }
        public int ProductId { get; set; }
        [Required]
        public string Url { get; set; }

        public Product Product { get; set; }
    }
}
```

We also need an OS entity to store the operating system of our phones:

```
namespace ECommerce.Data.Entities
{
    public class OS
    {
        public int Id { get; set; }
        [Required]
        public string Name { get; set; }

        public List<Product> Products = new List<Product>();
    }
}
```

We will need somewhere to store the different storage capacity options of our phones, which we'll use a Storage entity for:

```
namespace ECommerce.Data.Entities
{
    public class Storage
    {
        public int Id { get; set; }
        [Required]
        public string Capacity { get; set; }

        public List<ProductVariant> ProductVariants { get; set; } =
            new List<ProductVariant>();
    }
}
```


This completes the additional entities we need, but we still need a way of linking up these *options* with the products they relate to, so let's start with a `ProductFeature` table:

```
namespace ECommerce.Data.Entities
{
    public class ProductFeature
    {
        public int ProductId { get; set; }
        public int FeatureId { get; set; }

        public Product Product { get; set; }
        public Feature Feature { get; set; }
    }
}
```

This is a standard join table between the `Product` and `Feature` entities to create a many-to-many relationship. We need a similar, albeit more complicated, join table between a `Product` and a unique combination of `Colour` and `Storage`. We'll call this table our `ProductVariant` table:

```
namespace ECommerce.Data.Entities
{
    public class ProductVariant
    {
        public int ProductId { get; set; }
        public int ColourId { get; set; }
        public int StorageId { get; set; }
        [Required]
        public decimal Price { get; set; }

        public Product Product { get; set; }
        public Colour Colour { get; set; }
        public Storage Storage { get; set; }
    }
}
```

This is a pretty standard data model for an e-commerce application, where a product may have a number of configurable options that may alter the base cost of the product. Each unique combination of options is stored as a `ProductVariant` with an associated cost, and when a user places an order, we link the order to the specific variant they selected.

The final change we need to make to our entities is to tie all this together by updating the root `Product` entity to link to all of these new entities. Update it like so:

```
namespace ECommerce.Data.Entities
{
    public class Product
    {
        public int Id { get; set; }
        [Required]
        public string Name { get; set; }
        [Required]
        public string Slug { get; set; }
        [Required]
        public string Thumbnail { get; set; }
        [Required]
        public string ShortDescription { get; set; }
        [Required]
        public string Description { get; set; }
        [Required]
        public decimal ScreenSize { get; set; }
        [Required]
        public decimal TalkTime { get; set; }
        [Required]
        public decimal StandbyTime { get; set; }
        [Required]
        public int BrandId { get; set; }
        [Required]
        public int OSId { get; set; }

        public List<Image> Images { get; set; }
        public Brand Brand { get; set; }
        public OS OS { get; set; }
        public List<ProductFeature> ProductFeatures { get; set; } =
            new List<ProductFeature>();
        public List<ProductVariant> ProductVariants { get; set; } =
            new List<ProductVariant>();
    }
}
```

We've added a number of EF navigational properties for the new entities we just created, and also added a few extra decimal fields for the screen size, talk time, and standby time of the handsets.

Updating the DbContext class

We now need to update our context to make EF aware of the changes we've made so that we can reflect them in the database. Open up the `Data/EcommerceContext.cs` class and update the list of `DbSet<>` properties like so:

```
public DbSet<Brand> Brands { get; set; }
public DbSet<Colour> Colours { get; set; }
public DbSet<Feature> Features { get; set; }
public DbSet<Image> Images { get; set; }
public DbSet<OS> OS { get; set; }
public DbSet<Product> Products { get; set; }
public DbSet<ProductFeature> ProductFeatures { get; set; }
public DbSet<ProductVariant> ProductVariants { get; set; }
public DbSet<Storage> Storage { get; set; }
```

Next, we need to instruct EF on how to configure our many-to-many relationships by adding to our overridden `OnModelCreating` method:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<Product>()
        .HasIndex(b => b.Slug)
        .IsUnique();

    modelBuilder.Entity<ProductFeature>()
        .HasKey(x => new { x.ProductId, x.FeatureId });

    modelBuilder.Entity<ProductVariant>()
        .HasKey(x => new { x.ProductId, x.ColourId, x.StorageId });
}
```

Creating a migration to reflect the model changes

With these changes in place, we're now in a position to create a migration to apply these changes to the database. As we did before, open up a Terminal window and run the following command:

```
dotnet ef migrations add Catalogue
```

If all is well, the migration should be created successfully and two new files will have been created in the `Data/Migrations` directory. As usual, it's highly recommended to give them a brief glance over to make sure everything looks OK. Now, the next time we run the application, this migration will be run automatically, but if you'd rather be explicit and run it yourself, then execute the following command in your Terminal:

```
dotnet ef database update
```

You can then browse the new database structure using *pgAdmin4* if you're interested.



pgAdmin4 was installed along with PostgreSQL back in Chapter 2, *Setting Up the Development Environment*. As we've done before, you can use it to browse your local databases and check the structure/data in much the same way as you would with SQL Server Management Studio if that's what you're used to.

Updating the application's seed data

We still don't have a UI or API endpoints to create this data, so we need to extend our seed methods to create some sample data that we can consume from the frontend. In the `Data/DbContextExtensions.cs` class, we'll need two new methods that we'll call from the `EnsureSeeded` method. The first will be called `AddColoursFeaturesAndStorage`, and looks like this:

```
private static void AddColoursFeaturesAndStorage(EcommerceContext context)
{
    if (context.Colours.Any() == false)
    {
        var colours = new List<string>() { "Black", "White", "Gold",
            "Silver", "Grey", "Spacegrey", "Red", "Pink" };

        colours.ForEach(c => context.Add(new Colour
        {
            Name = c
        }));

        context.SaveChanges();
    }
    // rest of method omitted for brevity...
}
```

As these seed methods are pretty long and fairly uninteresting, I've omitted all but an example of how to seed the data we need. You can download the source code for this chapter, which contains the entire methods, but I've basically just made up a load of fake company and product names.

The next new method is the `AddOperatingSystemsAndBrands` method:

```
private static void AddOperatingSystemsAndBrands(EcommerceContext context)
{
    if (context.OS.Any() == false)
    {
        var os = new List<string>() { "Android", "iOS", "Windows" };

        os.ForEach(o => context.OS.Add(new OS
        {
            Name = o
        }));

        context.SaveChanges();
    }
    // rest of method omitted for brevity...
}
```

The `AddProducts` method has remained unchanged in structure, but we do need to update the data we're seeding on each product object to include the additional fields we added earlier. I won't show every product that we're seeding as they are all much the same. However, as an example, we need to add the `ScreenSize`, `TalkTime`, and `StandbyTime` decimal fields:

```
new Product
{
    //...
    ScreenSize = 5M,
    TalkTime = 8M,
    StandbyTime = 36M
    //...
}
```

We then need `Brand` and `OS` objects, as well as a list of `Image` objects, which we will display in our new gallery component on the product details page:

```
new Product
{
    //...
    Brand = context.Brands.Single(b => b.Name == "Acme"),
    OS = context.OS.Single(os => os.Name == "Android"),
    Images = new List<Image>
```

```
{
    new Image { Url = "/assets/images/gallery1.jpeg" },
    new Image { Url = "/assets/images/gallery2.jpeg" },
    new Image { Url = "/assets/images/gallery3.jpeg" },
    new Image { Url = "/assets/images/gallery4.jpeg" },
    new Image { Url = "/assets/images/gallery5.jpeg" },
    new Image { Url = "/assets/images/gallery6.jpeg" }
}
//...
}
```

Next is a list of product feature objects:

```
new Product
{
    //...
    ProductFeatures = new List<ProductFeature>
    {
        new ProductFeature
        {
            Feature = context.Features.Single(f => f.Name == "3G")
        },
        new ProductFeature
        {
            Feature = context.Features.Single(f => f.Name == "Bluetooth")
        },
        new ProductFeature
        {
            Feature = context.Features.Single(f => f.Name == "WiFi")
        },
        new ProductFeature
        {
            Feature = context.Features.Single(f => f.Name == "GPS")
        }
    }
    //...
}
```

The final thing we need is a list of product variant objects:

```
new Product
{
    //...
    ProductVariants = new List<ProductVariant>
    {
        new ProductVariant
        {
            Colour = context.Colours.Single(c => c.Name == "Black"),

```

```
        Storage = context.Storage.Single(s => s.Capacity == "32GB"),
        Price = 299M
    },
    new ProductVariant
    {
        Colour = context.Colours.Single(c => c.Name == "Black"),
        Storage = context.Storage.Single(s => s.Capacity == "64GB"),
        Price = 349M
    },
    new ProductVariant
    {
        Colour = context.Colours.Single(c => c.Name == "Gold"),
        Storage = context.Storage.Single(s => s.Capacity == "32GB"),
        Price = 319M
    },
    new ProductVariant
    {
        Colour = context.Colours.Single(c => c.Name == "Gold"),
        Storage = context.Storage.Single(s => s.Capacity == "64GB"),
        Price = 369M
    }
}
}
```

Again, all I've done is make up a number of fake product names and a bunch of different variants with different prices, as well as added a local URL to a selection of images that I downloaded from a free stock image website. You can grab these from the source code for this chapter, or simply download your own images and modify the URLs as necessary.

In order to make these image URLs work, I created a `wwwroot/assets/images` directory and dropped all of the product images in there. In a real application, I'd most likely make use of some kind of cloud file storage such as Microsoft Azure or Amazon S3, but that is beyond the scope of this book.

At this point, our data changes are complete, and, if you haven't already, you can run the application again now and you will be greeted with exactly the same application as before, but with our newly updated seed data.

Filtering on the server

As it stands, we have no way of filtering our list of products, which will soon become a problem when we have more than a handful of them. Users of our shop will expect to be able to filter down the list to help them find exactly what they are looking for, without scrolling through pages and pages of results.

With the data model we just set up, we can filter the list by brand, price, screen size, capacity, color, operating system, and feature. We have a few changes to make to achieve this as our API does not yet support filtering on the product list query, and we have no filter components in our client app. We'll start by making the changes we need to the API.

Updating controller actions to support filtering

The first thing we need to do is update our products controller action to support some optional filter parameters. Open up `Features/Products/Controller.cs`, locate the `Find` action, and change its method signature to the following:

```
[HttpGet]
public async Task<IActionResult> Find(string brands, int? minPrice, int?
maxPrice, int? minScreen, int? maxScreen, string capacity, string colours,
string os, string features)
{
    //method body omitted for brevity...
}
```

Here, we are specifying that the client needs to send string values for the brand, capacity, colors, and features filters. Multiple selections can be concatenated with some kind of delimiter value, which we will see in a moment. In addition to these, we are accepting optional min/max price integers, as well as optional min/max screen size integers. If no values are provided for any of these filters, they will be ignored and we simply return all products.

With these parameters in place, we need to do some processing of the strings. We'll say that multiple filter items will be concatenated with a `|` character, meaning we'll need to use the built-in `C# String.Split()` method to turn the string parameters into lists:

```
var Brands = string.IsNullOrEmpty(brands) ? new List<string>() :
brands.Split('|').ToList();
var Capacity = string.IsNullOrEmpty(capacity) ? new List<int>() :
capacity.Split('|').ToList();
var Colours = string.IsNullOrEmpty(colours) ? new List<string>() :
colours.Split('|').ToList();
var OS = string.IsNullOrEmpty(os) ? new List<string>() :
os.Split('|').ToList();
var Features = string.IsNullOrEmpty(features) ? new List<string>() :
features.Split('|').ToList();
```

First, we check if the string is null or empty, and if it is, we create a new empty list of string objects. Alternatively, if the string is passed a value, we call the `Split` method followed by the `ToList` method to return a new list of the individual filter items from the client.

Finally, we can update the database query to take these filters into consideration:

```
var products = await _db.Products
    .Where(x => Brands.Any() == false || Brands.Contains(x.Brand.Name))
    .Where(x => minPrice.HasValue == false || x.ProductVariants.Any(v =>
v.Price >= minPrice.Value))
    .Where(x => maxPrice.HasValue == false || x.ProductVariants.Any(v =>
v.Price <= maxPrice.Value))
    .Where(x => minScreen.HasValue == false || x.ScreenSize >=
Convert.ToDecimal(minScreen.Value))
    .Where(x => maxScreen.HasValue == false || x.ScreenSize <=
Convert.ToDecimal(maxScreen.Value))
    .Where(x => Capacity.Any() == false || x.ProductVariants.Any(v =>
Capacity.Contains(v.Storage.Capacity)))
    .Where(x => Colours.Any() == false || x.ProductVariants.Any(v =>
Colours.Contains(v.Colour.Name)))
    .Where(x => OS.Any() == false || OS.Contains(x.OS.Name))
    .Where(x => Features.Any() == false || Features.All(f =>
x.ProductFeatures.Any(pf => pf.Feature.Name == f)))
    .Select(x => new ProductListViewModel
    {
        Id = x.Id,
        Slug = x.Slug,
        Name = x.Name,
        ShortDescription = x.ShortDescription,
        Thumbnail = x.Thumbnail,
        Price = x.ProductVariants.OrderBy(v => v.Price).First().Price
    })
    .ToListAsync();
```

Note how I've chained multiple LINQ `Where` clauses in this query. This is an entirely personal preference, but I prefer to chain multiple `Where` clauses rather than use a whole load of logical operators within a single `Where` clause. I find it much easier to read, but feel free to change this if you don't.

Essentially, I've added a single `Where` clause per filter parameter being passed into the controller action. On each clause, if no value is passed to that particular filter, the first half of the logical OR operator will return `true`, and so the filter will have no effect on the resultant list of products. If a value *is* passed, we use it to return only those products that match based on the specific properties being filtered against.

For example, look at the following `Where` clause:

```
Where(x => Brands.Any() == false || Brands.Contains(x.Brand.Name))
```

This will limit the query results to those products whose associated brand name is contained within the list of brands passed to the controller action.

The final thing to note here is that we are no longer returning domain model entities directly to the client. Doing so is generally a bad idea and is frowned upon, so, instead, we'll use a LINQ `Select` call to project each returned product to an associated `ProductListViewModel`, which contains only the data we need to display in our catalog list page. This new view model class lives in the `Features/Products` folder and looks like this:

```
namespace ECommerce.Features.Products
{
    public class ProductListViewModel
    {
        public int Id { get; set; }
        public string Slug { get; set; }
        public string Name { get; set; }
        public string ShortDescription { get; set; }
        public decimal Price { get; set; }
        public string Thumbnail { get; set; }
    }
}
```

In a real application, we'd normally take our best practices to another level by removing this kind of logic from the controller entirely, and maybe even throw in a tool such as `AutoMapper` to make our lives a little easier when mapping domain models to view models in this way. However, to keep things simple and concise, I'm sticking to raw EF queries directly in controller actions for now.



My personal favorite approach to structuring .NET Core applications is using the excellent `MediatR` library to separate data access into distinct *commands* and *queries*. It also provides very useful ways of handling cross-cutting concerns such as error handling and logging.

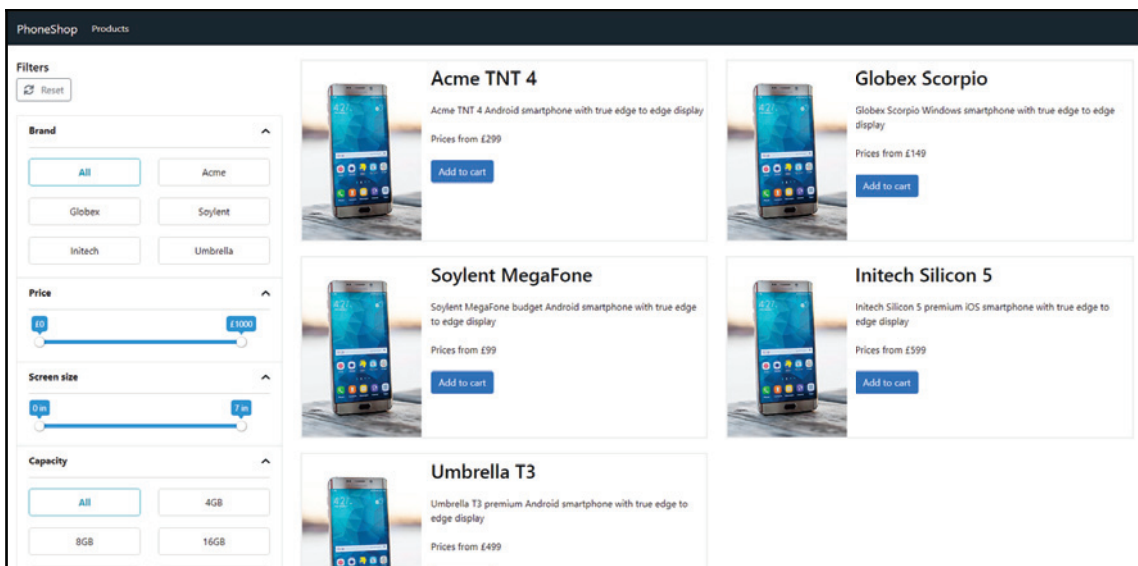
Testing our filtering logic

This is all we need to do on the server side to handle filtering in our product catalog. You can test this out by using your browser to navigate to URLs such as `http://localhost:5000/api/products`, followed by `http://localhost:5000/api/products?brands=Acme|Soylent` to see how the list of products returned changes based on the query string filters.

Filtering on the client

This is where things get a little trickier. We need to come up with a simple but powerful UI that provides suitable filter controls for each of our server-side filter parameters. Most of these are lists of string values concatenated with the pipe character, for which we can simply display a list or grid containing the available options that the users can click on to select. However, we also have a couple of numeric filters accepting a min/max value for each. It would be nice to include some kind of slider control for these range-based inputs. Finally, we need a way of clearing the selected filters.

The following screenshot is what we are aiming to build:



Product catalog filters component

Each filter will be in an accordion section so that the user can collapse them to save space, and the results panel should update in real time as they change their filters.

Installing additional dependencies

In order to build this UI, we'll need a few additional libraries installed and/or configured. First up, we can see that we need some icons for the reset button and the filter accordions; we'll make use of **Font Awesome** for these as it's so easy to install and use, and has become the standard for web icons.

Installing Font Awesome

We have a few options here. First and foremost, we should head over to the Font Awesome website at <https://fontawesome.com/> and check out their **Get Started** page. The recommended way of using Font Awesome via a CDN just happens to be the easiest, which is why we'll stick with it for the purposes of this book.

Installing Font Awesome using the CDN is as easy as opening up the `Features/Shared/_Layout.cshtml` file and adding the following link element at the very bottom of the head section:

```
<link href="https://use.fontawesome.com/releases/v5.0.6/css/all.css"
      rel="stylesheet" />
```

Now, if we follow the instructions on the Font Awesome website, they tell us to add a script reference to a JavaScript file rather than this CSS file. This is what I initially did until I noticed a strange bug whereby icons being conditionally rendered using the `v-if` directive didn't work properly. The resolution I found was to use a CSS file reference instead, which is why we've ignored the installation instructions here.

If you're interested, there are other Vue-specific ways of installing Font Awesome, which you can find on their website under the **Advanced Options** section of the **Get Started** page. Specifically, they have a couple of npm packages that we could have installed and made use of, but we'd need to make a few more changes than our one simple change so far.

Installing additional npm packages

With Font Awesome installed, we can look at our next requirement: accordions and sliders. Building a custom slider component would be fairly involved, and just not worth the effort when there are pre-built components on npm that we can leverage instead. However, it is pretty simple to build our own accordions, and this will also leave us with full control over how they are displayed.

For basic animation, we could make use of CSS transitions, but sooner or later we'll hit the limits of what we can easily do with CSS. Instead, we'll install the `velocity.js` library which provides accelerated JavaScript animations, and is much more powerful than CSS alone. In addition to this, we'll install the `vue-slider-component` library which, as the name suggests, provides us with a ready-made slider component.

Open a Terminal in the project root directory and run the following commands:

```
yarn add velocity-animate  
yarn add vue-slider-component
```

While we're here, we're going to change the library that we use for making API requests. The reason will become more apparent later, but essentially, there are some annoying limitations with `isomorphic-fetch`. We are going to replace it with a library called `axios`, which is an alternative HTTP client that works both in the browser and on the server.

Run the following commands in your Terminal:

```
yarn remove isomorphic-fetch  
yarn add axios
```

There is just one last change we need to make to completely replace `isomorphic-fetch` with `axios`, so open up the `webpack.config.vendor.js` file, locate the `vendor` array within the `entry` object, and replace the `isomorphic-fetch` line with `axios`. The completed `vendor` array should then look like this:

```
vendor: [  
  "event-source-polyfill",  
  "axios",  
  "vue",  
  "vue-router",  
  "bootstrap/dist/css/bootstrap.min.css",  
  "bootstrap-vue",  
  "nprogress/nprogress.css"  
]
```

Recalling the last time we changed this file, we need to force a `webpack` rebuild before these changes take effect, so run the following command in your Terminal:

```
yarn webpack
```

Building an accordion component

We now have everything we need to start making our filter components. We already stipulated that each filter component will be rendered as a collapsible accordion, but, with each filter having different contents, how do we achieve what we want without replicating the accordion logic for each filter? We need to build a reusable accordion component with enough flexibility to change what content we display in the collapsible sections. Luckily for us, Vue provides *slots* for doing just that.

Defining the accordion template structure

First of all, create a new component file in the `ClientApp/components/catalogue` directory, which I've named `FilterAccordion.vue`. The template section of this component looks like this:

```
<template>
  <b-list-group-item>
    <div :class="{ 'header': true, 'open': open }" @click="open =
      !open">
      <slot name="header"></slot>
      <i class="fas fa-chevron-down float-right"></i>
    </div>
    <transition @enter="onEnter" @leave="onLeave">
      <div class="body mt-3 pt-2" v-if="open">
        <slot name="body"></slot>
      </div>
    </transition>
  </b-list-group-item>
</template>
```

The root element we're using for this component is the `b-list-group-item` element from the `Bootstrap-Vue` library, and is used for rendering nicely formatted lists. We then have two `div` elements: one for the header of the accordion, and one for the body.

On the `div` header, we're using the `v-bind` directive to bind a dynamic `open` class based on a data property, also called `open`. We also include an `@click` event handler to toggle the `open` property when the user clicks on the header section of the accordion. As they click, the event handler fires and toggles the `open` property, which in turn toggles the `open` class on the header. We'll see why this is important shortly. The last thing to note in the header section is the `i` element that we're using. This is how we make use of the Font Awesome icons that we installed earlier. We can place an `i` element anywhere in our application, and provide appropriate class names to tell Font Awesome which icon we actually want to display. In this case, we're using the `fa-chevron-down` class to render a down arrow.

As this is an accordion, we know that the body section will not always be visible. As such, we use the `v-if` directive to control the visibility of it, based on the `open` data property that we've already discussed. This is all we need to do to get a basic accordion working, but it won't be the nicest UX without a nice transition to make it less jarring when toggled. To achieve this, we're wrapping the body section with a `transition` component, just like we saw around the `router-view` component earlier in this chapter. However, rather than passing a `name` prop to the transition component like we did before, we're attaching `@enter` and `@leave` event handlers, which will trigger a pair of JavaScript hooks instead. This is because we want to make use of Velocity for JavaScript-based animation, rather than the CSS animation we used before.

Finally, we're making use of the `slot` component in both the header and body sections of this template. This is how we create a placeholder, as it were, so that, when we call this component from a parent, we can insert different content into each accordion that we render. As we have multiple slots, we need to be able to differentiate between them and specify which slot we want to place our content into. To do so, we simply add a `name` prop to each slot.

Defining the accordion behavior

The `script` section of our accordion component looks like this:

```
<script>
import * as Velocity from "velocity-animate";

export default {
  name: "filter-accordion",
  data() {
    return {
      open: true
    };
  },
}
```

```
methods: {
  onEnter(el, done) {
    Velocity(el, "slideDown", {
      duration: 200,
      easing: "ease-in-out",
      complete: done
    });
  },
  onLeave(el, done) {
    Velocity(el, "slideUp", {
      duration: 250,
      easing: "ease-in-out",
      complete: done
    });
  }
};
</script>
```

This may look a little complicated at first, but when we break it down, it's really quite simple.

We start by importing the Velocity library that we installed earlier, and providing it with an alias of `Velocity`. We then have our standard `export default {}` object declaration that we've seen so many times before, containing a `data()` function and a `methods: {}` object—also just like we've seen many times before.

The `data` function is very simple, and only returns an object containing the `open` property that we saw being toggled and used to bind classes in the preceding template section. The `methods` object is a little more complicated, but essentially just contains the two JavaScript animation hooks that we referenced in the transition component in the template. The `onEnter` method will be called as the accordion is opened, and its body section will become visible; similarly, the `onLeave` method will be called as the accordion is closed. Each method receives an `el` argument, which is a reference to the element that is being transitioned, as well as a `done` argument which is a callback function we need to invoke at the end of the animation. If you've ever used jQuery `animate` before, the `Velocity` syntax will look very familiar to you. We call the `Velocity` function and pass an element reference, the name of the animation we want to apply, and an object containing any options we want to apply to the animation. In this case, we use either the `slideUp` or `slideDown` animation, and specify a duration value and an easing value, and pass the `done` function to be invoked as the animation completes.

Styling the accordion component

Our accordion component is now functionally complete, but there are just a few simple styles that we need to apply to get it looking the way we want it to. The `style` block for this component looks like this:

```
<style lang="scss" scoped>
.header {
  font-weight: bold;
  cursor: pointer;

  .fa-chevron-down {
    position: relative;
    top: 5px;
    transition: all 0.2s ease-in-out;
  }

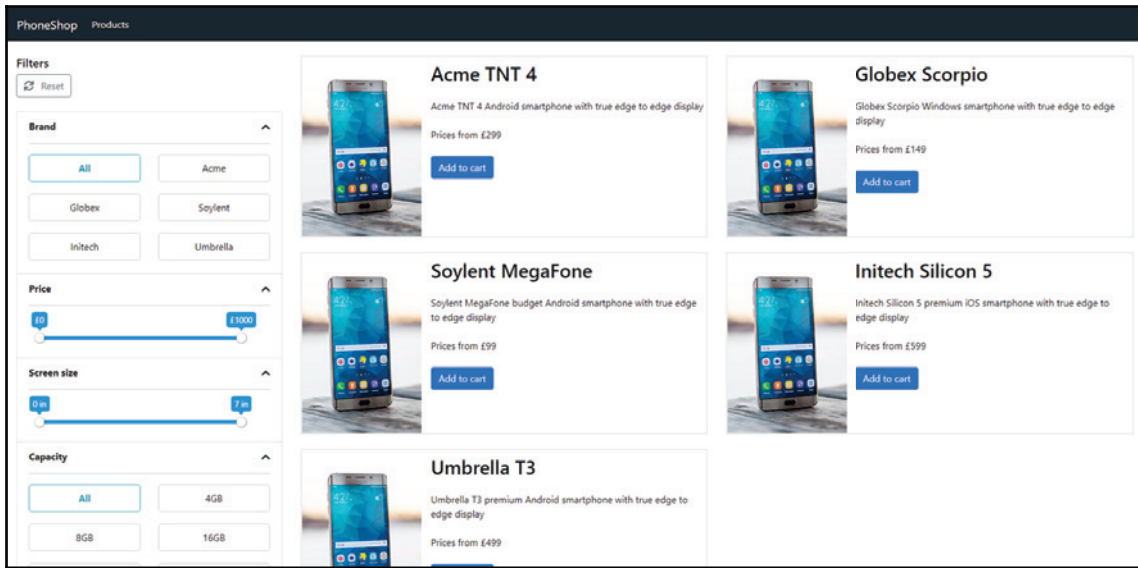
  &.open {
    .fa-chevron-down {
      transform: rotate(180deg);
    }
  }
}

.body {
  border-top: 1px solid #ccc;
}
</style>
```

There is nothing overly complicated going on here, but it's worth noting the `transform` style near the middle. Remember how we dynamically apply the `open` class to the accordion header depending on whether it's open or closed? We hook into that dynamic class here and rotate the Font Awesome icon 180 degrees to become an up arrow. Just above this line, there is also a `transition` style—this makes the icon rotation a nice smooth animation rather than being just an instant flicker between states.

Building the filters component

With a functioning accordion, we can now build the actual filters component that will make use of it. There will be a lot going on here, so we'll need to break things down to make sure we can see what's going on. As a reminder, the section on the left of the following screenshot shows the filters component we are about to build:



Product catalog filters component

Scaffolding the filters component template

Create a `ClientApp/components/catalogue/Filters.vue` file and add a barebones template section as follows:

```
<template>
  <div class="filters mb-4">
    <h5 class="mt-4">Filters</h5>
    <b-btn variant="outline-secondary" @click.prevent="reset">
      <i class="fas fa-sync mr-2"></i> Reset</b-btn>
    <b-list-group class="mt-4">
      // our list of filter accordions will go here...
    </b-list-group>
  </div>
</template>
```

We start with a small title and a reset button—we'll look at the `reset` method later, but note how we're making use of the Font Awesome library again by including the `sync` icon within the reset button. Apart from this, it's a fairly simple template so far.

Where it will get complicated is within the `b-list-group` element, which is another custom element from the `Bootstrap-Vue` library. This should now make things clearer as to why we used the `b-list-group-item` element as the root-level element for the accordion component. Now, we have a number of accordions to insert, but as they're not all exactly the same, we'll go over them one by one.

Adding a brand filter

The first accordion we need is the brands filter:

```
<filter-accordion>
  <span slot="header">Brand</span>
  <b-row slot="body">
    <b-col cols="6">
      <div :class="{ 'filter-item': true,
        'active': brands.length === 0 }" @click="clearBrands">All</div>
    </b-col>
    <b-col cols="6" v-for="item in filters.brands" :key="item">
      <div :class="{ 'filter-item': true, 'active':
        brands.indexOf(item) > -1 }" @click="filterBrand(item)">
        {{ item }}</div>
    </b-col>
  </b-row>
</filter-accordion>
```

As our custom `FilterAccordion.vue` component made use of slots, we can insert content between its opening and closing HTML tags. Furthermore, we added two different slots: one for the header and one for the body. In order to specify which slot we're targeting, we include the `slot` prop on any standard or custom HTML element within the template.

We're starting off by simply inserting a `span` element into the `header` slot in order to provide the text that displays as the title of the accordion panel. Next, we're inserting a bootstrap `b-row` element into the `body` slot, including a number of columns for each of the available brands that the user can filter by. We start the list of brands off with a hardcoded `All` filter item, with a dynamic `active` class binding based on whether or not we've selected any brands to filter on. We're also attaching a click handler to this item to call into a `clearBrands` method, which we'll see later on. We then use the `v-for` directive to loop over the `filters.brands` property, rendering the same HTML markup as we used for the preceding `All` item. However, this time, we base the `active` class binding on whether or not our list of selected brands contains the specific brand being rendered. The click handler also calls a different method called `filterBrand`, which we will use later to add or remove the brand depending on whether it has already been selected or not.

This is probably the most complex template code we've seen so far, so it is definitely worth taking the time to understand what is going on here before moving on. That being said, until we see what the `script` section looks like, it probably won't make complete sense.

Adding a price filter

The next accordion we need is the price filter:

```
<filter-accordion>
  <span slot="header">Price</span>
  <div class="slider" slot="body">
    <vue-slider
      :value="price"
      formatter="{value}"
      :min=0
      :max=1000
      :interval=50
      :lazy=true
      width="90%"
      @callback="filterPrice">
    </vue-slider>
  </div>
</filter-accordion>
```

Thankfully, it is a lot easier to see what's going on straight away. We start the same way we did with the brands accordion: by inserting a `span` element into the `header` slot. The `body` slot in this instance is a lot simpler, and only contains a single `div` element, which we use to wrap an instance of the `vue-slider` component that we installed earlier in this chapter. The slider component can optionally take many more props than we are using here, which makes it incredibly versatile. For our needs, though, this is all we need.

We're binding the value of the slider to a `price` property from our component state, and the `min`, `max`, and `interval` values to arbitrary values that make sense for the price of a phone. The `formatter` prop is used to specify the template of the labels at each end of the slider. As we are working with currency, we simply prefix the value passed with a `£` symbol.



For the purposes of this demonstration, I am not including any kind of localization of things such as currencies and date formats. For production apps that may be used in multiple locales, it is worth looking into a localization npm package such as `globalizejs` or `vue-globalize`.

The `@callback` event handler is used to call our `filterPrice` method as the user interacts with the slider and changes its value. Now, if we omit the `lazy` prop, the `@callback` event handler would fire each time the slider detects the value changing, even if the user hasn't let go of their mouse to signify the final position of their choice. By setting `lazy` to `true`, we're instructing the slider to wait until the mouse click is released before firing the `@callback` event, and thus preventing more API calls than we need to make.

Adding a screen size filter

The next filter in the list is the screen size filter:

```
<filter-accordion>
  <span slot="header">Screen size</span>
  <div class="slider" slot="body">
    <vue-slider
      :value="screenSize"
      formatter="{value} in"
      :min=0
      :max=7
      :lazy=true
      width="90%"
      @callback="filterScreenSize">
    </vue-slider>
  </div>
</filter-accordion>
```

This is almost identical to the price filter, so we won't dwell on it for long, but just note the slightly different `formatter` prop, as well as triggering a different method in the `@callback` event handler.

Adding the remaining color, OS, and feature filters

The next three filters, for color, operating system, and feature, are also almost identical to each other, along with the brand filter that we defined in the preceding section. The color filter looks like this:

```
<filter-accordion>
  <span slot="header">Colour</span>
  <b-row slot="body">
    <b-col cols="6">
      <div :class="{ 'filter-item': true,
        'active': colours.length === 0 }" @click="clearColours">
        All</div>
    </b-col>
    <b-col cols="6" v-for="item in filters.colours" :key="item">
      <div :class="{ 'filter-item': true, 'active':
        colours.indexOf(item) > -1 }" @click="filterColour(item)">
        {{ item }}</div>
    </b-col>
  </b-row>
</filter-accordion>
```

The operating system filter looks like this:

```
<filter-accordion>
  <span slot="header">Operating system</span>
  <b-row slot="body">
    <b-col cols="6">
      <div :class="{ 'filter-item': true,
        'active': os.length === 0 }" @click="clearOS">All</div>
    </b-col>
    <b-col cols="6" v-for="item in filters.os" :key="item">
      <div :class="{ 'filter-item': true, 'active': os.indexOf(item)
        > -1 }" @click="filterOS(item)">{{ item }}</div>
    </b-col>
  </b-row>
</filter-accordion>
```

And, finally, the feature filter looks like this:

```
<filter-accordion>
  <span slot="header">Features</span>
  <b-row slot="body">
    <b-col cols="6">
      <div :class="{ 'filter-item': true, 'active': features.length
        === 0 }" @click="clearFeatures">All</div>
    </b-col>
    <b-col cols="6" v-for="item in filters.features" :key="item">
```

```
    <div :class="{ 'filter-item': true, 'active':
      features.indexOf(item) > -1 }"
      @click="filterFeature(item)">{{ item }}
    </div>
  </b-col>
</b-row>
</filter-accordion>
```

The only differences in these filters are the data properties being looped over, the methods being called by the event handlers, and the properties being used for class binding. If you can understand one of them, you can understand all of them, so we won't say any more about these.

Scaffolding the filters component behavior

The script block of this template is equally long and complex, so we'll go over it in sections again! As an overview, this is what it looks like:

```
<script>
import FilterAccordion from "./FilterAccordion.vue";
import vueSlider from "vue-slider-component";

export default {
  name: "filters",
  props: {
    filters: {
      type: Object,
      required: true
    }
  },
  components: {
    FilterAccordion,
    vueSlider
  },
  computed: {
    // computed props omitted for brevity...
  },
  methods: {
    // methods omitted for brevity...
  }
};
</script>
```

First, we need to import the necessary child components that we'll make use of, including our own `FilterAccordion` component as well as the `vue-slider` component that we installed from npm earlier. Then, we will declare a standard component definition named `"filters"`, register the child components we just imported, and specify a required `filters` prop of type `Object`. This `filters` object prop will contain the collections of filters that we've been looping over in the preceding template sections, and we'll pass it in from the parent component after we retrieve the filter options from an API call later in this chapter.

Defining the filters component computed properties

The computed properties section will be where we store the selected filter items the user clicks on, as well as the current values of the price and screen size sliders. However, as we'll see in a moment, it is a little more complicated than that, as we'll also be keeping these values in sync with the browser URL query string.

Update the computed section to look like this:

```
computed: {
  brands() {
    return this.$route.query.brands || "";
  },
  price() {
    return [
      this.$route.query.minPrice || 0,
      this.$route.query.maxPrice || 1000
    ];
  },
  screenSize() {
    return [
      this.$route.query.minScreen || 0,
      this.$route.query.maxScreen || 7
    ];
  },
  capacity() {
    return this.$route.query.capacity || "";
  },
  colours() {
    return this.$route.query.colours || "";
  },
  os() {
    return this.$route.query.os || "";
  },
  features() {
    return this.$route.query.features || "";
  }
}
```



```
    }  
  }
```

If you haven't done much JavaScript before, this syntax may look a little strange, but in actual fact it maps pretty closely back to a similar concept in C#. For each of our filters, we're returning either a string or an array, based on query parameters from the current page route. If no query parameter is found matching that name, we default to either an empty string or a specific integer value. In essence, this:

```
return this.$route.query.brands || "";
```

Is very similar to writing this in C#:

```
return object.property ?? defaultValue;
```

We already saw that our filters API endpoint is expecting a single string value per filter, which is why we default to empty strings rather than empty arrays. Also, seeing as these are computed properties, if we update the query parameters at any point, Vue will detect the change in these computed properties and trigger a UI refresh wherever necessary.

This is all very well and good, but couldn't we make this a lot simpler and just store the users filter selections in standard component state using arrays? In short, we definitely could, and this is a perfectly acceptable approach for a lot of online shops that I've visited in the past. However, if we rely on local component state to store the user's selections, they will be lost if and when they force the browser to refresh the page. There are a few different ways that we can combat this, including pushing the selections into the browser's local storage, then retrieving them if they exist on page load, and pushing the selections to the URL query string. There are definitely pros and cons to each solution, but the deciding factor for me is that, if we make use of the query parameters, our users can share their search results with others by copying the URL.

Defining the filters component methods

So, we can populate our component state from the URL, but as it stands we can't update it. We're missing the `methods` section of the component that will fill in that gap. To get us started, add the following `methods` section:

```
methods: {  
  reset() {  
    this.$router.push({ query: {} });  
  }  
}
```

Recall the top of the template for this component; we have the reset button, which has a click event handler invoking a `reset` method. This is all we need to do to clear all the filters and have the list reset. Our computed properties that contain the selected filter items are directly tied to the query parameters in the URL. By pushing an empty `query` object without specifying a new URL, `vue-router` will overwrite the query parameters with this empty object. All of our computed properties will then update and the UI will refresh back to the default state.

For each filter, we need two methods to manipulate the URL and refresh the filter results. We need a method that takes a filter item and pushes it to the URL if it hasn't already been selected, or removes it if it has, and a method that removes all selected items for this filter. The following methods show how we do this for the brands filter:

```
clearBrands() {
  if (this.brands.length) {
    let query = Object.assign({}, this.$route.query);
    delete query.brands;

    this.$router.push({ query: query });
  }
},
filterBrand(brand) {
  let query = Object.assign({}, this.$route.query);
  let split = query.brands ? query.brands.split("|") : [];

  if (split.indexOf(brand) > -1) {
    let index = split.indexOf(brand);
    split.splice(index, 1);
  } else {
    split.push(brand);
  }

  if (split.length) {
    let joined = split.join("|");
    query.brands = joined;
  } else {
    delete query.brands;
  }

  this.$router.push({ query: query });
}
```

Again, there's quite a lot going on here, so we'll go through it line by line, starting with the `clearBrands` method.

First, we check if we've actually selected any brands or not. If we have, we use the `Object.assign` function to clone the `$route.query` object into a new local method variable, then delete the `brands` property from it before pushing the newly updated `query` object into the query string like we did in the preceding `reset` method. If we haven't selected any brands, we do nothing. But why do we need to bother cloning the query object rather than modifying it directly? The answer isn't entirely obvious, but essentially it just boils down to how the reactivity system works in Vue. If we were to try and modify the query object directly, and then push it back into `vue-router`, the component would not detect the change, and as such we would see no change in the UI. By taking a clone of the query object and pushing that back into `vue-router`, there is no dispute that we've changed something, and the reactivity system kicks in as we'd expect it to.

Moving on to the `filterBrand` method, we can see that it takes a `brand` parameter which is the specific filter item the user just clicked on. We start by taking a similar approach as before and taking a clone of the `$route.query` object. Now, we know that any selected brands will have been concatenated into a single string, so we call the `split` function on the `brands` property, but default to an empty array if it doesn't exist. We can now check to see if the `brand` parameter already exists in the array we just created, and either remove it or add it, depending on the outcome. To remove it, we first need to find its index within the array, then call the `splice` method to remove it based on that index. To add it, all we need to do is call the `push` method and pass it as a parameter. Next, we do another check to see if we still have any items in the array before deciding what to do next. If we do still have some selected brands, we concatenate them back together using the `join` method, passing the `|` character as the delimiter to use, before overwriting the `brands` property on our clone of the query object. Alternatively, if we no longer have any brands selected, we simply delete the `brands` property from the `query` object. Finally, we push the final version of the cloned `query` object back into the current URL using `vue-router`.

The methods for updating the slider-based numeric values are much simpler. The `filterPrice` method looks like this:

```
filterPrice(prices) {
  let query = Object.assign({}, this.$route.query);
  query.minPrice = prices[0];
  query.maxPrice = prices[1];

  this.$router.push({ query: query });
}
```

The slider in the filter component emits an array containing the current min and max values selected by the user. This array is what is passed to the `filterPrice` method as the `prices` parameter, as shown in the preceding code. As before, we clone the current `$route.query` object before setting the `minPrice` and `maxPrice` properties based on the two elements in the `prices` array, then push the new `query` object onto the current URL.

The `filterScreenSize` method looks almost identical:

```
filterScreenSize(sizes) {
  let query = Object.assign({}, this.$route.query);
  query.minScreen = sizes[0];
  query.maxScreen = sizes[1];

  this.$router.push({ query: query });
}
```

There is nothing new here, so we'll move on to the remaining filter methods, and in actual fact there is nothing new in any of them. The rest are almost identical to the preceding `clearBrands` and `filterBrands` methods, starting with the `clearCapacity` and `filterCapacity` methods:

```
clearCapacity() {
  if (this.capacity.length) {
    let query = Object.assign({}, this.$route.query);
    delete query.capacity;

    this.$router.push({ query: query });
  }
},
filterCapacity(capacity) {
  let query = Object.assign({}, this.$route.query);
  let split = query.capacity ? query.capacity.split("|") : [];

  if (split.indexOf(parsed) > -1) {
    let index = split.indexOf(parsed);
    split.splice(index, 1);
  } else {
    split.push(parsed);
  }

  if (split.length) {
    let joined = split.join("|");
    query.capacity = joined;
  } else {
    delete query.capacity;
  }
}
```

```
    this.$router.push({ query: query });
  }
```

Next up are the `clearColours` and `filterColour` methods:

```
clearColours() {
  if (this.colours.length) {
    let query = Object.assign({}, this.$route.query);
    delete query.colours;

    this.$router.push({ query: query });
  }
},
filterColour(colour) {
  let query = Object.assign({}, this.$route.query);
  let split = query.colours ? query.colours.split("|") : [];

  if (split.indexOf(colour) > -1) {
    let index = split.indexOf(colour);
    split.splice(index, 1);
  } else {
    split.push(colour);
  }

  if (split.length) {
    let joined = split.join("|");
    query.colours = joined;
  } else {
    delete query.colours;
  }

  this.$router.push({ query: query });
}
```

The following are the `clearOS` and `filterOS` methods:

```
clearOS() {
  if (this.os.length) {
    let query = Object.assign({}, this.$route.query);
    delete query.os;

    this.$router.push({ query: query });
  }
},
filterOS(os) {
  let query = Object.assign({}, this.$route.query);
  let split = query.os ? query.os.split("|") : [];
```

```
    if (split.indexOf(os) > -1) {
      let index = split.indexOf(os);
      split.splice(index, 1);
    } else {
      split.push(os);
    }

    if (split.length) {
      let joined = split.join("|");
      query.os = joined;
    } else {
      delete query.os;
    }

    this.$router.push({ query: query });
  }
}
```

And, finally, the following are the `clearFeatures` and `filterFeature` methods:

```
clearFeatures() {
  if (this.features.length) {
    let query = Object.assign({}, this.$route.query);
    delete query.features;

    this.$router.push({ query: query });
  }
},
filterFeature(feature) {
  let query = Object.assign({}, this.$route.query);
  let split = query.features ? query.features.split("|") : [];

  if (split.indexOf(feature) > -1) {
    let index = split.indexOf(feature);
    split.splice(index, 1);
  } else {
    split.push(feature);
  }

  if (split.length) {
    let joined = split.join("|");
    query.features = joined;
  } else {
    delete query.features;
  }

  this.$router.push({ query: query });
}
```

Styling the filters component

With these methods in place, our `script` block is complete. To finish this component off, we have some basic styles to complement the default Bootstrap styling:

```
<style lang="scss" scoped>
.filter-item {
  margin: 10px 0;
  border: 1px solid #ccc;
  border-radius: 5px;
  padding: 10px;
  text-align: center;
  cursor: pointer;

  &.active {
    font-weight: bold;
    color: #17a2b8;
    border-color: #17a2b8;
  }
}

.slider {
  padding: 35px 0 10px 10px;
}
</style>
```

Adding the filters component to the catalog page

We now have the capability of pushing our filter selections into the URL query string, and reacting to the changes to highlight the selected items using CSS. However, we're not actually displaying our filter component anywhere, let alone using the selected filters in an API call to the server. To tie everything together, we need to make some changes in the `ClientApp/pages/Catalogue.vue` file.

Updating the catalog page template

We need to start by updating the `template` section to look like this:

```
<template>
  <b-container fluid class="page">
    <b-row>
      <b-col cols="3">
        <filters :filters="filters" />
      </b-col>
```

```
      <b-col cols="9">
        <product-list :products="products" />
      </b-col>
    </b-row>
  </b-container>
</template>
```

Instead of a standard `div` element, we're now wrapping this template in a bootstrap `fluid` container which will make use of the full width of the screen. We then define a row with two columns: one for our new filters component, which is allocated a quarter of the screen width, and one for the existing product list component, which is allocated the other three quarters. We already know that the filters component has a required `filters` prop, so we are passing in an object that we will see defined in a moment.

Adding the catalog page filter behavior

In the `script` section, we first need to import both `axios` and our newly created filters component:

```
import axios from "axios";

import Filters from "../components/catalogue/Filters.vue";
```

Next, we update the `components` object to register the filters component so that we can make use of it:

```
components: {
  Filters,
  ProductList
}
```

We then need to update the object returned from the `data()` function to include the `filters` object that we are passing into the filters component:

```
data() {
  return {
    products: [],
    filters: {
      brands: [],
      capacity: [],
      colours: [],
      os: [],
      features: []
    }
  };
}
```


Then we update the `setData` method to set this property following a successful API call:

```
methods: {
  setData(products, filters) {
    this.products = products;
    this.filters = filters;
  }
}
```

Finally, we need to update the `beforeRouteEnter` hook, as well as add an additional `beforeRouteUpdate` hook. We are currently still using `isomorphic-fetch`, and need to replace it with `axios`. We also need to perform an additional API call to fetch the filter items we are passing into the `setData` method. The updated `beforeRouteEnter` hook looks like this:

```
beforeRouteEnter(to, from, next) {
  axios
    .all([
      axios.get("/api/products", { params: to.query }),
      axios.get("/api/filters")
    ])
    .then(
      axios.spread((products, filters) => {
        next(vm => vm.setData(products.data, filters.data));
      })
    );
}
```

This is where we see the benefits of using `axios` over `isomorphic-fetch`. When we have multiple API calls to make at once, like we do here, we can make use of the `axios.all()` function to kick off multiple requests at once. Furthermore, this still returns a single JavaScript promise, which will not resolve until all of the API requests have completed. If you aren't familiar with JavaScript promises, this simply means that the preceding `.then()` function will not be called until both API calls have returned. When they do, we must make use of the `axios.spread()` function to process the results of both API calls simultaneously. If we had a third or fourth API call to make, the `spread` function here would receive an additional parameter for each of them, too. In this instance, we just have the two: a `products` parameter containing the response from the products request, and a `filters` parameter containing the response from the filters request. We pass the `data` property from each of these parameters into our `setData` method, which binds the data onto the component state to be rendered into the UI.



Axios.all is very similar to using **Promise.all**, except the `axios` version is cross-browser compatible out of the box, without the need for a polyfill in older browsers such as IE11.

The final thing to note in this hook is the way we pass the current query parameters into the URL for the products API request. With the `axios.get()` function, we can pass an optional second argument, which must be an object. One of the properties this object can contain is the `params` property, which is yet another object that represents the query string parameters we want to pass to the request. Because of this `params` property, it is a simple case of assigning the `to.query` object, which contains all of the selected filter items that we've been pushing onto the URL from the filters component. This means that, when we select a couple of brands in the brands filter, we end up with a URL that looks like `http://localhost:5000/products?brands=Acme%7CSoylent`.

If we then open up Chrome DevTools, flip over to the network tab, and refresh the page, we can see that an API call is triggered with the following URL: `http://localhost:5000/api/products?brands=Acme%7CSoylent`.

The `vue-router` and `axios` libraries work perfectly with one another to quickly and easily map object-based representations of query string parameters onto the URLs of API requests and client-side routing requests alike. If we were still using `isomorphic-fetch`, we'd have to manually build up the query string instead.

The final piece to put in place is the new `beforeRouteUpdate` hook, which will look like this:

```
beforeRouteUpdate(to, from, next) {
  axios.get("/api/products", { params: to.query }).then(response => {
    this.products = response.data;
    next();
  });
}
```

This is pretty similar to the preceding `beforeRouteEnter` hook, except we only have a single API request to trigger, so there is no need to use the `axios.all()` or `axios.spread()` functions. This hook will get fired every time we make a change to the URL for this page. In other words, every time the user interacts with our filter components, or clicks the reset button, we push a new query object to the URL, which will cause this function to be invoked. Therefore, we can use this function to make a new API request for a list of products that match the current filter selections. As soon as we hit the `this.products = response.data;` line, Vue detects the change and propagates it down through our component tree, refreshing the UI and displaying the new list of products as it goes.

Currently, our API call to the `/api/filters` route will fail, as we have not yet created the controller that will serve it. To fix this, add a new `Features/Filters/Controller.cs` file, add a `private readonly EcommerceContext _db;` property as we've done before, and add the following action method:

```
[HttpGet]
public async Task<IActionResult> Get()
{
    var brands = await _db.Brands
        .Select(x => x.Name)
        .ToListAsync();

    var storage = await _db.Storage
        .Select(x => x.Capacity)
        .ToListAsync();

    var colours = await _db.Colours
        .Select(x => x.Name)
        .ToListAsync();

    var os = await _db.OS
        .Select(x => x.Name)
        .ToListAsync();

    var features = await _db.Features
        .Select(x => x.Name)
        .ToListAsync();

    return Ok(new FiltersListViewModel
    {
        Brands = brands,
        Storage = storage,
        Colours = colours,
        OS = os,
        Features = features
    });
}
```

```
}
```

Nothing too complicated here; we simply query the database for all of the filter options we need to allow our customers to select from, then combine them into a single view model, which we return in an `Ok` result. This view model belongs in the `Features/Filters/FiltersListViewModel.cs` file, and looks like this:

```
namespace ECommerce.Features.Filters
{
    public class FiltersListViewModel
    {
        public IEnumerable<string> Brands { get; set; }
        public IEnumerable<string> Storage { get; set; }
        public IEnumerable<string> Colours { get; set; }
        public IEnumerable<string> OS { get; set; }
        public IEnumerable<string> Features { get; set; }
    }
}
```

Tidying up our existing components

We are very nearly ready to boot the application and test that everything is working as expected, but we just have a few more very minor changes to make. First of all, we need to utilize `axios` on the `ClientApp/pages/Product.vue` API call, then make some very subtle changes to a couple of component templates now that we're using a full-width container on the catalog page.

Open up `ClientApp/pages/Product.vue` and make the following changes:

1. Add an import line for `axios` at the top of the `script` section:

```
import axios from "axios";
```

2. Update the `beforeRouteEnter` hook as follows:

```
beforeRouteEnter(to, from, next) {
  axios.get(`/api/products/${to.params.slug}`).then(response => {
    next(vm => vm.setData(response.data));
  });
}
```

3. Add a `v-if` directive to the product details component declaration in the template section:

```
<template>
  <div class="page">
    <product-details v-if="product" :product="product" />
  </div>
</template>
```

Step 3 isn't crucial, but it does prevent some warnings being thrown in the browser console. Next, open up `ClientApp/components/catalogue/ProductList.vue` and update the template section to look like this:

```
<template>
  <div class="products">
    <b-row class="mt-4">
      <b-col class="mb-4" sm="6" v-for="product in products"
        :key="product.id">
        <b-media class="product">
          

          <h2 class="mt-2" @click="view(product)">{{ product.name }}
          </h2>

          <p class="mt-4 mb-4">
            {{ product.shortDescription }}
          </p>

          <p class="mt-4 mb-4">Prices from £{{ product.price }}</p>

          <b-button variant="primary">Add to cart</b-button>
        </b-media>
      </b-col>
    </b-row>
  </div>
</template>
```

All we've done is remove the title and container, then add a `Prices from £` prefix to the price, seeing as we now have different prices based on which variant is selected.

Finally, open up `ClientApp/components/App.vue` and remove the `<b-container>` component from the template section. After doing so, it should look like this:

```
<template>
  <div class="app">
    <b-navbar toggleable="md" type="dark" variant="dark">
      <b-navbar-toggle target="nav_collapse"></b-navbar-toggle>
      <b-navbar-brand to="/">PhoneShop</b-navbar-brand>
      <b-collapse is-nav id="nav_collapse">
        <b-navbar-nav>
          <b-nav-item to="/products">Products</b-nav-item>
        </b-navbar-nav>
      </b-collapse>
    </b-navbar>

    <transition name="fade" mode="out-in">
      <router-view />
    </transition>
  </div>
</template>
```

We no longer need an app-wide container, seeing as we are specifying the container width on a page-by-page basis. For example, the catalog page is using 100% of the available screen width, whereas the product page is constrained to a maximum width.

Testing the completed filtering logic

We have now finished making all of the changes necessary for catalog filtering, so, if you haven't done so already, start up the application and give everything a test to make sure it's working. Select a few filters and watch the products list update, then refresh the browser and make sure the same results are displayed after the refresh. You can even try selecting a product to go to its details page, then hit the back button in the browser, and you will see your previous filters still intact. Finally, hit the reset button and make sure that the URL is cleared of all filters, and that the product list returns to displaying all products.

Refactoring the filters component

So, at this point, we have a perfectly acceptable solution for filtering our products, including persisting user selections across page refreshes and supporting the sharing of URLs among our customers. However, as it stands, the filters component is very large and responsible for handling every filter that we provide. There is also a vast amount of code duplication, or at the very least, code that is incredibly similar aside from naming conventions or object properties it relies on. Recall the first chapter, when we talked about UI composition; this is a fairly normal situation to be in, as it is no easy task to plan out a perfect component tree before writing any code. We can certainly make our current solution a lot cleaner and more maintainable if we take the time to refactor it, so let's see what we can do.

Highlighting duplication in our existing implementation

Glancing over the current `ClientApp/components/catalogue/Filters.vue` file, most of our filters are based on the concept of selecting multiple values from a list. In the template section, we have quite a lot of repetition where we loop over different lists of filter items and render near enough the same HTML for each. As an example, in the brands filter accordion, we have this:

```
<b-row slot="body">
  <b-col cols="6">
    <div :class="{ 'filter-item': true,
      'active': brands.length === 0 }" @click="clearBrands">All</div>
  </b-col>
  <b-col cols="6" v-for="item in filters.brands" :key="item">
    <div :class="{ 'filter-item': true,
      'active': brands.indexOf(item) > -1 }" @click="filterBrand(item)">{{
item }}</div>
  </b-col>
</b-row>
```

Then, slightly lower down, in the colors filter accordion, we have this:

```
<b-row slot="body">
  <b-col cols="6">
    <div :class="{ 'filter-item': true, 'active': colours.length === 0
      }" @click="clearColours">All</div>
  </b-col>
  <b-col cols="6" v-for="item in filters.colours" :key="item">
    <div :class="{ 'filter-item': true, 'active': colours.indexOf(item)
      > -1 }" @click="filterColour(item)">{{ item }}</div>
  </b-col>
</b-row>
```

The only differences between these two template sections is the filters property we loop over for the items, the property we check for class binding, and the names of the methods we call to manipulate the query object. It would be much nicer if we could have something like this instead:

```
<multi-select-filter slot="body" :items="filters.brands" />
<multi-select-filter slot="body" :items="filters.colours" />
```

We can then reuse this element elsewhere in this component, passing different filter items depending on which filter accordion we are in.

Extracting a common multi-select filter component

The first step in refactoring to extract a new component is to scaffold a new empty component. Create a

ClientApp/components/catalogue/MultiSelectFilter.vue file and give it a template section that looks like this:

```
<template>
  <b-row>
    <b-col cols="6">
      <div :class="{ 'filter-item': true,
        'active': selected.length === 0 }" @click="clear">All</div>
    </b-col>
    <b-col cols="6" v-for="item in items" :key="item">
      <div :class="{ 'filter-item': true, 'active':
        selected.indexOf(item) > -1 }"
        @click="filter(item)">{{ item }}</div>
    </b-col>
  </b-row>
</template>
```


Note how similar this is to both of the template sections we just evaluated previously. All we've done is take a direct copy of that HTML, then tweaked some of the naming conventions to make them more generic. After all, this is going to be a reusable component rather than one specific to a type of filter, so there is no point in having variables with names such as `brands` or `colors`. Instead, we simply have `items` for the list we want to loop over, and `selected` for the computed property that maps to the URL query parameters containing our selected options.

Next, we need to add a `script` block to define the logic of the component:

```
<script>
export default {
  name: "multi-select-filter",
  props: {
    // props omitted for brevity...
  },
  computed: {
    // computed omitted for brevity...
  },
  methods: {
    // methods omitted for brevity...
  }
};
</script>
```

Based on the existing filters component, we know that each of our filters relies on a computed property and a couple of methods to manipulate the query object, so we've stubbed out those sections here ready for our generic implementations. We also know we'll need to pass in some data via props, so we've added that section ready as well.

Let's start with the computed section, whereby in this case I've copied the `brands` computed property directly from the filters component:

```
computed: {
  brands() {
    return this.$route.query.brands || "";
  }
}
```

Again, this is no good in a generic reusable component, as we are accessing a specific property from the query object. We need a way of accessing a different property depending on which filter is being rendered, which of course is another use case for props. But how do we access a JavaScript object property when we don't know what it's called until runtime? It's actually pretty easy:

```
return this.$route.query["brands"] || "";
```

Accessing JavaScript object properties based on their name can also be done using a similar syntax to accessing a specific index in an array—`$route.query["brands"]` is the same as `$route.query.brands`. This enables us to pass a string directly in here to access a different property based on a string prop from the calling component. The updated computed property looks like this:

```
computed: {
  selected() {
    return this.$route.query[this.queryKey] || "";
  }
}
```

We're now expecting a `queryKey` prop, and we've renamed the computed property to make it generic. Next, we need our two methods for manipulating the query object, starting with a generic `clear` method:

```
clear() {
  if (this.selected.length) {
    let query = Object.assign({}, this.$route.query);
    delete query[this.queryKey];

    this.$router.push({ query: query });
  }
}
```

Again, I copied and pasted one of the `clear` methods from the filters component, then generalized it by changing the name, accessing the new `selected` property, and making use of the `queryKey` prop to determine which key to delete from the query object. We can do a very similar thing with the `filter` method:

```
filter(item) {
  let query = Object.assign({}, this.$route.query);
  let split = query[this.queryKey] ?
    query[this.queryKey].split("|") : [];

  if (split.indexOf(item) > -1) {
    let index = split.indexOf(item);
    split.splice(index, 1);
  } else {
    split.push(item);
  }

  if (split.length) {
    let joined = split.join("|");
    query[this.queryKey] = joined;
  } else {

```

```
        delete query[this.queryKey];
    }

    this.$router.push({ query: query });
}
```

Nothing new here, really; we've made the method name and argument name more generic, and made use of the `queryKey` prop again to decide which `query` object properties to manipulate. The last piece to add is the `props` section, which looks like this:

```
props: {
  queryKey: {
    type: String,
    required: true
  },
  items: {
    type: Array,
    required: true
  }
}
```

We're declaring two required props, the `queryKey` string prop which we've already seen used, but also the `items` array which we're now looping over to render the filter items our users can select from. Finally, we can add the following `style` block to complete this component:

```
<style lang="scss" scoped>
.filter-item {
  margin: 10px 0;
  border: 1px solid #ccc;
  border-radius: 5px;
  padding: 10px;
  text-align: center;
  cursor: pointer;

  &.active {
    font-weight: bold;
    color: #17a2b8;
    border-color: #17a2b8;
  }
}
</style>
```

This is taken directly from the filters component, from which we will delete these styles later as they are no longer relevant.

Extracting a common range filter component

This generic component will now work for all of the multi-select filters, drastically reducing how much code we need in the filters component. However, we still have some duplication in the numeric slider-based filters, so let's fix that up first. Create a `ClientApp/components/catalogue/RangeFilter.vue` file with a template section that looks like this:

```
<template>
  <div class="slider">
    <vue-slider
      :value="value"
      :formatter="formatter"
      :min="min"
      :max="max"
      :interval="interval || 1"
      :lazy=true
      width="90%"
      @callback="filter">
    </vue-slider>
  </div>
</template>
```

Essentially, this is another direct copy from the filters component, but we are now basing the props that we pass down into the slider component on the props that we receive into this component. The only exception is the `interval` prop, where we default to 1 if no value is passed to this optional prop.

The script section looks like this:

```
<script>
import vueSlider from "vue-slider-component";

export default {
  name: "range-filter",
  components: {
    vueSlider
  },
  props: {
    // props omitted for brevity...
  },
  computed: {
    // computed omitted for brevity...
  },
  methods: {
    // methods omitted for brevity...
  }
}
```

```
};  
</script>
```

We start by importing the `vue-slider` component, and then scaffold another standard component definition including the slider as a child component. We need quite a few props on this component, but most of them simply map directly to the props required by the slider:

```
props: {  
  min: {  
    type: Number,  
    required: true  
  },  
  max: {  
    type: Number,  
    required: true  
  },  
  interval: {  
    type: Number  
  },  
  formatter: {  
    type: String,  
    required: true  
  },  
  minQueryKey: {  
    type: String,  
    required: true  
  },  
  maxQueryKey: {  
    type: String,  
    required: true  
  }  
}
```

The only exceptions are `minQueryKey` and `maxQueryKey`. You can probably already guess what these are for, but, like we did earlier, we're just passing in the query object keys we want to manipulate inside this component. The difference is that we need two of them, as this is a range slider with two values to push to the URL.

Like we did previously, I've generalized the name of the computed property:

```
computed: {  
  value() {  
    return [  
      this.$route.query[this.minQueryKey] || this.min,  
      this.$route.query[this.maxQueryKey] || this.max  
    ];  
  }  
}
```

```

    }
  }

```

I've also generalized the name of the method and its arguments:

```

methods: {
  filter(values) {
    let query = Object.assign({}, this.$route.query);
    query[this.minQueryKey] = values[0];
    query[this.maxQueryKey] = values[1];

    this.$router.push({ query: query });
  }
}

```

To finish this component, add the following `style` block at the bottom:

```

<style lang="scss" scoped>
.slider {
padding: 35px 0 10px 10px;
}
</style>

```

And there we have it: a generic range filter component that can replace the remainder of the duplicated code inside the filters component.

Rendering the new multi-select and range filter components

The final step is to now update the `ClientApp/components/catalogue/Filters.vue` file to make use of these two new components, and remove any obsolete code.

In the `template` section, we need to make a few changes, starting by updating the brand filter accordion section to look like this:

```

<filter-accordion>
  <span slot="header">Brand</span>
  <multi-select-filter slot="body" query-key="brands"
  :items="filters.brands" />
</filter-accordion>

```

Then, update the price filter accordion to look like this:

```

<filter-accordion>
  <span slot="header">Price</span>

```

```

    <range-filter slot="body"
      :min=0
      :max=1000
      :interval=50
      formatter="{value}"
      min-query-key="minPrice"
      max-query-key="maxPrice"
    />
  </filter-accordion>

```

And, similarly, the screen size filter accordion should look like this:

```

<filter-accordion>
  <span slot="header">Screen size</span>
  <range-filter slot="body"
    :min=0
    :max=7
    formatter="{value} in"
    min-query-key="minScreen"
    max-query-key="maxScreen"
  />
</filter-accordion>

```

Finally, update the capacity, colour, operating system, and feature filter accordions to look like this:

```

<filter-accordion>
  <span slot="header">Capacity</span>
  <multi-select-filter slot="body" query-key="capacity"
    :items="filters.storage" />
</filter-accordion>
<filter-accordion>
  <span slot="header">Colour</span>
  <multi-select-filter slot="body" query-key="colours"
    :items="filters.colours" />
</filter-accordion>
<filter-accordion>
  <span slot="header">Operating system</span>
  <multi-select-filter slot="body" query-key="os"
    :items="filters.os" />
</filter-accordion>
<filter-accordion>
  <span slot="header">Features</span>
  <multi-select-filter slot="body" query-key="features"
    :items="filters.features" />
</filter-accordion>

```

Don't run the application just yet, as we still need to import these new components into the `script` block, as well as remove a whole load of obsolete methods and computed properties. Take a look at how long the `script` section is currently, because it's about to be reduced to this:

```
<script>
import FilterAccordion from "./FilterAccordion.vue";
import MultiSelectFilter from "./MultiSelectFilter.vue";
import RangeFilter from "./RangeFilter.vue";

export default {
  name: "filters",
  props: {
    filters: {
      type: Object,
      required: true
    }
  },
  components: {
    FilterAccordion,
    MultiSelectFilter,
    RangeFilter
  },
  methods: {
    reset() {
      this.$router.push({ query: {} });
    }
  }
};
</script>
```

We no longer need to import or declare the `vue-slider` component as a child, because the new `RangeFilter` component now takes that responsibility. Instead, we import the two new components we just created and declare them inside the `components` block. We've then removed the computed section entirely, and trimmed the methods section right down to a single `reset` method. Finally, we can remove the `style` block entirely, as those styles now belong in their respective child `components` style block.

Testing that everything still works

At this point, our refactoring is complete, and if you start the app again now, it should be fully functioning as before. However, if you open your browser devtools, you'll likely see a Vue warning about a required prop being undefined. This is because there is a very brief time after our page-level API call returns where the filters property still has its default empty array values, which are being passed down into one of the new components we just created. To prevent this warning, open up the `ClientApp/pages/Catalogue.vue` file and modify the `template` section as follows:

```
<template>
  <b-container fluid class="page">
    <b-row>
      <b-col cols="3">
        <filters v-if="filters.brands.length" :filters="filters" />
      </b-col>
      <b-col cols="9">
        <product-list :products="products" />
      </b-col>
    </b-row>
  </b-container>
</template>
```

All we've done is add a `v-if` directive on the filters component, instructing it not to display until we have some items in filters object brands array.

And there we have it: fully functioning catalog filters using nice, clean, reusable components to minimize how much duplication we have.

Client-side sorting

Most product catalogs won't just contain ways of filtering their product list, but also ways of sorting it as well. Luckily for us, the sorting side is a lot simpler than the filtering side. We're going to add a very simple sort component that follows the same theme of pushing the user's selected value into the URL `query` object. We'll then make sure our product list abides by the selected sort property and direction using computed properties.

Building a sort component

Start by creating a `ClientApp/components/catalogue/ProductSort.vue` file, and add a template section with the following content:

```
<template>
  <div>
    <span class="label mr-2">Sort by</span>
    <b-dropdown class="dropdown" right :text="items[selected]">
      <b-dropdown-item v-for="(item, index) in items" :key="index"
        @click="select(index)">{{ item }}</b-dropdown-item>
    </b-dropdown>
  </div>
</template>
```

It should be fairly easy to work out what we're doing here, but essentially we're just displaying a label and a drop-down element side by side. As this is a custom drop-down component rather than a standard select box, we're binding the `text` property to a value from an array named `items`. We're also looping over that same array to render out the drop-down items, and attaching a click handler to fire a `select` method based on the item being selected.

Scaffold out a standard component definition and add the following `data()` function:

```
data() {
  return {
    items: [
      "Cost (Low to high)",
      "Cost (High to low)",
      "Name (A - Z)",
      "Name (Z - A)"
    ]
  };
}
```

The only local component state we need here is the list of items to choose from, which we're keeping simple and only allowing price and name sorting in both ascending and descending directions.

As previously mentioned, we're going to be pushing the selected sort index into the URL query object so that we can maintain the selection after a page change, so add a computed object like this:

```
computed: {
  selected() {
    return this.$route.query.sort || 0;
  }
}
```

```
    }  
  }  
}
```

Now, we could push some kind of string value to make it more obvious what it is at a glance, but, for the purposes of this demo application, we'll keep it simple and just use the numeric index of the selected array item instead. Finally, add a `methods` object with a single `select` method like so:

```
methods: {  
  select(index) {  
    if (index === 0) {  
      let query = Object.assign({}, this.$route.query);  
      delete query.sort;  
  
      this.$router.push({ query: query });  
    } else {  
      let query = Object.assign({}, this.$route.query);  
      query.sort = index;  
  
      this.$router.push({ query: query });  
    }  
  }  
}
```

This is incredibly similar to the filter logic we've already seen, so we don't need to say too much. The `index` argument to this method is the numeric index of the array item the user just selected. All we do is either push it to the current URL `query` object using the same clone technique we've seen before, or, if the index is the first item in the array, we remove the `sort` property altogether, as the default sort values will take effect if we don't provide one.

To complete this component, add the following `style` block:

```
<style lang="scss" scoped>  
  .label,  
  .dropdown {  
    vertical-align: middle;  
  }  
</style>
```

This is a very simple style just to align the label with the middle of the drop-down box next to it.

Adding the sort component to the catalog page

With the sorting component in place, we need to add it to the catalog page to make use of it. Open up the `ClientApp/pages/Catalogue.vue` file and insert the following HTML elements into the `template` section just above where we're already rendering the product list component:

```
<div class="mt-4 clearfix">
  <product-sort class="float-right" />
</div>
```

Next, in the `script` block, we need to add an `import` statement at the top to import our new component:

```
import ProductSort from "../components/catalogue/ProductSort.vue";
```

Next, we add the component declaration to the `components` object:

```
components: {
  Filters,
  ProductSort,
  ProductList
}
```

We now need to create two computed properties: one for listening to the `sort` property on the URL query object, and one to filter the product list based on that `sort` property. Add the following computed object to your component:

```
computed: {
  sort() {
    return this.$route.query.sort || 0;
  },
  sortedProducts() {
    switch (this.sort) {
      case 1:
        return this.products.sort((a, b) => {
          return b.price > a.price;
        });
        break;
      case 2:
        return this.products.sort((a, b) => {
          return a.name > b.name;
        });
        break;
      case 3:
        return this.products.sort((a, b) => {
          return b.name > a.name;
        });
        break;
    }
  }
}
```

```

    });
    break;
  default:
    return this.products.sort((a, b) => {
      return a.price > b.price;
    });
  }
}
}

```

The first of these computed properties is very simple, and exactly like those we've seen before which look for a specific property in the URL `query` object, but have a default value if it does not exist. However, the `sortedProducts` property is a little more involved. We're defining a `switch` statement based on the previously discussed `sort` computed property, then, depending on the value, we use a standard JavaScript `Array.sort()` function to sort the products array and return a new version of it. The default option in this `switch` statement corresponds to the "Cost (High to low)" sort value, and has the following logic:

```

return this.products.sort((a, b) => {
  return a.price > b.price;
});

```

The JavaScript `Array.sort()` function receives two arguments representing the two objects that are about to be compared. In this case, we simply compare the price values, but we could make this comparison based on any of the properties on our `ProductListViewModel`.

To finish wiring things up, we need to bind the product list component onto this newly computed property rather than the original products array:

```
<product-list v-if="products.length" :products="sortedProducts" />
```

After doing so, the final template section will look like this:

```

<template>
  <b-container fluid class="page">
    <b-row>
      <b-col cols="3">
        <filters v-if="products.length" :filters="filters" />
      </b-col>
      <b-col cols="9">
        <div class="clearfix">
          <product-sort />
        </div>
        <product-list v-if="products.length"

```

```
        :products="sortedProducts" />
      </b-col>
    </b-row>
  </b-container>
</template>
```

Run the application again now and check that everything is all working as expected.

Creating a search bar component

To finish off our product catalog, we're going to add a basic search bar to give our users even more control over the product list they're browsing. Let's start by creating a new component named `SearchBar.vue` in the `ClientApp/components/catalogue` directory. The `template` section for this component is very simple:

```
<template>
  <div>
    <b-form-input
      :value="query"
      type="text"
      placeholder="Search..."
      @change="update"
      @keyup.enter.native="search">
    </b-form-input>
  </div>
</template>
```

All we're rendering is a text input field, with a few standard attributes such as a placeholder of "Search...". However, usually, we would use the `v-model` directive on text inputs, so why are we binding the `value` of the text input here instead? In order to transport whatever the user types into this search box to the server for processing, we'll need to push the value into the URL `query` object again. As such, we'll also want the search box value to persist if the user refreshes the page or navigates away to a product details page and back again. If we were to use a standard local component state variable and the `v-model` directive, we would not be able to achieve this goal without persisting to the browser's local storage. Instead, we bind the initial value of the text input to a property called `query`, which, as we'll see shortly is a computed property that is bound to a URL query object property. We then attach the `@change` event handler to the input, which calls a method called `update`—we'll see what this does in a moment. Finally, we also attach a second event handler, which fires on the `keyup` event of the *Enter* keyboard key, and calls a `search` method.

The script block of this component looks like this:

```
<script>
export default {
  name: "search-bar",
  data() {
    return {
      value: ""
    };
  },
  computed: {
    query() {
      return this.$route.query.q || "";
    }
  },
  methods: {
    update(newVal) {
      this.value = newVal;
    },
    search() {
      let query = Object.assign({}, this.$route.query);

      if (this.value.trim()) {
        query.q = this.value;
      } else {
        delete query.q;
      }

      this.$router.push({ query: query });
    }
  }
};
</script>
```

We're defining a local component state variable called `value` and initializing it with an empty string, as well as the `query` computed property which should look pretty familiar to you by now. The `update` method is the one which is fired every time the input changes value, and receives the new value as an argument which we simply assign to our local `value` variable. Finally, when the user hits the *Enter* key, the `search` method is invoked. Here, we do our usual trick of cloning the current URL query object before either pushing our search query onto it, or removing it if the text input is currently empty.

There are no styles for this component, so all we need to do now is import it and render it into the `ClientApp/pages/Catalogue.vue` file. Look at the `template` section and find the part that looks like this:

```
<div class="clearfix">
  <product-sort />
</div>
<product-list :products="sortedProducts" />
```

Now, update it to look like this:

```
<div class="mt-4 flex">
  <search-bar class="search" />
  <product-sort class="ml-4" />
</div>
<product-list :products="sortedProducts" />
```

It is important to note the class changes on the wrapping `div` element around the search bar and sort components. In a moment, we're going to add some CSS styles that display these components nicely side by side, with the search bar expanding to fill any remaining space.

In the `script` section, import the new `SearchBar` component and register it in the `components` object:

```
import SearchBar from "../components/catalogue/SearchBar.vue";

components: {
  Filters,
  SearchBar,
  ProductSort,
  ProductList
}
```

Finally, add the following flexbox-based CSS styles to position things nicely:

```
<style lang="scss" scoped>
.flex {
  display: flex;
  flex-direction: row;

  .search {
    flex: 1;
  }
}
</style>
```


At this point, you can run the application, type something in the search bar, and press the *Enter* key. Note how the URL changes, and if you refresh the page, the search bar will populate from your previously entered value in the query object. Similarly, if you hit the *reset* button, the search bar will be cleared. This is great, but as yet we aren't honoring the search query in our API calls on the server. Open the `Features/Products/Controller.cs` file and update the `Find` action declaration to include an additional `q` string parameter:

```
public async Task<IActionResult> Find(string q, string brands, int?
minPrice, int? maxPrice, int? minScreen, int? maxScreen, string capacity,
string colours, string os, string features)
```

Next, create a new string variable at the top of the method, above the existing filter variables:

```
var Query = $"{q?.ToLower()}%";
```

We'll see what this is for in a moment, but the important part is the new null-conditional check, which prevents this line throwing an error if we don't provide a search term. Finally, add an additional LINQ `Where` clause to the top of the products query:

```
.Where(x =>
    string.IsNullOrEmpty(q) ||
    (
        EF.Functions.Like(x.Name.ToLower(), Query) ||
        EF.Functions.Like(x.ShortDescription.ToLower(), Query) ||
        EF.Functions.Like(x.Description.ToLower(), Query) ||
        EF.Functions.Like(x.Brand.Name.ToLower(), Query) ||
        EF.Functions.Like(x.OS.Name.ToLower(), Query) ||
        x.ProductFeatures.Any(f =>
            EF.Functions.Like(f.Feature.Name.ToLower(), Query)
        )
    )
)
```

Here, we check if the search query is empty, and if it is then we ignore the rest of this expression and move on. If it isn't empty, we use the variable we just created as an argument to the `EF.Functions.Like()` method. As the name suggests, this is one of the new features added in EF Core 2.0 and it allows us to use native SQL `like` statements directly in our data access code. In a native SQL query, we can use `%` characters either before, after, or before and after a piece of text to see if a column starts with, ends with, or contains the text we are looking for. This concept is exactly the same in the `EF.Functions.Like()` method, which is why we've declared our search query variable in the way that we have:

```
$"%{q?.ToLower()}%";
```

All we want to know is if either the product name, short or long description, brand name, operating system name, or any of its feature names contain the text that the user has searched for.

If you rerun the application now, everything should be working. Filtering, sorting, and searching should all be controlled via the URL query object.

Triggering API requests using watchers

The last thing we are going to do with our product catalog is configure the search bar to react to the user's input in real time, without waiting for them to press the *Enter* key. To achieve this, we only need to make a couple of quick and simple changes, but it provides a nice UX for our users. Open the

`ClientApp/components/catalogue/SearchBar.vue` file that we just created. The first thing we need to do is change the `@change` event handler on the input to an `@input` event handler instead:

```
<b-form-input
  :value="query"
  type="text"
  placeholder="Search..."
  @input="update"
  @keyup.enter.native="search">
</b-form-input>
```

The `@change` event handler only fires as the user navigates away from the input, or when they hit the *Enter* key while it's focused. Either way, we need our `update` method to be fired as soon as the user enters some text into the box so that we have access to that text straight away. The `@input` event handler fulfills this need as it fires on every keypress.

The only other change we need to add is a `watch` object at the bottom of the component:

```
watch: {
  value(newValue) {
    this.search();
  }
}
```

Recall Chapter 1, *Understanding the Fundamentals*, any function defined inside the `watch` object uses the function name as a convention for the `data` property it is *watching* for changes on. In this instance, with a function named `value`, we are watching the `value` data property, which now gets updated immediately after a user enters a value in the search box. We already have the `search` method defined, which takes the search value and pushes it to the URL query object, which in turn fires the API request to the server. Therefore, all we need to do when the search box value changes is to call the `search` method and leave it to do its thing.

Debouncing API requests to limit how often they fire

As you can probably imagine already, with the current implementation, we are going to be making a vast amount of requests to the server, seeing as every key press invokes the search method. A better approach would be to `debounce` the requests to limit how often they actually fire. To do so, we'll install an npm package called **lodash**, which is a JavaScript utility library that provides some incredibly useful functions to save us the time of writing them ourselves.

Open a Terminal and run the following command:

```
yarn add lodash
```

Next, still in the search bar component, import `lodash` at the top of the script block:

```
import _ from "lodash";
```

Then, finally, update the existing `search` method to look like this:

```
search: _.debounce(function() {
  let query = Object.assign({}, this.$route.query);

  if (this.value.trim()) {
    query.q = this.value;
  } else {
```

```
        delete query.q;
    }

    this.$router.push({ query: query });
}, 500)
```

This may look a little strange at first, but all we've done is wrap our existing function with the `debounce` function from `lodash`, specifying a delay of 500 milliseconds. Now, when the user first types something into the search box, there will be a delay of half a second before the API request triggers, and, if they keep typing beyond that period, the API request will only ever fire twice every second at most.

Summary

This has been by far the longest and most complex chapter yet, and we've covered an awful lot of ground. Let's have a quick recap before moving on to building a shopping cart.

We started out by installing and configuring some additional dependencies in order to add some styles to the existing application. We also looked at the difference between fetching data before and after navigation occurs, before refactoring our current pages to pre-fetch their data. To round off the UX improvements, we added a page-level loading indicator between page changes, as well as a nice fade transition to make things smoother.

We then dropped back to the server side of the application and extended our existing data model to include a number of additional entities and model properties to support our needs of filtering and sorting the product list. We also created a whole load of fake product seed data to start us off.

Sticking to the server side initially, we added support for filtering the product list based on a number of different product attributes such as brand, color, and price. Moving over to the client side, we built a custom accordion component to house our individual filters, before creating the actual filters themselves along with all of the logic that goes with them. We talked about the different ways we could store the selected filter items, but settled on pushing everything into the `URL query` object for the benefits it provides of state persistence and accessibility for our users wanting to share their search results with others. We saw a lot of duplication with our initial approach, so we looked at how we could refactor and extract common functionality into new and reusable components, drastically reducing the complexity and the amount of code in some of our existing components.

Next, we added simple sort and search bar components to finish off the standard functionality that we would expect from most e-commerce product catalogs. We carried on the existing trend and made the decision to push the sort field and search query text into the URL query object as well, meaning the entire catalog state can be reset with the click of a single button.

Finally, we added a Vue `watch` function to configure our search bar to respond to the user's input in real time, triggering API requests on every keypress. We decided that this wasn't the best approach, and installed the `lodash` utility library so that we could make use of its `debounce` function in order to limit the API requests to one every half a second.

In the next chapter, we'll spend some time finishing off the product details page with an image gallery and variant selection drop-down menus. This will prepare us to move on to the main focus of the chapter—building out a fully featured shopping cart for our users to begin the process of placing an order.

6 Building a Shopping Cart

With the product catalog complete, it's time to build a shopping cart so that our customers can store the products they intend to purchase. There are a number of ways we can decide to implement a shopping cart, so we'll start by evaluating these options before we start building one. In summary, the topics we'll cover in this chapter are as follows:

- Installing and configuring Vuex for client-side state management
- Vuex actions, mutations, and getters
- Binding component state to a centralized Vuex store
- Creating custom filters
- Persisting state to local storage
- Providing feedback with toast messages

Evaluating our options

Before we can start building a shopping cart, we need to decide how we're going to do it. We have a number of options available to us, each with their own respective pros and cons. We need a way of storing selected products somewhere until the user is ready to complete their purchase and we can persist their order in the database. Without including any additional technologies in our stack, we have three main options to choose from—one of which is client-side only, and the other two require API calls to the server.

Persisting to the database

Persisting to the database is probably the most obvious option, but is also the most complicated. On the face of it all, what we need to do is add a couple of new tables for shopping carts and associated cart items, and provide the API endpoints necessary to store and retrieve the data in those tables. However, generally speaking, most e-commerce websites do not force users to create an account before adding items to their cart, and as such, how do we identify which cart in the database we want to retrieve for a specific user?

This is by no means an insurmountable problem, but it does require adding additional complexity to our application, and so we should try and avoid it unless there is a significant benefit to going down this route. The solution would be to provide unauthenticated users with an anonymous user ID, most likely a GUID to ensure uniqueness, and link the shopping cart to that ID. We'd then need to return this ID to the client so that it can use it later to identify the shopping cart that belongs to the user who stored it. We could either use a cookie or local storage to store this ID, but as it stands we're not using cookies for any other purpose, so it would make more sense to use local storage instead.

The main benefit of storing shopping cart data in the database is for reporting purposes. If we only store cart data on the client side of our application, we have no knowledge of it and as such cannot report on it. If this is an important factor for your application, then it is certainly worth exploring this approach in more detail for your own use case. However, for the sample application in this book, we have no requirement for reporting, and as such the overhead of managing anonymous user IDs and making additional API requests is not worth it.

Persisting to session state

The next approach we could follow still requires a round trip to the server, but does not require any changes to our current database structure. Rather than pushing shopping carts into database tables, we could store them in session state instead. This approach does reduce the complexity of the previous option, as we have no need to try and provide IDs for unauthenticated users. Session state is automatically associated with a cookie, which gets stored on the user's computer by the web browser.

However, this approach forces us to introduce session state and cookies to our application, which breaks the stateless nature of our current API setup. The configuration involved with setting up session state is also more tricky than you would expect. You are likely to find problems with AJAX requests having different session ID values for subsequent requests, meaning previously stored session data is lost. Combine this with the overhead of needing to perform API requests to store and retrieve shopping cart data, and this approach simply isn't worth the additional complexity and setup time. Session state is also time limited, meaning it will only exist for a short period of time—usually around 30 minutes or so. It would be pretty frustrating for our customers to spend hours browsing our catalog and filling their shopping cart, only to have their items discarded because their session timed out before they could finish the checkout process.

Persisting to local storage

Finally, we could avoid involving the server at all and simply persist shopping cart data into local storage within the user's browser. This approach is by far the simplest to implement, and only has a couple of minor downsides. Firstly, users are able to clear their browser cache, which could potentially delete their cart data from local storage depending on the settings they choose. However, the same issue applies to both of the previously discussed options, which use either cookies or local storage for one reason or another. Secondly, we cannot report on the shopping cart data unless users actually proceed to the checkout and place an order. As reporting is not a concern for this application, we'll keep things simple and make use of local storage.

Finishing the product details page

Before we can let users add products to their shopping carts, we have some work to do to finish off the product details page. For a start, we have an `Add to cart` button, which doesn't do anything, and we have no way of determining which product variant to add to the cart even when it does.

Our products also have an array of associated images, which we aren't doing anything with, so let's start off by displaying those on the page before creating a basic image gallery component. The first step is to modify the existing controller action to return a view model for the same reasons as with the product list action. This means we'll need the following new `ProductDetailsViewModel.cs` class in the `Features/Products` folder:

```
namespace ECommerce.Features.Products
{
    public class ProductDetailsViewModel
```



```
{
    public int Id { get; set; }
    public string Slug { get; set; }
    public string Name { get; set; }
    public string ShortDescription { get; set; }
    public string Thumbnail { get; set; }
    public IEnumerable<string> Images { get; set; }
    public IEnumerable<string> Features { get; set; }
    public decimal Price { get; set; }
    public string Description { get; set; }
    public IEnumerable<SelectListItem> Colours { get; set; }
    public IEnumerable<SelectListItem> Storage { get; set; }
    public IEnumerable<ProductVariantViewModel> Variants { get; set; }
}
}
```

As you may have spotted already, this class depends on another new class, `ProductVariantViewModel.cs`, which belongs in the same folder:

```
namespace ECommerce.Features.Products
{
    public class ProductVariantViewModel
    {
        public int ProductId { get; set; }
        public string Name { get; set; }
        public string Thumbnail { get; set; }
        public int ColourId { get; set; }
        public string Colour { get; set; }
        public int StorageId { get; set; }
        public string Capacity { get; set; }
        public decimal Price { get; set; }
    }
}
```

We'll see why we need this additional view model shortly. With these in place, we can update the query in the Get action of the `Features/Products/Controller.cs` controller:

```
var product = await _db.Products.Select(x => new ProductDetailsViewModel
{
    Id = x.Id,
    Slug = x.Slug,
    Name = x.Name,
    ShortDescription = x.ShortDescription,
    Description = x.Description,
    Price = x.ProductVariants.OrderBy(v => v.Price).First().Price,
    Thumbnail = x.Thumbnail,
    Images = x.Images.Select(i => i.Url),
```

```
Features = x.ProductFeatures.Select(f => f.Feature.Name),
Colours = x.ProductVariants.Select(v => new SelectListItem
{
    Value = v.ColourId.ToString(),
    Text = v.Colour.Name
}).Distinct(),
Storage = x.ProductVariants.Select(v => new SelectListItem
{
    Value = v.StorageId.ToString(),
    Text = v.Storage.Capacity.ToString() + "GB"
}).Distinct(),
Variants = x.ProductVariants.Select(v => new ProductVariantViewModel
{
    ProductId = x.Id,
    Name = x.Name,
    Thumbnail = x.Thumbnail,
    ColourId = v.ColourId,
    Colour = v.Colour.Name,
    StorageId = v.StorageId,
    Capacity = $"{v.Storage.Capacity}GB",
    Price = v.Price
})
})
).FirstOrDefaultAsync(x => x.Slug == slug);
```

With this in place, we can now make some changes to the `template` section of the `ClientApp/components/product/Details.vue` component. First of all, we need a way of returning to the list of products so that it doesn't reset any filters we may have applied. Navigating using the browser's `back` button does exactly this, but there's also a `Products` link in the navbar, which will reset the filters if it's used. To encourage our users not to use the link from this page, we'll add the following button as the first element within the existing `b-container` element (above the existing `b-media` element):

```
<b-button variant="outline-secondary" @click.prevent="back">
  <i class="fas fa-arrow-left"></i>
  Back to results
</b-button>
```

This is a simple use of the `b-button` component from `Bootstrap-Vue`, with an `@click.prevent` event handler attached, which prevents the default action and instead invokes the `back` method, which we'll define in just a moment. The only other thing to note is another use of the `Font Awesome` icons library, specifically the `fa-arrow-left` icon.

Next, we're going to replace the existing `b-media` element with a `b-row` element and two nested `b-col` elements:

```
<b-row class="pt-4">
  <b-col cols="5">
    // first column
  </b-col>
  <b-col cols="7">
    // second column
  </b-col>
</b-row>
```

Inside the first column, we're going to display the product images in a grid layout, where the first image fills the full width of this column, and the remaining images are displayed underneath it in rows of three:

```
<b-row>
  <b-col class="mb-2" v-for="(image, index) in product.images"
    :key="index" :cols="index === 0 ? 12 : 4">
    
  </b-col>
</b-row>
```

We start off with another `b-row` element, then loop over all of the images in the `product.images` array and render a `b-col` element for each one. Rather than specifying a static value for the `cols` prop, we use the `v-bind` directive to specify a width of 12 columns if this is the first image in the array, and a width of four columns for the rest. It is then a simple case of rendering a standard `img` element and binding the `src` attribute to the image URL in the array, as well as attaching an `@click` event handler to invoke an `openGallery` method, which we'll define later.

The second column holds the remainder of the content from the `b-media` element we're replacing, with the exception of some minor formatting changes and the addition of a list of the features this product has:

```
<h2>{{ product.name }}</h2>
<p class="mt-4 mb-4">
  {{ product.shortDescription }}
</p>
<h5>Features</h5>
<ul>
  <li v-for="feature in product.features" :key="feature">{{ feature }}</li>
</ul>
```

```
<p class="mt-4 mb-4">€{{ product.price }}</p>
<b-button variant="primary">Add to cart</b-button>
```

In order to make sure everything lines up properly, we need to wrap the final `Product` details heading and paragraph in another `b-row` and `b-col` element:

```
<b-row>
  <b-col cols="12">
    <h3 class="mt-4">Product details</h3>
    <p class="mt-4 mb-4">
      {{ product.description }}
    </p>
  </b-col>
</b-row>
```

Finally, we know that we want to include some sort of image gallery, which will display our product images at full size, so we'll add the following `transition` element to the bottom of the `template` section, right before the closing `b-container` tag:

```
<transition name="fade" mode="out-in">
  <gallery v-if="open" :images="product.images" :initial="index"
    @close="open = false" />
</transition>
```

The gallery component does not yet exist, and as such we'll need to create it and reference it before we can run the application. However, when building a component-based UI in Vue, sometimes it helps to define the "interface" of new components before actually creating them. In other words, we mock out how we expect to consume the component from a parent, as we've just done here.

We know we only want to display this component after a user clicks on one of the images, so we conditionally display it based on an `open` property that we're yet to define. We also know we'll need to close the gallery at some point, so we're binding an `@close` event handler, which will set the same `open` property back to `false`. A gallery is not much use without a list of images to display, so we're binding an `images` prop to the `product.images` array, as well as an `initial` prop to the index of the image array we wish to display when the gallery opens. Finally, we wrap this component in a `transition` component to make it fade nicely in and out as we toggle the `open` property.

In the `script` section, we need to define the methods and data properties we've just made use of in the template. Let's start with the `data` function, which looks like this:

```
data() {
  return {
    open: false,
    index: 0
  };
}
```

The `open` property, which toggles the rendering of the gallery, is initially set to `false` and we default the `index` array to display 0. Next, we need the `methods` object with the following two functions defined:

```
methods: {
  back() {
    this.$router.go(-1);
  },
  openGallery(index) {
    this.index = index;
    this.open = true;
  }
}
```

The `back` method simply invokes the `go` function on the `Vue-Router` object, instructing it to navigate backwards by one step. We can pass any number to this function, either positive or negative, to move backwards and forwards through the browser's URL history. For example, if we were to pass `-2` instead of `-1`, we would navigate two steps backwards, whereas passing `2` would navigate two steps forwards.

The `openGallery` method receives the `index` of the `image` array the user just clicked on and sets it to the `index` property, which will be passed into the gallery component via `props`. It also sets the `open` property to `true` in order to actually display the gallery.

To finish our changes on this component, we just need to add a simple `style` section in order to make it obvious to our users that the images are clickable, by giving them a pointer when they hover over them:

```
<style lang="scss" scoped>
img {
  cursor: pointer;
}
</style>
```

Creating the gallery component

Now that we know how the gallery component will be consumed, we can start to build it. Create a new `ClientApp/components/product/Gallery.vue` component and add a template section like this:

```
<template>
  <div class="gallery" @click="close">
    <span @click.stop="prev">
      <i class="fas fa-chevron-circle-left fa-3x prev" />
    </span>

    <div class="slide" @click.stop="next">
      <transition name="fade" mode="out-in">
        
      </transition>
    </div>

    <span @click.stop="next">
      <i class="fas fa-chevron-circle-right fa-3x next" />
    </span>
  </div>
</template>
```

First, we declare a wrapping `div` element, which we will style with CSS to cover the entire screen with a semi-transparent black background. We're also attaching an `@click` event handler to this element to invoke a `close` method that is yet to be defined. This will ultimately enable our users to close the gallery by clicking anywhere other than the image on display and the two buttons that will rotate between the images in the gallery.

Next, we render a `span` element, which contains a Font Awesome `fa-chevron-circle-left` icon to act as the button to go to the previous image in the gallery. Due to an issue where `click` events do not get fired from the `i` element directly, the `span` element is only necessary so that we can attach a `@click.stop` event handler to it to invoke the `prev` method, which we'll define shortly.

The main element inside the gallery component is a `div` element with a class of `slide`, which we'll use to horizontally and vertically center the nested `img` element on any screen size. We will also attach a `@click.stop` event handler to invoke the `next` method, which is still to be defined. We're also using the `v-bind` directive to bind the `src`, `key`, and `alt` attributes to the URL of the image at the currently displayed `index` of the array. We need the `key` attribute so that when we change the active gallery image, Vue can detect that the element inside the wrapping `transition` component has changed, and the associated *fade* animation is applied.

Finally, we have another `span` element housing a Font Awesome icon. However, this time it is the `fa-chevron-circle-right` icon that will invoke the `next` method, rather than the `prev` method as in the previous icon. Notice how on both of these left and right icons, and on the main `div class="slide"` element itself, we use the `.stop` modifier on the `@click` event handler declarations. We need this modifier to stop the event from propagating up to the parent `div class="gallery"` element, which also has a `@click` event handler that is used to close the gallery. Without it, even though the image would change as we want it to, the click would also cause the parent event handler to fire and call the `close` method, which is not what we want to happen.

Moving on to the `script` section of the component, start by defining a standard component definition with `props`, `data`, and `methods` sections, as well as a `created` life cycle hook. The `props` object contains the two items we defined earlier when looking at how we'd render this component, and ultimately looks like this:

```
props: {
  images: {
    type: Array,
    required: true
  },
  initial: {
    type: Number,
    required: true
  }
}
```

The data function is very simple, and the returned object only has a single `index` property, which is used to instruct the gallery as to which image in the array to display:

```
data() {
  return {
    index: 0
  };
}
```

The created lifecycle hook is used to copy the `initial prop` value onto the local `index` data property in order to display a specific image as the gallery is opened, as well as invoke the `window.addEventListener` method to add a global `keyup` event listener:

```
created() {
  this.index = this.initial;
  window.addEventListener("keyup", this.onKeyUp);
}
```

As we're adding a global event listener here, we need to make sure that we remove it again to avoid memory leaks. To do so, we'll use another life cycle hook, which is fired as the component is removed from the DOM. The `beforeDestroy` life cycle hook needs to look like this:

```
beforeDestroy() {
  window.removeEventListener("keyup", this.onKeyUp);
}
```

And finally, the `methods` object contains the `onKeyUp`, `next`, `prev`, and `close` methods, which we've already attached to event handlers at one point or another:

```
methods: {
  onKeyUp(event) {
    switch (event.keyCode) {
      case 27:
        this.close();
        break;
      case 37:
        this.prev();
        break;
      case 39:
        this.next();
        break;
    }
  },
  next() {
    if (this.index < this.images.length - 1) {
      this.index++;
    }
  }
}
```



```
    } else {
      this.index = 0;
    }
  },
  prev() {
    if (this.index > 0) {
      this.index--;
    } else {
      this.index = this.images.length - 1;
    }
  },
  close() {
    this.$emit("close");
  }
}
```

The `onKeyUp` method is what we invoke from the global event listener that we registered as soon as the gallery is displayed, and is used to enable our users to move between images or close the gallery altogether using their keyboards. We use the `event.keyCode` property in a switch statement to detect the **Escape**, **left**, and **right** arrow key presses, and invoke the `close`, `prev`, and `next` methods, respectively. This is all very well and good, but why can't we use the `v-on` directive as we've done before for attaching event listeners?

The `keyup` event is not one that is usually fired from the HTML elements we're using in this component, the `div` and `img` elements. As such, we don't actually have anything to add the `v-on` directive to. There are ways to trick them into firing the `keyup` event, such as adding a `tabindex` property. However, this still wouldn't work as the event wouldn't fire until the element in question had received focus from the browser. The only way to guarantee the element would get that focus is to make use of the `created` life cycle hook to force it as soon as the component is rendered. This is a lot of unnecessary hacking around, so instead, we'll simply attach and remove a global event listener, as we've done here.

The `next` method compares the current `index` value with the length of the `images` array and either increments the value to move to the next image, or resets it back to 0 if there are no more images to display. These changes to the `index` property cause the `image src` attribute binding value to change, which in turn changes the image currently displayed in the gallery. The `back` method is identical to the `next` method, but in reverse.

The `close` method is used to emit the custom `close` event we are listening for in the parent component in order to hide the gallery.

To complete the gallery component, we need to add a fair few CSS styles to get things displaying nicely. Add a `style` section with the following styles:

```
<style lang="scss" scoped>
.gallery {
  position: fixed;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  background: rgba(0, 0, 0, 0.8);
  z-index: 15000;

  .prev,
  .next {
    position: absolute;
    color: white;
    cursor: pointer;
  }

  .prev,
  .next {
    top: 50%;
    transform: translateY(-50%);
  }

  .prev {
    left: 20px;
  }

  .next {
    right: 20px;
  }

  .slide {
    position: relative;
    width: 750px;
    max-width: 90%;
    height: 422px;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    overflow: hidden;

    img {
      position: relative;
      top: 50%;
      transform: translateY(-50%);
    }
  }
}
```

```
    }  
  }  
}</style>
```

Again, this isn't a book about CSS so I won't go into detail here, and most of these are fairly self-explanatory. However, a nice way of vertically aligning content is with the following three lines of CSS, which I've used a few times before:

```
position: relative;  
top: 50%;  
transform: translateY(-50%);
```

This essentially positions the top of the element in question to 50% of the screen height away from the top of the screen, then uses the `transform` property to shift the whole element up by 50% of its height—perfect vertical alignment in three lines of CSS!

With this, our new gallery component is complete. The last thing we need to do is go back into the `ClientApp/components/product/Details.vue` file and import it at the top of the script section:

```
import Gallery from "../Gallery.vue";
```

Then, we need to add a `components` object to declare it for use:

```
components: {  
  Gallery  
}
```

And we are done. Restart the application now and click through in to the details page for any of the products to see the gallery in action.

Adding variants to the product details component

Before we can allow users to add products to their shopping cart, we need a way of telling which variant they are interested in buying. The actual price of a product is dictated by the combination of the color and storage capacity options, so we need to add these options to the product details component.

We only have a couple of easy changes to make, so open up the `ClientApp/components/product/Details.vue` file and let's get started. In the `template` section, right beneath the `Features` heading in the `ul` element, we need to add the following:

```
<h5>Variants</h5>
<b-form-group label="Colour">
  <b-form-select :options="product.colours" v-model="colour" />
</b-form-group>

<b-form-group label="Capacity">
  <b-form-select :options="product.storage" v-model="capacity" />
</b-form-group>
```

All we're doing here is rendering two standard Bootstrap *form groups* with a label and `select` element in each. As part of our product details query on the server side, we're already returning a list of `SelectListItem` objects for both the available colors and capacity options. As such, we can bind the `options` prop of each `b-form-select` component to these lists, and the properties from our model are already compatible with what the `select` component expects. However, in each case, we need to provide a `data` property to bind the selected value of the dropdown to. In this case, we need to add the `colour` and `capacity` properties to our local component state:

```
data() {
  return {
    open: false,
    index: 0,
    colour: null,
    capacity: null
  };
}
```

These won't do much good with `null` as their value, so we'll default each one to the first item in their respective options list using a `created` life cycle hook:

```
created() {
  this.colour = this.product.colours[0].value;
  this.capacity = this.product.storage[0].value;
}
```

The variant dropdowns will now be working just fine, but changing the values will have no effect on anything at all. To change this, we're going to add a `computed` property, which tracks which variant from the `product.variants` array matches the selected values. We can then use this `computed` property to show a dynamic price based on the selected options, as well as determine which variant to push to the shopping cart when the user clicks the `Add to cart` button. This `computed` property looks like this:

```
computed: {
  variant() {
    return this.product.variants.find(
      v => v.colourId == this.colour && v.storageId == this.capacity
    );
  }
}
```

We simply invoke the `Array.find` function using fat arrow syntax to look for and return a variant where the `colourId` property matches our selected color value and the `storageId` property matches our selected capacity value. With this `computed` property in place, we can now make use of it in the `template` section by displaying the variant price, rather than the minimum price from the `product` object that we've been using so far:

```
<p class="mt-4 mb-4">
  <b>Price:</b> £{{ variant.price }}
</p>
```

I also added the `b` element to make it stand out a little more, but this is all that's required, and we are now ready to add products to a shopping cart.

Introduction to Vuex

We talked about **state** back in [Chapter 1, *Understanding the Fundamentals*](#), and until now we've only had the need for local component state, which we sometimes pass to related components via **props**. However, with the introduction of our shopping cart feature, we will need to display the same state in multiple components that have no direct relationship with one another. Think about almost any e-commerce website you've ever purchased something from; typically, you'll have a dedicated page for your shopping cart items, as well as some kind of cart summary widget, which will be displayed in the sidebar of every page. These two locations have no direct connection to one another without traversing all the way up the component tree to the very top, to the root level `App` component.

Do we really need to store our shopping cart state in the `App` component, then pass it all the way down to those components that care about it using props? Think about how tedious this would be, and how much of a maintenance nightmare it would be each time we refactor the component tree. We'd need to make sure that we add the same props and event handlers to every component to ensure we don't break the chain. This is where **Vuex** comes in.

What is Vuex?

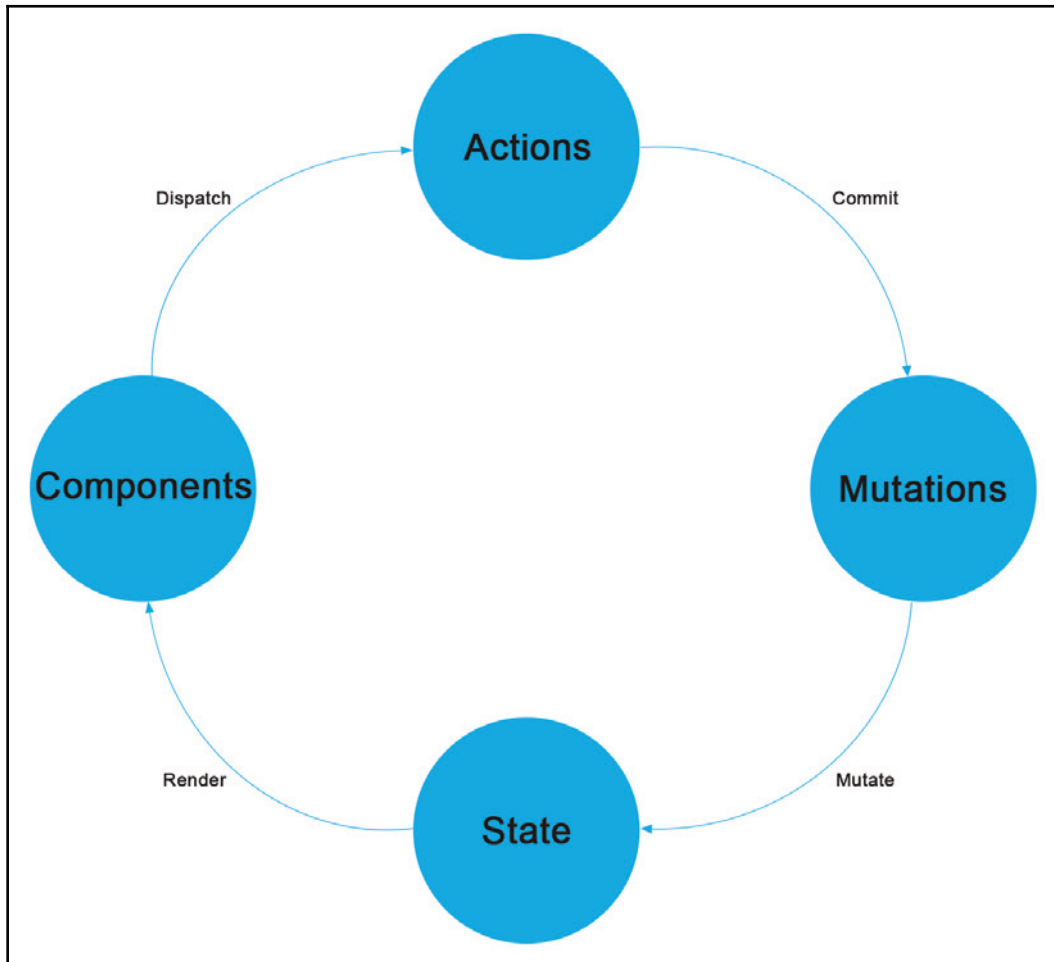
According to the official Vuex documentation (<https://vuex.vuejs.org/>), "Vuex is a state management pattern + library for Vue.js applications. It serves as a centralized store for all the components in an application, with rules ensuring that the state can only be mutated in a predictable fashion."

As with all of the official Vue.js documentation, this is already a great explanation of what Vuex is, but let's break it down to make sure it makes perfect sense, as follows:

- Vuex is a **centralized store**: This means that we have one central place to put state, and every component in the application has direct access to it if they need it. There's no need to pass it around using props, and no need to attach event handlers to listen to changes on it. We can bind data directly to the store, and every time the underlying state changes, the UI will be kept in sync just as if the state were local to the component.
- It has **rules** that ensure that state changes, or **mutations**, are predictable: This means that we should always know exactly how and why a change in state occurred. Vuex enforces the concept of a one-way flow of data, where the only way of manipulating the state of the store is using a mutation. These mutations are tracked using the official devtools extension that we installed earlier, so we can always tell how and why the data is changing.

How does Vuex work?

Vuex contains four main concepts: **actions**, **mutations**, **state**, and **getters**. We've already discussed state at length, but actions, mutations, and getters will all be new concepts to you. We'll come back to getters shortly, but along with state and components, which we already know about, actions and mutations feed directly into the one-way data flow paradigm that we touched on earlier. Take a look at the following diagram:



Actions are functions that can be *dispatched* from our existing **components**, which then *commit* mutations. **Mutations** are also functions, but have the sole purpose of *mutating* the state, and are the only way of doing so. When the **state** changes, any component that is bound to the store will be *rendered* again, which completes the cycle of data flow.

At first glance, it may seem unnecessary to have both actions and mutations. However, there is a very big difference between the two, which gives both a very specific purpose within the application.

Mutations

A mutation is like an **event**, whereby we define a **handler** function to perform state modifications when we receive a specific **type** of mutation. They should not contain any kind of logic or update more than a single piece of state to ensure there is no ambiguity in our history of state changes. Mutation handler functions are the only way to manipulate state in Vuex, and the only way to invoke these handler functions is to commit a mutation of the appropriate type—they cannot be invoked directly. Mutations are also always run synchronously, so cannot contain any kind of asynchronous operation, such as API calls.

Actions

Actions can contain asynchronous operations, making them the perfect place for interacting with our backend API if we need them to. They can also contain logic to decide which mutations, if any, they need to commit. You can think of actions as functions that can orchestrate multiple mutations in a single operation. This will make more sense when we look at an example shortly!

Getters

Getters are like computed properties for store state, and can be used at times when we need to derive state based on the state within the store. This helps eliminate duplication, where the same derived state is needed in multiple components. As an example, we might have an array of shopping cart items, each with their own price and quantity values. To work out the total shopping cart cost, we need to iterate over each cart item and multiply the price by the quantity, the sum of which gives us the total cost. Rather than store this value in the state and recalculate it each time the cart items change, we can define a getter function that computes it for us. We can then data bind onto this function, and it will behave exactly like a standard component-level computed property, whereby the UI will update every time Vue detects a change to the computed value of the function.

Putting it all together

Let's look at an example to help explain the concepts we've been talking about so far. To define a Vuex store, it is a simple case of calling the `Vuex.Store` constructor function, and passing an object with the `state`, `mutations`, `actions`, and `getters` that you wish to define:

```
const store = new Vuex.Store({
  strict: true,
  state: {
    cart: []
  },
  mutations: {
    addProductToCart (state, product) {
      state.cart.push(product);
    },
    incrementProductQuantity (state, product) {
      state.cart.find(p => p.id === product.id).quantity++;
    }
  },
  actions: {
    addToCart({ state, commit }, product) {
      if (state.cart.some(p => p.id === product.id)) {
        commit('incrementProductQuantity', product);
      } else {
        commit('addProductToCart', product);
      }
    }
  },
  getters: {
    cartTotal(state) {
      const reducer = (accumulator, cartItem) =>
        accumulator + cartItem.price * cartItem.quantity;

      return state.cart.reduce(reducer, 0);
    }
  }
});
```

In this example, we define a simple store that is keeping track of a single state property: an array of `cart` items. If we think about how we'd expect a shopping cart to work, when we click the **Add to cart** button on a product details page, one of two things should happen. If we have already added this product to our cart, it should increment the quantity of that cart item rather than add a duplicate entry. Otherwise, it should simply add the product to the cart. From this, we can determine that we need two mutations, which we've defined in the preceding store. The `addProductToCart` mutation simply pushes the product in question to the `cart` array, and the `incrementProductQuantity` mutation finds the existing product in the cart based on its `id` property, then increments its `quantity` value.

We know that mutations can't contain any logic, which is why we can't have a single mutation that decides whether to push the product to the array or increment its quantity if it already exists. This is where the `addToCart` action comes in, which we use to perform the required logic and decide which mutation to commit. We first check to see whether any of the existing cart items match the product in question and commit the `incrementProductQuantity` mutation if they do. Otherwise, we commit `addProductToCart` instead. Finally, we define a single `cartTotal` getter function, which computes the total cost of the cart items using a reducer function.



In the preceding store initialization, we also set the `strict` property to `true`. With strict mode on, if we try to mutate the store state outside of a mutation, an error will be thrown. However, don't leave this on in production as the performance hit can be quite high!

With a store this small, we could easily leave its definition in a single file to keep things simple. However, as we start to add additional actions, mutations, and getters, this single file can rapidly increase in size and become difficult to manage. A better idea is to break the actions, mutations, and getters out into their own separate files, and then import them back into the main store file to add them to the definition. This is what we'll do when we build the real store for our sample application shortly.

Installing and configuring Vuex

As Vuex is an official companion library made by the Vue team, it is very quick and easy to install and configure. We need to install a single npm package, then create a handful of new files, most of which will be empty until we add our store functionality. Start by running the following command in your Terminal:

```
yarn add vuex
```

Then, as this is another third-party library, we need to add it to the `vendor` array in the `webpack.config.vendor.js` file:

```
vendor: [  
  "event-source-polyfill",  
  "axios",  
  "vue",  
  "vue-router",  
  "vuex",  
  "bootstrap/dist/css/bootstrap.min.css",  
  "bootstrap-vue",  
  "nprogress/nprogress.css"  
]
```

And because we've made another change to this file, we need to run the following command from the Terminal again:

```
yarn webpack
```

With Vuex installed, we now need to configure our application to use it. Create a new `store` folder in the `ClientApp` directory, and add a `ClientApp/store/index.js` file to it. At the top of this file, add the following three lines:

```
import Vue from "vue";  
import Vuex from "vuex";  
  
Vue.use(Vuex);
```

The `Vue.use` function is how we install plugins that extend the default functionality of the `Vue` instance. Vuex is one of these plugins, so by adding the preceding code, we have done all we need to do to install it in our application. This only works because of how the Node module system treats the `import` statements in our files. After an npm module is imported for the first time, every other `import` statement receives the same copy of that module. You can think of this like registering singletons with a .NET DI framework. This means that the `Vue` instance that we receive in this file is the same instance that we initialized after importing it into our application entry point, that is, `ClientApp/boot.js`.

Next, we need three more new files in the `ClientApp/store` directory for defining the actions, mutations, and getters of our Vuex store. We need to create the following empty files:

- `ClientApp/store/actions.js`
- `ClientApp/store/mutations.js`
- `ClientApp/store/getters.js`

We can now import these files from the `index.js` file as we discussed earlier, right beneath the line where we installed Vuex previously:

```
import * as actions from "./actions";
import * as mutations from "./mutations";
import * as getters from "./getters";
```

This looks a little different to our previous `import` statements, so what exactly is going on here? When we start adding our individual actions, mutations, and getters to these files, each one will be an individually exported function. This means that each of these files will have multiple `export` statements, and as such we need to import multiple exports from a single file. We can do this using the `*` wildcard character as we've done here, then provide an alias for the imported items using the `as` keyword. We can now use these aliased imports in the object we export from this file:

```
const store = new Vuex.Store({
  strict: true,
  actions,
  mutations,
  getters,
  state: {
    cart: []
  }
});

export default store;
```

Here, we initialize a new Vuex store object, passing it the references to our actions, mutations, and getters. We could make things simpler and define these functions directly within this file, but sooner or later we'll have a lot of functions in one place, which become hard to manage and maintain. It is a much better idea to separate the different aspects of the store into separate files. We also set the `strict` property to `true`, which will cause our store to raise console errors if we try to change its state outside of any mutations we define. Finally, we add a `state` object, which is where we actually store any centralized state that we can't—or don't want to—store in local component state. In this instance, we are defining a `cart` array where we will push the items the user wants to add to their shopping cart.

The last thing we need to do is import this `store` object in our entry point file, then register it with our application's `Vue` instance. Open the `ClientApp/boot.js` file and add the following `import` statement somewhere near the top where we imported `Vue` and `Vue-Router`:

```
import store from "./store";
```

Finally, update the `Vue` instance initialization at the bottom of the file to include this `store` object:

```
new Vue({
  el: "#app-root",
  router: router,
  store,
  render: h => h(require("./components/App.vue"))
});
```

This completes our `Vuex` installation and configuration.

Adding products to the cart

The first piece of functionality our cart needs is to allow users to add their chosen product variants to it. Thinking about how this will be invoked from the UI, there are two things that could happen when a user clicks the **Add to cart** button from the product details page. First, if the chosen product variant does not yet exist in the cart, a new cart item is pushed to the array; second, if there is already a matching variant in the cart, we need to increment its quantity.

Creating the mutations

Now, we could create a single mutation that contains the logic required to work out whether the selected product variant already exists in the cart or not, then either push a new item or update an existing one as necessary. However, mutations should be very small and focused functions that only update a single piece of state. They certainly shouldn't contain any logic. Mutations are tracked by the `Vue devtools` extension so that we can see which mutations were fired. However, if those mutations have different outcomes depending on some business logic, we have no record of that outcome and as such lose all the benefits of tracking the mutations in the first place.

Based on this knowledge, there are two mutations we need to create: `addProductToCart` and `updateProductQuantity`. These are very simple and need to be added to the `ClientApp/store/mutations.js` file:

```
export const addProductToCart = (state, product) => {
  product.quantity = 1;
  state.cart.push(product);
};

export const updateProductQuantity = (state, index) => {
  let cartItem = Object.assign({}, state.cart[index]);
  cartItem.quantity++;

  state.cart.splice(index, 1, cartItem);
};
```

Both of these functions receive a `state` property as their first argument. All mutations in Vuex are passed this same object as their first argument to provide access to `state` so that it can be mutated. The second argument to any mutation function is always the object that we pass to it when invoking it from an action or directly from a component. In the `addProductToCart` mutation, we simply add an additional `quantity` property with a value of 1 to the `product` object that we receive as an argument, then push it straight into the `cart` array in the store.

The `updateProductQuantity` mutation is far more complicated. We start by cloning the cart item we wish to update using the passed in array `index` and the `Object.assign` function before we increment its `quantity` property. We then need to explicitly modify the `cart` array in such a way that Vue can tell we've changed it. This is why we make use of the `Array.splice` function again, as we know from prior experience that it will be detected and propagated down to our components. If we didn't clone the original object before passing it to the `splice` function, it still wouldn't be enough to trigger a UI update.

At this point, we still have the problem of two mutations to fire and a single `Add to cart` button to trigger them. We have no logic to decide which mutation to fire when the button is clicked; this is where actions come into play.

Creating an action

Vuex actions are where we should place any logic or asynchronous operations, such as API requests, that don't belong in mutations. As such, they are perfect for enforcing the business rules of our application, such as whether to add a new cart item or update the quantity of an existing item. In the `ClientApp/store/actions.js` file, add the following exported function:

```
export const addProductToCart = ({ state, commit }, product) => {
  const index = state.cart.findIndex(
    i =>
      i.productId === product.productId &&
      i.colourId === product.colourId &&
      i.storageId === product.storageId
  );
  if (index >= 0) {
    commit("updateProductQuantity", index);
  } else {
    commit("addProductToCart", product);
  }
};
```

Actions always receive a `context` object as their first argument. The `context` object provides an API similar to the `store` object itself, so we can access the `state` property or `commit` mutations using `context.state` and `context.commit`, respectively. Alternatively, we can use ES6's **argument destructuring** to extract only the parts of the `context` object we wish to use. This is what we've done here using the `{ state, commit }` syntax. The second argument is an optional object parameter that we can pass when dispatching the action from our component(s), which in this case is the product variant the user wants to add to their cart.

We start by checking to see whether the product variant already exists, based on the ID values that make up its composite key in our database, and then commit a mutation depending on that outcome. If the product already exists, we commit the `updateProductQuantity` mutation, or alternatively the `addProductToCart` mutation if it does not.

The only thing left to do is to actually dispatch this action from our product details page. Open up the `ClientApp/components/product/Details.vue` file, then add the following function to the `methods` object in the `script` section:

```
addProductToCart() {
  this.$store.dispatch("addProductToCart", this.variant);
}
```

By installing `Vuex` earlier, we are now given access to the `$store` property on the `Vue` instance, and therefore to each component's `this` context. We can use this property to access the `dispatch` function as we have here, passing the name of the action we want to dispatch, as well as the optional object parameter that we discussed earlier. In this case, we pass the computed `variant` property.

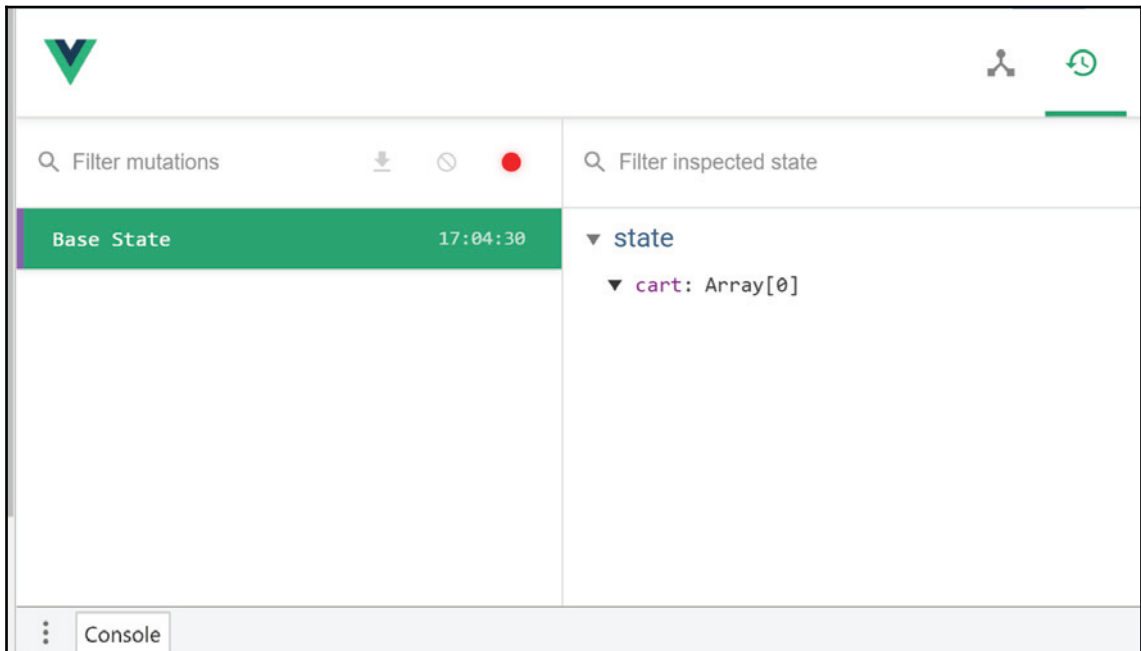
The final step is to invoke this method from the `Add to cart` button in the `template` section:

```
<b-button variant="primary" @click="addProductToCart">Add to cart
</b-button>
```

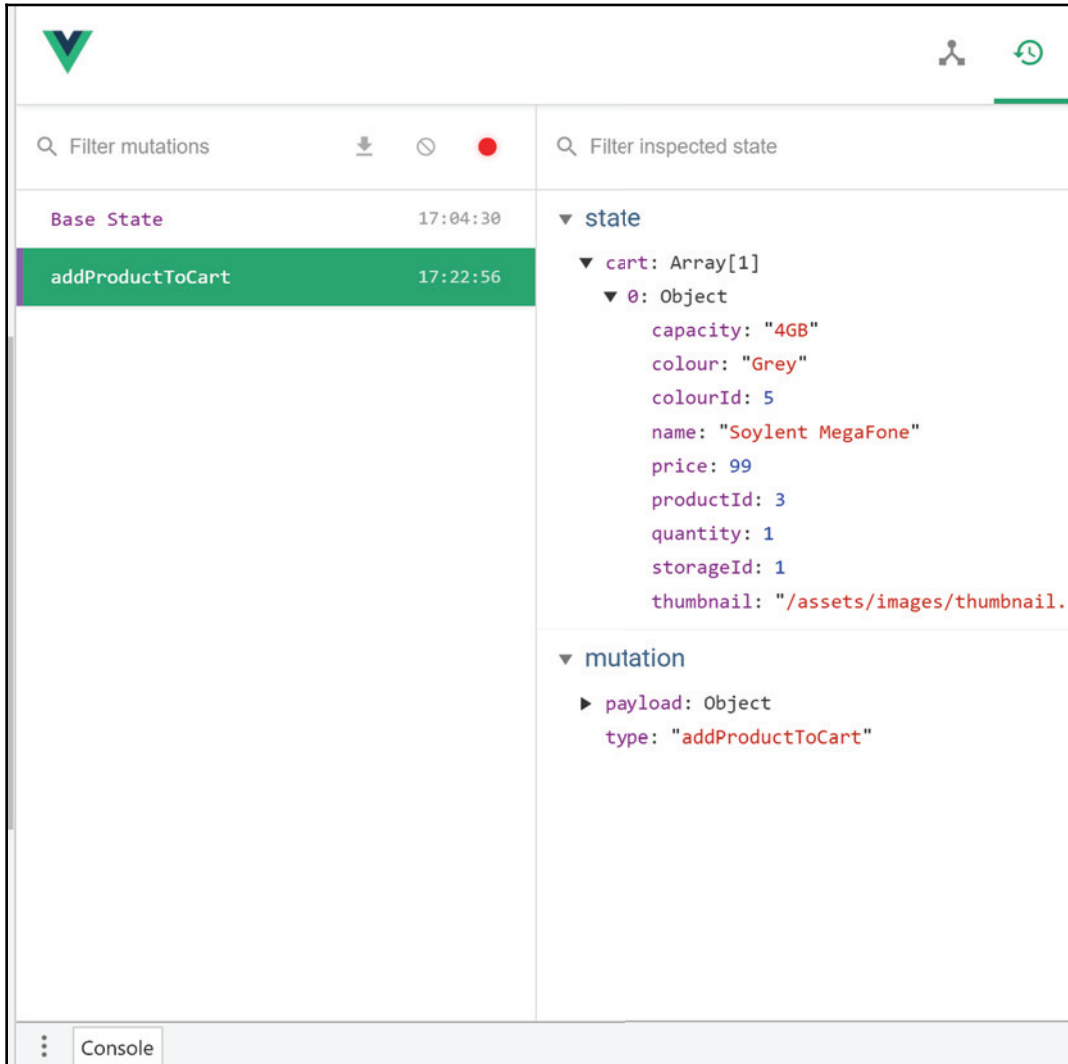
We can now add products to our cart, and if the selected product already exists, then its quantity is incremented instead. However, as we don't yet have a shopping cart page to display these cart items, the only way we can verify that everything is working correctly is by using the `Chrome devtools` extension that we installed way back in [Chapter 2, Setting Up the Development Environment](#).

One thing to note here is that when we make changes to the files in the `store` directory, the `webpack HMR` functionality will not work. As such, in order to see our changes to these files, we must force a full browser refresh before they'll take effect. If you haven't already, make sure you are viewing the application in `Chrome`; press `F12` to open the developer tools page and then perform a full browser refresh. Next, navigate to the `Vue` tab in the developer tools window, then locate the **Switch to Vuex** button inside it. If you're not sure where to find it, it's shown in the following screenshots.

The following screenshot shows what the Vuex page of the devtools extension looks like before we add any products to our cart:



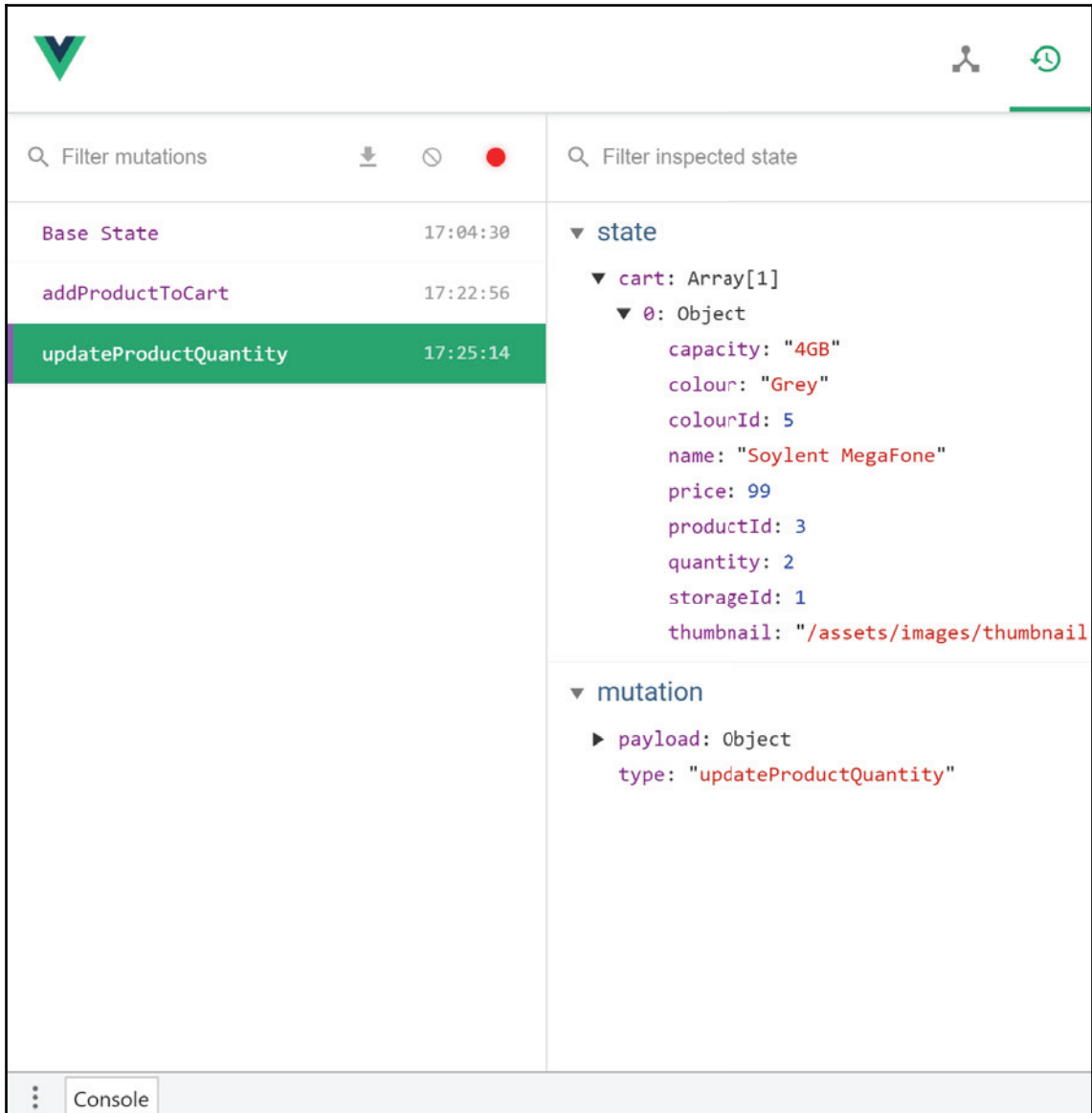
After finding a product we wish to buy and clicking the **Add to cart** button for that product, we get a single cart item with a **quantity** value of **1**. We can also see that an **addProductToCart** mutation is listed:



The screenshot displays the Redux DevTools interface. On the left, the 'Mutations' panel shows a list of actions. The 'addProductToCart' action, timestamped at 17:22:56, is highlighted in green. The right panel shows the 'Inspected State' for this action, which is a JSON object representing the state of the application. The state includes a 'cart' array with one item and a 'mutation' object with a 'payload' and a 'type'.

```
state
  cart: Array[1]
    0: Object
      capacity: "4GB"
      colour: "Grey"
      colourId: 5
      name: "Soylent MegaFone"
      price: 99
      productId: 3
      quantity: 1
      storageId: 1
      thumbnail: "/assets/images/thumbnail."
  mutation
    payload: Object
      type: "addProductToCart"
```

If we then click the **Add to cart** button again for the same product variant, instead of the **addProductToCart** mutation that we saw before, we get an **updateProductQuantity** mutation added to the list. In the state section, we can also see that we still only have one cart item in the array, and that item now has a **quantity** value of 2:



The screenshot displays the Redux DevTools interface. On the left, a list of mutations is shown:

Mutation	Time
Base State	17:04:30
addProductToCart	17:22:56
updateProductQuantity	17:25:14

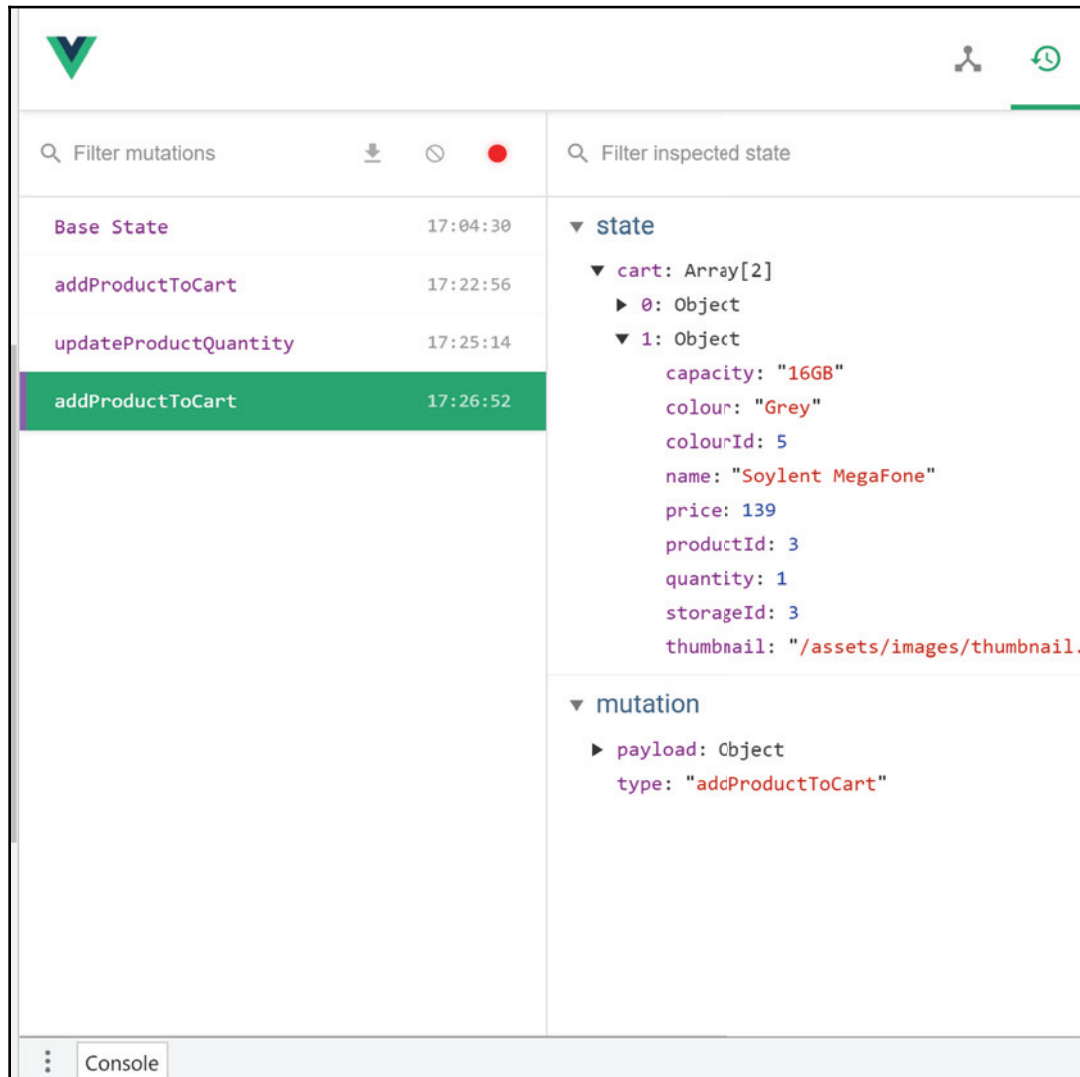
The 'updateProductQuantity' mutation is selected and highlighted in green. On the right, the inspected state is shown:

```
state
  └─ cart: Array[1]
      └─ 0: Object
          ├── capacity: "4GB"
          ├── colour: "Grey"
          ├── colourId: 5
          ├── name: "Soylent MegaFone"
          ├── price: 99
          ├── productId: 3
          ├── quantity: 2
          ├── storageId: 1
          └── thumbnail: "/assets/images/thumbnail"

mutation
  └─ payload: Object
      └─ type: "updateProductQuantity"
```

At the bottom, a 'Console' tab is visible.

Finally, if we change one of the variant dropdowns on the details page and click the **Add to cart** button again, a new cart item is pushed to the array. We can see this is the case as another `addProductToCart` mutation was fired, and our `state` object now contains two items in the `cart` array:



The screenshot displays the Redux DevTools interface. On the left, a list of mutations is shown, with the most recent one, `addProductToCart` at 17:26:52, highlighted in green. On the right, the inspected state is shown, which is an object with a `state` property. The `state` property is an array of two objects. The first object has the following properties: `capacity: "16GB"`, `colour: "Grey"`, `colourId: 5`, `name: "Soylent MegaFone"`, `price: 139`, `productId: 3`, `quantity: 1`, `storageId: 3`, and `thumbnail: "/assets/images/thumbnail..."`. The second object is partially visible. Below the state, the mutation object is shown, with `payload: Object` and `type: "addProductToCart"`.

Creating a shopping cart page

Now that we have the ability to add products to a shopping cart, we need a page for viewing what we've added before deciding whether to proceed to the checkout or not. Create an empty component called `Cart.vue` in the `ClientApp/pages` directory, then open the `ClientApp/boot.js` file and import it along with the other page components:

```
import Cart from "./pages/Cart.vue";
```

In the same file, we'll also need to add a route definition to the `routes` array:

```
{ path: "/cart", component: Cart }
```

And finally, we'll need a link to this new page from our navbar that's defined in the `ClientApp/components/App.vue` file. Find the existing `b-navbar-nav` element and add the following additional right-aligned `b-navbar-nav` element directly preceding it:

```
<b-navbar-nav class="ml-auto mr-4">
  <b-nav-item to="/cart">Cart</b-nav-item>
</b-navbar-nav>
```

Before we build the `ClientApp/pages/Cart.vue` page itself, we're going to build a `ClientApp/components/cart/CartItem.vue` component instead. If we think about what functions we're going to want for the cart page, they include updating and deleting individual cart items. These functions could reside in the cart page itself, and take a reference to the specific cart item we want to update or delete, but we can also simplify that logic by extracting a `CartItem` component, which is only responsible for displaying a single cart item record. If the `update` and `delete` functions reside in this nested `CartItem` component, they can only update or delete an individual item, and as such the logic is simpler. We also gain the other standard benefits of child components, such as reducing the amount of code in a single file and making sure each component only has a single responsibility.

Creating a CartItem component

Create the `ClientApp/components/cart/CartItem.vue` component and add a `template` section that looks like this:

```
<template>
  <tr>
    <td>
      <b-row align-v="center">
        <b-col cols="auto">
```

```

        
    </b-col>
    <b-col class="align-middle">
        <h5>{{ item.name }}</h5>
        <div>
            Colour: <strong>{{ item.colour }}</strong>
        </div>
        <div>
            Capacity: <strong>{{ item.capacity }}</strong>
        </div>
    </b-col>
</b-row>
</td>
<td>
    £{{ item.price }}
</td>
<td>
    <b-form-input type="number" :value="item.quantity">
    </b-form-input>
</td>
<td>
    £{{ item.price * item.quantity }}
</td>
<td>
    <b-button variant="danger">
        <i class="fas fa-trash-alt"></i>
    </b-button>
</td>
</tr>
</template>

```

There is nothing new here, and the majority of this HTML is only necessary to get things presented and arranged nicely. We'll use a standard HTML `table` element for our shopping cart when we create the parent component, so for each individual cart item, we're using a `tr` element as its root. We're defining five columns within this table row: one for the product name, one for the price, one for the quantity, one for the calculated subtotal, and one for any actions that can be performed on this row.

The product name column is the most complicated, but only because we're nesting a Bootstrap grid row with two columns to position the product thumbnail side by side with its name, selected color, and capacity options. For the price, we just render the raw value prefixed with a £ sign as we did in the product details component, and for the quantity, we're using a simple HTML number input. For the calculated subtotal, we're keeping things simple for now and rather than using a computed property, we're just using a string interpolation expression. Finally, in the actions column, we have a single button, which will eventually be used to remove the item from the cart.

The `script` section for this component is incredibly simple, and only defines a single `item` prop to represent the cart item being displayed:

```
<script>
export default {
  name: "cart-item",
  props: {
    item: {
      type: Object,
      required: true
    }
  }
};
</script>
```

And finally, the `style` section is mainly used for specifying the minimum widths of each column:

```
<style lang="scss" scoped>
td {
  vertical-align: middle;
}

td:nth-child(1) {
  width: auto;
  min-width: 350px;

  img {
    width: 100px;
  }
}

td:nth-child(2) {
  width: 75px;
}

td:nth-child(3) {
  width: 75px;
}

td:nth-child(4) {
  min-width: 150px;
}

td:nth-child(5) {
  width: 125px;
}
</style>
```

This is all we need for the `CartItem` component, so let's finish off the shopping cart page itself.

Displaying the list of cart items

Back in the `ClientApp/pages/Cart.vue` file, we need to add a `template` section that renders a HTML table element and loops over the items in our cart to render an instance of the `CartItem` component that we just created. Modify the `template` section like this:

```
<template>
  <b-container class="pt-4 page">
    <h1 class="pb-4">Shopping cart</h1>
    <p v-if="!items.length" class="mt-4 text-center">
      You don't have anything in your cart yet!
    </p>
    <div v-else class="table-responsive">
      <table class="table">
        // shopping cart items will be rendered here...
      </table>
    </div>
  </b-container>
</template>
```

As with our other pages, we will start with a `b-container` element with a class of `page` to make sure transitions to and from this page display correctly. After the `h1` element, we use the `v-if` and `v-else` directives to conditionally display either a `p` element or a `div` element, depending on whether a data property called `items` has any array elements or not. If it does, we display a Bootstrap responsive table to hold our cart items, and if not, we just display a message to state that there are no items in the user's cart.

Inside the `table` element, we'll need a `thead` section with column headers matching those we defined in the `CartItem` component earlier:

```
<thead>
  <tr>
    <th>Product</th>
    <th>Price</th>
    <th>Quantity</th>
    <th>Subtotal</th>
    <th></th>
  </tr>
</thead>
```


We'll also need a `tbody` element to loop over the `items` property that we're about to define:

```
<tbody>
  <cart-item v-for="(item, index) in items" :key="index"
    :item="item" />
</tbody>
```

As we have no single unique value for a cart item due to the composite key that identifies a product variant, we're using the index of the `items` array as the `key` prop for each cart item. We then use the `v-bind` directive to pass the item reference down into the cart item component to be displayed.

Finally, we need a `tfoot` section to display the total of the items in the cart, as well as two buttons for returning to the catalog or proceeding to the checkout:

```
<tfoot>
  <tr>
    <td>
      <b-button variant="secondary" @click="continueShopping">
        <i class="fas fa-chevron-left"></i>
        Continue shopping
      </b-button>
    </td>
    <td colspan="2"></td>
    <td>
      <strong>Total: £0.00</strong>
    </td>
    <td>
      <b-button variant="success">
        Checkout
        <i class="fas fa-chevron-right"></i>
      </b-button>
    </td>
  </tr>
</tfoot>
```

The script section for this page component is relatively simple, but does involve fetching data from our centralized Vuex store, which we've never done before:

```
<script>
import CartItem from "../components/cart/CartItem.vue";

export default {
  name: "cart",
  components: {
    CartItem
  },
}
```

```
    computed: {
      items() {
        return this.$store.state.cart;
      }
    },
    methods: {
      continueShopping() {
        this.$router.go(-1);
      }
    }
  };
</script>
```

We start by importing and declaring the `CartItem` component in the `components` object. We then have the `computed items` property, which we already know needs to hold our list of cart items from the store, and this is where we deviate from any `computed` property that we've seen before. As we now have access to the `$store` property in the component's `this` context, we can directly access the state of our store. In this instance, we simply return the `cart` array. Finally, in the `methods` object, we have a single function to go back a page in our router history when the user clicks on the **Continue shopping** button.

Our shopping cart page is now complete. Restart the application now and add a few different product variants to your cart, then follow the **Cart** link on the right of your navbar to see what the shopping cart looks like.

Creating a currency filter

At this point, we have multiple places within our UI where we want to display a price. Currently, we're simply printing the raw string value of the number representing that price and prefixing it with the `£` character. Apart from repeating the logic of prefixing the value with a currency character, there is another limitation here in that unless the number already has decimal places, that is, it's a decimal rather than an integer, the value we render won't have decimal places either. When we see prices, we expect to not only see the currency of that price, but also the value after the decimal place, even if it's zero. For example, if the price of a phone is 99 GBP, we'd expect to see this rendered like this: `£99.00`.

We could repeat this rendering logic everywhere that we display a price, but we already have three or four occurrences already, and that number is only ever going to increase. Instead, we should try and find some reusable way of defining this logic so that we can use it anywhere we need to. The answer to how we do this in Vue is filters. A filter is a simple function that takes an arbitrary list of arguments, does some processing on those arguments, and returns a value. The returned value is what ends up being rendered in the UI of the application.

Let's create a simple currency filter to demonstrate how this works. Create a new `filters` folder in the `ClientApp` directory and add an `index.js` file to it. Within this file, we can add as many filter functions as we like, exporting each one individually like we did with the actions and mutations of our Vuex store. The `currency` function looks like this:

```
export const currency = value => {
  return "£" + parseFloat(value).toFixed(2);
};
```

This function takes a single `value` argument, passes it through the `parseFloat` function to ensure we're working with a float value with decimal places, then chains the `toFixed` function on the end to limit the number of those decimal places to two. Finally, we return this newly parsed value as a string with the £ prefix.



We're hard-coding a currency prefix again here, but just remember that in a real application you may need to localize this prefix if you support users from multiple locales.

We now need to register this filter function with our global `Vue` instance so that we can make use of it anywhere in our application, without needing to manually import it into every file. Open up the `ClientApp/boot.js` file and add the following code right below the line where we install `Bootstrap-Vue`:

```
// rest of file as before...

Vue.use(VueRouter);
Vue.use(BootstrapVue);

// filters
import { currency } from "./filters";

Vue.filter("currency", currency);

// rest of file as before...
```

We will import the `currency` function using named imports, then register it with the `Vue` instance as a filter using the `Vue.filter` function. This is all we need to do. Now, we have access to this filter in all of our components, but how do we use it? To demonstrate, in the `ClientApp/components/cart/CartItem.vue` component, we have a line that looks like this:

```
£{{ item.price }}
```

We can use the currency filter by removing the `£` character, and use the filter syntax, which uses the pipe character and looks like this:

```
{{ value | filter }}
```

In our case, the updated item price rendering looks like this:

```
{{ item.price | currency }}
```

A little lower down in the same template, we have a slightly more complex expression:

```
£{{ item.price * item.quantity }}
```

This is no problem, and our currency filter can be used here as well:

```
{{ item.price * item.quantity | currency }}
```

Essentially, this filter can now be used anywhere that we're rendering a price in a template string expression. There are a few more places to make this change, which I won't go into detail about here as it's exactly the same process, but these files are:

- `ClientApp/components/catalogue/ProductList.vue`
- `ClientApp/components/product/Details.vue`

Removing products from the cart

We already have the UI elements we need to trigger the removal of products from the cart, so all we need to do now is create the necessary mutation and action in our store and wire everything up. In the `ClientApp/store/mutations.js` file, add the following exported function:

```
export const removeProductFromCart = (state, index) => {  
  state.cart.splice(index, 1);  
};
```

This mutation is a simple case of calling the `Array.splice` function and passing it the `index` argument as the position to start deleting from, and instructing it to only delete a single item by passing `1` as the second parameter. Again, by modifying the `cart` array using the `splice` function, Vue can detect the change and react to it. This causes any components that are observing this array to update and refresh their UI automatically, the same way they do if a local component's state changes.

The associated action for this mutation is quite simple, and simply finds the `index` of the item we wish to remove, then commits the mutation and passes it as an argument:

```
export const removeProductFromCart = ({ commit }, product) => {
  const index = state.cart.findIndex(
    i =>
      i.productId === product.productId &&
      i.colourId === product.colourId &&
      i.storageId === product.storageId
  );

  commit("removeProductFromCart", index);
};
```

This action may seem fairly pointless as we could have found the array index within the mutation itself. In that instance, we could actually just commit the mutation directly from a component as well, which ultimately makes the action unnecessary. The idea behind Vuex is that components dispatch actions, and actions commit mutations, which update the state in the store. However, there is a difference of opinion among Vue developers that I've spoken to around whether actions are 100% necessary if they don't add any kind of value.

If we aren't performing any kind of asynchronous operation or committing multiple mutations based on some kind of business logic, my opinion is that there is no harm in committing mutations directly from our components, and this is exactly what I do in a lot of cases. However, it is important to avoid the temptation of putting any kind of logic, or starting to update multiple pieces of state, in your mutations. For this reason, we will make sure to always use actions for the purposes of this sample application.

The final step in wiring up the *remove from cart* functionality is to dispatch this action in our `ClientApp/components/cart/CartItem.vue` component. In the template section, locate the final `td` element in the row and update the `b-button` element inside to include an `@click` event handler, like this:

```
<b-button variant="danger" @click="removeProductFromCart">
  <i class="fas fa-trash-alt"></i>
</b-button>
```

Finally, in the `script` section, add a `methods` object that includes the following function:

```
methods: {
  removeProductFromCart () {
    this.$store.dispatch("removeProductFromCart", this.item);
  }
}
```

Refresh the application and test out the *remove from cart* functionality by adding a couple of items to your cart, then clicking the trash icon on one of the rows to ensure that the correct item is deleted. You can also verify that the correct mutations were fired as we did before, using the Chrome devtools extension.

Updating cart items

The last function we need to perform with our cart items is manually updating the item quantity. As we've just done with removing cart items, we'll need a mutation and an action, and we'll have to perform some minor updates on the `CartItem` component to wire everything up.

In the `ClientApp/store/mutations.js` file, add the following new mutation:

```
export const setProductQuantity = (state, payload) => {
  let cartItem = Object.assign({}, state.cart[payload.index]);
  cartItem.quantity = payload.quantity;

  state.cart.splice(payload.index, 1, cartItem);
};
```

This mutation is another fairly complicated one due to how the reactivity system in Vue works, but it is virtually identical to the `updateProductQuantity` mutation we saw earlier, so I won't explain it again this time around. I'm conscious that I've made the same statement about the reactivity system a few times now, and it may be coming across as though I'm suggesting there are some fundamental flaws with it. However, this isn't the case, as the reason why the reactivity system has to work the way it does is down to limitations in JavaScript itself, rather than with the Vue or Vuex libraries.

The only thing to note is that instead of an explicitly named argument such as `product`, like we've seen before, we're receiving an object called `payload`. We're using an object here because we can only pass a single user-defined argument to a mutation, and we need multiple pieces of data. On this object, we're expecting another integer property called `index`, which represents the array index of the item we wish to update, and another integer property specifying the quantity to set.

In the `ClientApp/store/actions.js` file, add the following new action:

```
export const setProductQuantity = ({ state, commit }, payload) => {
  const index = state.cart.findIndex(
    i =>
    i.productId === payload.product.productId &&
    i.colourId === payload.product.colourId &&
    i.storageId === payload.product.storageId
  );

  if (payload.quantity > 0) {
    payload.index = index;
    commit("setProductQuantity", payload);
  } else {
    commit("removeProductFromCart", index);
  }
};
```

We use the `payload.product` object to find the array index of the cart item we wish to update, then interrogate the `payload.quantity` property to decide which mutation to commit. If the quantity is more than zero, we commit the mutation we just created, passing the `payload` object after assigning the `index` property we just found. However, if the user has entered zero or less into the quantity field, we commit the `removeProductFromCart` mutation instead, passing just the index of the item we wish to remove.

Open `ClientApp/components/cart/CartItem.vue` and find the `b-form-input` element that we're using for the quantity number input. Modify it and add a `@change` event handler like so:

```
<b-form-input type="number" :value="item.quantity"
@change="setProductQuantity"></b-form-input>
```

Finally, in the `methods` object, add the following new function:

```
setProductQuantity(quantity) {
  const payload = { product: this.item, quantity: parseInt(quantity) };
  this.$store.dispatch("setProductQuantity", payload);
}
```

All we need to do here is create a new `payload` object, including the properties and values we're expecting in the action and the mutation we just created, then pass this new object as the argument to the `dispatch` call. The only other thing to note is that we need to parse the `quantity` value passed in to ensure it goes through the pipeline as an integer rather than a string.

Refresh the application and make sure everything is still working. You should be able to change the quantity value either by typing a new value in and tabbing away from the field, or clicking the buttons inside the field to increase or decrease its value. Either way, the subtotal will update to represent the new value based on the quantity entered.

Adding a getter to display the cart total

The final change we need to make to the shopping cart itself is to display the calculated grand total at the bottom. As it stands, we have no means of calculating it, so we're just displaying a hardcoded `£0.00` value instead. To fix this, we'll add a Vuex getter to our store, which will calculate the total in a reusable function that we can use anywhere in our application.

Getters have a similar use case as computed properties: creating derived state for specific display purposes. In this case, the grand total of the shopping cart is the sum of a specific calculation on each cart item. We could store the grand total as a `state` property, but we'd then have the overhead of remembering to keep it up to date after any mutation has run that could alter that value. Instead, we derive the value from the cart items, and as they change, the getter will update automatically.

In the `ClientApp/store/getters.js` file, add the following exported function:

```
export const shoppingCartTotal = state => {
  const reducer = (accumulator, cartItem) =>
    accumulator + cartItem.price * cartItem.quantity;

  return state.cart.reduce(reducer, 0);
};
```


As we did with our actions and mutations, we export each getter function individually, and each one receives the store `state` object as its first argument. There are multiple ways of calculating this value, but the most concise is probably to use the `Array.reduce` function, as we have here. However, if this function is new to you, it may look incredibly strange at first. The `reduce` function executes another function against each element in the array, accumulating the result and passing it to the next iteration. In our getter, we define a `reducer` function, which takes two arguments: `accumulator` and `cartItem`. As the names would suggest, the `accumulator` argument tracks the accumulated value and `cartItem` represents the array item currently being processed. The actual function is simple: take the accumulated value and add the cart item subtotal ($price * quantity$) to it. To make use of this function, we must pass it to the `Array.reduce` function as we have here, which causes each element of the array to be *reduced* in turn, each down to a single value. As our cart may be empty, we have to provide the optional second argument, which is the starting value to use.

To make use of this getter from our cart page, we need to make two very simple changes. We'll add a computed property, which retrieves the value from the getter, then bind the UI to that computed property. Open the `ClientApp/pages/Cart.vue` file and add the following computed property:

```
total() {
  return this.$store.getters.shoppingCartTotal;
}
```

Finally, find the `Total: £0.00` line in the `template` section and change it to an expression that makes use of the currency filter we made earlier:

```
<strong>Total: {{ total | currency }}</strong>
```

Refresh the application now and test that everything works as expected. The total at the bottom of the cart should update whenever you update an item's quantity or remove an item from the cart.

Creating a cart summary component

To further demonstrate the benefits of centralized store state and getters, we're going to add a cart summary component that displays in a popup when a user hovers over the cart link in the navbar. We'll add another getter that calculates the total number of items in the cart, then use it along with the one we already have inside the popover to show how useful it is to have reusable calculated properties like this.

Add the following exported function to the `ClientApp/store/getters.js` file:

```
export const shoppingCartItemCount = state => {
  const reducer = (accumulator, cartItem) => accumulator +
    cartItem.quantity;
  return state.cart.reduce(reducer, 0);
};
```

This is very similar to the previous getter we created, so it should look fairly familiar, but ultimately we're doing the equivalent of a `Sum` calculation on a LINQ collection in C#. The total number of items in the cart is equal to the sum of the cart item `quantity` values.

Next, create a new empty component called `CartSummary.vue` in the `ClientApp/components/cart` directory. The `template` section of this component looks like this:

```
<template>
  <b-nav-item id="Cart" to="/cart">
    <i class="fas fa-shopping-cart"></i>
    Cart ({{ count }})

    <b-popover target="Cart" triggers="hover">
      <div>
        Items: {{ count }}
      </div>
      <div>
        Total: {{ total | currency }}
      </div>
    </b-popover>
  </b-nav-item>
</template>
```

As we'll be replacing the navbar link to the cart page with this component, we're using a `b-nav-item` element as the root element, with the same `to` prop of the cart page's relative URL. To make things look a little nicer, I've added a Font Awesome shopping cart icon before the link text, as well as a `count` variable displayed in brackets at the end. Note that I also added an `id` attribute, which we'll use as the target of the popover component we're about to add.

Bootstrap-Vue has a built-in popover component called `b-popover`, which we're using here. We're passing a `target` prop equal to the `id` attribute of `b-nav-item` above it, and passing `hover` as the `triggers` prop to cause the popover to display when the link is hovered over. Inside the popover, we can render whatever content we wish, and could even render a reduced-size table of cart items like the one on the cart page itself. However, to keep things simple for the sake of this demonstration, all we're rendering is the item count and total cost of the items in our cart.

The `script` section of this component is very simple, and only contains two computed properties that map to the store getters we defined earlier:

```
<script>
export default {
  name: "cart-summary",
  computed: {
    total() {
      return this.$store.getters.shoppingCartTotal;
    },
    count() {
      return this.$store.getters.shoppingCartItemCount;
    }
  }
};
</script>
```



Rather than creating a computed property each time we wish to data bind to a Vuex getter or state property, we could utilize some helper functions that come baked into Vuex itself. If you're interested, look up the `mapState` and `mapGetters` functions to see how they work. However, we won't be using them in the interests of keeping things simple!

The last step is to make use of this new component in the `ClientApp/components/App.vue` file by replacing this line:

```
<b-nav-item to="/cart">Cart</b-nav-item>
```

With this line:

```
<cart-summary />
```

Then, import and declare the new component for use in the `script` section:

```
<script>
import CartSummary from "../cart/CartSummary.vue";

export default {
  name: "app",
  components: {
    CartSummary
  }
};
</script>
```

That's all there is to it. By abstracting certain pieces of functionality away into a centralized store, we can reuse them across any component in the entire application with very little effort or complexity involved. If we were to carry on using props to pass data down from parent components to their children, we'd have to declare all of our shared application state in the root `App.vue` component and pass it down through the tree of components as far as it needs to go. Each time we add new components to the tree, we'd need to remember to pass those props on again, which is very repetitive and error-prone. By centralizing shared application state, we alleviate these issues.

Now, test the application again to make sure everything is still working, and remember that as we've made changes to the `ClientApp/store/getters.js` file, a full browser refresh is required before the changes will take effect. Add some products to your cart, then hover over the cart link to see the summary component we just created.

Persisting the cart to local storage

As you will have noticed already, every time we refresh the browser, our shopping cart's contents are lost. Both component- and application-level state are only stored in memory, apart from the catalog filter selections, which we pushed into the browser URL. As such, as soon as the browser is refreshed, that memory is cleared and we lose the state of our shopping cart.

There are a number of ways that we could solve this problem, including persisting the state into the browser's local storage, or pushing the state up to our API and persisting it in the database. Both of these options are completely acceptable and widely used approaches, but they also have their own pros and cons. Persisting to local storage in the browser is simple and effective, but if the user clears their browser cache, then they will still lose their cart items. On the other hand, persisting to the database via the API is permanent, and unless they explicitly clear their cart, then its contents are safe from browser memory loss. However, it is a far more complicated approach, as unless we enforce user registration and authentication before adding items to a shopping cart, we have to put additional measures in place to allow anonymous user access so that we can identify which cart belongs to which user session.

Generally speaking, it is normally acceptable to allow the shopping cart's contents to be wiped if a user explicitly clears their browser cache, so this usually isn't a reason to avoid using local storage. One of the main benefits of persisting a shopping cart into the database is for reporting purposes. Companies will find information about how many carts actually result in a completed sale highly valuable, and this information is only available if the cart is stored on the server side of the application. For this reason, and to keep things simple, we'll be persisting the cart items to local storage for this demo application.



If you prefer to store the cart in the database, you can write ASP.NET Core middleware that attaches a GUID to unauthenticated response headers in order to identify an anonymous user. Link the cart to this GUID in the database, then transfer it to a real user account when they register!

You might think this is going to be a fairly tricky process of extending every mutation to also persist the changes into local storage. However, thanks to the use of Vuex, we only have three simple changes to make. First, open the `ClientApp/store/index.js` file, and right before the export line at the bottom, we need to add the following:

```
store.subscribe((mutation, state) => {
  localStorage.setItem("store", JSON.stringify(state));
});
```

As the `store.subscribe` function name suggests, we are subscribing to any changes in the store and firing a callback that uses the `localStorage.setItem` function to persist the entire `store` object. Note that the browser's local storage is only capable of storing key/value pairs, where the values are strings. In order to store a complex object like our `state` object, we have to convert it to a JSON string by using the `JSON.stringify` function.

Next, we need an additional mutation, which will check whether we have anything stored in local storage, and initialize our state with it if we do. Add the following exported function to the `ClientApp/store/mutations.js` file:

```
export const initialise = state => {
  const store = localStorage.getItem("store");
  if (store) {
    Object.assign(state, JSON.parse(store));
  }
};
```

If we find a key named `store` in local storage, we use the `Object.assign` function that we've used so many times before to copy the values from the local storage onto the `state` object from our Vuex store. As we had to convert this object to a string on the way into the local storage, here we have to do the opposite and call `JSON.parse` to convert it back into an object. Note that we must use the `Object.assign` function here or the application will not react to the state change.

Finally, in the `ClientApp/components/App.vue` file, we need to add a `beforeCreate` life cycle hook to commit this new mutation:

```
export default {
  name: "app",
  components: {
    CartSummary
  },
  beforeCreate() {
    this.$store.commit("initialise");
  }
};
```

Notice how we haven't bothered to define an action this time, as like we previously discussed, there really is no need for such a simple mutation.

Refresh the application again now and the next time you add products to your cart and refresh the browser, they should remain in your cart. You can also verify that the mutation is firing by checking the devtools extension, like we did earlier.

Improving the UX with add to cart feedback

Our cart is now fully functional, and as such we could leave it as is. However, when clicking the **Add to cart** button, there is no feedback as to whether or not something happened. Some online shops take you to the shopping cart page as and when you add a product to your cart, and if that's your preferred approach, then it should be a fairly simple change for you to add a page change as part of the button click handler. However, to demonstrate a different approach, we'll use a library called `toastr` to provide instant feedback to the user that something positive happened.



This section is completely optional, and will have no negative effects on the rest of the application if you choose to skip it for any reason!

First, we need to download a new npm module. The original `toastr` library has a dependency on jQuery, so instead we'll use another open source alternative that is built specifically for use with Vue instead. Open Terminal and run the following command:

```
yarn add @deveodk/vue-toastr
```

Next, in the `ClientApp/boot.js` file, we need to install and configure this module like we did before with other Vue plugins. At the top of the file with the rest of the `import` statements, add the following two lines:

```
import VueToastr from "@deveodk/vue-toastr";
import "@deveodk/vue-toastr/dist/@deveodk/vue-toastr.css";
```

Then, after the `Vue.use(BootstrapVue);` line, add the following:

```
Vue.use(VueToastr, {
  defaultPosition: "toast-top-right"
});
```

This is similar to what we've done before, but this particular installation has an optional second argument where we can override some default settings. In this case, we just specify that the toast dialogs should appear at the top-right of the screen by default.

Finally, in the `ClientApp/components/product/Details.vue` file, locate the `addProductToCart` function in the `methods` object of the script section, and right after committing the mutation, add the following line:

```
this.$toastr("success", "Product added to cart successfully.");
```

Refresh the page and add another product to your cart to check that the toast messages appear as expected.

The last thing I'm going to do is remove the **Add to cart** button from the catalog list page, as it currently doesn't do anything and there is no way for our users to specify a product variant until they enter the details page anyway. For reference, this is in the `template` section of the `ClientApp/components/catalogue/ProductList.vue` file.

Summary

It was another long and fairly complicated feature to implement in this chapter, so let's take a minute to review what we've achieved. We started out by looking at our options for persistent shopping cart data, both on the client side and server side, before settling on using the local storage feature of the web browser.

We then prepared our product details page by creating a custom image gallery component and adding drop-down lists to provide our users with a way of specifying which variant of the product they wish to purchase. We then installed and configured Vuex for centralized client-side state management, before adding the selection of actions, mutations, and getters required to provide the basic functionality of a shopping cart.

We also added a new shopping cart page to our application and looked at how we can fetch data from the Vuex store inside our components. We also built a cart summary component to show how easily we can reuse store logic from anywhere in our application. We saw how custom filters can reduce duplication by building a currency filter, and how easy it is to add toast notifications to improve the UX of the application.

Finally, with just a few lines of code, we saw how we can persist the entire centralized state of the application into local storage, and retrieve it again each time the application loads or the browser refreshes.

7

User Registration and Authentication

We now have a functional product catalog that our users can browse, as well as a shopping cart that they can utilize to store their chosen products before making a purchase. However, before we allow them to proceed to the checkout page and place their orders, we need to make sure they have a user account to link the order to in the database. In this chapter, we're going to add authentication and user registration to the app, by making use of the latest features of ASP.NET Core 2.0 to issue and validate JWT tokens.

In this chapter, we're going to look at the following topics:

- Configuring JWT authentication in ASP.NET Core 2.0
- Issuing and validating JWTs
- Persisting authentication state with Vuex and local storage
- Configuring global HTTP headers with `axios`
- Creating login/register UI components

We have a fair bit of ground to cover, so let's dive right in.

Adding JWT authentication to the API

In previous versions of ASP.NET Core, configuring authentication was far more cumbersome and not particularly intuitive. However, with the 2.0 release, Microsoft did a lot of work to refactor how authentication works in ASP.NET Core, and it's now a very simple process to add it to an ASP.NET Core web app.

Why JWTs?

For standard server-side web applications built with MVC, we'd most likely use cookies rather than JWTs, which is the default option if we don't specify one. However, as we're building a stateless web API with an SPA frontend, it makes much more sense to use JWTs in order to maintain the stateless nature of the application.

In traditional MVC applications, when a user logs in, a session is created on the server and a cookie is returned to the user's browser, which identifies that session on subsequent HTTP requests. This is a stateful authentication mechanism that doesn't really fit in with modern applications that use JavaScript SPA frameworks for the UI. The sample application we're building throughout this book is based on the simple approach of hosting both frontend and backend on a single web server, accessed via a single domain. However, it is fairly common with modern applications to use multiple web servers to completely separate the frontend from the backend, and access each from different domains or subdomains. For example, we could host the frontend of our e-commerce application at `http://app.phoneshop.com`, and the backend at `http://api.phoneshop.com`. With cookie-based authentication, this isn't necessarily possible, depending on the configuration of the server and the way it sets cookies. It certainly isn't at all possible if we want to use an entirely different domain, or allow access to our API for other applications or services on different domains.

In comparison, when a user logs in to an application using JWT authentication, no server-side session is required. All the server needs to do is generate a JWT and return it to the client. It doesn't matter what type of client it is, such as web applications versus mobile applications; as long as the JWT is included with any HTTP request to the API, it can be verified and used to authenticate the user. We also lose the restriction on how we host our applications and the domains we use, and we can easily allow access to external applications or services.

There are a number of other benefits to using JWTs, such as scalability and performance, but these are beyond the scope of this book. Generally speaking, it is safe to say that any application that makes use of a JavaScript SPA framework with a backing API is far better suited to using JWTs rather than cookies.

Configuring JWT authentication

Now that we know why we're using JWTs, we need to make a few configuration changes to enable authentication and specify the mechanism we wish to use. We'll start by opening the `Startup.cs` file, and finding the following piece of code within it:

```
services.AddIdentity<AppUser, AppRole>()
    .AddEntityFrameworkStores<EcommerceContext>()
    .AddDefaultTokenProviders();
```

Directly beneath this section, add the following:

```
services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme =
        JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
        JwtBearerDefaults.AuthenticationScheme;
})
```

If we were using cookie-based authentication, we wouldn't need this code at all, but since we're using JWTs, we need to override the default authentication mechanism by specifying the JWT bearer authentication scheme as the default challenge and authentication schemes. With this in place, we need to finish off the configuration by chaining on the following code directly:

```
.AddJwtBearer(options =>
{
    options.RequireHttpsMetadata = false;
    options.SaveToken = true;
    options.ClaimsIssuer = Configuration["Authentication:JwtIssuer"];
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidIssuer = Configuration["Authentication:JwtIssuer"],
        ValidateAudience = true,
        ValidAudience = Configuration["Authentication:JwtAudience"],
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new
            SymmetricSecurityKey(Encoding.UTF8.GetBytes
                (Configuration["Authentication:JwtKey"])),

        RequireExpirationTime = true,
        ValidateLifetime = true,
        ClockSkew = TimeSpan.Zero
    };
});
```

Most of these options are fairly self-explanatory, and Microsoft's own documentation is pretty good for setting up authentication, so we won't dwell too long on this. Essentially, we are just defining what the standard JWT claim values should be set to, such as the `iss` and `aud` claims, as well as overriding some of the defaults, such as disabling the HTTPS requirement. As previously discussed, our sample app is running from a single domain URL, so the issuer and audience values will be the same, but we could just as easily have different URLs for these values. We also make sure that the JWT middleware will validate the issuer, audience, and signing key in order to ensure that the JWTs haven't been tampered with in any way, or been issued from a different application.

Quite a few of these settings are making use of the `Configuration` object of the class, but the properties we're trying to access don't actually exist yet, so let's add those in next. Open up the `appsettings.json` file, and, just before the closing curly brace at the bottom of the file, add the following JSON key/value pair:

```
"Authentication": {  
  "JwtKey": "ECOMMERCE_SUPER_SECRET_KEY",  
  "JwtIssuer": "http://localhost:5000",  
  "JwtAudience": "http://localhost:5000",  
  "JwtExpireDays": 30  
}
```

The `JwtKey` value is used here as the issuer signing key, and can be any string you like. However, if malicious users were to find out what this key was, they could forge tokens that would give them authenticated access to the system. Therefore, it is recommended to make this value fairly random and hard to guess, and some suggestions even go as far as to recommend changing it on a fairly frequent basis. At the very least, it should be stored in a secure fashion on any production environment—we'll see how to do this in a later chapter where we'll use environment variables to override it in production.

We've already discussed the `JwtIssuer` and `JwtAudience` values, but to reiterate, this is how we ensure that only tokens that were issued from our trusted domain are valid, as well as ensuring that client requests originate from the domain URL we set for the audience claim.

Finally, we set the expiration window of our JWTs to 30 days. You can obviously set this to whatever you like, depending on your preferences, but it's worth noting that, once a JWT has been issued, it cannot be invalidated. The longer the expiration, the bigger the risk if a malicious user gained access to a JWT that didn't belong to them, as they have a much longer period of time they can access the system for. The simple way of reducing that risk is to go the opposite way and reduce the length of the expiration period, but the trade-off is that users then have to log in more frequently.

A better, albeit more complicated, approach is to introduce an additional token called a **refresh** token, which as its name suggests is used to refresh an access token. The idea is that access tokens (what we're using currently) should have a short expiration value, usually measured in minutes, and once it expires, we use the refresh token to get a new one. Refresh tokens can have much longer expiration values, because they are stored in the database, and as such can be invalidated at will. We then have more control and can minimize the window in which a compromised access token can be used to access the system. We'll look at this in more detail in a later chapter when we introduce refresh tokens to make our API far more secure.

With these settings in place, the last step is to actually make use of the JWT bearer middleware by adding it to our middleware pipeline. Back in the `Startup.cs` file, locate the `Configure` method and add the following line:

```
//...rest of file as before
app.UseStaticFiles();

app.UseAuthentication();

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

Remember that ordering matters when configuring the middleware pipeline, so make sure you add it before adding the MVC middleware. At this point, we have everything we need to start preventing unauthenticated access to specific endpoints within our API, but we can't test it properly until we're also able to actually issue a JWT to use on authenticated endpoints.

Issuing JWTs

Ultimately, we need an endpoint for new users to register, as well as an endpoint for existing users to authenticate and retrieve a JWT in return. We already added a test user into the database as part of our seed data, so we can jump right into issuing JWTs to make sure our authentication configuration is working properly. Under the `Features` directory, create a new folder called `Authentication`, and then add a `Controller.cs` file with the following contents:

```
namespace ECommerce.Features.Authentication
{
    [Route("api/[controller]")]
    public class TokenController : Controller
    {
        private readonly SignInManager<AppUser> _signInManager;
        private readonly UserManager<AppUser> _userManager;
        private readonly IConfiguration _configuration;

        public TokenController(SignInManager<AppUser> signInManager,
            UserManager<AppUser> userManager, IConfiguration configuration)
        {
            _signInManager = signInManager;
            _userManager = userManager;
            _configuration = configuration;
        }
    }
}
```

All we've done so far is to declare the dependencies we'll need, namely the ASP.NET Identity `SignInManager<T>` and `UserManager<T>` classes for loading users from the database and validating the supplied password, as well as an instance of the `IConfiguration` interface, which we'll use for reading the app settings we need to create a JWT if authentication is successful. These classes are all available using the built-in DI container so we simply declare them as parameters to the controllers constructor.

Directly beneath this constructor, we'll add the one and only action method that we need on this controller, which will respond to a `HttpPost` request containing the email address and password of the user attempting to log in, and return a JWT if successful. This action method looks like this:

```
[HttpPost]
public async Task<IActionResult> GetToken([FromBody]
    LoginViewModel model)
{
    var errorMessage = "Invalid e-mail address and/or password";
```

```
    if (!ModelState.IsValid)
        return BadRequest(errorMessage);

    var user = await _userManager.FindByEmailAsync(model.Email);

    if (user == null)
        return BadRequest(errorMessage);

    if (await _userManager.IsLockedOutAsync(user))
        return BadRequest(errorMessage);

    var result = await _signInManager.PasswordSignInAsync(user,
        model.Password, true, true);

    if (!result.Succeeded)
        return BadRequest(errorMessage);

    var token = await GenerateToken(user);
    return Ok(token);
}
```

Web app security best practices suggest that, regardless of what the actual error is, if authentication fails, we should always return a standard error message to tell the user their email or password was incorrect. Therefore, we start this action method off by defining this error message variable before checking whether the model state is valid or not, and returning the error if validation fails. Next, we use the `UserManager<T>` class to look for a user in the database based on the supplied email address, returning our generic error message if we fail to find one. If a user is found, we have two different checks to make: we first check if the user is locked out, which occurs when they enter the wrong password five times in a row; and we use the `SignInManager<T>` class to validate the supplied password. If either check fails, we return the same generic error message in a `BadRequest` response. Finally, if all is well, we call a `GenerateToken` method, which we're yet to define, returning the generated token model within an `Ok` response.

Before we look at the `GenerateToken` method, you may have noticed the `LoginViewModel` class we're expecting as a parameter to this action method. This view model class looks like this:

```
namespace ECommerce.Features.Authentication
{
    public class LoginViewModel
    {
        [Required]
        public string Email { get; set; }

        [Required]
    }
}
```

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

The `GenerateToken` method we're using to create the JWT after a successful login attempt also belongs in the `Features/Authentication/Controller.cs` class and looks like this:

```
private async Task<TokenViewModel> GenerateToken(AppUser user)  
{  
    var claims = new List<Claim>  
    {  
        new Claim(JwtRegisteredClaimNames.Sub, user.UserName),  
        new Claim(JwtRegisteredClaimNames.Jti,  
            Guid.NewGuid().ToString()),  
        new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),  
        new Claim(ClaimTypes.Name, user.UserName)  
    };  
  
    var roles = await _userManager.GetRolesAsync(user);  
  
    foreach (var role in roles)  
    {  
        claims.Add(new Claim(ClaimTypes.Role, role));  
    }  
  
    var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes  
        (_configuration["Authentication:JwtKey"]));  
    var creds = new SigningCredentials(key,  
        SecurityAlgorithms.HmacSha256);  
    var expires = DateTime.Now.AddDays(Convert.ToDouble  
        (_configuration["Authentication:JwtExpireDays"]));  
  
    var token = new JwtSecurityToken(  
        _configuration["Authentication:JwtIssuer"],  
        _configuration["Authentication:JwtAudience"],  
        claims,  
        expires: expires,  
        signingCredentials: creds  
    );  
  
    return new TokenViewModel  
    {  
        AccessToken = new JwtSecurityTokenHandler().WriteToken(token),  
        AccessTokenExpiration = expires,  
        FirstName = user.FirstName,  
        LastName = user.LastName  
    };  
}
```


We start by creating a new list of `Claim` objects containing the standard claims that are expected in most JWT authentication implementations. We use the `AppUser.UserName` property for the `Sub` claim, and a GUID value for the `Jti` claim, which is used as a unique identifier for the token itself. We also use the `AppUser.Id` and `AppUser.UserName` properties for the `NameIdentifier` and `Name` claims, respectively.

Next, we retrieve a list of roles the user belongs to using the `UserManager<T>` class, then loop over these, and add them to the list of claims we just created. We're not yet using roles in our app, but we will be soon, so we'll cover on roles shortly.

The next few lines are used to create an instance of the `SigningCredentials` class, which is required to generate a JWT. First up, we create an instance of the `SymmetricSecurityKey` class using the signing key from the configuration that we discussed earlier, by converting it to a `byte[]` array using the `System.Encoding.UTF8` static class. We then pass this object into the constructor of the `SigningCredentials` class, specifying the `SecurityAlgorithms.HmacSha256` algorithm as the signature algorithm to apply.

The last piece of information we need to generate a JWT is the expiry date of the token. We determine this date by using the `JwtExpireDays` configuration setting that we defined earlier, adding the variable amount of days to the current timestamp. With this in place, we create an instance of the `JwtSecurityToken` class, passing in the `JwtIssuer` and `JwtAudience` configuration settings as discussed earlier, as well as the list of claims, expiry date, and signing credentials we just defined. Finally, we return an instance of a `TokenViewModel` class, which we are about to define, using the `JwtSecurityTokenHandler.WriteToken` method to write the `JwtSecurityToken` object we just created to a string. We also add the expiry date to this view model to make it easier for the client application to determine if a token is expired without needing to perform a HTTP request to the server. We could decode the token on the client and take the expiry from there, but this method is far simpler for now, and there are no downsides to doing it this way either.

The `TokenViewModel` class we just used is another simple class, and looks like this:

```
namespace ECommerce.Features.Authentication
{
    public class TokenViewModel
    {
        [JsonProperty("access_token")]
        public string AccessToken { get; set; }
        [JsonProperty("access_token_expiration")]
        public DateTime AccessTokenExpiration { get; set; }
        public string FirstName { get; set; }
    }
}
```

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

You'll notice that we've used the `JsonProperty` attribute to override the JSON serializer settings for these properties. If you've ever used OAuth or Open ID Connect before, you'll probably have noticed that they tend to use the underscore naming convention for variables, rather than camel case as we do with JSON generated by C# and .NET. In this instance, we're simply demonstrating how to override the default generated JSON property names should you wish to follow the OAuth naming conventions—feel free to omit these if you prefer, but remember to change the frontend appropriately as well.

Adding user role support

Before we can properly test our JWT setup, we need to introduce user roles into our system. Any roles a user is assigned to are included in the claims that we bake into the JWT itself, so we need to add a role to the test user we seeded earlier in order to properly test that our JWTs contain all the information we expect them to.

The first step is to modify our seed data to include the roles we'll need, which will simply be `Admin` and `Customer`. Open up the `Data/DbContextExtensions.cs` file, and right at the top, add the following new property:

```
public static RoleManager<AppRole> RoleManager { get; set; }
```

We then need to create a new static method to check if the roles we want already exist, and create them if they don't:

```
private static void AddRoles(EcommerceContext context)  
{  
    if (RoleManager.RoleExistsAsync("Admin").GetAwaiter()  
        .GetResult() == false)  
    {  
        RoleManager.CreateAsync(new  
            AppRole("Admin")).GetAwaiter().GetResult();  
    }  
  
    if (RoleManager.RoleExistsAsync("Customer")  
        .GetAwaiter().GetResult() == false)  
    {  
        RoleManager.CreateAsync(new  
            AppRole("Customer")).GetAwaiter().GetResult();  
    }  
}
```

This method now needs invoking from the `EnsureSeeded` method as we've done before, so update this as follows:

```
public static void EnsureSeeded(this EcommerceContext context)
{
    AddRoles(context);
    AddUsers(context);
    AddColoursFeaturesAndStorage(context);
    AddOperatingSystemsAndBrands(context);
    AddProducts(context);
}
```

Make sure this is invoked before the `AddUsers` method, as we're about to update this to add our existing user to the admin role. This will obviously fail if the roles haven't been added before the `AddUsers` method is invoked. The updated `AddUsers` method looks like this:

```
private static void AddUsers(EcommerceContext context)
{
    if (UserManager.FindByEmailAsync("stu@ratcliffe.io")
        .GetAwaiter().GetResult() == null)
    {
        var user = new AppUser
        {
            FirstName = "Stu",
            LastName = "Ratcliffe",
            Username = "stu@ratcliffe.io",
            Email = "stu@ratcliffe.io",
            EmailConfirmed = true,
            LockoutEnabled = false
        };

        UserManager.CreateAsync(user,
            "Password1*").GetAwaiter().GetResult();
    }
    var admin = UserManager.FindByEmailAsync("stu@ratcliffe.io")
        .GetAwaiter().GetResult();

    if (UserManager.IsInRoleAsync(admin,
        "Admin").GetAwaiter().GetResult() == false)
    {
        UserManager.AddToRoleAsync(admin, "Admin");
    }
}
```

As our database and test user already exist, the first `if` statement here will not run. Therefore, we need to find the user by email address, and then add it to the admin role if it doesn't already belong to it. Seeing as we're not yet production-ready, we could easily drop the entire database and modify the original `if` statement to also add the new user to the admin role at the point it's originally created. This would keep things simple and make most of the code we just added unnecessary. However, if our application was already in production with a live database, we wouldn't be able to do this, so it's worth thinking about how to make these kinds of changes when the database can't be dropped.

Finally, as we now have a dependency on the `RoleManager<T>` class, we need to provide an instance as we did with the `UserManager<T>` class before. In the `ConfigureServices` method of the `Startup.cs` class, find the following line:

```
DbContextExtensions.UserManager =  
services.BuildServiceProvider().GetService<userManager<AppUser>>();
```

And change it to the following:

```
var provider = services.BuildServiceProvider();  
DbContextExtensions.UserManager =  
provider.GetService<userManager<AppUser>>();  
DbContextExtensions.RoleManager =  
provider.GetService<RoleManager<AppRole>>();
```

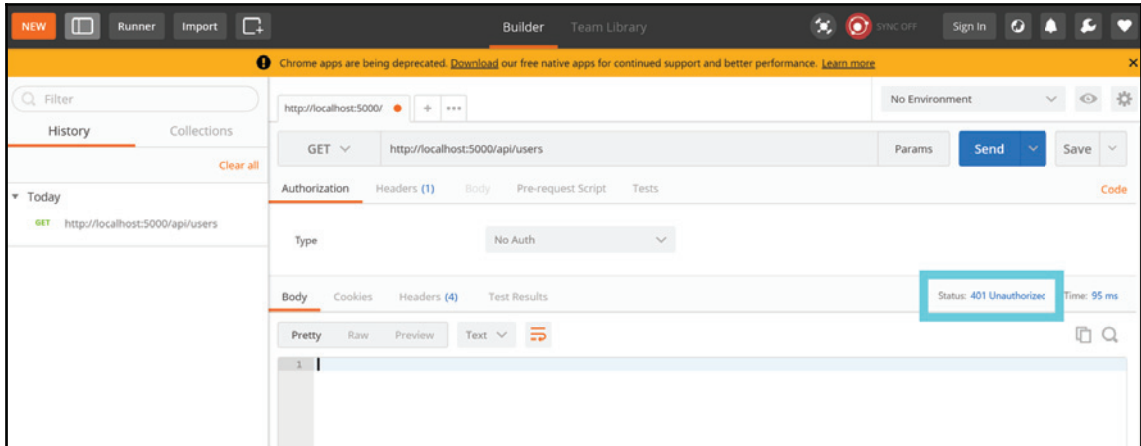
As we now have two dependencies to provide, we only build the service provider once and reuse it for both dependencies.

Testing JWT authentication

Our JWT authentication setup and configuration is complete, so let's give our API endpoints a quick test and make sure everything is working properly. Currently, none of our controllers require authentication, so let's add an `[Authorize]` attribute to the `Features/Users/UsersController.cs` file and use it for testing our setup again. Modify the controller declaration as follows:

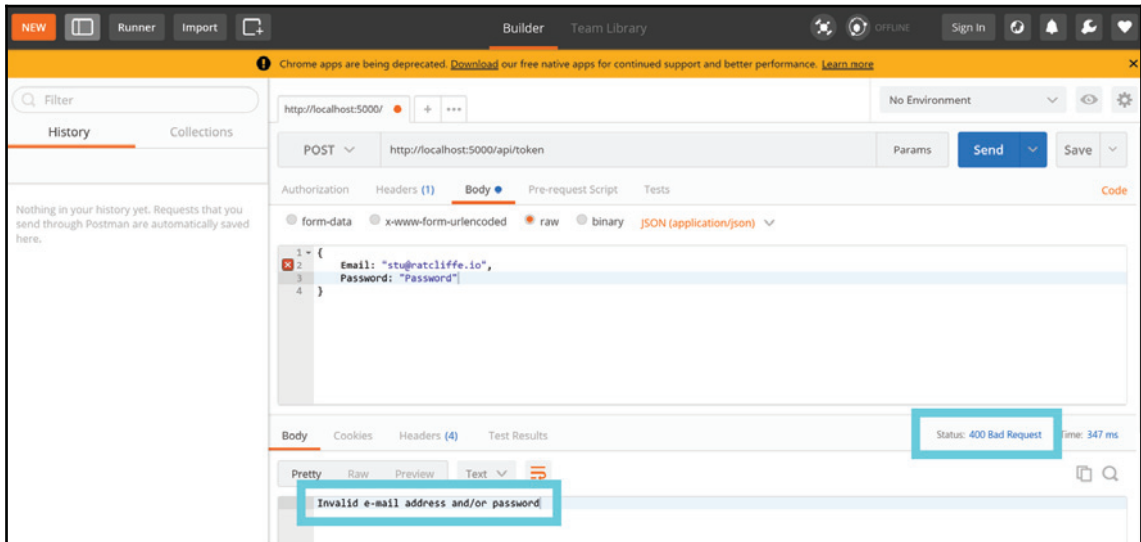
```
[Authorize]  
[Route("api/[controller]")]  
public class UsersController : Controller  
{  
    // rest of class unchanged  
}
```

If we restart the application now, we can test that this is having the desired effect using Postman again. Performing a HTTP GET request to `localhost:5000/api/users` yields the following 401 HTTP response code:



Testing anonymous access with Postman

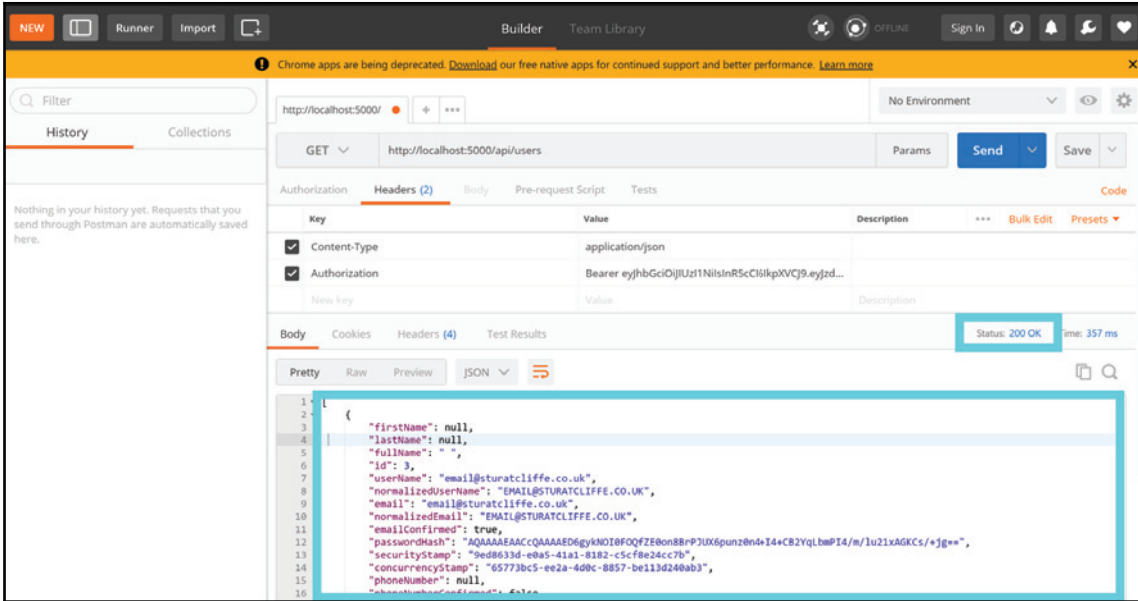
Now we'll try an invalid request to our token endpoint to make sure it doesn't return a JWT if we get the password wrong:



Testing invalid login credentials with Postman

Next, a valid request to the same endpoint to retrieve a JWT using a valid email/password combination based on the details we used in the seed data:

All looking good so far, so let's now attach this token to the Authorization header of the previously failing GET request to `http://localhost:5000/api/users` and see if we can now retrieve our list:



Testing authenticated access with Postman

And there we have it; JWT authentication is up and running.

User registration

Being able to authenticate is all very well and good, but so far the only user account we have to authenticate is the admin account we seeded for ourselves. Let's change this by creating an API endpoint for our customers to register for new user accounts.

Create a new `Features/Account` directory and add a standard controller class like so:

```
namespace ECommerce.Features.Account
{
    [Route("api/[controller]")]
    public class AccountController : Controller
    {
        private readonly UserManager<AppUser> _userManager;
```



```
public AccountController(UserManager<AppUser> userManager)
{
    _userManager = userManager;
}
}
```

Within this controller, we need a single action method for registering new users:

```
[HttpPost]
public async Task<IActionResult> Register([FromBody]
RegisterViewModel model)
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState);

    var user = await _userManager.FindByEmailAsync(model.Email);

    if (user != null)
        return BadRequest("A user with that e-mail address
already exists!");

    user = new AppUser
    {
        FirstName = model.FirstName,
        LastName = model.LastName,
        Email = model.Email,
        EmailConfirmed = true,
        UserName = model.Email,
        LockoutEnabled = true
    };

    var registerResult = await _userManager.CreateAsync(user,
model.Password);

    if (!registerResult.Succeeded)
        return BadRequest(registerResult.Errors);

    await _userManager.AddToRoleAsync(user, "Customer");

    return Ok();
}
```

There's nothing particularly new going on here. We use methods from the `userManager<T>` class that we've used before to search for existing users by email address, and we do a number of validation checks along the way. The difference here from the token controller we just saw is that we actually return specific error messages depending on what's gone wrong. The only other thing to note is that we're setting the `AppUser.EmailConfirmed` property to `true` for all new users. In a real application, I'd always enable email verification, and leave this property set to its default `false` value, but to keep things simple we won't be going down that route here.

The `RegisterViewModel` class that we receive as a parameter looks like this:

```
namespace ECommerce.Features.Account
{
    public class RegisterViewModel
    {
        [Required]
        public string FirstName { get; set; }
        [Required]
        public string LastName { get; set; }

        [Required]
        [EmailAddress]
        public string Email { get; set; }

        [Required]
        public string Password { get; set; }

        [Required]
        [Compare(nameof(Password))]
        public string ConfirmPassword { get; set; }
    }
}
```

You can test this API endpoint in much the same way as we did with the previous one. Try sending requests with missing parameters, invalid data types, and finally with valid data to ensure everything is working correctly.

Authentication and user registration in the client app

We now have fully functional API endpoints to enable users to register for an account, and then authenticate themselves using their email address and password. However, this is only half of the problem, as we still need to build a UI to perform these actions. We'll start by extending the Vuex store that we built in the last chapter to include the actions, mutations, and getters necessary to perform the API calls to the server, and store the authentication status of our users.

Vuex state properties for authentication

The first step is very simple—we need a few additional properties added to the `state` object within our Vuex store. Open up the `ClientApp/store/index.js` file and modify the `state` object as follows:

```
state: {  
  auth: null,  
  showAuthModal: false,  
  loading: false,  
  cart: []  
}
```

Starting at the top, we're going to use the `auth` property to store a simple object that will contain the JWT access token and expiry date properties that we return from the login action of our API. As its name suggests, the `showAuthModal` property has a Boolean value to control the display of a modal component that we're going to use for housing the login and register forms. Finally, we'll use the `loading` Boolean value to track when an API request is in progress so we can do things like disabling buttons, or showing loading icons.

Vuex mutations for managing authentication state

The next thing we need is a set of Vuex mutations for manipulating the `state` properties we just created. All of the following mutation functions belong in the `ClientApp/store/mutations.js` file.

The first two mutations we need are for setting the visibility of the modal we'll be building later:

```
export const showAuthModal = state => {
  state.showAuthModal = true;
};

export const hideAuthModal = state => {
  state.showAuthModal = false;
};
```

Next, we need a set of mutations to track and store the status and outcome of a login API request:

```
export const loginRequest = state => {
  state.loading = true;
};

export const loginSuccess = (state, payload) => {
  state.auth = payload;
  state.loading = false;
};

export const loginError = state => {
  state.loading = false;
};
```

This is a common pattern in Vuex when performing API requests. Before performing the AJAX call, it's common to commit a mutation to signify the start of the request, which in this case sets the `loading` property to `true`. Then, depending on whether the API request returns a success or failure result, we commit a success/error mutation to signify the end of the request. In this case, if the API call is successful, we simply assign the response object to the `auth` property, and then set the `loading` property to `false`. If the request fails, all we do is set the `loading` property to `false`.

We'll also need a similar set of mutations to track the user registration API request:

```
export const registerRequest = state => {
  state.loading = true;
};

export const registerSuccess = state => {
  state.loading = false;
};

export const registerError = state => {
  state.loading = false;
};
```

It may seem like overkill to use three mutations here rather than two, seeing as both the `registerSuccess` and `registerError` mutations are identical. However, in a more complicated application, it is common to perform some additional state changes in each one, and it's also more explicit when we check the mutation history using the Vue devtools in Chrome.

Finally, we need a single mutation to commit when the user logs out, in order to clear the `auth` object:

```
export const logout = state => {
  state.auth = null;
};
```

Vuex authentication getters

We only need a single Vuex getter to determine if the current user is authenticated or not. Open up the `ClientApp/store/getters.js` file and add the following exported function:

```
export const isAuthenticated = state => {
  return (
    state.auth !== null &&
    state.auth.access_token !== null &&
    new Date(state.auth.access_token_expiration) > new Date()
  );
};
```

In order to tell if a user is authenticated, we first need to check that the `auth` property is not null, and that it has an `access_token` property set. Finally, we check that the `access_token_expiration` property is a valid date in the future. If any of these checks fail, either the user is unauthenticated, or their access token has expired.

Vuex login, register, and logout actions

The final step in extending our Vuex store is to add the login and register actions that actually coordinate the API requests to the server, as well as committing the relevant mutations. These belong in the `ClientApp/store/actions.js` file, and the first of these is the login action:

```
export const login = ({ commit }, payload) => {
  return new Promise((resolve, reject) => {
    commit("loginRequest");
    axios
      .post("/api/token", payload)
      .then(response => {
        const auth = response.data;
        axios.defaults.headers.common["Authorization"] = `Bearer ${
          auth.access_token
        }`;
        commit("loginSuccess", auth);
        commit("hideAuthModal");
        resolve(response);
      })
      .catch(error => {
        commit("loginError");
        delete axios.defaults.headers.common["Authorization"];
        reject(error.response);
      });
  });
};
```

This action returns a promise, so that, when we dispatch it from a component, we can wait for it to resolve before performing any additional component-level operations. We start by committing the `loginRequest` mutation that we talked about earlier, before performing an HTTP POST request to the token controller using `axios`. `axios` requests also return promises, so we can wait for them to resolve successfully using a `.then(response)` block.

In the success block in this case, we retrieve the `access_token` from the response and attach it to the authorization header in the `axios.defaults.headers.common` property. From this point on, all HTTP requests performed using `axios` will have the bearer token set so that the request can be authenticated by our server-side API. After this, we commit the `loginSuccess` and `hideAuthModal` mutations, passing the `auth` object that we retrieved from the response as the payload for the former. We finish the success block by calling the `resolve` function of the returned promise, passing it the entire `response` object for maximum flexibility.

In order to handle request failures, we can use a `.catch(error)` block, which will be triggered by the `reject` function of the promise returned by the `axios` request. In this instance, we commit the `loginError` mutation, delete the authorization header from future `axios` requests, and then call our own `reject` method of the promise being returned by the action. The user registration action is very similar, albeit slightly simpler:

```
export const register = ({ commit }, payload) => {
  return new Promise((resolve, reject) => {
    commit("registerRequest");
    axios
      .post("/api/account", payload)
      .then(response => {
        commit("registerSuccess");
        resolve(response);
      })
      .catch(error => {
        commit("registerError");
        reject(error.response);
      });
  });
};
```

As with the login action, we return a promise, which we either resolve or reject based on the response of an API call performed using `axios`. We commit the `registerRequest` mutation before triggering the API call, the `registerSuccess` mutation if it succeeds, and the `registerError` mutation if it fails.

The final action we need is the logout action, which in comparison to the previous two is incredibly simple:

```
export const logout = ({ commit }) => {
  commit("logout");
  delete axios.defaults.headers.common["Authorization"];
};
```

All we need to do is commit the `logout` mutation in order to clear the authentication state, and then delete the authorization header from the default `axios` configuration used by any subsequent HTTP requests. This completes our Vuex store changes, so we are ready to start building out the components we need to trigger these actions.

Authentication modal component

As previously discussed, rather than building out full login and register pages, including the `VueRouter` configuration that goes with them, we're going to put our login and register forms into separate tabs within a modal. As this component doesn't really belong to any single page, it doesn't belong in any of our sub-directories beneath `ClientApp/components` either. Instead, create a `ClientApp/components/app` directory to hold all of our app-level components, and then create an `AuthModal.vue` component file inside. The `template` section for this component looks like this:

```
<template>
  <b-modal v-model="show" hide-header hide-footer
    no-close-on-backdrop no-close-on-esc>
    <b-tabs v-model="index">
      <b-tab title="Login" active>
        <login-form :registered="registered" @close="close" />
      </b-tab>
      <b-tab title="Register" >
        <register-form @success="success" @close="close" />
      </b-tab>
    </b-tabs>
  </b-modal>
</template>
```

The root-level element here is the modal component from the `Bootstrap-Vue` library. To control visibility of this component, it expects a Boolean property to be bound to the `v-model` directive, so we're stating here that we'll need some kind of `show` property with a Boolean value. We're also passing a number of Boolean props to this component, such as `hide-header` and `no-close-on-backdrop`. All we're doing here is preventing the default Bootstrap modal header/footer sections from appearing, and preventing the modal from being closed if the user presses the *Escape* key or clicks away from the modal itself. We need full control over the closing of this modal so we can decide what to do based on the authentication status of the user.

Inside the modal component, we're nesting two more components from `Bootstrap-Vue`: the `b-tabs` and `b-tab` components. The tabs component optionally accepts an integer value for the `v-model` directive, which we can use to programmatically switch tabs—we'll need to do this after a user successfully registers and needs to log in. This doesn't prevent the user from manually switching tabs by clicking the headers; it just gives us extra flexibility.

We then render two `b-tab` components, one for the login form and one for the register form, each of which is rendered as a custom component that we're yet to define. By default, the login tab is active when the modal is first displayed. The `login-form` component takes a `registered` prop, which we'll use to show a success message to notify the user that their registration was successful. We also listen for a `close` event and invoke a `close` method, which we'll define in a moment. The `register-form` component emits a `success` event, which we'll listen for and programmatically switch to the login tab, and a `close` event that will also trigger the `close` method.

The `script` section of this component is a little more complicated, so we'll go over it in sections. We start by importing the login and register form components:

```
import LoginForm from "./LoginForm.vue";
import RegisterForm from "./RegisterForm.vue";
```

These don't actually exist yet, so don't worry if you start seeing errors in your Terminal or browser console. Next, we need to do our standard practice of exporting a default object which will be our component definition:

```
export default {
  name: "auth-modal",
  components: {
    LoginForm,
    RegisterForm
  }
}
```

All we've done so far is give the component a name, and declare the two custom components we just imported as child components of this one. We still need to add a fair few pieces of behavior though, starting with the `show` property, which controls the visibility of this modal. We'll use the following prop declaration for this:

```
props: {
  show: {
    type: Boolean,
    required: true
  }
}
```

We also need an `index` data property for controlling the active tab, and a `registered` Boolean property, which we pass down into the `login-form` component to show a success message after a user registers for an account:

```
data() {
  return {
```

```
    index: 0,  
    registered: false  
  };  
}
```

And, finally, we need the `success` and `close` methods to manipulate these data properties:

```
methods: {  
  success() {  
    this.registered = true;  
    this.index = 0;  
  },  
  close() {  
    this.$store.commit("hideAuthModal");  
    let query = Object.assign({}, this.$route.query);  
    delete query.redirect;  
    this.$router.push({ query: query });  
  }  
}
```

The `success` method is fairly self-explanatory. Remembering that this gets called after a new user successfully registers for an account, we set the `registered` property to `true` and the `index` property to `0` in order to switch tabs back to the login tab. The user will then be presented with a successful registration confirmation message and asked to log in to continue.

The `close` method is a little more complicated, but this is actually very similar to the code we've seen before, when building out the product catalog. We start by committing the `hideAuthModal` mutation to close the modal, and then use the `Object.assign` function to clone the current query string object representation. We then delete the `redirect` property if it exists, before pushing the modified `query` object back onto the query string, as we've done before. This won't make much sense yet, but, as we'll be preventing navigation to protected routes if the user isn't authenticated, we'll use the query string to track the URL to redirect to, just like we would with a full fat login page implementation. However, unlike with a full page for logging in, where the user would navigate back to their previous page if they didn't want to complete the login process, we need to manually delete the `redirect` parameter from the query string if they close out of the modal rather than proceeding to log in.

Login form component

The `LoginForm.vue` component also lives under the `ClientApp/components/app` directory, and has the following template section:

```
<template>
  <form @submit.prevent="login" class="p-2">
    <b-alert variant="danger" :show="error !== null" dismissible
      @dismissed="error = null">
      {{ error }}
    </b-alert>
    <b-alert variant="success" :show="registered &&
      error === null">
      Registration successful. Please login to continue.
    </b-alert>
    <p>Login with your e-mail address and password.</p>
    <b-form-group label="E-mail">
      <b-form-input v-model.trim="email" />
    </b-form-group>
    <b-form-group label="Password">
      <b-form-input v-model.trim="password" type="password" />
    </b-form-group>
    <b-form-group>
      <b-button variant="primary" type="submit"
        :disabled="loading">Login</b-button>
      <b-button variant="default" @click="close"
        :disabled="loading">Cancel</b-button>
    </b-form-group>
  </form>
</template>
```

We start by rendering a form where we prevent the default `submit` action, choosing to invoke a `login` method instead. We then render two (initially hidden) alert boxes, one for displaying an error message if the form submission fails, and one that shows the successfully registered message if the `registered` prop is set to `true`. The error alert is configured as *dismissible*, which allows us to click a Close icon to hide the alert once we've seen it. As it's hidden, it emits a `dismissed` event, which we use to clear the `error` property. Both of these alert boxes have their visibility controlled by a `show` prop, which in the instance of the error alert only resolves to `true` if the `error` property is anything other than `null`, and for the success alert resolves to `true` if the `registered` prop is `true` and the `error` property is `null`.

The rest of this form is very simple, consisting of three Bootstrap form groups for the email address input, password input, and action buttons, respectively. The text and password inputs use the Bootstrap-Vue's `b-form-input` component, using the `v-model.trim` directive to bind the values to appropriate data properties, as well as automatically trimming any additional whitespace entered by mistake. Both of the action buttons have their `disabled` prop bound to a local `loading` property, which we'll bind up to our Vuex store state shortly.

Again, the `script` section for this component is more complicated, so we'll go over it one piece at a time:

```
<script>
export default {
  name: "login-form",
  props: {
    registered: {
      type: Boolean,
      required: false
    }
  }
}
</script>
```

As always, we've started with a default component export, specifying the component name and the props that we expect to be passed in. In this case, it's the single `registered` prop that controls the alert box visibility, as discussed previously. The local component state that we require includes the `email` and `password` properties that we bind to the form inputs, as well as an `error` property for any validation errors:

```
data() {
  return {
    email: "",
    password: "",
    error: null
  };
}
```

We're also binding the `disabled` state of the action buttons to a `loading` property, which we need to fetch from our Vuex store, so we'll use a `computed` property for this:

```
computed: {
  loading() {
    return this.$store.state.loading;
  }
}
```

We need two functions in the `methods` object of this component, the first of which is the `login` function, which performs the main action of the entire component:

```
login() {
  const payload = {
    email: this.email,
    password: this.password
  };

  this.$store
    .dispatch("login", payload)
    .then(response => {
      this.error = null;
      this.email = "";
      this.password = "";

      if (this.$route.query.redirect) {
        this.$router.push(this.$route.query.redirect);
      }
    })
    .catch(error => {
      this.error = error.data;
    });
}
```

We start by constructing a `payload` object consisting of the `email` and `password` properties, which we need on the API action method to authenticate the user. We then dispatch the Vuex `login` action that we defined earlier, passing it the `payload` object we just created. This is where we see the benefit of returning a promise from Vuex actions, as we can now wait for it to complete before doing different operations, depending on whether it succeeded or failed. If the action succeeds, the `.then()` block will be run, where we clear out the user input fields and error property, and then redirect to the URL in the query string if one exists. If the action fails, the `.catch()` block is run, where we simply assign the error data from the server to the local `error` property.

The second function we need is the `close` function, which gets called when the user exits the form rather than submitting it:

```
close() {
  this.$emit("close");
}
```

All we need to do here is emit the `close` event and let the parent `AuthModal` component handle the `close` logic.

Register form component

The RegisterForm.vue component lives in the same directory, is very similar to the login form component we just discussed, and, as such, does not need much explanation. The template section looks like this:

```
<template>
  <form @submit.prevent="submit" class="p-2">
    <b-alert variant="danger" :show="errors !== null" dismissible
      @dismissed="errors = null">
      <div v-for="(error, index) in errors" :key="index">
        {{ error[0] }}
      </div>
    </b-alert>
    <b-form-group label="First Name">
      <b-form-input v-model.trim="firstName" />
    </b-form-group>
    <b-form-group label="Last Name">
      <b-form-input v-model.trim="lastName" />
    </b-form-group>
    <b-form-group label="E-mail">
      <b-form-input v-model.trim="email" />
    </b-form-group>
    <b-form-group label="Password">
      <b-form-input v-model.trim="password" type="password" />
    </b-form-group>
    <b-form-group label="Confirm Password">
      <b-form-input v-model.trim="confirmPassword"
        type="password" />
    </b-form-group>
    <b-form-group>
      <b-button variant="primary" type="submit"
        :disabled="loading">Register</b-button>
      <b-button variant="default" @click="close"
        :disabled="loading">Cancel</b-button>
    </b-form-group>
  </form>
</template>
```

As with the login form, we start with an alert box that will display if the form submission fails in order to feed back to the user what the validation errors were. At first glance, it looks like we're looping over an array of errors inside this alert box, but we're actually looping over the keys of an object. The value that corresponds to those keys is an array of error messages, so, in order to keep things simple, we just display the first message in the array. This error message structure is the default for ASP.NET Core MVC validation, which is why we need to handle things in this way. The rest of this template is much the same as the login form: a group of Bootstrap form input elements, and a pair of action buttons to submit the form or cancel and close the modal.

The bulk of the `script` section of this component is also very similar to the login form component we just looked at:

```
<script>
export default {
  name: "register-form",
  data() {
    return {
      firstName: "",
      lastName: "",
      email: "",
      password: "",
      confirmPassword: "",
      errors: null
    };
  },
  computed: {
    loading() {
      return this.$store.state.loading;
    }
  }
};
</script>
```

Where things get slightly different is in the main `submit` function within the `methods` object:

```
submit() {
  const payload = {
    firstName: this.firstName,
    lastName: this.lastName,
    email: this.email,
    password: this.password,
    confirmPassword: this.confirmPassword
  };
};
```

```
    this.$store
      .dispatch("register", payload)
      .then(response => {
        this.errors = null;
        this.firstName = "";
        this.lastName = "";
        this.email = "";
        this.password = "";
        this.confirmPassword = "";
        this.$emit("success");
      })
      .catch(error => {
        if (typeof error.data === "string" || error.data
            instanceof String) {
          this.errors = { error: [error.data] };
        } else {
          this.errors = error.data;
        }
      });
  }
}
```

As we did with the login method, we construct a `payload` object to match the parameters we need for the user registration API action method, and pass it to a Vuex `dispatch` call. The `success` block is also very similar in that it clears out the local state properties, but also emits the `success` event so the parent component can switch tabs to the login form. The main difference is in the `error` block. As we've kept our API very simple, it can return either a single string value as the error message, or an object with a collection of arrays of error messages.

As such, we first determine if the response is a string or not, and then make sure the structure we assign to the local `errors` property is the same so that the UI displays the correct data. If the response is a single string value, we create an anonymous object with a single `error` key, where the value is an array containing the single error message returned in the `error.data` property.

As with the login form component, we also need a `close` method to emit the `close` event:

```
close() {
  this.errors = null;
  this.$emit("close");
}
```


Auth navigation item component

Our `AuthModal` component is now complete, but we have no way of opening it to test it out. We're going to build a simple component that will reside in the navigation menu, and either display a login/register link, or a drop-down menu with account-specific links, depending on the authentication status of the user.

Create a `ClientApp/components/app/AuthNavItem.vue` file, and give it a `template` section that looks like this:

```
<template>
  <b-nav-item-dropdown v-if="isAuthenticated" right>
    <template slot="button-content">
      <i class="fas fa-user"></i>
      {{ fullName }}
    </template>
    <b-dropdown-item @click="logout">
      <i class="fas fa-sign-out-alt"></i>
      Logout
    </b-dropdown-item>
  </b-nav-item-dropdown>
  <b-nav-item v-else @click="login">
    <i class="fas fa-user"></i>
    Login / register
  </b-nav-item>
</template>
```

Note how we actually have two root-level elements in this template. Recall [Chapter 1, Understanding the Fundamentals](#), where we looked at the fundamentals of Vue; we can only have a single root element in a component's `template` section. The trick here is that we're using the `v-if` and `v-else` directives on these elements, meaning only one will ever be rendered, which meets the requirements of the Vue template engine.

If the user is authenticated, we display a drop-down `nav item`, where the toggle button is a Font Awesome user icon, and if not, it simply contains a single logout link. If the user is not yet logged in, we display the login/register link, which we'll use to display the modal component we just created.

The `script` section for this component looks like this:

```
<script>
export default {
  name: "auth-nav-item",
  computed: {
    isAuthenticated() {
      return this.$store.getters.isAuthenticated;
    }
  }
}
```

```
    },
    fullName() {
      return `${this.$store.state.auth.firstName} ${
        this.$store.state.auth.lastName
      }`;
    }
  },
  methods: {
    login() {
      this.$store.commit("showAuthModal");
    },
    logout() {
      this.$store.dispatch("logout").then(() => {
        if (this.$route.meta.requiresAuth) {
          this.$router.push("/");
        }
      });
    }
  }
};
</script>
```

Most of this is fairly standard stuff by this point, so should be fairly self-explanatory. We have the `isAuthenticated` computed property, which hooks directly into the Vuex getter of the same name, and determines whether to show the login/register link or the user account drop-down menu in the preceding template. We also have the `fullName` computed property, which simply concatenates the logged-in user's first and last names to display in the navbar at the top of the page. We then specify two methods, one for each of the links we could be displaying, depending on authentication status.

The `login` method quite simply commits the `showAuthModal` mutation, which we defined earlier, to control the visibility of the login modal we just created. The `logout` method is a little more complex, and may not look particularly familiar yet. As we've done before, we dispatch a Vuex action, which we're expecting to return a promise (which it does) as we chain on a `.then()` block to process the successful response. We don't have to have a corresponding `.catch()` block unless we need to handle the error, so in this case we're keeping things simple. We also have no use for whatever parameter is passed to the `success` block, so we use a pair of empty parentheses for the first part of the fat arrow syntax callback function that we declare like this: `.then(() => { //..process the response })`.

The next part is where we've changed things quite a bit by checking the `this.$route.meta.requiresAuth` property. We'll see how to do this in a moment, but for now, you just need to know that we can assign any arbitrary values we'd like to the `meta` property of a VueRouter's `route` object. In this case, if a particular route requires authentication, we set the `requiresAuth` meta property to `true`. Now that we can tell if the page the user is currently on is locked down to authenticated users, we can decide whether or not to redirect them back to the home page after they've logged out.

Wiring up the new components in App.vue

With these two new components in place, we need to wire them up so that they're actually displayed and can be interacted with. We have a few changes to make in the `ClientApp/components/App.vue` file, starting by importing our new components at the top of the `script` section:

```
import AuthNavItem from "./app/AuthNavItem.vue";
import AuthModal from "./app/AuthModal.vue";
```

Then we declare them as child components:

```
components: {
  AuthNavItem,
  AuthModal,
  CartSummary
}
```

We also need a computed property, which checks the visibility of the modal in the store:

```
computed: {
  showAuthModal() {
    return this.$store.state.showAuthModal;
  }
}
```

Finally, in the `template` section, we need to render the new components in the appropriate locations. Start by finding the `<cart-summary />` element, and, directly beneath it, add the following:

```
<auth-nav-item />
```

Then, at the bottom of the template, before the final closing `div` tag, add the following:

```
<auth-modal :show="showAuthModal" />
```

This is enough to get us started with testing our UI changes. If you refresh the page now, you should see the login/register link in the upper-right corner of the screen. Try logging in with the credentials we used earlier, and make sure that, once you have logged in, the UI refreshes to show the drop-down menu item with a nested logout link. Similarly, try logging out and make sure that the UI refreshes back to showing the login/register link. You can also try registering for a new user account and make sure the login page is displayed with the appropriate success message if registration was successful.

Protecting pages with navigation guards

At this point, we can open the login modal manually, but we don't yet have any protected routes to prevent access until the user has logged in. Let's remedy this by creating an empty checkout page ready for the next chapter, when we start processing payments. Create a `ClientApp/pages/Checkout.vue` component with a barebones `template` section as follows:

```
<template>
  <b-container class="page pt-4">
    <h1>Checkout</h1>
  </b-container>
</template>
```

Next, open up `ClientApp/boot.js` and, right beneath the other page component imports, add the following:

```
import Checkout from "../pages/Checkout.vue";
```

Finally, in the same file, change the `routes` array to include a route definition for this new checkout page:

```
const routes = [
  { path: "/products", component: Catalogue },
  { path: "/products/:slug", component: Product },
  { path: "/cart", component: Cart },
  { path: "/checkout", component: Checkout, meta:
    { requiresAuth: true } },
  { path: "*", redirect: "/products" }
];
```

As promised earlier, this is where/how we add meta properties to route definitions. We'll see how to actually prevent unauthorized access to this in just a moment, but with this convention all we need to do to prevent unauthorized access is to add the `requiresAuth` meta property to a route definition like this.

A little further down in this same file, we have a `router.beforeEach` function hook configured to start the page transition loading animation. We can hook into this same function to check the meta properties of the page being requested, and ensure that the user is authenticated if it's a page that requires it. Change this function to look like this:

```
router.beforeEach((to, from, next) => {
  NProgress.start();
  if (to.matched.some(route => route.meta.requiresAuth)) {
    if (!store.getters.isAuthenticated) {
      store.commit("showAuthModal");
      next({ path: from.path, query: { redirect: to.path } });
    } else {
      next();
    }
  } else {
    next();
  }
});
```

This might look pretty scary at first, but if we step through it, it should be fairly easy to grasp what's going on here. As previously discussed, in this hook we receive a `to` parameter, which corresponds to the route we're about to change to, and a `from` parameter for the current route we're navigating away from. What we've not talked about previously is that there could be multiple routes matched by the path we push to the router, and these are all available in the `to.matched` array. As such, we first use the built-in `Array.some` function to check if some of those matched routes have the `requiresAuth` meta property set to `true`. The syntax of this function probably looks pretty familiar, as it's similar to the `LINQ Any` function in C#, which performs the same action.

Once this first line makes sense, the rest of the function should as well. If none of the matched routes require authentication, or the user is already authenticated, we allow the navigation to continue by calling the `next` parameter function. Otherwise, we display the login modal by committing the `showAuthModal` mutation. Finally, we override the default call to `next` by instructing it to navigate to a route with the current URL path, as well as setting the `redirect` query parameter to the path of the route we were trying to access. This `redirect` parameter is what we then use after a successful login to decide where to send the user.

To test that everything is working correctly, open up the `ClientApp/pages/Cart.vue` file, locate the checkout button markup in the `template` section, and modify it to include a `click` handler like so:

```
<b-button variant="success" @click="checkout">
  Checkout
  <i class="fas fa-chevron-right"></i>
</b-button>
```

Then, add the following `checkout` function to the `methods` object in the `script` section:

```
checkout () {
  this.$router.push("/checkout");
}
```

Refresh the application again now. Ensure you are logged out and have at least one product added to your shopping cart, then navigate to the shopping cart page and click on the checkout button to make sure you are forced to log in.

Setting the authentication state on app startup

At this point, you might think we are done. However, if you try refreshing the browser after navigating to the checkout page, you'll probably notice some unexpected behavior, as you'll be asked to log in again. There is also another far less obvious issue in that, after refreshing the page, the default `axios` headers will be cleared, including the bearer token we attached to the authorization header. Currently, the only way this header is set is after a successful login request—not something that the user should have to do after every page refresh. This also means that any future API requests would fail if those endpoints required authentication.

To fix the `axios` configuration issue, we need to check if the user is authenticated at the earliest point of the app startup, and set the authorization header if they are. This fix will also lead us onto the right path for solving the authentication state issue as well—the problem we're seeing is due to the current place that we're initializing our Vuex store if it already exists in local storage.

We can fix both of these issues in one go by adding the following block of code in the appropriate location of the `ClientApp/boot.js` file. Find the existing `routes` array, and then add the following code directly *above* it:

```
import axios from "axios";

const initialState = localStorage.getItem("store");

if (initialStore) {
  store.commit("initialise", JSON.parse(initialStore));

  if (store.getters.isAuthenticated) {
    axios.defaults.headers.common["Authorization"] = `Bearer ${
      store.state.auth.access_token
    }`;
  }
}
```

As we'll be setting the default `axios` headers here, we first need to import `axios` itself. We then check local storage for any initial state that we configured to be persisted on every change to the store itself. If it exists, we commit the `initialise` mutation that sets this state. We can then use the `isAuthenticated` getter to check if the user is already authenticated, and if they are, we attach the access token in much the same way as we do after a successful login attempt.

You might be thinking that this code looks incredibly similar to what we originally put inside the `initialise` mutation itself, minus the `axios` configuration part, and you'd be right. However, we're not currently committing this mutation until the `beforeCreate` lifecycle hook of our main `App.vue` component. The issue is that the navigation guard we set up on our routes has already run before this hook does, so it thinks we're not authenticated when we actually are, and, as such, we get directed to the login modal when we shouldn't. By committing this mutation before the router navigation guard is even declared, we ensure that our store is initialized before the router decides if we're authenticated or not.

The mutation itself can now be simplified as well. It currently checks local storage for our initial state, and only performs an action if it finds it. Mutations shouldn't contain any logic, so if we check our mutation history it should be explicitly obvious what changes have been made. In this case, even though the mutation is always fired, it doesn't necessarily make a change to the state. We're now checking local storage for initial state outside of the mutation, and only commit it if we find it, so the mutation itself can be simplified to the following:

```
export const initialise = (state, payload) => {  
  Object.assign(state, payload);  
};
```

And, seeing as we're now committing it from the `ClientApp/boot.js` file, we no longer need the `beforeCreate` hook in `ClientApp/components/App.vue`, and as such it can be deleted.

This concludes our authentication and user registration feature, so make sure to give the application a good test to make sure everything is working properly. Refreshing the application should also no longer leave you as an unauthenticated user if you've previously logged in.

Summary

Authentication is one area of building modern web applications that is made considerably more complicated by the separation of server-side API and client side SPA frontends. It's no longer a fairly simple case of using a cookie-based approach where the browser will automatically attach the appropriate cookies to every HTTP request by default. However, more and more applications are being developed in conjunction with other external applications and services, so moving down the route of token-based authentication using JWTs adds a great deal of flexibility to make the additional complications more beneficial to us.

In this chapter, we started out by looking at what's involved with setting up and configuring JWT-based authentication in an ASP.NET Core web application. We also added the necessary API endpoints for issuing JWTs when users successfully log in, as well as one for users to register for a new account. We then moved on to the client-side of the application and expanded our existing Vuex store to include the necessary actions, mutations, and getters necessary to consume these API endpoints and store the returned JWT in the user's browser for authenticating future requests.

We then built a custom authentication modal component that contains the HTML login and user registration forms that our users need to interact with the API. For now, we are leaving form validation to the server side of the application only, so we looked at a simple way of displaying server-side validation errors in our client-side components. We also built a second custom component for displaying either a login or logout link in the navbar, depending on the authentication state of the user.

We added an empty checkout page in preparation for the following chapter on payment processing, and locked it down to access by authenticated users only using VueRouter navigation guards and route meta properties. We also configured our application's router to trigger the login modal if an unauthenticated user attempts to access a protected page. Finally, we configured our app startup process to check if the user has already retrieved a valid access token before attempting to process a page request, so that they don't get forced to log in again every time they do a full browser refresh.

With authentication in place, we're now in a position to allow our users to start placing their orders and processing their payments. In the next chapter, we'll look at how to process payments using the brilliant Stripe payment service, which helps us to remain PCI-compliant with an incredibly simple-to-use library.

8 Processing Payments

With our shopping cart and user registration/authentication features complete, we are now in a position to allow our users to complete their purchases and start processing their payment information using Stripe.

In this chapter, we're going to look at the following topics:

- Building a checkout form using Stripe elements
- Client-side form validation using `VeeValidate`
- Storing orders in the database
- Processing payments with Stripe's .NET client library
- Building a my account page with a list of previously placed orders

Let's start by discussing the reasons for using Stripe to process our payments.

Why use Stripe?

Payment processing, or more specifically the handling of payment card data, is a very sensitive part of building an online shop of any kind. There are a lot of hoops to jump through to make sure you are PCI DSS compliant if you process payment card data on your own servers. However, luckily for us we don't need to, as we can make use of the Stripe online payment service to process that data on our behalf.

Simple PCI compliance

By using Stripe, we make PCI compliance infinitely simpler, because our user's payment card data is sent to Stripe servers for processing rather than ours. Getting your head around how this works takes a little thought, but it is actually a pretty simple concept. Payment card details are first posted off to Stripe servers where they validate them and get pre-approval for the amount you wish to charge them. If all is successful, a token is returned, which can then be posted to our own server along with the rest of the order information, such as products, quantities, and delivery information. As part of our order creation process, we can call back the Stripe API and pass in the token we just received from the client in order to trigger the actual payment.

This begs the question, *why don't we just charge the card on the first trip to the Stripe API instead of fetching a token?* The answer is simply that the charge amount sent to Stripe from the client cannot be trusted due to the fact that it was calculated on the client side and could have been tampered with. Instead, we can recalculate the amount on our server to ensure its correctness, and then send this value to Stripe for the actual payment amount.

Easy integration

PayPal is another very popular option for payment processing in online shops, and seeing as it's used so widely, you'd expect it to be fairly simple to set up and use. However, I've not found this to be the case when compared to how simply we can integrate Stripe into our apps. On top of this, unlike with PayPal, we don't need to redirect our users to an external website, or display a checkout form in a new modal window. With Stripe, we can build our checkout flow entirely within the boundaries of our own app domain; no redirects or modal windows required.

Excellent dashboard

The Stripe API is fully featured and incredibly easy to use, and as such you can easily create pages in your app for all of your payment-related data, such as customers, charges, and refunds. However, the dashboard they provide is so good that there really is no need to if you don't explicitly have the requirement to do so. All the data you need is easily accessible, and the interface is very intuitive, making recurring payment management and issuing refunds an absolute breeze.

Getting started with Stripe and client-side validation

Before we go any further, there are a few things that we need to do. We need to sign up for an account on the Stripe website and install a couple of third-party dependencies that we'll be using along the way as we build out the client-side payment processing feature.

Registering for a Stripe account

This one's easy. Head over to <https://stripe.com/> and click on the big green **Create account** button on their landing page. It's a simple signup form, so it doesn't really require any explanation.

Once you've registered and logged in, you'll be greeted with the Stripe dashboard where you'll have access to the **Developers** menu on the left-hand side, which contains an **API Keys** section beneath it. In here, you'll find both your public and secret Stripe API keys for testing. You'll need both eventually, but the public key belongs in your JavaScript code for the first part of the payment process, and the secret key belongs safely in the server-side C# code for the second part.

Including the Stripe checkout JavaScript library

Unlike most client-side JavaScript packages, Stripe specifically recommends loading their library from their CDN. As they're the experts, we'll happily take their advice and add it to our `Features/Shared/_Layout.cshtml` file. It belongs at the bottom of the `body` tag like so:

```
<body>
  @RenderBody ()

  <script src="https://js.stripe.com/v3/"></script>
  <script src="~/dist/vendor.js" asp-append-version="true"></script>
  @RenderSection("scripts", required: false)
</body>
```

Installing VeeValidate for client-side validation

Finally, we need to install the `VeeValidate` npm package for performing client-side validation in our Vue components. Open up a Terminal in the root of your project and run the following command:

```
yarn add vee-validate
```

Next, we have a couple of changes to make to the `ClientApp/boot.js` file. At the top, beneath our list of `import` statements, add the following:

```
import VeeValidate from "vee-validate";
```

Then, just below the rest of the `Vue.use(...)` statements, add the following:

```
Vue.use(VeeValidate);
```

This installs `VeeValidate` for use in every one of our components, but we'll come back to this and discuss it in more detail later on in the next section.

Building the checkout components

Enough theory! Let's start building our client-side checkout components to complete the checkout process of our phone shop. We'll start by making the modifications we need for the containing checkout page, which we'll use to control the display of the new child components we're about to create. Open up the `ClientApp/pages/Checkout.vue` file, then modify the `template` section as follows:

```
<template>
  <b-container class="page pt-4">
    <h1>Checkout</h1>
    <checkout-success v-if="success" :order="order" />
    <b-row v-else>
      <b-col cols="4" order="2">
        <cart-summary />
      </b-col>
      <b-col cols="8">
        <checkout-form @success="onSuccess" />
      </b-col>
    </b-row>
  </b-container>
</template>
```

At this point, everything we just added should be fairly self-explanatory. We're simply rendering three custom components (that we're yet to define), two of which are laid out in columns using `Bootstrap-Vue` layout components. We are also using the `v-if` and `v-else` directives to show either the `<checkout-success />` component, or the `<checkout-form />` and `<cart-summary />` components side by side. To finish things off, we add a `@success` event handler on the checkout form component, and bind an `order` prop to the `CheckoutSuccess` component.

The script section for this component does not yet exist, but needs to look like this:

```
<script>
import CartSummary from "../components/checkout/CartSummary.vue";
import CheckoutForm from "../components/checkout/CheckoutForm.vue";
import CheckoutSuccess from "../components/checkout/CheckoutSuccess.vue";

export default {
  name: "checkout",
  components: {
    CartSummary,
    CheckoutForm,
    CheckoutSuccess
  },
  data() {
    return {
      success: false,
      order: null
    };
  },
  methods: {
    onSuccess(order) {
      this.success = true;
      this.order = order;
      window.scrollTo(0, 0);
    }
  }
};
</script>
```

Again, nothing particularly new here, but we start by importing the three new components we need before registering them with a standard component definition, just like we've done before. The local component state we need for this component includes a `success` boolean property to control which components to display, and an `order` property that will represent the successfully placed order object after a successful checkout request. Finally, we need a single `onSuccess` method that is invoked when the checkout form component emits the `success` event. All we do is set the `success` property to `true` in order to hide the checkout form and cart summary components, and show the success component, then set the `order` property to the object associated with that same event. As the checkout form will be quite long, we can also scroll back to the top of the page using `window.scrollTo(0, 0)`.

Building a cart summary component

We already created a cart summary component, which we're using in the navbar of every page. However, in the context of the checkout page, we need a more detailed view of the cart's contents. There is no issue with having multiple components with the same name, as long as they reside in different directories, so we can specify exactly which one to import. As such, create a `ClientApp/components/checkout/CartSummary.vue` file with a template section as follows:

```
<template>
  <div>
    <h4 class="d-flex justify-content-between align-items-center mb-3">
      <span class="text-muted">Your cart</span>
      <span class="badge badge-secondary badge-pill">
        {{ itemCount }}</span>
    </h4>
    <ul class="list-group mb-3">
      <li v-for="(item, index) in items" :key="index" class="list-group-item d-flex justify-content-between lh-condensed">
        <div>
          <h6 class="my-0">{{ item.name }} <span class="text-muted">
            ({{ item.quantity }})</span></h6>
          <small class="text-muted">{{ item.colour }},<br>
            {{ item.capacity }}</small>
        </div>
        <span class="text-muted">{{ item.price * item.quantity |
          currency }}</span>
      </li>
      <li class="list-group-item d-flex justify-content-between">
        <span>Total:</span>
        <strong>{{ total | currency }}</strong>
      </li>
    </ul>
  </div>
</template>
```

```
        </li>
      </ul>
    </div>
  </template>
```

There is quite a lot of markup here, but it looks far more complicated than it actually is due to the number of Bootstrap classes we need to apply to lay things out nicely. In fact, this is simply a slightly modified version of one of the official Bootstrap checkout page examples, which you can find here: <https://getbootstrap.com/docs/4.0/examples/checkout/>.

We will display a title that includes a Bootstrap badge containing the item count of the user's cart, then use a Bootstrap list group to loop over the items in the cart, and display some minimal information about each one. The last item in the list is the calculated total of the cart. The final thing to note is that we're making use of the `currency` filter that we created in the previous chapter. This just goes to show how useful Vue filters can be for following DRY principles and only writing this kind of presentation logic once before using it everywhere in our app.

The script section for this component is pretty simple:

```
<script>
export default {
  name: "cart-summary",
  computed: {
    itemCount() {
      return this.$store.getters.shoppingCartItemCount;
    },
    total() {
      return this.$store.getters.shoppingCartTotal;
    },
    items() {
      return this.$store.state.cart;
    }
  }
};
</script>
```

All we need are three computed properties to read the `itemCount`, `total`, and `items` properties from our central store, which we've already made use of in the preceding template.

Building a checkout form component

As we saw from the checkout page template, side by side with the cart summary we just created sits the checkout form. The `template` section for this component is fairly long as there are quite a few fields for us to render. We'll go over it in sections, but first we need to create the `ClientApp/components/checkout/CheckoutForm.vue` file. As the name of the component suggests, it needs to contain our `HTML form` element, so let's start with that:

```
<template>
  <form @submit.prevent="submit">
    // ...rest of form omitted for brevity
  </form>
</template>
```

We prevent the default form `submit` action as we've done before, choosing to call a `submit` method instead, which we will define shortly. Delivery information is pretty important when selling physical products online, so we start by collecting the name of the customer we're delivering to:

```
<h4 class="mb-4">Delivery address</h4>
<b-row>
  <b-col>
    <b-form-group>
      <label>First name</label>
      <b-form-input v-model="firstName" data-vv-name="first name"
        v-validate="'required|min:3'" :state="state('first name')" />
      <b-form-invalid-feedback>
        {{ errors.first('first name') }}
      </b-form-invalid-feedback>
    </b-form-group>
  </b-col>
  <b-col>
    <b-form-group>
      <label>Last name</label>
      <b-form-input v-model="lastName" data-vv-name="last name"
        v-validate="'required|min:3'" :state="state('last name')" />
      <b-form-invalid-feedback>
        {{ errors.first('last name') }}
      </b-form-invalid-feedback>
    </b-form-group>
  </b-col>
</b-row>
```

When you read through this snippet, you'll notice we're only rendering two form fields, so it may look like a lot of markup for something so simple. However, there's quite a lot going on here that we've not seen before. Aside from Bootstrap grid components to lay these fields out in two columns, most of the preceding markup is validation-related.

First look at client-side validation

So far, we've only dealt with two simple forms—login and user registration—both of which had their validation requirements handled entirely on the server. This isn't exactly making the most of a client-side UI framework such as Vue, so, seeing as this form is far more substantial, we'll add rich client-side validation using `VeeValidate`.

We've made use of the `b-form-input` component from `Bootstrap-Vue` before, but this time we've added a couple of extra directives:

```
<b-form-input v-model="firstName" data-vv-name="first name" v-validate="'required|min:3'" :state="state('first name')" />
```

The `v-model` directive is what we've used before to add two-way data binding to a local data property, but what we've not seen before are the `data-vv-name` and `v-validate` directives. These are both part of the `VeeValidate` library.

Let's start with the `v-validate` directive, seeing as the `data-vv-name` directive is entirely optional. We can use the `v-validate` directive to instruct `VeeValidate` on how to validate each individual form input. In this case, we use `v-validate="'required|min:3'"` to specify that this field is required, and has a minimum length of three characters. Each validation rule is delimited by a pipe character, and there are no limits as to how many rules can be applied to each field. There is a very comprehensive list of rules that we can use, and these are very similar to those builtin to the .NET framework via the `System.ComponentModel.DataAnnotations` namespace. The full list can be found at <https://vee-validate.logaretm.com/index.html#available-rules>.

As with validation in .NET, a set of sensible default validation messages come baked into the `VeeValidate` library; these will be used unless you explicitly specify a custom message. The only issue with these messages is that they use the `data` property name without any kind of modification. For example, in the preceding form field, we are validating the `firstName` property, and as such the default validation message will be something along the lines of `The firstName field is required`. This isn't ideal, but we can work around this by using the `data-vv-name` directive, like we did earlier. By using `data-vv-name="first name"`, the error message is changed to `The first name field is required`.

Finally, we also made use of the `state` prop of the `b-form-input` component. We haven't used this prop before either, but it's simply used to set specific CSS classes to differentiate between different input states, for example, success and error. We're binding it to the result of a method that we'll define shortly, which takes the name of the form input to work out the state. Here, we are using `state('first name')` due to our usage of the `data-vv-name` directive, but, had we not overridden the name of the field, we'd need to use `state('firstName')` instead.

After each input field, we've used another component from `Bootstrap-Vue` that we've not used before—the `b-form-invalid-feedback` component:

```
<b-form-invalid-feedback>
  {{ errors.first('first name') }}
</b-form-invalid-feedback>
```

Here, we use the `errors` property, which is automatically added to the component by the `VeeValidate` library. More specifically, we use `errors.first('first name')` to find the first error message—if it exists—for the `firstName` field. Again, as we used the `data-vv-name` directive to override the default property name, we use `first name` rather than `firstName`. As we only show a single error message at a time, the order that we declare the validation rules in the `v-validate` directive matters. As we specified the required rule before the minimum length rule, the required field error message is displayed until at least one character is entered into the textbox. At this point, the minimum length rule kicks in until at least three characters are entered. Finally, once more than three characters have been entered, there are no error messages remaining and, as such, the field turns green to indicate its valid state.

The markup for the `lastName` field is almost identical, aside from the property name differences, so we won't spend any more time explaining it.

Finishing the delivery address form fields

Directly beneath the closing `</b-row>` tag, we need a very similar form field rendering for the first line of the customer's delivery address:

```
<b-form-group>
  <label>Address</label>
  <b-form-input v-model="address" data-vv-name="address" v-
    validate="'required'" :state="state('address')" />
  <b-form-invalid-feedback>
    {{ errors.first('address') }}
  </b-form-invalid-feedback>
</b-form-group>
```

No need to explain any of this as it's exactly the same as the previous two fields, so we'll move straight on to the second line of the customer's delivery address:

```
<b-form-group>
  <label>Address 2 <span class="text-muted">(Optional)</span></label>
  <b-form-input v-model="address2" data-vv-name="address 2"
    :state="state('address 2')" />
</b-form-group>
```

The `address 2` field is optional, so there is no associated error component. Next up, we have the `townCity` field:

```
<b-form-group>
  <label>Town / city</label>
  <b-form-input v-model="townCity" data-vv-name="town / city" v-
    validate="'required'" :state="state('town / city')" />
  <b-form-invalid-feedback>
    {{ errors.first('town / city') }}
  </b-form-invalid-feedback>
</b-form-group>
```

This is followed by the `county` field:

```
<b-form-group>
  <label>County</label>
  <b-form-input v-model="county" data-vv-name="county"
    v-validate="'required'" :state="state('county')" />
  <b-form-invalid-feedback>
    {{ errors.first('county') }}
  </b-form-invalid-feedback>
</b-form-group>
```

Finally we have the postcode field:

```
<b-form-group>
  <label>Postcode</label>
  <b-form-input v-model="postcode" data-vv-name="postcode" v-
    validate="'required'" :state="state('postcode')" />
  <b-form-invalid-feedback>
    {{ errors.first('postcode') }}
  </b-form-invalid-feedback>
</b-form-group>
```

Capturing payment information

The next field on our form isn't necessarily required, but it helps identify our customers in the Stripe dashboard when we're reviewing the list of payments we've received. The `nameOnCard` field is almost identical to those mentioned previously:

```
<h4 class="mb-4">Payment details</h4>
<b-form-group>
  <label>Name on card</label>
  <b-form-input v-model="nameOnCard" data-vv-name="name on card"
    v-validate="'required'" :state="state('name on card')" />
  <b-form-invalid-feedback>
    {{ errors.first('name on card') }}
  </b-form-invalid-feedback>
</b-form-group>
```

We could have assumed that the name of the purchaser is the same name that's entered as part of the delivery address, but that isn't always the case, so we've included this field to cover more bases. We'll send the value entered into this field to the Stripe API, which will embed it in the token we get back so it's included with the payment information when we submit it from the server.

The final field we need on the checkout form is the most important field—the card details field:

```
<b-form-group>
  <label>Credit/debit card details</label>
  <div ref="card" class="form-control"></div>
</b-form-group>
```

This is quite different from the previous fields we've defined. For a start, we don't use the `b-form-input` component as we've done for the others. Instead, we simply render an empty `<div ref="card" class="form-control"></div>` element. This is because we're going to configure the Stripe elements library to mount one of their custom credit card elements into this `div`—we'll see how to do this in the next subsection.

To finish the `template` section of this component, the last thing we need is the `submit` button to actually complete the checkout form:

```
<b-button :disabled="loading" type="submit" variant="primary"
size="lg" block class="mt-4 mb-4">
  Checkout
  <span v-if="loading" class="fas fa-spinner fa-spin"></span>
</b-button>
```

The only thing of note here is that we bind the `disabled` prop of the button to a `loading` property in our component, and we conditionally display a Font Awesome loading spinner based on the same property.

Initializing Stripe elements

We have the containing `div` element that we wish to mount the credit card input to, but we've not yet done anything with it. We'll use a couple of life cycle hooks to set up and tear down the credit card input—add a `script` section to the checkout form component with the following contents:

```
<script>
import axios from "axios";

let stripe = Stripe(`PUBLIC_STRIPE_KEY_HERE`),
  elements = stripe.elements(),
  card = null,
  style = {
    base: {
      lineHeight: "24px"
    }
  };

export default {
  name: "checkout-form",
  mounted() {
    card = elements.create("card", { style: style });
    card.mount(this.$refs.card);
    this.firstName = this.$store.state.auth.firstName;
    this.lastName = this.$store.state.auth.lastName;
```

```
    },
    beforeDestroy() {
      card.destroy();
    }
  }
</script>
```

We start by initializing some global variables outside of the component definition, the first of which is an instance of the `Stripe` object, which takes our public API key as an argument. You'll need to copy your own public key in here, which as we've already seen is available from the **Developers | API Keys** section of the Stripe dashboard. From this object, we can create an instance of the `elements` object using the `stripe.elements()` function. We can also create an empty `card` variable, as well as a `style` object, which we'll use to override the default styles applied to the credit card's input.

Inside the component definition, we define the `mounted` and `beforeDestroy` life cycle hooks. The `mounted` hook is used to call the `elements.create()` function, passing in `card` as the first argument to make sure we're creating a credit card input. As the second argument, we pass a new object with a `style` property, which references the `style` object we just declared. The output from this function is assigned to the empty `card` variable, before we call the `card.mount()` function in order to mount the element into the DOM. To do so, we have to pass a reference to the element we wish to mount to, which in this case is the empty `div` element that we rendered in the preceding template. Because we gave this element a `ref="card" prop`, we can access the reference to it using `this.$refs.card`. The `beforeDestroy` hook is more simple, and is only used to call the `card.destroy()` function to make sure that, the next time we reach the checkout page, we get a fresh credit card input created. We also retrieve the logged-in user's first and last names from our Vuex store to pre-fill the appropriate fields on the form.

Validating form input state

The local component state we need includes a `data` property for each form field, as well as the `loading` property that we use to track when an API request is in progress to disable the submit button:

```
data() {
  return {
    nameOnCard: "",
    firstName: "",
    lastName: "",
    address: "",
    address2: "",
    townCity: "",
```

```
    county: "",
    postcode: "",
    loading: false
  };
}
```

`VeeValidate` will automatically validate our form fields as their values change based on the rules we applied using the `v-validate` directive. As errors are found, they will be picked up by our validation messages in the template and displayed automatically. However, we've also bound the `state` prop of each input to a method that determines that state based on whether there are any errors or not. This method looks like this:

```
methods: {
  state(field) {
    if (this.errors.has(field)) {
      return false;
    } else {
      return this.fields[field] && this.fields[field].dirty ?
        true : null;
    }
  }
}
```

There are three potential values that we are interested in passing as the `state` prop to a `b-form-input` component. If the input is in an error state, we need to pass `false`; if it is in a valid state, we need to pass `true`; and if it's in a neutral or untouched state, we need to pass `null`. Bearing this in mind, the preceding method takes the name of a field based on the `data-vv-name` directive we've been using, and then does a number of checks to determine which of the three states to apply.

If the `errors` property from `VeeValidate` contains any errors for this field, our decision is easy and we simply return `false` to set the input as invalid and give it a red border. However, if there are no errors for this field, our checks become slightly more complicated. We need a way of determining if the field is *dirty* or not—that is, whether it has been interacted with by our users. We can do this using another `VeeValidate` property called `fields`. This keeps track of all the fields that have validation rules applied, and also sets a `dirty` boolean property as soon as a user focuses on the input. This means that, to determine whether to set the input state as valid or not, we can check whether the field we are interested in is dirty or not. If it is, we return `true` to set the input as valid and apply a green border. Otherwise, we return `null` as the field has not yet been touched, so we leave it with the default border color.

At this point, our validation will be firing in real time as our users enter data into the form fields, and they will be presented with immediate feedback as to whether or not their input is valid. However, we still need to prevent the form from submitting if there are any errors, which we will do now as we define the `submit` method that we fire as the form is submitted. Add the following function to the `methods` object we just defined:

```
submit() {
  this.$validator.validateAll().then(result => {
    if (result) {
      this.loading = true;
      //...rest of function omitted for brevity
    }
  })
}
```

When we installed `VeeValidate` earlier, the `$validator` object was added to every component within the application. As such, we can use it to force the validation of all input fields in the component using the `validateAll` function, which returns a promise. When that promise resolves, we get a Boolean result that we can check to see if the form was valid or not. In this case, we set the `loading` property to `true` if it is, as we'll be performing an API request that we want to provide feedback to the user on. We don't need to do anything if the form is invalid, as all of the errors that have occurred will already be on display to the user due to the real-time feedback that we get out of the box with `VeeValidate`.

Verifying payment details with Stripe

After we've checked that the form is valid, we can submit the details to Stripe's API and validate the card details. As previously discussed, if they are valid, we'll receive a token in return that we can process from our own server-side API later. Immediately after we set the `loading` property to `true`, add the following code:

```
submit() {
  this.$validator.validateAll().then(result => {
    if (result) {
      this.loading = true;
      const details = {
        name: this.nameOnCard
      };

      stripe.createToken(card, details).then(result => {
        if (result.error) {
          this.loading = false;
        } else {
          //...submit order here
        }
      })
    }
  })
}
```

```
    }  
  }  
}  
}
```

We start by creating a new `details` object with a single `name` property to which we assign the value from the `nameOnCard` form field. We then call the `stripe.createToken` function, passing the `card` object that we initialized earlier, along with the `details` object that we just created. The second argument is optional, but by passing these additional details, they show up in the Stripe dashboard, making it easier to identify which customer the payment belongs to.

The `createToken` function returns a promise, so we wait for it to resolve by chaining a call to `.then`, before determining if the card details were valid or not by interrogating the `result.error` property. If it evaluates as a *truthy* value, the card details were invalid, so we stop the loading animation by setting the `loading` property back to `false`. The Stripe elements card component will already be highlighting the details in red to signify an error, so there is nothing more we need to do. Otherwise, if the `result.error` property is not a truthy value, we are ready to submit the order details to our API for processing.

Submitting the order to the API

At this point, we've successfully performed our own client-side validation of the checkout form, and validated the payment card details with the Stripe API. Therefore, we are ready to submit the order to our API for processing. The first thing we need to do is construct an object to represent the order using a mixture of data from the local checkout form component state, and the cart items array from the global store. Add the following object declaration:

```
stripe.createToken(card, details).then(result => {  
  if (result.error) {  
    this.loading = false;  
  } else {  
    const order = {  
      stripeToken: result.token.id,  
      firstName: this.firstName,  
      lastName: this.lastName,  
      address1: this.address,  
      address2: this.address2,  
      townCity: this.townCity,  
      county: this.county,  
      postcode: this.postcode,  
      items: this.$store.state.cart.map(item => {
```

```
        return {
            productId: item.productId,
            colourId: item.colourId,
            storageId: item.storageId,
            quantity: item.quantity
        };
    })
};
}
```

Most of the properties on this `order` object are self-explanatory, as we're just mapping the local form data that we're collecting for the delivery address of the customer. The `stripeToken` property is pretty important as, without this, we can't actually trigger the payment when we receive the order on the server. We get it by accessing the `result.token.id` property, which in this instance is the only piece of information we're interested in from Stripe's response. It could be argued that it's worth sending the full response object back to the server so that we can persist it in our database if we ever need additional information, but, to keep things simple, for now, we're only interested in the token.

As the name suggests, the `items` array contains the items the user has placed in their shopping cart. We loop over these using the built-in `Array.map` function, where we can use a lambda expression as the argument to access the properties of each `item` object within the array. The only properties we need are those required to identify the specific product variant being ordered, namely the `productId`, `colourId`, and `storageId` properties, as well as the `quantity` they wish to order. As such, we return a new object containing these properties, which ultimately results in a new array of objects containing only the properties we care about on the server. This is a common pattern in JavaScript that can be likened to performing a LINQ `Select` operation on a C# collection in order to project it to a different data structure. This is exactly what we've been doing with our EF database operations when we project our entity models to view models.

With our `order` object constructed and ready to go, the last piece of the puzzle is to actually send this off to our API for processing. Directly beneath the `order` object declaration, add the following API call:

```
axios
    .post("/api/orders", order)
    .then(response => {
        this.$store.commit("clearCartItem");
        this.$emit("success", response.data);
        this.loading = false;
    })
```

```
.catch(error => {  
  //process server errors  
  console.log(error.response);  
  this.loading = false;  
});
```

As always, we're making use of Axios to perform the API request, but this time we're doing it directly from a component method rather than using a Vuex action. There is nothing wrong with this approach, as the Vuex store is intended for global application state and functionality that is reused in multiple components. We don't need to reuse the state or methods in this component anywhere else in the application, so there is no need to add the additional overhead of writing Vuex actions and mutations. However, it can also be argued that we should be consistent and perform all API requests in the same fashion, that is, using Vuex actions. This also means we'd be using mutations as well, and as such we'd be able to use the Chrome devtools to see the history of everything that's happened. I don't feel particularly strongly either way so I'm happy to keep things simple, but if you'd rather use Vuex, then feel free to take the time to refactor things at this point.

After posting the `order` object to the `/api/orders` endpoint, we wait for the promise to resolve, then, as long as it was successful, we have a number of things to do. First of all, we commit the `clearCartItems` mutation (which we're yet to define), which as the name suggests will empty the user's shopping cart. We then emit a custom `success` event, passing the server's `response.data` property as its argument—recall that we listen for this event in the parent component to hide the checkout form and display a success message containing the order number. Finally, we stop the loading animation by setting the `loading` property to `false`. The `clearCartItems` mutation belongs in the `ClientApp/store/mutations.js` file and looks like this:

```
export const clearCartItems = state => {  
  state.cart = [];  
};
```

If the API call fails, we handle the failure by chaining the `.catch` block to the request. Even though we've done some client-side form validation, we should always be validating on the server as well. We've already seen how to handle server-side validation errors so we're leaving it out here, but it's worth noting that if this were a real app, we'd need to handle any server-side validation errors at this point too. All we do for now is log the response to the console so that we can see it easily, and then stop the loading animation, just like we did in the success block.

Adding basic Bootstrap styling to Stripe elements

The very last thing we need to do with this component is override the styles of the Stripe card element so that it fits in with the styling of our other form fields. We've already fixed the height of the textbox, but if we focus it, we don't get the same outline as we do with the rest. Add the following `style` section to fix this:

```
<style lang="scss" scoped>
.StripeElement--focus {
  border-color: #80bdff;
  outline: 0;
  box-shadow: 0 0 0 0.2rem rgba(0, 123, 255, 0.25);
  transition: border-color 0.15s ease-in-out,
  box-shadow 0.15s ease-in-out;
}
</style>
```

Building a checkout success component

The last component we need is a checkout success component to notify the user that their order has been placed, and provide them with the order number. We've already wired up the logic in the parent checkout page component to display this component after a successful API request has been made to place an order. Create the `ClientApp/components/checkout/CheckoutSuccess.vue` file, and add a `template` section as follows:

```
<template>
  <b-alert :variant="variant" show>
    <h3>Order placed successfully.</h3>
    <p>
      Your order number is <strong>{{ order.orderId }}</strong>.
      <span v-if="order.paymentStatus !== 'Paid'">
        However, your payment card was declined -
        we will not ship your order until payment has been made.
      </span>
    </p>
    <p>
      Click <router-link to="/account">here</router-link>
      to see your orders.
    </p>
  </b-alert>
</template>
```

All we're displaying here is a standard Bootstrap alert box, but even though the order has been placed, the payment could still have failed. As such, we bind the `variant` prop of the alert to a `variant` computed property, which will change the style of the box depending on the result. If the payment was successful, we display a **success** alert, which has a green background and border, and if the payment failed, we display a **warning** alert, which has a pale yellow background and border.

The rest of the template is simple—we have a heading to confirm the order was placed, as well as some text containing the new order number which we get from an `order` object which we will receive as a prop. As well as the order number, this object also contains the payment status of the order, which we then use to conditionally display a message stating that the order will not be shipped until payment has been taken. Finally, we display a link to an account page, which we'll build later to include a list of all the orders we have placed. The `router-link` component that we're using here ultimately renders a standard `a` tag with the appropriate `href` attribute. As such, it might make far more sense for us to use standard HTML elements wherever possible. However, seeing as we're using HTML5 history mode for client-side routing, we should always use the `router-link` component instead, as it intercepts the click event on the `a` element in order to prevent full page refreshes.

The `script` section for this component looks like this:

```
<script>
export default {
  name: "checkout-success",
  props: {
    order: {
      type: Object,
      required: true
    }
  },
  computed: {
    variant() {
      return this.order.paymentStatus === "Paid" ?
        "success" : "warning";
    }
  }
};
</script>
```

This is another standard component definition with a single `order` prop and `variant` computed property. We've already talked about these so we won't dwell on them for too long, but to reiterate, the `order` prop is the response we receive from the server after successfully placing an order. We use the `paymentStatus` property from this object to decide which `variant` to use for the alert box, giving success if payment was successful, and a warning if not.

Building a my account page

Now that we're allowing our users to place orders, we should really give them a page to view all of their previously placed orders as well. You'd normally find this information somewhere within a `My Account` section of most online shops, so we'll build a my account page here as well. The following `template` section belongs in a new `ClientApp/pages/Account.vue` file:

```
<template>
  <b-container class="page pt-4">
    <h1>My Account</h1>
    <order-list :orders="orders" />
  </b-container>
</template>
```

Nothing special here, just the `My Account` heading and a new `order-list` component, which we'll define shortly. Note that we're binding an `orders` prop to a component property of the same name, which will eventually be populated with the list of orders that we'll retrieve from our backend API.

The `script` section for this page component is a little more involved, so we'll build it up in stages. Start by adding the following standard component definition:

```
<script>
import axios from "axios";
import OrderList from "../components/account/OrderList.vue";

export default {
  name: "account",
  components: {
    OrderList
  }
};
</script>
```

We know that we'll be making an API call to fetch our list of orders, so we'll start by importing the `axios` object. We also know that we've included the `order-list` component element in the preceding `template` section, so we will also import the `OrderList` component (again, that we're yet to define), and then register it as a child in the `components` object of our default export. Next, we'll define the `data` properties that we need for this component:

```
data() {
  return {
    orders: null
  };
}
```

As this is a simple component, we just need the single `orders` property, which will hold our list of orders from the server, which we then pass down into the order list component via props. As we've done with our previous page-level components, which require data from the server, we're going to add the following `setData` method:

```
methods: {
  setData(orders) {
    this.orders = orders;
  }
}
```

Finally, we need the following `beforeRouteEnter` hook to trigger our API request and call the `setData` method when it resolves:

```
beforeRouteEnter(to, from, next) {
  const vm = this;
  axios.get("/api/orders").then(response => {
    next(vm => vm.setData(response.data));
  });
}
```

Before we can use this page, we need to register it with our router configuration in the `ClientApp/boot.js` file. Find the section where we import our list of page components and add the following line:

```
//import page components
import Catalogue from "./pages/Catalogue.vue";
import Product from "./pages/Product.vue";
import Cart from "./pages/Cart.vue";
import Checkout from "./pages/Checkout.vue";
import Account from "./pages/Account.vue";
```


Next, find the `routes` array and add the following line:

```
const routes = [
  { path: "/products", component: Catalogue },
  { path: "/products/:slug", component: Product },
  { path: "/cart", component: Cart },
  { path: "/checkout", component: Checkout, meta:
    { requiresAuth: true } },
  { path: "/account", component: Account, meta:
    { requiresAuth: true } },
  { path: "*", redirect: "/products" }
];
```

Remember that ordering matters here, so make sure that this line sits above the catch-all wildcard route.

Building the OrderList component

The only component we need for our basic `My Account` section is the order list component, which we're trying to render from the main page. Create a `ClientApp/components/account/OrderList.vue` file, and then add the following template section:

```
<template>
  <div>
    <h3 class="mt-4">Orders</h3>
    <table class="table table-striped table-hover">
      <thead>
        <tr>
          <th>Order #</th>
          <th>Placed</th>
          <th>Items</th>
          <th>Total</th>
          <th>Payment status</th>
        </tr>
      </thead>
      <tbody>
        <tr v-if="orders && orders.length > 0"
          v-for="order in orders" :key="order.id">
          <td>{{ order.id }}</td>
          <td>{{ order.placed | date }}</td>
          <td>{{ order.items }}</td>
          <td>{{ order.total | currency }}</td>
          <td>{{ order.paymentStatus }}</td>
        </tr>
        <tr v-else>
```

```
        <td colspan="5">You haven't placed any orders yet!</td>
    </tr>
</tbody>
</table>
</div>
</template>
```

This might look quite long, but all it is is a simple `div` element containing a heading and a table. The `tbody` section of the table contains the only really interesting things that we need to look at, so we'll ignore the rest, as it's just standard HTML. We start by using the `v-if` and `v-else` directives on two table rows to display one or the other, depending on whether the user has placed any orders yet or not. If they have, we loop over the `orders` array, and for each one, we render a table row containing a number of columns describing the order. If they haven't, we just display a single table row with one column, which spans the full width of the table and contains a simple message to show that there are no orders to display.

In the row that we repeat for each of the user's orders, notice how we're making use of the `currency` filter that we created earlier for the order total column. You may have also noticed that we're using a new `date` filter that we've not seen before. We'll come back and see what this looks like shortly, but for now, let's look at the very simple `script` section of this component:

```
<script>
export default {
  name: "order-list",
  props: {
    orders: {
      type: Array,
      required: false
    }
  }
};
</script>
```

All we need to do is define the `orders` prop, which we specify is of type array and is not required for the case where a user has not yet placed any orders.

Formatting dates with a reusable date filter

It is very common in web applications to need to display dates in a user-friendly format. We could format the dates on the server side of our application using the C# `ToString` overloads where we can specify the exact format we want. However, formatting data for display purposes is really a concern for the client-side applications that are consuming REST APIs. It would be very difficult for an API to start catering to all of the needs of any client application consuming it, so it is best to leave this type of logic for the clients to provide themselves. You could also be consuming an external API that you have no control over, so it is worth knowing how to handle this use case in JavaScript anyway.

In this sample application, our requirement is extremely simple, as all we want to do is display dates in the `dd/MM/yyyy` format. The following filter function, which belongs in the `ClientApp/filters/index.js` file, will do just that:

```
export const date = value => {
  const date = new Date(value);
  const dd = (date.getDate() < 10 ? "0" : "") + date.getDate();
  const MM = (date.getMonth() + 1 < 10 ? "0" : "") +
    (date.getMonth() + 1);
  const yyyy = date.getFullYear();

  return dd + "/" + MM + "/" + yyyy;
};
```

Unfortunately, this isn't anywhere near as simple as doing the same thing in C#, but at least this logic is now reusable across the whole of our application. Fundamentally, what we're doing here is extracting the day, month, and year parts of the date in order to put them back together in the right order.

We start by passing the argument value into the constructor of JavaScript's built-in `Date` object in order to make sure we are definitely dealing with a valid date, and to enable us to use some of the functions defined on that object. Next, we extract the day and month values, using the `getDate` and `getMonth` functions, which JavaScript gives us on `Date` objects. However, if the day or month is less than 10, these functions will return a single-digit value. When displaying dates, we expect that both the day and month values are always two digits long, that is, they get prefixed with a zero if they only have a single digit. As such, we start by checking if these values are less than 10, and prefix the value with a zero if they are. For the year, all we need to do is call the `getFullYear` function, which returns the four-digit value that we expect. Finally, we build our date string back up in the correct order, delimiting each part with a `/` character.

We still need to register this filter for use in our application, so open up the `ClientApp/boot.js` file and find and modify the following section where we configured our currency filter, as follows:

```
// filters
import { currency, date } from "../filters";

Vue.filter("currency", currency);
Vue.filter("date", date);
```

Linking to the my account page

At this point, our my account page is complete, but the only link to it is only visible after a user successfully places an order. Let's add a link to it in the navbar so that our users can access it whenever they like. Open up the `ClientApp/components/app/AuthNavItem.vue` file, and then add the following `b-dropdown-item`:

```
<b-nav-item-dropdown v-if="isAuthenticated" right>
  <template slot="button-content">
    <i class="fas fa-user"></i>
  </template>
  <b-dropdown-item to="/account">
    <i class="fas fa-user"></i>
    My Account
  </b-dropdown-item>
  <b-dropdown-item @click="logout">
    <i class="fas fa-sign-out-alt"></i>
    Logout
  </b-dropdown-item>
</b-nav-item-dropdown>
```

Fixing the register form component

Now that we've installed `VeeValidate` globally in our application, all of our components have an `errors` property by default. This has now broken the user registration form component, as we defined a local state property called `errors`, which now conflicts with the `VeeValidate` version. There are some open pull requests with the library creators on GitHub to prevent this kind of property declaration on components where we don't actually need validation. However, for now, we just need to deal with this by changing our property names.

Start by opening up the `ClientApp/components/app/RegisterForm.vue` file, and then modify the `b-alert` element at the top of the form tag in the template section as follows:

```
<b-alert variant="danger" :show="regErrors !== null"
dismissible @dismissed="regErrors = null">
  <div v-for="(error, index) in regErrors" :key="index">{{ error[0]
  }}</div>
</b-alert>
```

Next, in the data function of the script section, make the following changes:

```
data() {
  return {
    email: "",
    password: "",
    confirmPassword: "",
    regErrors: null
  };
}
```

In the submit method, we have a couple of changes to make, starting by making the following changes after dispatching the `register` action:

```
this.$store
  .dispatch("register", payload)
  .then(response => {
    this.regErrors = null;
    this.email = "";
    this.password = "";
    this.confirmPassword = "";
    this.$emit("success");
  })
```

Next are the following changes in the corresponding catch block of the same dispatch call:

```
.catch(error => {
  if (typeof error.data === "string" || error.data instanceof String)
  {
    this.regErrors = { error: [error.data] };
  } else {
    this.regErrors = error.data;
  }
});
```

Finally, we need to make the following changes to the `close` method:

```
close() {
    this.regErrors = null;
    this.$emit("close");
}
```

This completes the changes we need to make to fix this component, as well as completing all of our client-side changes for processing payments, so let's move on to the backend.

Server-side payment processing

As part of our server-side payment processing, we're going to actually persist the user's orders into the database, and to do that we need to make some changes to our data model. We have a couple of new entities to create, and a few minor tweaks for our existing ones. We'll also be looking at some of the new features introduced in EF Core 2.0 such as owned entity types.

Adding orders to the data model

The first and most obvious new entity we need is the `Order` entity. Create a new `Data/Entities/Order.cs` file with the following contents:

```
namespace ECommerce.Data.Entities
{
    public class Order
    {
        public int Id { get; set; }
        public int UserId { get; set; }
        public DateTime Placed { get; set; } = DateTime.UtcNow;

        public List<OrderItem> Items { get; set; } = new List<OrderItem>();
        public Address DeliveryAddress { get; set; }
        public PaymentStatus PaymentStatus { get; set; } =
            PaymentStatus.Pending;
        public AppUser User { get; set; }
    }
}
```

As with our other entities, we have an integer primary key named `Id`; in this case, we also need to link each order with a single user. As such, we have an integer foreign key named `UserId`, which will be wired up automatically by the default EF conventions. In addition to knowing which user the order belongs to, we also need to know the date the order was placed, so we also include a `Placed DateTime` property.

Next up, we have a list of `OrderItem` objects, which are navigation properties for a one-to-many relationship, which we'll set up when we create the `OrderItem` entity shortly. We then have the `DeliveryAddress` property, which we will soon define as an owned entity type—more on this later. The `PaymentStatus` property is an enum which, as the name suggests, is used to model the payment status of the order. Finally, we have the `AppUser` property, which is the navigation property that pairs with the `UserId` foreign key that we discussed earlier.

The `OrderItem` entity belongs in a new `Data/Entities/OrderItem.cs` file, and looks like this:

```
namespace ECommerce.Data.Entities
{
    public class OrderItem
    {
        public int Id { get; set; }
        public int OrderId { get; set; }
        public int ProductId { get; set; }
        public int ColourId { get; set; }
        public int StorageId { get; set; }
        public int Quantity { get; set; }

        public ProductVariant ProductVariant { get; set; }
    }
}
```

Again, as usual, we have the `Id` primary key property, but this time we will include an `OrderId` foreign key property to associate each order item with the order it belongs to. We then have the `ProductId`, `ColourId`, and `StorageId` properties, which together form another foreign key to the specific product variant that was ordered. Even though this is a composite key, EF is still capable of determining the relationship by convention without any kind of manual configuration on our part. We then have the `Quantity` property, which tells us how many of this product variant was ordered, enabling us to calculate the order total based on the price of the variant multiplied by the quantity. We'll need to be able to perform this calculation on the server side in order to verify the order total that we send to Stripe for processing. Finally, we have the `ProductVariant` navigation property, which ties in with the composite foreign key before it.

The `PaymentStatus` enum that we used on the `Order` entity is another new file that we need to create in the `Data/Entities/PaymentStatus.cs` directory. It looks like this:

```
namespace ECommerce.Data.Entities
{
    public enum PaymentStatus
    {
        Pending,
        Paid,
        Declined
    }
}
```

Nothing to explain here really, as this is just a standard C# enum class. By default, we set all new orders to use the `Pending` value, but we will update this to either `Paid` or `Declined` depending on the response from Stripe when we charge the customer later on.

We've already defined all the data properties we need for EF to infer the foreign key between orders and users. However, we have currently only defined the appropriate navigation properties to a single user from each order, and not the list of orders that belong to each user. Open up the `Data/Entities/AppUser.cs` file and add the following navigation property to resolve this:

```
namespace ECommerce.Data.Entities
{
    public class AppUser : IdentityUser<int>
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }

        [NotMapped]
        public string FullName
        {
            get { return $"{FirstName} {LastName}"; }
        }

        public List<Order> Orders { get; set; } = new List<Order>();
    }
}
```

By defaulting this property to a new empty list of `Order` objects, we avoid the potential null reference exceptions that would occur if we tried to perform any LINQ operations on this property before the user has actually placed any orders.

Owned entity types in EF Core 2.0

The last new entity that we need is the `Address` entity, which, as previously discussed, will be configured as an owned entity type. These have been newly added in the latest version of EF Core, which at the time of writing is version 2.0, and can be compared with the similar concept of complex types in older versions of EF.

Why use owned entity types?

Owned types enable us to model complex types such as addresses that can be reused in as many different owning parent types as we need, including multiple references to the same owned type from a single parent.

For example, in the context of most online shops, users will have both delivery and billing addresses. Each of these complex types will likely have the exact same fields, so we can save duplication by defining an `Address` type and referencing it twice for billing and delivery variants.

By default, owned types are stored as additional fields in the same table as their parent type. As an example, we could have an `Address` owned type with `Address1` and `Address2` fields. If we then referenced this type from a parent entity with a property named `BillingAddress`, we'd see these properties in the parent database table named `BillingAddress_Address1` and `BillingAddress_Address2`. However, if you'd prefer, you can manually override the default in order to use a feature called **table splitting**, which splits the owned type into its own database table instead.

Defining an owned type

Owned type classes are almost identical to normal entity classes. Following on from the preceding examples, and recalling the `Order` entity that we defined earlier, we need a new `Data/Entities/Address.cs` entity file, which looks like this:

```
namespace ECommerce.Data.Entities
{
    public class Address
    {
        [Required]
        public string FirstName { get; set; }
        [Required]
        public string LastName { get; set; }
        [Required]
        public string Address1 { get; set; }
    }
}
```

```
    public string Address2 { get; set; }
    [Required]
    public string TownCity { get; set; }
    [Required]
    public string County { get; set; }
    [Required]
    public string Postcode { get; set; }
}
}
```

Based on this class, the only clue that this isn't a standard entity is the lack of the `Id` primary key property. Owned types do not need primary keys, as they are given a **shadow property** as their primary key, which is assigned the same value as the primary key of their parent entity. Owned types are only ever accessible as navigation properties of the entities that own them, so there is never any confusion as to which entity they belong to. They are also included by default in every EF query for the parent entity, without the need for eagerly loading the navigation property.

Configuring owned types

At the time of writing, a new version of EF Core has been released that gives us additional options in configuring owned types. As of EF Core 2.1, the `OwnedAttribute` class was added, meaning that we can now decorate the `Address` class with it in order to declare it as an owned type. However, my personal preference is to stick with the original method of explicit configuration by overriding the `OnModelCreating` method of our `DbContext` class as we have for our existing entity-related configurations. Open up the `Data/EcommerceContext.cs` class and update our `OnModelCreating` override as follows:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<Product>()
        .HasIndex(b => b.Slug)
        .IsUnique();

    modelBuilder.Entity<ProductFeature>()
        .HasKey(x => new { x.ProductId, x.FeatureId });

    modelBuilder.Entity<ProductVariant>()
        .HasKey(x => new { x.ProductId, x.ColourId, x.StorageId });

    modelBuilder.Entity<Order> ()
```

```
        .OwnsOne(x => x.DeliveryAddress);  
    }
```

If you'd rather use the `OwnedAttribute` method instead, feel free to make sure you're running EF Core 2.1 and add the attribute to the `Address` class instead, as this really is a matter of personal preference. However, it is worth noting that owned entity types are never inferred by default, and must be configured in one way or the other.

While we're in this file, we might as well add the following orders table declaration to the bottom of the current list of `DbSet<T>` values:

```
public DbSet<Order> Orders { get; set; }
```

Creating the orders migration

As with any change to our entity data model, we need to add a migration before the changes will be persisted to the database. Open a Terminal at the root of the project and run the following command:

```
dotnet ef migrations add Orders
```

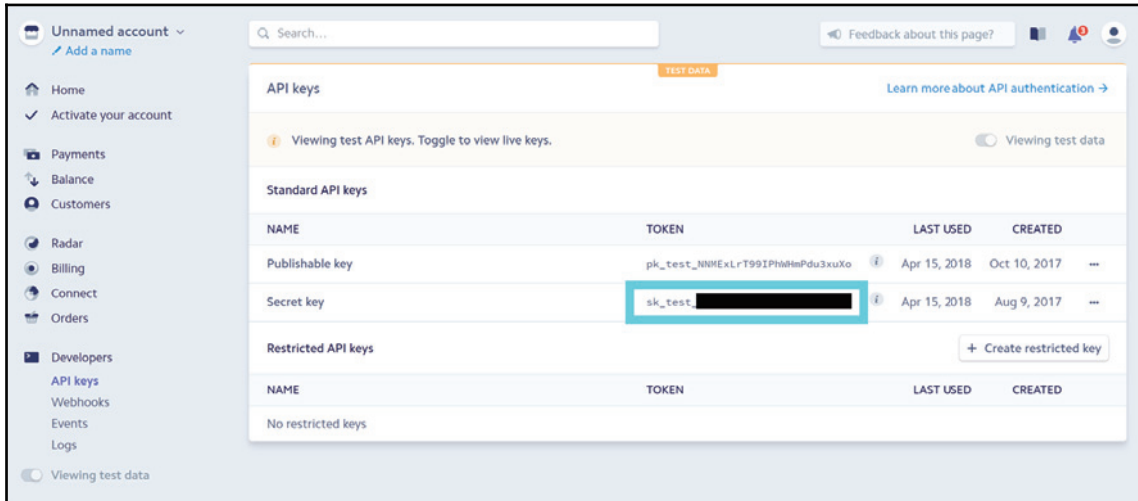
Again, as with any migration, it is worth giving the generated migrations file a check to make sure everything looks OK, as it's easier to remove a migration before it reaches the database than it is to roll back and remove it afterwards.

Installing and configuring the Stripe.net NuGet package

At this point, we're ready to add a controller and action method to receive orders and subsequently process the payment information of the user placing that order. Before we get that far, though, we need to install the `Stripe.net` NuGet package. Open the **Add Package** window of the NuGet package manager VSCode extension and type `Stripe.net` into the search bar. Make sure it is `Stripe.net` that you install and not just `Stripe`. At the time of writing, the latest version is 15.3.1.

Configuring Stripe

If you followed along from the first section of this chapter, you should have already registered for an account on the Stripe website, but if not, you'll need to go back and do that now. If you have, we're going to need your Stripe secret key from the **API Keys** section under the **Developers** option in the left-hand menu of the Stripe dashboard. See the following screenshot to find what you're looking for:



Next, open up the `Startup.cs` file and locate the `Configure` method. At the very bottom, below the `app.UseMvc(...)` line, add the following:

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

    routes.MapSpaFallbackRoute(
        name: "spa-fallback",
        defaults: new { controller = "Home", action = "Index" });
});

StripeConfiguration.SetApiKey("YOUR_STRIPE_SECRET_KEY_HERE");
```

You'll need to add the appropriate Stripe import statement to the top of the file, and replace the placeholder string with your own secret key from the Stripe dashboard.

Processing orders and payments

We're finally ready to actually receive an order, persist it to the database, and process the user's payment for that order. To do so, we'll create a new controller in the `Features/Orders/Controller.cs` file. The initial content for this controller contains a single `Create` action method, which looks like this:

```
namespace ECommerce.Features.Orders
{
    [Authorize]
    [Route("api/[controller]")]
    public class OrdersController : Controller
    {
        private readonly EcommerceContext _db;
        public OrdersController(EcommerceContext db)
        {
            _db = db;
        }

        [HttpPost]
        public async Task<IActionResult> Create([FromBody]
        CreateOrderViewModel model)
        {
            if (!ModelState.IsValid)
                return BadRequest(ModelState);

            //...rest of method omitted for brevity
        }
    }
}
```

This is a standard controller definition that will be very familiar to you, so we won't dwell on it for too long. The only things worth mentioning are that every action will require authentication, so we decorate the controller itself with the `Authorize` attribute, and that we're prefixing each action with `api/orders` using the `Route` attribute as well. Finally, we're expecting an instance of the `EcommerceContext` class to be supplied using constructor injection from the built-in DI framework.

For now, the only action method we need is the `Create` action, where we expect to bind an instance of the—as yet undefined—`CreateOrderViewModel` class. We'll come back to it later, but for now just know that it contains the same properties that we added to the `Order` object that we're sending from the `CheckoutForm` component on the client side. Finally, as with most action methods, we're failing early if there are any validation errors.

Persisting the order object

The first thing we need to do to complete the `Create` action method is to persist the user's order into the database. The following code needs to be added immediately below the `ModelState` check:

```
var user = await _db.Users.SingleAsync(x => x.UserName ==
HttpContext.User.Identity.Name);

var order = new Order
{
    DeliveryAddress = new Address
    {
        FirstName = model.FirstName,
        LastName = model.LastName,
        Address1 = model.Address1,
        Address2 = model.Address2,
        TownCity = model.TownCity,
        County = model.County,
        Postcode = model.Postcode
    },
    Items = model.Items.Select(x => new OrderItem
    {
        ProductId = x.ProductId,
        ColourId = x.ColourId,
        StorageId = x.StorageId,
        Quantity = x.Quantity
    }).ToList()
};

user.Orders.Add(order);

await _db.SaveChangesAsync();
```

We start by retrieving the currently logged-in user based on their username, which we find on the `HttpContext.User.Identity.Name` property. By using the LINQ `SingleAsync` method, we decide to throw an exception if either zero or more than one user matches the query we supplied. This is something that should never happen, or the user wouldn't have been able to log in in the first place, so I'm happy to let an exception be thrown. Normally, we'd be logging these types of exceptions so that we can handle them appropriately.

Next, we will build up the `Order` object from the properties on the view model that we receive as a parameter to the action method. Most of this is plain object mapping, but for the order items, we use a LINQ `Select` statement to project the list of items from the view model into a new list of `OrderItem` objects that we can store in the database. This is very similar to how we project entity models into view models when querying the database.

Finally, we add the `Order` object to the `Orders` collection in our database context, then save changes to ensure that the order is persisted regardless of anything else that happens in the rest of the action method. The next and final thing we'll do is process the user's payment, but regardless of the outcome to that action, we want to make sure that the order is persisted so that we have a record of it should the user want to retry their payment at a later date. Remember that the `PaymentStatus` property of this order will default to `Pending`, so if anything goes wrong with the payment code, we'll still know not to process the order until the payment status changes.

Calculating the total order price

Before we charge the payment to the customer, we need to calculate the total order price. We've already talked about this but, to reiterate, the calculation we did earlier cannot be trusted as it was done on the client side of our application. Client-side code is inherently vulnerable to malicious users, seeing as they have full access to it if they know how. Therefore, we didn't even bother to send the order total as part of the form submission, as we'll be calculating it again now that we're safely on the server side of the application.

Directly after saving the order to the database, add the following:

```
var total = await _db.Orders
    .Where(x => x.Id == order.Id)
    .Select(x => Convert.ToInt32(x.Items.Sum(i =>
        i.ProductVariant.Price * i.Quantity) * 100))
    .SingleOrDefault();
```

Since we called `SaveChangesAsync`, the `order.Id` property was generated for us so that we can now use it in this query. We find the single order in the database that matches the order ID that was just created, then use a LINQ `Select` statement to return an integer value that is calculated as the sum of the product variant price multiplied by the quantity of each order item. When sending this price to Stripe for charging the user, we need to send it as a whole number in pence (GBP), if you're in the UK like me. As such, I've multiplied the total by 100 to get the price in pence rather than pounds. Stripe supports multiple currencies, so if you're not in the UK, then do check their documentation for how to use your own national currency.

Processing the payment with Stripe

With the order total calculated, we can send the details to Stripe to confirm the payment. Add the following code to the same action method:

```
var charges = new StripeChargeService();
var charge = await charges.CreateAsync(new StripeChargeCreateOptions
{
    Amount = total,
    Description = $"Order {order.Id} payment",
    Currency = "GBP",
    SourceTokenOrExistingSourceId = model.StripeToken
});

if (string.IsNullOrEmpty(charge.FailureCode))
{
    order.PaymentStatus = PaymentStatus.Paid;
}
else
{
    order.PaymentStatus = PaymentStatus.Declined;
}

await _db.SaveChangesAsync();

return Ok(new CreateOrderResponseViewModel(order.Id, order.PaymentStatus));
```

We start by creating a new instance of the `StripeChargeService` class before invoking its `CreateAsync` method. Stripe refers to payments as **charges**, so when we create a charge, we are committing the payment to Stripe for processing, and the user's card details that we validated earlier will be charged for the total we provide. The `CreateAsync` method takes an instance of the `StripeChargeCreateOptions` class, which models the details that we need to create a charge with Stripe.

The `Amount` property of this options class is the total that we just calculated, again remembering that it must be in pence rather than pounds. The `Description` property can be anything you like, but it is worth setting this to something that makes it immediately obvious what the payment was for if you intend to use the Stripe dashboard regularly. We've discussed currency already, but you'll need to set the `Currency` property depending on where you live. In this case, it's GBP. The `SourceTokenOrExistingSourceId` property is what Stripe will use to identify the card details to use for the payment, and must be set to the token that we received from our client-side Stripe API call.

In this case, we receive that token on the `model.StripeToken` property, which is what we assign. As a quick recap, when we submitted the payment card details to Stripe from the client side of our app, the card details were verified and stored on Stripe's servers, ready to be charged when we were ready to do so. The token we received in return is now used to identify those card details to be used with this payment request.

After calling the `CreateAsync` method, we receive an instance of the `StripeCharge` class in return, which contains a `FailureCode` property to work out if the payment was successful or not. If it was successful, we change the payment status of the order to `Paid`, and to `Failed` if it wasn't. We can then call `SaveChangesAsync` again to persist this change into the database. As previously mentioned, we're keeping things simple here and choosing not to persist any of the Stripe charge response details, as all of this information is readily available on the Stripe dashboard. However, if you'd prefer, you could extend your data model further in order to store the full response if you really wanted to.

Finally, we return an instance of our own `CreateOrderResponseViewModel`, which consists of the order ID and payment status values, which we then use in the client-side `CheckoutSuccess` component that we created earlier.

In the preceding code, we used a number of view models that we haven't created yet, so let's start by creating the `Features/Orders/CreateOrderViewModel.cs` file, which looks like this:

```
namespace ECommerce.Features.Orders
{
    public class CreateOrderViewModel
    {
        [Required]
        public string StripeToken { get; set; }
        [Required]
        public string FirstName { get; set; }
        [Required]
        public string LastName { get; set; }
        [Required]
        public string Address1 { get; set; }
        public string Address2 { get; set; }
        [Required]
        public string TownCity { get; set; }
        [Required]
        public string County { get; set; }
        [Required]
        public string Postcode { get; set; }
        public List<OrderItemViewModel> Items { get; set; }
    }
}
```

The associated `Features/Orders/OrderItemViewModel.cs` file, which looks like this:

```
namespace ECommerce.Features.Orders
{
    public class OrderItemViewModel
    {
        public int ProductId { get; set; }
        public int ColourId { get; set; }
        public int StorageId { get; set; }
        public int Quantity { get; set; }
    }
}
```

Finally, we'll create the `Features/Orders/CreateOrderResponseViewModel.cs` file, which looks like this:

```
namespace ECommerce.Features.Orders
{
    public class CreateOrderResponseViewModel
    {
        public int OrderId { get; set; }
        public string PaymentStatus { get; set; }

        public CreateOrderResponseViewModel(int orderId,
            PaymentStatus paymentStatus)
        {
            OrderId = orderId;
            PaymentStatus = paymentStatus.ToString();
        }
    }
}
```

This now completes everything we need for placing orders and processing payments. The last API we need to provide for our client-side functionality is the list of orders our users will see on their my account page.

Adding an order list API endpoint

We have one last action method to add, which, thankfully, is a very simple one. If you've closed it, open the `Features/Orders/Controller.cs` file up again, and add the following action method beneath the one we already defined:

```
[HttpGet]
public async Task<IActionResult> List()
{
```

```
var orders = await _db.Orders
    .Where(x => x.User.UserName == User.Identity.Name)
    .Select(x => new OrderListViewModel
    {
        Id = x.Id,
        Placed = x.Placed,
        Items = x.Items.Sum(i => i.Quantity),
        Total = x.Items.Sum(i => i.ProductVariant.Price * i.Quantity),
        PaymentStatus = Enum.GetName(typeof(PaymentStatus),
            x.PaymentStatus)
    })
    .ToListAsync();

return Ok(orders);
}
```

We simply query the database for all orders where the associated user's `UserName` property matches the username of the currently logged in user, then project them to a new list of `OrderListViewModel` objects. Again, we'll create this in a moment, but it just contains the properties that we display in the table of orders on the my account page of our UI. The `Id` and `Placed` properties are very simple, but the rest are not so obvious. The `Items` property is the total count of items ordered, so we use the LINQ `Sum` operator on the `Quantity` field of all the items in the order. This totals all of the item quantities and gives us the total number of phones for each individual order. The `Total` field is essentially the same calculation that we did before sending the payment off to Stripe for processing. However, this time, we don't multiply by 100 at the end as we want to display this total in pounds and pence, rather than just pence.

The `PaymentStatus` field is the most interesting, because it requires a bit of hacking to get the string representation of the enum when querying directly from the database. Normally, we could have just done `x.PaymentStatus.ToString()` to get the string value that we want, but in this case, it always returns the numeric value instead. Presumably, this is because enums are stored as numeric values in the database when using them with EF models, and this operation by default will run at the database level. Therefore, by calling `x.PaymentStatus.ToString()` at this point, we'd just be calling it on a numeric SQL value rather than a C# enum. To fix this, we have to use the static `Enum.GetName` method, passing it the type of the enum we want to evaluate, and the value we want the name string for. This will force the query to hit the database first, before returning back to our C# code to evaluate this expression in order to return the data we want, but there isn't really a much nicer way of doing this with enums.



At the time of writing, EF Core 2.1 has been released, which includes value converters that enable us to store enum string representations within the database!

The view model we need for the preceding action method is the `Features/Orders/OrderListViewModel.cs` class, which looks like this:

```
namespace ECommerce.Features.Orders
{
    public class OrderListViewModel
    {
        public int Id { get; set; }
        public DateTime Placed { get; set; }
        public int Items { get; set; }
        public decimal Total { get; set; }
        public string PaymentStatus { get; set; }
    }
}
```

And, with this, we have finished our server-side API changes that are required for the client-side features we added earlier.

Summary

We've covered a lot of ground in this chapter. We started off by looking at the reasons why we'd use Stripe for payment processing over competitors such as PayPal. We then proceeded to build a number of new client-side UI components for collecting the information we need in order to handle order processing on the server, as well as to keep our users informed of their order state. We also included rich client-side validation for the first time, using the popular `VeeValidate` package.

We looked at how Stripe processes payments and how we remain PCI-compliant by not storing any sensitive payment card information on our own servers. Instead, we send these to Stripe's API to deal with, meaning that we can deal with simple tokens to pass between our client and server in order to trigger the final payment from the safety of our server-side C# code.

We then moved on to the server side of our application and extended the data model so that we can store orders in the database. We also added a new controller for handling new orders, as well as listing all of the orders a user has placed. We looked at how to trigger payments from the server using our client-generated Stripe token, and how to tell if the payment was successful or not.

It's been another long chapter, but we really have only scratched the surface of what we can do with the combination of Stripe, Vue, and ASP.NET Core. In the next chapter, we're going to build an admin panel so that you can actually get your new products into the database without manually seeding them, as we've done so far.

9

Building an Admin Panel

From the perspective of a customer, our online shop is now feature complete. They can browse our product catalog with the added abilities of sorting, filtering, and searching to help them find what they are looking for. They can then add their chosen products to their shopping cart before proceeding through the checkout process to pay for their order using the Stripe payment service. Finally, they can visit their account page to view a list of all of their previously placed orders.

However, right now, all of our products are hardcoded to be seeded manually into the database when the application starts. It's not ideal to need to deploy a new version of the application every time we need to add a new product, so in this chapter we're going to build a very basic admin panel where only those users with the Admin role can add new products to the catalog at any time.

The topics we're going to cover in this chapter include the following:

- Role-based access on both the client and server
- Reducing duplication using Vue component inheritance
- Building a custom type ahead component
- Building a custom multi-select component
- Vee-Validate remote validation
- Nested Vue-Router route definitions

Extending the authentication endpoint with user roles

The current implementation of our authentication API endpoint already adds the users' roles to the JWT token in the form of claims. This is all we needed to do in order to support role-based authorization on the server side of the application, as ASP.NET Identity automatically decodes the JWT for us so that the claims we added are available on the `HttpContext.User.Identity` claims principle object.

However, on the client side of our application, those claims are still encoded into the JWT and we cannot access them without decoding it. This isn't a huge issue, and we could certainly bring in a new npm package that would be able to decode a JWT for us, but we don't really need to. Instead, what we'll do is return the list of roles a user is assigned to as part of the view model we send back to the client when they first authenticate. We can then store these roles in the Vuex store state of our application so that we have access to them from any client-side component. This is by far the simplest approach, with no real downside. The only thing to note is that we must always remember that client-side code is inherently insecure. As such, we still need to ensure that any API endpoint that serves role-specific data is also locked down using role-based authorization on the server, just in case a malicious user is able to bypass our page-level access restrictions on the client.

Start by opening up the `Features/Authentication/Controller.cs` file, then change the `return` statement at the bottom of the `GenerateToken` method as follows:

```
return new TokenViewModel
{
    AccessToken = new JwtSecurityTokenHandler().WriteToken(token),
    AccessTokenExpiration = expires,
    FirstName = user.FirstName,
    LastName = user.LastName,
    Roles = roles
};
```

We already created the `roles` variable to encode them into the JWT as claims, so it's just a simple matter of returning the raw list as part of the `TokenViewModel` we return to the client. Speaking of which, the updated version of said view model looks like this:

```
namespace ECommerce.Features.Authentication
{
    public class TokenViewModel
    {
        [JsonProperty("access_token")]
        public string AccessToken { get; set; }
        [JsonProperty("access_token_expiration")]
        public DateTime AccessTokenExpiration { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public IEnumerable<string> Roles { get; set; }
    }
}
```

That's all there is to it. After a user successfully authenticates, we've already configured our Vuex mutation in such a way that the full object response from the API call is persisted into the global application state. This means that the `roles` property we just added will now also be available to us on the client, as part of the `auth` object.

Client-side role-based authorization

Now that we have access to the set of roles the current user belongs to, we can start preventing them from accessing certain pages of the application, depending on which roles they are assigned to. To make this process easier, we'll add the following Vuex getter function to the `ClientApp/store/getters.js` file:

```
export const isInRole = (state, getters) => role => {
    const result = getters.isAuthenticated && state.auth.roles.indexOf(role)
    > -1;
    return result;
};
```

As the name suggests, this getter function determines whether the current user is in a specific role, which we pass in as an argument. This is quite different to the getter functions we've defined before, as in addition to the `state` parameter we also make use of the optional `getters` parameter that we also have access to if we need it. On top of this, we then use fat arrow syntax to declare the `role` argument, which represents the value we pass in to the function to check for.

The function body itself is more straightforward. In order to determine whether a user is in a specific role, we first need to check whether the user has even logged in yet. The `getters` parameter gives us access to all of the other `getter` functions we've defined, including the `isAuthenticated` function, which we use to check whether the user is logged in or not. If they are, we proceed to check the `state.auth.roles` property to see whether the role we pass in exists within it or not, by checking whether its index is greater than `-1`. If both of these checks resolve to `true`, then the user is in the role we received as the function argument; if they don't both resolve to `true`, they aren't.

Adding role checks to client-side routes

Now that we have an easy way of determining whether the current user is in a specific role or not, we need to extend our route guard functionality to include role checking as well as authentication status. To make sure only logged in users are allowed to proceed the checkout, we added a `meta` property to the checkout page route definition, as well as a router hook that checks whether the user is authenticated before allowing them to access the page. We need to extend this functionality to support an additional `meta` property, which we can use to specify which user role is required to access a specific page.

Let's start by modifying the router's `beforeEach` hook to check for a `meta.role` property on the route we're trying to navigate to, and if it finds one, ensure that the current user is in the correct role using the Vuex getter we just defined. Open up the `ClientApp/boot.js` file and locate the `router.beforeEach` hook that we defined earlier. Start by making the following amendments:

```
if (to.matched.some(route => route.meta.requiresAuth)) {
  if (!store.getters.isAuthenticated) {
    store.commit("showAuthModal");
    next({ path: from.path, query: { redirect: to.path } });
  } else {
    if (
      to.matched.some(
        route => route.meta.role && store.getters.isInRole(route.meta.role)
      )
    ) {
      next();
    } else if (!to.matched.some(route => route.meta.role)) {
      next();
    } else {
      next({ path: "/" });
    }
  }
} else {
```

```
//...omitted for brevity
//...we'll make some more changes here shortly...
}
```

This looks fairly involved at first glance, but if we break it down, it's actually pretty simple. Notice that the code we've added is wrapped inside an `if` block that's already checked whether the route we're trying to navigate to requires authentication. Following that, we've also verified that the user is authenticated by triggering the login modal if they haven't. So, now that we know the user is authenticated, we can check to see whether the route we're trying to access has a role restriction. We can do this with three separate checks to decide what to do:

1. `If`: The `to` route has the `meta.role` property set, and the user is in the role specified by its value; we allow them to proceed by invoking the `next` callback directly.
2. `Else if`: The `to` route does not have the `meta.role` property set; we also allow them to proceed by invoking the `next` callback directly.
3. `Else`: The user does not have access to this route, so we redirect them to the home page by invoking the `next` callback and overloading it with a different `path` property.

At this point, it might seem as though we are done. However, so far we've only added role checking to routes that require authentication. It may not be immediately obvious that we also need to perform the same checks on routes that don't necessarily require authentication. As an example, think about the shopping cart page of our app; you don't have to be authenticated to add products to your cart or access the shopping cart page to view them, but we may not want admin users placing orders. As such, we can allow them to add products to their cart before they authenticate, but once they log in, we prevent them from accessing the shopping cart page, so they can't proceed with the order.

In order to fix this, we need to make some similar changes in the `else` block from the preceding code example (there's a comment to make it more obvious!). These changes look like this:

```
if (to.matched.some(route => route.meta.requiresAuth)) {
  //...previous sample omitted for brevity
} else {
  if (
    to.matched.some(
      route =>
        route.meta.role &&
        (!store.getters.isAuthenticated ||
          store.getters.isInRole(route.meta.role))
    )
  )
```

```
) {
  next ();
} else {
  if (to.matched.some(route => route.meta.role)) {
    next({ path: "/" });
  }

  next ();
}
}
```

Again, this looks more complex than it actually is. This time, we only have two checks to make:

1. If the `to` route has the `meta.role` property set, and either the user has not yet authenticated or they are in the role defined as the value of the `meta.role` property, we allow them to proceed by invoking the `next` callback directly.
2. Else, we check if the `to` route has the `meta.role` property set, and if so overload the `next` callback with a different `path` value because the user cannot access this route. Otherwise, we simply invoke the `next` callback directly because the user can proceed.

To test out these changes, we need to actually go ahead and add the `meta.role` property to a few of our existing route definitions. Slightly further up in the `ClientApp/boot.js` file, locate the `routes` array and modify it as follows:

```
const routes = [
  { path: "/products", component: Catalogue },
  { path: "/products/:slug", component: Product },
  { path: "/cart", component: Cart, meta: { role: "Customer" } },
  {
    path: "/checkout",
    component: Checkout,
    meta: { requiresAuth: true, role: "Customer" }
  },
  {
    path: "/account",
    component: Account,
    meta: { requiresAuth: true, role: "Customer" }
  },
  { path: "*", redirect: "/products" }
];
```

If you re-run the application again now, then log in as our existing admin user account, you should be prevented from accessing either the **Cart**, **Checkout**, or **My Account** pages.

Server-side role-based authorization

Remembering that we can't trust client-side authorization checks alone, the final change we need to make to prevent admin users placing orders is to protect the API endpoint that stores the order and processes the payment information. Open up the `Features/Orders/Controller.cs` file and amend it as follows:

```
[HttpPost, Authorize(Roles = "Customer")]
public async Task<IActionResult> Create([FromBody] CreateOrderViewModel
model)
{
    //...method body omitted for brevity
}
```

That's all there is to it. In this instance, we only allow the single `Customer` role to place orders, but if we had multiple roles, then we could pass a comma-separated list of roles here instead.

Hiding UI elements based on role

So, we're now preventing admin users from placing orders by locking down the API endpoint to customer users only; we are also preventing admin users from accessing the shopping cart, checkout, and my account pages, seeing as they are no use to a user who can't place orders. However, the links to those pages are still visible, and this isn't a good user experience, so let's go ahead and hide the links that admin users shouldn't be able to see.

The first thing we're going to do is hide the cart summary widget in the main navbar. Open up the `ClientApp/components/App.vue` file, then make the following amendments to the `template` section:

```
<b-navbar-nav class="ml-auto mr-4">
  <cart-summary v-if="isCustomer" />
  <auth-nav-item />
</b-navbar-nav>
```

Very simply, we conditionally render the cart summary component based on some kind of `isCustomer` property, which we're yet to define. This property will be defined as a new computed property, and looks like this:

```
computed: {
  ...
  isCustomer() {
```

```
    return (
      this.$store.getters.isInRole("Customer") ||
      !this.$store.getters.isAuthenticated
    );
  }
}
```

Again, we make use of the `isInRole` getter function we defined earlier, but in this instance, it isn't enough to simply check whether the user is in the `Customer` role. We still need to show the cart summary widget to users who have not yet authenticated. As such, we assume that the current user is a customer if they belong to the `Customer` role or they are not yet logged in. This takes care of the only link to the shopping cart page, and subsequently the only link to the checkout page, seeing as it's situated on the shopping cart page itself.

Next, we need to hide the link to the my account page in the dropdown that appears after a user has logged in. Open up the `ClientApp/components/app/AuthNavItem.vue` file, and make the following changes to the `template` section:

```
<b-dropdown-item v-if="isCustomer" to="/account">
  <i class="fas fa-user"></i>
  My Account
</b-dropdown-item>
```

As we did before, we conditionally display this drop-down item based on an `isCustomer` property, which we'll now define as another computed property:

```
computed: {
  ...
  isCustomer() {
    return this.$store.getters.isInRole("Customer");
  }
}
```

In this instance, we only need to check whether the user belongs to the `Customer` role, as this component is only ever visible after a user has already logged in. Re-running the application again now should show that when logged in as an admin user, you no longer see links to the pages we prevented access to.

Building the admin panel components

The admin panel section of our app is going to be a little different to what we've done so far, whereby we have a distinct page of the application for each feature. Instead, we're going to define a single entry point route definition, which contains a collection of nested routes for each section of the admin panel.

This should make more sense when we take a look at the `template` section of a new page-level component that we need to make. It belongs in the `ClientApp/pages/admin/Index.vue` file and looks like this:

```
<template>
  <b-container class="page pt-4">
    <b-row>
      <b-col cols="3">
        <b-list-group>
          <b-list-group-item to="/admin/orders">
            <i class="fas fa-shopping-cart mr-2"></i>
            Orders
          </b-list-group-item>
          <b-list-group-item to="/admin/products">
            <i class="fas fa-mobile mr-2"></i>
            Products
          </b-list-group-item>
        </b-list-group>
      </b-col>
      <b-col cols="9">
        <router-view />
      </b-col>
    </b-row>
  </b-container>
</template>
```

Essentially, all we're doing here is rendering two standard columns using the Bootstrap grid system. The first of these columns contains a Bootstrap list group control, which we're using as a type of subnavigation menu, and the second (wider) column contains a `<router-view />` component. You may be wondering how this can possibly work, seeing as we're already rendering this page level component into a similar `<router-view />` component in the root level `App.vue` component. The answer is simply that Vue and Vue-Router support nested routes in much the same way that ASP.NET supports nested master or layout pages.

We'll look at the route definitions in a moment, but essentially what we are doing here is defining another (nested) layout component specific to the admin panel. The subnavigation menu in the first column will be displayed on any child route of the `/admin` URL, and the `<router-view/>` component in the second column will be used to display the nested page component. If you look closer at the list group in the first column, you'll notice we've defined links to both the `/admin/orders` and `/admin/products` URLs—these are the two main pages we'll be creating to form the admin panel.

Configuring nested route definitions

So, we've seen how to render child route level components in the `template` section of a parent component, but how do we configure this parent-child relationship with Vue-Router? As with most things in the Vue ecosystem, it's actually incredibly easy. Open up the `ClientApp/boot.js` file, as we have a few changes to make here. Firstly, find our existing list of page level component imports, and add the following additional imports, as shown in the following code snippet:

```
//import admin pages
import AdminIndex from "../pages/admin/Index.vue";
import AdminOrders from "../pages/admin/Orders.vue";
import AdminProducts from "../pages/admin/Products.vue";
import AdminCreateProduct from "../pages/admin/CreateProduct.vue";
```

Don't worry if you start seeing errors at this point, as only one of these components actually exists—we'll create the others shortly. Next, find the existing `routes` array and modify it as follows:

```
const routes = [
  ...
  {
    path: "/admin",
    component: AdminIndex,
    meta: { requiresAuth: true, role: "Admin" },
    redirect: "/admin/orders",
    children: [
      {
        path: "orders",
        component: AdminOrders
      },
      {
        path: "products",
        component: AdminProducts
      },
      {
```

```
      path: "products/create",
      component: AdminCreateProduct
    }
  ],
  { path: "*", redirect: "/products" }
];
```

All we have to do is declare a `children` array property containing a list of nested route definitions that belong to the parent route. In this case, we define a root level `/admin` route, which renders the `Index.vue` component we just created but automatically redirects to the `/admin/orders` route by default. It has three child routes declared in its `children` array, one of which is the `/admin/orders` route we just discussed. The next is a `/admin/products` route, which will display our existing list of products. Finally, we have the `/admin/products/create` route, which will display a form for creating new products. Now, there is no limit to how many levels deep you can nest route definitions. We could easily have declared the last route as a child of the `/admin/products` route, but to keep things simple, we've left it at a single level of nesting.

There are a few caveats to be aware of when declaring child route definitions, the first of which is that the `path` value we assign is appended to that of the parent route. In this case, the parent route path is `/admin`, and the first child route path is `orders`. When accessing the nested orders page, we must use the full `/admin/orders` URL. With this knowledge, you'd expect that child route definitions would also inherit the `meta` property of their parent as well. Unfortunately, this isn't the case, so if you wanted to be completely explicit and ensure that each child route of the admin panel requires authentication, you'd need to duplicate the `meta` property on each one. However, it isn't necessary as long as we put some thought into how our role checking route guard works—which we already have! As a reminder, we do things like this:

```
if (to.matched.some(route => route.meta.requiresAuth) { ... }
```

The key thing to note here is that the `matched` property from the `to` route is an array of route definitions that match the URL we're trying to access. This means that if we try to navigate to the `/admin/orders` URL, we'd get both the `/admin` and `/admin/orders` route definitions in the `to.matched` array. Then, because we use the native JavaScript `Array.some` function, we're looking to see whether any of the matched routes require authentication. In our case, it means we don't need to duplicate the `meta` property, as navigating to any child route will result in the parent being checked for authentication restrictions.

Refactoring components for reuse

The first page in our admin panel is the orders page, where we need to display a list of all orders placed by our customers. This sounds incredibly similar to the order list component that we built for displaying the customer-specific list of orders on the my account page. Rather than repeat ourselves, we can make some minor tweaks to the existing component so that it can be used in both contexts.

From the perspective of a customer, it's already doing what it needs to do, as the orders it displays are passed in via props, and are limited to only those orders placed by the currently logged in customer. From the perspective of an admin, we can display the orders for all customers by simply passing in a different list of orders. However, it would be good to quickly see which customer each order belongs to. We can add an additional column for this, but that column doesn't make sense in the context of a customer as they can only see their own orders anyway. We already have a simple way of determining whether a user is in a specific role, so we can conditionally display this column based on the users' role.

The first thing we're going to do is move the `OrderList.vue` component out of the `ClientApp/components/account` folder and into the new `ClientApp/components/shared` folder, seeing as we're now using it from multiple pages. Next, in the `template` section, we need to modify the `thead` section of the orders table as follows:

```
<thead>
  <tr>
    <th>Order #</th>
    <th v-if="isAdmin">Customer</th>
    <th>Placed</th>
    <th>Items</th>
    <th>Total</th>
    <th>Payment status</th>
  </tr>
</thead>
```

As discussed, we conditionally display an additional table header based on an `isAdmin` computed property that we'll define shortly. We need to make a couple of very similar changes to the `tbody` section of the same order's table element:

```
<tbody>
  <template v-if="orders && orders.length > 0">
    <tr v-for="order in orders" :key="order.id">
      <td>{{ order.id }}</td>
      <td v-if="isAdmin">{{ order.customer }}</td>
      <td>{{ order.placed | date }}</td>
      <td>{{ order.items }}</td>
```

```
        <td>{{ order.total | currency }}</td>
        <td>{{ order.paymentStatus }}</td>
    </tr>
</template>
<tr v-else>
    <td v-if="isAdmin" colspan="6">There are no orders to display.</td>
    <td v-else colspan="5">You haven't placed any orders yet!</td>
</tr>
</tbody>
```

Again, we conditionally display the customer column based on the `isAdmin` computed property. We've also used the `v-if` and `v-else` directives on a pair of `td` elements to change the text we use if there are no orders to display, again based on whether or not the user belongs to the admin role. As you've probably already guessed, the `isAdmin` computed property is very similar to the `isCustomer` computed property we used earlier:

```
computed: {
  isAdmin() {
    return this.$store.getters.isInRole("Admin");
  }
}
```

This completes the changes we need to make to the component itself, but now that we've moved it to a different folder, we need to update the `my account` page to import it from the shared folder instead. Open up the `ClientApp/pages/Account.vue` file and make the following minor change:

```
...
import OrderList from "../components/shared/OrderList.vue";
...
```

We can now create the `ClientApp/pages/admin/Orders.vue` file, which is the next of our nested page components that will reuse the same `OrderList.vue` component that we've just refactored. The `template` section for this new page component looks like this:

```
<template>
  <order-list :orders="orders" />
</template>
```

This is so simple that it needs no explanation, so we'll move straight on to the associated `script` section. This is a bit more involved, so like we did previously, we'll build it up in stages. The base component definition looks like this:

```
<script>
import axios from "axios";
import OrderList from "../..//components/shared/OrderList.vue";
```

```
export default {
  name: "orders",
  components: {
    OrderList
  },
  ...
};
</script>
```

We will start by importing `axios`, as we know we'll need to fetch the order list from the API, and then we'll import the `OrderList.vue` component that we just refactored and moved into the shared components folder. Finally, we will export a default component definition object with the name `orders` and a `components` declaration object to include the `OrderList` component we just imported.

The data requirement of this component is very simple, as all we need is an `orders` property to store the orders that we pass down as a prop to the order list component:

```
data() {
  return {
    orders: null
  };
}
```

As with our other page level components, we need a `setData` method for assigning the data that we fetch from the API to our data object:

```
methods: {
  setData(orders) {
    this.orders = orders;
  }
}
```

And finally, we need a `beforeRouteEnter` hook to fetch the data we need from the API, before calling the `setData` method we just defined:

```
beforeRouteEnter(to, from, next) {
  const vm = this;
  axios.get("/api/orders").then(response => {
    next(vm => vm.setData(response.data));
  });
}
```

Notice how we fetch the orders list from the `/api/orders` endpoint. This is the same endpoint we use for the order list on the my account page, which currently only returns the orders placed by the user who calls it. As such, we'll need to tweak this to return all customers orders if that user belongs to the admin role. We also need to add a new `Customer` property to the model we return to cater for the additional needs of the order list component in the context of an admin user. The updated `List` action method on the `Features/Orders/Controller.cs` file looks like this:

```
[HttpGet]
public async Task<IActionResult> List()
{
    var orders = await _db.Orders
        .Where(x => User.IsInRole("Admin") || x.User.UserName ==
        User.Identity.Name)
        .Select(x => new OrderListViewModel
        {
            Id = x.Id,
            Customer = x.User.FullName,
            Placed = x.Placed,
            Items = x.Items.Sum(i => i.Quantity),
            Total = x.Items.Sum(i => i.ProductVariant.Price * i.Quantity),
            PaymentStatus = Enum.GetName(typeof(PaymentStatus),
            x.PaymentStatus)
        })
        .ToListAsync();

    return Ok(orders);
}
```

We've updated the LINQ `Where` statement in the preceding query to first check whether the current user belongs to the admin role. If they do, the `or` statement will terminate early and simply return all orders in the database. If they don't, it will continue to work as it did before, by only returning the orders that belong to the logged in user. As mentioned previously, we've also added the `Customer` property to the model we return, which now looks like this:

```
public class OrderListViewModel
{
    public int Id { get; set; }
    public string Customer { get; set; }
    public DateTime Placed { get; set; }
    public int Items { get; set; }
    public decimal Total { get; set; }
    public string PaymentStatus { get; set; }
}
```

Product list component

The order list is only the first part of our admin panel, and by far the simplest. The main feature we're ultimately aiming for here is a form component for creating new products. However, before we get that far, we need to display the existing list of products. We've already added the route definition for this component, as well as a link to it from the parent component.

Create a new `ClientApp/pages/admin/Products.vue` file, and start it off with the following template section:

```
<template>
  <div>
    <h3 class="float-left">Products</h3>
    <b-button variant="primary" to="/admin/products/create"
      class="float-right mb-4">
      <i class="fas fa-plus"></i>
      Add new product
    </b-button>
    <table class="table table-striped table-hover">
      ...
    </table>
  </div>
</template>
```

Nothing particularly special here, just the page title and a button, floated to the left and right, respectively, followed by an empty table element which will hold our list of products. This button will eventually link to the Create Product page, but until we define that component, it won't work. The missing section of the products table looks like this:

```
<thead>
  <tr>
    <th>#</th>
    <th>Name</th>
    <th>Short description</th>
    <th>Price</th>
  </tr>
</thead>
<tbody>
  <template v-if="products && products.length > 0">
    <tr v-for="product in products" :key="product.id">
      <td>{{ product.id }}</td>
      <td>{{ product.name }}</td>
      <td>{{ product.shortDescription }}</td>
      <td>{{ product.price | currency }}</td>
    </tr>
  </template>
```

```
<tr v-else>
  <td colspan="3">There are no products to display.</td>
</tr>
</tbody>
```

Again, there's nothing new here; the template markup of this component is virtually the same as the order list component, so we won't dwell on it any more. The `script` section for this component is also very similar to the Order List page, so much so that it would be a good test for you to try and complete it before reading the following code sample.

Done? Now, compare it with the following:

```
<script>
import axios from "axios";

export default {
  name: "products",
  data() {
    return {
      products: null
    };
  },
  methods: {
    setData(products) {
      this.products = products;
    }
  },
  beforeRouteEnter(to, from, next) {
    const vm = this;
    axios.get("/api/products").then(response => {
      next(vm => vm.setData(response.data));
    });
  }
};
</script>
```

As a quick recap:

1. We import `axios` for fetching data from our API
2. We export a default component object
3. We name the component `"products"`
4. We define a data object that contains a single `products` property for storing the data returned from the API

5. We define a single `setData` method for assigning the data returned from the API to the `products` property of the component's data object
6. We define a `beforeRouteEnter` hook to trigger the API call before invoking the `setData` method

Creating a product form component

The last page of our admin panel is the Create Product page, which on the face of it should be fairly simple as it's just a standard form that posts some data back to our API. However, when adding a new product, we have fairly complex requirements whereby we need to select a value from a variable length list of options—potentially more than we'd want to add to a standard HTML `select` element. We also need to support the addition of a variable-length list of complex product variant objects.

Create a new `ClientApp/pages/admin/CreateProduct.vue` file, with an initial template section that looks like this:

```
<template>
  <div>
    <div class="clearfix">
      <h3 class="float-left">Add a new product</h3>
      <b-button variant="link" class="float-right"
        to="/admin/products">Back to product list</b-button>
    </div>

    <form class="mt-4 mb-4" @submit.prevent="save">
      ...
    </form>

    <add-variant-modal
      :colours="colours"
      :storage="storage"
      @submit="addVariant" />
  </div>
</template>
```

The full template section of this component is going to be pretty long due to the amount of fields we have to render, so we're going to build it up in stages. What we've added in the preceding code snippet should be fairly self-explanatory, but so far we have:

- A title and link back to the product list page, floated to the left and right, respectively

- An empty form shell with an `@submit` event handler preventing the default action and invoking a `save` method instead
- A custom `<add-variant-modal />` component, which is yet to be defined, but will ultimately show a modal window containing a secondary form for adding a product variant

Before we begin filling in the missing form fields, cast your mind back to the previous chapter where we built out the checkout form component. We had a fair amount of duplication in the way we rendered each form field, which looked something like this:

```
<b-form-group>
  <label>First name</label>
  <b-form-input v-model="firstName" data-vv-name="first name" v-
    validate="'required|min:3'" :state="state('first name')" />
  <b-form-invalid-feedback>
    {{ errors.first('first name') }}
  </b-form-invalid-feedback>
</b-form-group>
```

That's a fair amount of HTML markup for a single form field, especially considering most of it does not change much, aside from the label text and the data property we bind to the input. Even if we were to render a `select` element rather than a text input, we'd still need the wrapping `<b-form-group />` component, the `<label>` element, and the `<b-form-invalid-feedback />` component for displaying validation errors. The other issue here is that we used a component level method called `state` for working out whether the input should be highlighted as valid/invalid, or left in its default state. Now that as we're declaring form fields in a different component, we'd likely need to duplicate that method to use here, which is certainly not very DRY.

It would be much cleaner if we could do something more like this instead:

```
<form-input
  label="Name"
  name="first name"
  :error="errors.first('first name')"
  v-model="firstName"
  v-validate="'required|min:3'" />
```


Much more concise and easy to read! In fact, the only reason this isn't a single line of code is to make it more legible on narrower screen widths. For now, let's assume that this is exactly how we can render each of the form fields of the Create Product form, as we'll come back and define this component shortly. Back in the `template` section of the `ClientApp/pages/admin/CreateProduct.vue` file, start adding the following form fields between the empty form element tags:

```
<form-input
  label="Name"
  name="name"
  :error="errors.first('name') "
  v-model="product.name"
  v-validate="'required|min:10|uniqueProductName' " />

<form-input
  label="Short description"
  name="short description"
  :error="errors.first('short description') "
  v-model="product.shortDescription"
  v-validate="'required|min:10' " />

<form-text-area
  label="Description"
  name="description"
  :rows="5"
  :error="errors.first('description') "
  v-model="product.description"
  v-validate="'required|min:10' " />
```

Each of these components expects a number of props to control the differences in the content that they render. The `label` prop specifies the text in the label element, the `name` prop specifies the name of the field in the context of client-side validation, the `error` prop controls the visibility and text of the validation error displayed if we pass a value, and we then use `v-model` and `v-validate` as we would with any other native input element. In addition to this, the last component renders a text area rather than an input, and as such it has an additional `rows` prop to specify the number of rows to display.

Next, add the following markup directly after the previous components:

```
<b-row>
  <b-col>
    <form-input
      type="number"
      label="Talk time"
      name="talk time"
      :error="errors.first('talk time') "
```

```

        append="hours"
        v-model="product.talkTime"
        v-validate="'required|decimal'" />
</b-col>
<b-col>
  <form-input
    type="number"
    label="Standby time"
    name="standby time"
    :error="errors.first('standby time')"
    append="hours"
    v-model="product.standbyTime"
    v-validate="'required|decimal'" />
</b-col>
<b-col>
  <form-input
    type="number"
    label="Screen size"
    name="screen size"
    :error="errors.first('screen size')"
    append="inches"
    v-model="product.screenSize"
    v-validate="'required|decimal'" />
</b-col>
</b-row>

```

Here, we're rendering three of our new `<form-input />` components side-by-side using a standard Bootstrap grid. You may have noticed that on each of these three components we've specified the `type` prop, which we're using here to override the default input type of text to number instead. We've also used an `append` prop, which may not make much sense just yet. If you're familiar with Bootstrap at all, you'll know that they have styles for creating input groups. These are custom elements of sorts, where things like text or buttons can be displayed either before or after an input element, in a way that it looks as though it's part of the input itself. This is exactly what we want to do here, so we're assuming we'll have some props to enable us to specify the data to either append or prepend.

After these three inputs, we'll need the following:

```

<b-row>
  <b-col>
    <typeahead
      label="Brand"
      name="brand"
      :items="brands"
      :error="errors.first('brand')"
      v-model="product.brand"
      v-validate="'required|min:3'" />

```

```
</b-col>
<b-col>
  <typeahead
    label="Operating system"
    name="operating system"
    :items="os"
    :error="errors.first('operating system')"
    v-model="product.os"
    v-validate="'required|min:3'" />
</b-col>
</b-row>
```

This is where things get a little more interesting, as instead of `<form-input />` or `<form-text-area />` components, where it's pretty easy to guess what will ultimately be displayed, we're using another custom `<typeahead />` component instead. The brand and operating system fields are the first in which we need to allow a selection from an unknown amount of potential options. Right now, our lists of brands and operating systems are fairly small, but as the product catalogue grows in size, so too could these lists of options. As such, rather than displaying them in a standard HTML `select` element where they cannot be searched or filtered on, we'll create a custom typeahead component. This will give us the added flexibility that we can allow brands or operating systems that don't yet exist to be entered, at which point we can enter a new value into the database rather than linking the product to an existing one.

The only new prop on the `<typeahead />` component is the `items` prop. To make sure this component is completely reusable, it won't be in charge of managing its own list of options, so we'll need to pass them down into the component from the parent using the `items` prop instead.

Next up, we have the product features field:

```
<multi-select
  name="features"
  label="Features"
  :items="features"
  :error="errors.first('features')"
  v-model="product.features"
  v-validate="'required'" />
```

Every product can have multiple features associated with it. We're not allowing new features to be added in order to keep things simple, but we still need a way of selecting multiple values for the same field. We could have rendered a standard checkbox list, but with rich SPA frameworks such as Vue, we can do much better. Here, we're using yet another custom `<multi-select />` component, which has an almost identical interface to that of the `<typeahead />` component we just saw.

We've now defined all of the standard inputs we need on this form, so all that's left is defining a way of adding multiple product variants. Displaying the variants that we've already added is easy, as it's a perfect fit for a standard HTML table element:

```
<div class="clearfix mt-4 mb-2">
  <h4 class="float-left">Variants</h4>
  <b-button size="sm" class="float-right" v-b-modal.variantModal>
    <i class="fas fa-plus"></i>
  </b-button>
</div>

<table class="table">
  <thead>
    <tr>
      <th>Colour</th>
      <th>Capacity</th>
      <th>Price</th>
    </tr>
  </thead>
  <tbody>
    <template v-if="product.variants.length > 0">
      <tr v-for="(item, index) in product.variants" :key="index">
        <td>{{ item.colour }}</td>
        <td>{{ item.storage }}</td>
        <td>{{ item.price | currency }}</td>
      </tr>
    </template>
    <tr v-else>
      <td colspan="3">You haven't added any variants yet</td>
    </tr>
  </tbody>
</table>
```

We start with a title and a button for toggling a modal, which we'll use to add new variants to the following table. In this case, the button makes use of the `v-b-modal` directive to specify the ID of the modal component we wish to toggle when the button is clicked—in this case, it's the modal we already rendered directly beneath the form we are currently filling out. Next, we render a simple HTML table to display the color, capacity, and price values of each variant we add by looping over the items in the `product.variants` array and rendering a table row for each. If there are no items in that array, we display a single row with appropriate text to notify the user.

The only thing to note is that as we've done before, we must wrap the first `tr` element here in a `template` tag. When using the `v-if` and `v-else` directives, if you wish to control the visibility of multiple elements with the same condition, you can wrap them in a template to avoid duplicating the `v-if` statement. In this case, we are only writing that statement once anyway, as we only declare a single table row, which may or may not be rendered multiple times. However, in this case, we still need to include the wrapping template element, or the conditionals simply don't work due to the way Vue processes them.

The last piece of this form includes a validation message for the product variants property, and the form submit button:

```
<div v-if="variantsError" class="error">
  {{ variantsError }}
</div>

<div class="clearfix">
  <b-button class="float-right" variant="primary"
  @click.prevent="save">Save product</b-button>
</div>
```

But why are we suddenly back to including explicit error messages rather than using the Vee-Validate computed `errors` property as we've done so far? With standard inputs such as text, number, and selections, Vee-Validate can handle their validation needs out of the box, without much work from us. However, we've not actually been using standard inputs in this form, as we've been using our own custom inputs instead (even though we're yet to see what they look like). We'll cover more of this later, but for now you just need to know that when building your own custom input components in Vue, if we want them to *behave* like an input, and therefore be *validated* like an input, we have to build them in a specific way. This is what we will be doing for our custom `form-input`, `form-text-area`, `typeahead`, and `multi-select` components, and ultimately we could also do it for the product variants as well.

We could have abstracted the table of product variants, along with the modal to add new ones, into another custom component that we could build to behave like an input. However, it does add additional complexity, so it is always worth weighing up the pros and cons of making the effort to do so. It's also worth seeing what the alternative approach is, so we can see clearly what the benefits of our current set of custom inputs truly are. Think about how many form inputs we've needed to render on this form, and how many individual error properties we'd need to create and track. We could do what Vee-Validate does and have a single `errors` collection, but we'd still need to manually validate each input and decide when and if an error is pushed into that collection. That being said, if this were the only custom input you needed to build, it may not be worth it to save the effort of tracking a single validation error.

The `template` section for this component is now complete, so add a `script` section with the following initial contents:

```
<script>
import axios from "axios";
import FormInput from "../../components/shared/FormInput.vue";
import FormTextArea from "../../components/shared/FormTextArea.vue";
import Typeahead from "../../components/shared/Typeahead.vue";
import MultiSelect from "../../components/shared/MultiSelect.vue";
import AddVariantModal from "../../components/admin/AddVariantModal.vue";

export default {
  name: "create-product",
  components: {
    FormInput,
    FormTextArea,
    Typeahead,
    MultiSelect,
    AddVariantModal
  }
}
</script>
```

As usual, we start by importing `axios` for our API calls, as well as each of the new components we've discussed while defining the template of the component. We then export a default component object named `create-product`, specifying each of the imported components as children using the `components` object.

The data requirements of this component are fairly simple:

```
data() {
  return {
    product: {
      name: "",
      shortDescription: "",
      description: "",
      talkTime: "",
      standbyTime: "",
      screenSize: "",
      brand: "",
      os: "",
      features: [],
      variants: []
    },
    brands: [],
    os: [],
    features: [],
    colours: [],
    storage: [],
    variantsError: null
  };
}
```

We have a `product` object with empty properties for each of the fields we need to send off to our API to create a new product. Then, we have empty `brands`, `os`, `features`, `colours`, and `storage` arrays for storing the options we're passing down into our custom typeahead and multi-select components. Finally, we have the `variantsError` property, which as we already discussed will track the error state of the `product.variants` array.

Ultimately, there will be three methods that we need, the first of which is the `setData` method, which looks like this:

```
methods: {
  setData(data) {
    this.brands = data.brands;
    this.os = data.os;
    this.features = data.features;
    this.colours = data.colours;
    this.storage = data.storage;
  }
  ...
}
```

As with our other page level components, we use this method as a way of assigning the data returned from an API call in a `beforeRouteEnter` hook. Next, we have a `save` method, which we'll use to save the product data by calling an API endpoint:

```
methods: {
  ...
  save() {
    if (this.product.variants.length <= 0) {
      this.variantsError = "You must add at least one product variant.";
    } else {
      this.variantsError = null;
    }

    this.$validator.validateAll().then(result => {
      if (result && !this.variantsError) {
        axios
          .post("/api/products", this.product)
          .then(response => {
            this.$router.push("/admin/products");
          })
          .catch(error => {
            //handle server side validation
            console.log(error.data);
          });
      }
    });
  }
}
```

As we're not using Vee-Validate to validate the `product.variants` array, we start by ensuring there is at least one item in the array. If not, we set an appropriate error message against the `variantsError` property. Next, we validate the rest of our data properties using the `$validator.validateAll` function, wait for it to resolve, then check the `result` property.

If the `result` property is true and the `variantsError` property does not have a value, we can proceed to make our API call since the form is in a valid state. Otherwise, we do nothing, as our errors are already visible to the user so they know what they need to fix. To save the product to the database, we HTTP POST the `product` object to the `/api/products` endpoint. If the request is successful, we simply redirect back to the products list page. If it fails, all we do is log the result to the console for debugging purposes in order to keep things simple. Again, in a real application, this is where we'd handle server-side validation.

The last method we need on this component is the `addVariant` method:

```
methods: {
  ...
  addVariant(variant) {
    this.product.variants.push(variant);
  }
}
```

This is the method we invoke when our custom `<add-variant-modal />` component emits the `submit` event, and we use it to push the new `variant` object into the `product.variants` array.

To finish off the `script` section for this component, we need the following `beforeRouteEnter` hook:

```
beforeRouteEnter(to, from, next) {
  const vm = this;
  axios.get("/api/filters").then(response => {
    next(vm => vm.setData(response.data));
  });
}
```

Nothing to explain here as we've used a similar hook on most of our page level components. The only thing to note is that we're reusing the `/api/filters` API endpoint that we defined way back in Chapter 5, *Building a Product Catalog*—it already has all the data we need, and nothing else that we don't, so we might as well make the most of it.

To finish this component entirely, we need the following simple `style` section in order to make our product variants error match the default Bootstrap styling for error messages:

```
<style lang="scss" scoped>
.error {
  font-size: 80%;
  color: #dc3545;
}
</style>
```

Creating an add variant modal component

In the previous component, we rendered a custom `<add-variant-modal />` component, which we are still to define. Create a

`ClientApp/components/admin/AddVariantModal.vue` file and give it the following template section:

```
<template>
  <b-modal ref="modal" id="variantModal" title="Add a new variant" ok-
title="Add" @ok="submit">
    <form @submit.prevent="submit">
      <typeahead
        label="Colour"
        name="colour"
        :items="colours"
        v-validate="'required|min:3'"
        :error="errors.first('colour')"
        v-model="colour" />

      <typeahead
        label="Capacity"
        name="capacity"
        :items="storage"
        v-validate="'required|min:3'"
        :error="errors.first('capacity')"
        v-model="capacity" />

      <form-input
        type="number"
        label="Price"
        name="price"
        :error="errors.first('price')"
        prepend="£"
        v-model="price"
        v-validate="'required|decimal'" />
    </form>
  </b-modal>
</template>
```

Most of this will look very familiar by now, as we are simply rendering a group of our new custom input components inside a `<b-modal />` component. The only things to note are that for the first time we use the `prepend` prop of our `<form-input />` component, we assign a `ref` of `modal` to the `<b-modal />` component, and we listen for the `@ok` event of the `<b-modal />` component in order to invoke the `submit` method, which we'll look at shortly.

The script section for this component is also fairly simple. The main component definition looks like this:

```
<script>
import FormInput from "../shared/FormInput.vue";
import Typeahead from "../shared/Typeahead.vue";

export default {
  name: "add-variant-modal",
  components: {
    FormInput,
    Typeahead
  },
  props: {
    colours: {
      type: Array
    },
    storage: {
      type: Array
    }
  }
};
</script>
```

We need to make use of our custom form input and typeahead components, so we start by importing those and then export another default component object. This time, it is named `add-variant-modal`, and optionally has two props: a `colours` array and a `storage` array. These are the predefined options that we can pick from when adding a new product variant, but they aren't required as we can still add a new value for each of these if there aren't any to begin with.

In addition to the preceding components and props definitions, we need the following data function:

```
data() {
  return {
    colour: "",
    capacity: "",
    price: ""
  };
}
```

This should be fairly self-explanatory, as each product variant has individual `colour`, `capacity`, and `price` properties.

Finally, we need the following `submit` method:

```
methods: {
  submit() {
    this.$validator.validateAll().then(result => {
      if (result) {
        const payload = {
          colour: this.colour,
          storage: this.capacity,
          price: this.price
        };

        this.$emit("submit", payload);
        this.$refs.modal.hide();
        this.colour = "";
        this.capacity = "";
        this.price = "";
      }
    });
  }
}
```

As with our other form-based components, we start by validating the inputs using `$validator.validateAll`. If all is well, we construct a `payload` object, which represents the product variant we need to pass back up to the parent form so that it can be added to the list of product variants. In order to achieve that, we emit a `submit` event, passing the `payload` object as an argument. Once this is done, we can close the modal using the `ref` property we assigned earlier by calling the `$refs.modal.hide()` function. Finally, we can clean up by resetting the values of the `colour`, `capacity`, and `price` properties.

Vue component inheritance

Our `CreateProduct.vue` page is now complete, but the vast majority of its complexity has been abstracted away into a set of reusable custom input components. We are still to define these, but they include `FormInput.vue`, `FormTextArea.vue`, `MultiSelect.vue`, and `Typeahead.vue`. As a quick recap, the idea here is to reduce duplication. When rendering a single form field with Bootstrap styling, we tend to have multiple nested `div` elements with different classes, a label, an input of some kind, and a validation message. We also need some logic in order to work out which class to apply to the input to give immediate feedback as to whether the input is valid or not—this would also be duplicated, at least at the containing component level.

As an example, without our custom components, if we were to render a text input field, we'd need the following markup:

```
<b-form-group>
  <label>First name</label>
  <b-form-input v-model="firstName" data-vv-name="first name" v-
validate="required|min:3" :state="state('first name')" />
  <b-form-invalid-feedback>
    {{ errors.first('first name') }}
  </b-form-invalid-feedback>
</b-form-group>
```

The bold highlighted sections are really the only bits of that markup that change depending on the specific form field in question. The `label` text obviously needs to change, as does the name of the property in the context of anything to do with validation. Also, the input element itself can change. For example, we could swap out the `<b-form-input />` component for a `<b-form-select />` instead, or even one of our custom `<typeahead />` or `<multi-select />` components.

So, how do we go about creating a component to encapsulate this duplicated functionality, while still being flexible enough to handle different types of input, all while still playing nicely with Vee-Validate? The obvious option is to take a single custom component that takes a `type` prop, then render a collection of different inputs, each using a `v-if` directive to control visibility based on the type passed in. Something like this:

```
<template>
  <div class="form-group">
    <label>{{ label }}</label>
    <input v-if="type === 'text'" type="text" />
    <input v-if="type === 'number'" type="number" />
    <select v-if="type === 'select'">
      ...
    </select>
  </div>
</template>

<script>
export default {
  name: "input-example",
  props: {
    type: {
      type: String,
      required: true,
      default: 'text'
    }
  }
}
```

```
}  
</script>
```

However, the problem we'd face here is that the more input types we need to support, the bigger the template section will get. Aside from readability, if you are interested in SOLID principles, you'll also notice that it would violate the open/closed principle. We'd need to modify this component every time we hit an input type that we haven't yet catered for.

Instead, a better approach would be to use inheritance. Just like in C#, Vue components can extend, or inherit from, another component. Now, there are some gotchas that we need to be aware of, the biggest of which is that only the *behavior* of a component can be inherited—that is, the `script` section. The `template` section cannot be inherited with our current HTML only templates; we'd need to bring in some kind of rendering engine such as Jade or Pug, and use partials to achieve a similar goal. However, we still have a fairly clean option of declaring the base component as a child of the inheriting component, and using slots to render child-specific content within the parent template. This will make more sense if we look at an example.

Defining a form input base component

Bearing everything we've already discussed in mind, let's go ahead and define a base component for all of our form input elements. Create a

`ClientApp/components/shared/FormInputBase.vue` file with the following template section:

```
<template>  
  <div class="form-group">  
    <label v-if="label">{{ label }}</label>  
    <slot><!--input will eventually go here --></slot>  
  </div>  
</template>
```

So far, this is very simple. We render a standard `div` element with a Bootstrap-specific class of `form-group`, containing an optional `label` element, and a single `slot` where we can insert input-specific markup in any parent components. This way, if a consuming component does not pass in a `label` prop, the label simply does not get rendered.

The `script` section is a little more interesting:

```
<script>  
export default {  
  name: "form-input-base",  
  props: {
```

```
    label: {
      type: String
    },
    name: {
      type: String
    },
    value: {
      type: [String, Array]
    },
    error: {
      type: String
    }
  },
  computed: {
    classes() {
      return {
        "form-control": true,
        "is-valid": !!!this.error && this.value.length > 0,
        "is-invalid": !!this.error
      };
    }
  }
};
</script>
```

Remembering that the properties and methods that we define here can be inherited, we declare a selection of props that are not directly used within this component. In fact, the only one that is is the `label` prop. So, knowing that we'll be using this component as a base for our other input components, we declare the `label`, `name`, `value`, and `error` props, as well as a `classes` computed property.

Regardless of the type of input we are trying to render, we must give it a name so that it can be validated by Vee-Validate—hence the `name` prop. Also, validation will be done by the parent form component, which means any errors will be placed in the `errors` computed property of that component. Remember that, by default, Vee-Validate injects this property into every component we create, meaning the errors we'd see in this component are different to those of the parent. Therefore, we must rely on the parent to pass any error message that we need to display down within the `error` prop. This is absolutely fine, as in order to make it as reusable as possible, it probably shouldn't be responsible for its own validation anyway. The `value` prop should be self-explanatory: we'll be using it to bind the value of the input! When building custom input components, if we want to make use of the `v-model` and `v-validate` directives from consuming components, there are two rules to abide by. First, the child component must declare a `value` prop, and second, it should emit an `input` event when the underlying value actually changes. More on this later.

The `classes` computed property is how we determine whether the input is in an untouched, valid, or invalid state. We're returning an object so that we can do something like `:class="classes"` on one of our inheriting input controls. Every input in a Bootstrap-styled application has a class of `form-control`, so we always set that class by giving it a value of `true`. However, the `is-valid` and `is-invalid` classes may look a little strange to you if you are not familiar with JavaScript. By putting double exclamation marks in front of a statement, it coerces that statement into a Boolean. For example, if we write `!!this.error`, it treats the string value of the `error` property as a Boolean. As such, if we provide a value in the `error` prop, then this statement will return `true`. On the other hand, if we provide a value but change the statement to `!!!this.error`, then it will return `false`. Basically, if we don't provide an `error` prop, and the `value` prop has something in it, the input is valid. If we do provide an `error` prop, then the input is invalid.

Inheriting from a base component

Now that we have our base component in place, let's look at how to make use of it. We'll start with the most commonly used `FormInput.vue` component, which belongs in the `ClientApp/components/shared` folder. Start by giving it the following `template` section:

```
<template>
  <form-input-base :label="label">
    // inheriting component content will go here...
  </form-input-base>
</template>
```

This is our workaround for the fact that templates aren't inherited by subcomponents that extend a base component. Instead, we declare the base component as a child of the subcomponent, then render it as part of the template, as we've done here. This only works because we included a `slot` component as part of the base component's own `template` section, which we can now use to inject subcomponent-specific markup. In this case, this markup looks like this:

```
<div class="input-group">
  <div v-if="prepend" class="input-group-prepend">
    <span class="input-group-text">{{ prepend }}</span>
  </div>

  <input
    :class="classes"
    :type="type"
    :name="name"
    :value="value"
```



```
@focus="$emit('focus')"  
@blur="$emit('blur')"  
@input="$emit('input', $event.target.value)"  
@change="$emit('change', $event.target.value)" />  
  
<div v-if="append" class="input-group-append">  
  <span class="input-group-text">{{ append }}</span>  
</div>  
  
<div v-if="error" class="invalid-feedback">  
  {{ error }}  
</div>  
</div>
```

From our add product form component template, we know that this component needs to support Bootstrap input groups. As such, we start by rendering a `div` element with the `input-group` class, followed by an optional `div` with the `input-group-prepend` class. We only display this element if the consuming component passes in the `prepend` prop.

Next, we have the most important part, the actual `input` element itself. We use the `v-bind` directive to bind the `class`, `name`, and `value` properties based on the component's props that we're going to inherit from the base input component. We also bind the `type` property, but this time to a local `type` prop, which doesn't belong to the base because it is specific to this component. Remembering that if we want this component to behave like a standard input, we need to emit an `input` event when the value changes. This is exactly what we do here, but we take it a few steps further by directly emitting the `focus`, `blur`, and `change` events also. Aside from enabling us to listen for these events like we would with a standard input and triggering our own actions on them, we need to emit the `focus` and `blur` events so that Vee-Validate can provide real-time feedback as we progress through the form. It does this by automatically validating a field when it detects that the user has entered and subsequently moved away from it, by listening to the `focus` and `blur` events.

We follow the input with another optional `append` `div` element, which again is only displayed if we pass in the `append` prop. Finally, we render another `div` element to conditionally display an error message if we pass one in using the `error` prop. Most, if not all of our custom input components will have an error message like this, but unfortunately due to the way Bootstrap works, we cannot move this to the base component template. This is because it won't receive proper styling unless it is rendered in exactly the right location for Bootstraps, CSS selectors. In this case, it is rendered inside the input group element, and will not work properly if we were to move it outside instead. Unfortunately, this duplication is unavoidable, so we just have to live with it!



We're using standard HTML with Bootstrap classes rather than Bootstrap-Vue components here due to a bug in Bootstrap-Vue where the CSS styling breaks when using input groups!

The script section for this component looks like this:

```
<script>
import FormInputBase from "./FormInputBase.vue";

export default {
  name: "form-input",
  extends: FormInputBase,
  components: {
    FormInputBase
  },
  props: {
    type: {
      type: String,
      default: "text"
    },
    prepend: {
      type: String
    },
    append: {
      type: String
    }
  }
};
</script>
```

Seeing as we're extending the base form component, we will start by importing it. Next, we will export a default component object where we also declare the base form component as a child, so that we can render it in the preceding `template` section. All we need to do to actually inherit from the base component is use the `extends` property, passing in the imported base component object. Finally, we declare any subcomponent-specific props that don't already exist on the base component. In this case, these are the `type`, `prepend`, and `append` props.

In addition to the `FormInput.vue` component, we also need a `FormTextArea.vue` component, which lives in the same `ClientApp/components/shared` folder. This component is so similar to the last that we're not going to discuss it, and you could actually have a go at creating it yourself before reading any further.

Once you've done this, compare it with the following template section:

```
<template>
  <form-input-base :label="label">
    <textarea
      :class="classes"
      :name="name"
      :value="value"
      :rows="rows"
      @focus="$emit('focus')"
      @blur="$emit('blur')"
      @input="$emit('input', $event.target.value)"
      @change="$emit('change', $event.target.value)">
    </textarea>

    <div v-if="error" class="invalid-feedback">
      {{ error }}
    </div>
  </form-input-base>
</template>
```

And also compare it with this script section:

```
<script>
import FormInputBase from "./FormInputBase.vue";

export default {
  name: "form-text-area",
  extends: FormInputBase,
  components: {
    FormInputBase
  },
  props: {
    rows: {
      type: Number,
      default: 3
    }
  }
};
</script>
```

Building custom input controls

So far, our input components are fairly simple wrappers around standard HTML input elements in order to reduce duplication. However, we are yet to define the `Typeahead.vue` and `MultiSelect.vue` components, which need to go beyond the standard functionality of a HTML input element.

Now, there are plenty of third-party components that would do the job we want without us going to the extent of building them ourselves. However, it isn't always appropriate to use someone else's component if it doesn't quite look or behave the way we want. It's also quite easy and fun to build our own, and definitely something worth learning for times when you can't find a ready-rolled component to do the job you want.

Building a custom typeahead control

Create a `ClientApp/components/shared/Typeahead.vue` file and then start off the template section with the following:

```
<template>
  <form-input-base :label="label">
    <div class="typeahead">
      ...
    </div>
  </form-input-base>
</template>
```

As we did with our other input components, we're going to inherit from the base form component, so we'll start by rendering it here and placing our subcomponent content within it. Next, we need a standard HTML input where the user will begin typing their value:

```
<input
  ref="input"
  :value="value"
  :name="name"
  :class="classes"
  @blur="blur"
  @focus="focus"
  @keydown.enter.prevent="enter"
  @keydown.down.prevent="down"
  @keydown.up.prevent="up"
  @input="input" />
```

Again, as we did with the form input component, we bind the `value`, `name`, and `class` properties to the values of the associated props that we're inheriting from the base component. However, in addition to this, we're also giving this input a `ref` of `input`. We'll see why when we get to the `script` section shortly.

In this case, we do something slightly different with regards to event handling. For starters, we're listening for additional `keydown` events on the `enter`, `up`, and `down` keys. Also, rather than simply emitting the event directly so that it bubbles up to the consuming component, we invoke component methods instead. This is because we need more control in this component, as we'll need to perform some logic before we eventually emit these events like we did before.

Directly beneath the `input` element, we need the following:

```
<div v-if="error" class="invalid-feedback">
  {{ error }}
</div>
```

No explanation needed here as it's exactly the same as we've done in the previous form input components. Finally, we need some kind of UI element for displaying the list of suggestions that we'll show as the user starts typing in the input:

```
<b-list-group v-if="open" class="suggestions">
  <b-list-group-item
    v-for="(suggestion, index) in matches"
    :key="index"
    :active="index === current"
    @mousedown.prevent="click(index)"
    button>

    {{ suggestion }}
  </b-list-group-item>
</b-list-group>
```

The Bootstrap list group element is ideal for this scenario, so that's what we're using here. We conditionally render it based on some kind of `open` property on the component, and give it a class of `suggestions` so that we can specifically target it with some CSS styles later on.

There's no point in a list group without a nested collection of list group items, so we'll loop over an array of `matches` (another component property of some kind), rendering a `b-list-group-item` component for each one. As we've done before, we bind the `key` property to the unique array index of the item, and the `active` property to an inline expression that checks whether the array `index` is equal to a component property called `current`. More on this shortly, but for now just know that the `current` property is a numeric value used to track which suggested item the user has currently selected if navigating with the keyboard. We will also prevent the default `mousedown` behavior, choosing instead to invoke a `click` method, passing in the item's array `index` array as an argument. Finally, inside each list group item, we display the suggestion text so that the user knows what they are selecting from.

The `script` section for this component is pretty complex in comparison to what we've seen so far, so we're going to build it up in stages. Start with the following:

```
<script>
import FormInputBase from "./FormInputBase.vue";

export default {
  name: "typeahead",
  extends: FormInputBase,
  components: {
    FormInputBase
  },
  props: {
    items: {
      type: Array,
      required: true
    }
  }
  ...
};
</script>
```

None of this should need any explanation by now, as we're simply defining another custom input component that inherits from the base input component. As it's a typeahead control, we'll need to display suggestions based on what the user has typed, and those suggestions will be computed from a list of items passed in via a required `items` prop of type `Array`.

The data requirements of the component are fairly simple:

```
data() {
  return {
    current: -1,
    focused: false
  };
}
```

As we discussed previously, as we're supporting keyboard navigation in order to highlight a suggestion before selecting it, we need the `current` property to track the index of the currently highlighted suggestion. We also need a `focused` property to keep track of whether or not the user currently has their cursor inside the input—we'll see why when we look at the `computed` properties object:

```
computed: {
  matches() {
    return this.items.filter(str => {
      return str.toLowerCase().indexOf(this.value.toLowerCase()) >= 0;
    });
  },
  open() {
    return (
      this.value.length > 0 &&
      this.focused &&
      !this.items.some(item => item === this.value)
    );
  }
}
```

The `matches` property is a subset of the full `items` collection. We display them in the suggestions box so that the user can select one to automatically complete the rest of the input value. We only suggest the item if it contains the value the user has begun to type into the input box, making sure the search is case-insensitive by converting both sides of the comparison to lowercase.

The `open` property determines whether or not the suggestions box should be visible or not. The rule is quite simple: the user must have entered at least a single character into the input, the input must be focused, and none of the items in the `items` prop should have their text value equal to the current `value` of the input. As soon as they either navigate away from the input, make a selection from the suggestions box, or type a complete value that matches one of the predefined items, the suggestions box is hidden.

We need quite a few methods on this component, so we'll add them one at a time so that we can cover each one individually. The first is the `input` method:

```
methods: {
  input(event) {
    this.$emit("input", event.target.value);
    this.items.map(item => {
      if (item.toLowerCase() === event.target.value.toLowerCase()) {
        this.$emit("input", item);
      }
    });
  },
  ...
}
```

Rather than directly emit the `input` event inline as we've done before, this time we invoke this method every time the `input` element emits a native `input` event. After emitting the event so that it bubbles up to the consumer, we do another case-insensitive search of the `items` array to see whether what the user typed matches any of the items in the array. If it does, we emit another `input` event, this time passing the matched array item. This might seem slightly pointless unless you're a little OCD like I am, but if I have a brand named `Brand` and a user types `brand` into the input, I would rather have `Brand` be displayed in the input.

Next, we need the `click` method:

```
click(index) {
  this.$emit("input", this.matches[index]);
  this.$refs.input.$el.focus();
}
```

This method is invoked when a user clicks on one of the suggested items in the suggestions box. We receive the selected `index` of the `matches` array as an argument so that we can emit an `input` event with the selected value to notify the consumer that a selection has been made. We then need to force the component to focus back on the `input` element using `this.$refs.input.$el.focus()`, seeing as a click made outside of the input will have caused it to emit a `blur` event. This enables the user to immediately carry on typing should they wish to, or at the very least tab into the next input quickly and easily.

The next method we need is the `enter` method:

```
enter() {
  if (this.current >= 0) {
    this.$emit("input", this.matches[this.current]);
  }
}
```

This method is invoked when the user hits the *Enter* key on their keyboard. The idea is that once they begin typing and the suggestions box is displayed, they can use the arrow keys to highlight a suggestion, then hit *Enter* to actually make the selection. As such, in this method, we ultimately emit the `input` event and pass the currently selected suggestion as an argument. We do this by accessing a specific item of the `matches` array, using the `current` value as the array index to access. However, before we do any of this, we need to make sure the user has actually highlighted a suggestion by validating that the `current` property has a value greater than or equal to zero.

Next, we need the `up` and `down` methods to handle the arrow key presses:

```
up() {
  if (this.current >= 0) this.current--;
},
down() {
  if (this.current < this.matches.length - 1) this.current++;
}
```

The currently selected suggestion is highlighted based on the `current` property being compared with the index of the `matches` array when its items are rendered into the template section. As such, in order to change the currently selected (and highlighted) item, all we need to do is change the `current` property. This is exactly what the `up` and `down` methods do; they decrement the `current` property if its value is greater than or equal to zero, and increment it if its value will not exceed the number of items in the `matches` array.

Finally, we need the `focus` and `blur` methods:

```
focus() {
  this.focused = true;
  this.$emit("focus");
},
blur() {
  this.focused = false;
  this.$emit("blur");
}
```

Other than emitting the appropriate native events, the only other thing these methods do is keep the `focused` property in sync with whether or not the user is currently focused on this component.

To finish off this component, add the following `style` section:

```
<style lang="scss" scoped>
.typeahead {
  position: relative;
  .suggestions {
    position: absolute;
    top: 42px;
    right: 0;
    left: 0;
    z-index: 1;
    -webkit-box-shadow: 0px 2px 10px 0px rgba(204, 204, 204, 1);
    -moz-box-shadow: 0px 2px 10px 0px rgba(204, 204, 204, 1);
    box-shadow: 0px 2px 10px 0px rgba(204, 204, 204, 1);
  }
}
</style>
```

We want the suggestions box to be displayed over the top of any content, directly beneath the typeahead components, so we use relative and absolute positioning to achieve what we want. Again, this isn't a book on CSS, so I'm not going to explain any further than that. The only other thing we've done is give the suggestions box a slight drop shadow to make it stand out from any input elements that it may display.

With this, our typeahead component is complete. In summary, we pass in a list of items that will be used for the autocomplete functionality, and as the user begins typing, we filter those items based on the current value of user input and display the suggestions in a box beneath the input. To select a suggestion, the user can click on it with a mouse, touchpad, or touchscreen, or they can use the arrow keys on their keyboard before pressing the *Enter* key to select the highlighted item. If the value they are looking for does not exist, they can still enter whatever they like and we'll handle the creation of new items in the API when we get there.

Building a multi-select control

The typeahead control we've just built is used for most of the complex fields on the create product form, but for the features of that product, we've used a multi-select control instead. We won't be supporting the addition of new features on the fly, so the user will only be able to select from the preexisting list.

Create an empty `ClientApp/components/shared/MultiSelect.vue` file, and then start the template section off with the following:

```
<template>
  <form-input-base :label="label">
    <div class="multi-select">
      <input
        ref="input"
        :value="text"
        :name="name"
        :class="classes"
        @focus="focus"
        @blur="blur"
        @keydown.prevent="keydown" />
      ...
    </div>
  </form-input-base>
</template>
```

Again, this should look pretty familiar by now, but in summary, we are:

1. Rendering a `div` element with a class of `multi-select` inside the slot of a `<form-input-base />` component
2. Rendering a standard HTML input with a `ref` of `input`
3. Binding the `value` property to a computed `text` value
4. Binding the `name` and `class` properties based on the props we'll inherit from the base form component
5. Handling the `focus`, `blur`, and `keydown` events with component methods rather than default behavior

Next, as with all of our custom form components, we need an optional validation feedback `div` element:

```
<div v-if="error" class="invalid-feedback">
  {{ error }}
</div>
```

And just like with our typeahead component previously we need a Bootstrap list group to display the items the user can select from:

```
<b-list-group v-if="open">
  <b-list-group-item
    v-for="(item, index) in items"
    :key="index"
    @mousedown.prevent="check(item)">
```

```
<b-form-checkbox
  class="checkbox"
  :checked="isChecked(item)"
  disabled>

  {{ item }}
</b-form-checkbox>
</b-list-group-item>
</b-list-group>
```

This time, we loop over the `items` prop directly, as there is no filtering to be done. We also display a Bootstrap formatted checkbox instead of a plain text value so that we can show which items have already been selected. We do this by binding the `checked` property of the checkbox to an `isChecked` component method, passing in the item in question so that it can be calculated.

Notice how we're actually disabling each checkbox by passing in an empty `disabled` prop. As with the typeahead control, a `click` event in the items box causes the input element to emit a `blur` event. This then hides the items, as we only want them to appear when the user is focused on the control. We combat this by disabling the checkbox, then adding a `mousedown` event handler to the containing list group item element, preventing the default behavior and invoking our `check` component method. In this method, we can force the focus back onto the input element, preventing the items box from disappearing each time the user selects one.

The `script` section for this component is another reasonably complex one, so we'll build it up in stages again. Start with the following:

```
<script>
import FormInputBase from "../FormInputBase.vue";

export default {
  name: "multi-select",
  extends: FormInputBase,
  components: {
    FormInputBase
  },
  props: {
    items: {
      type: Array,
      required: true
    }
  }
};
</script>
```

In summary, we are:

1. Importing the `FormInputBase` component
2. Exporting a default component object
3. Naming it `multi-select`
4. Extending the `FormInputBase` component
5. Declaring the `FormInputBase` component as a child
6. Defining a required `items` prop of type `Array`

Next, we need to define the `data` requirements of the component, which in this case are incredibly simple:

```
data() {
  return {
    open: false
  };
}
```

The only data we need to track is the `open` property, which controls whether or not we display the items box. We also need a single computed property:

```
computed: {
  text() {
    return this.value.join(", ");
  }
}
```

Since the `value` prop we pass in to this component is an array of selected items, we cannot simply bind the `value` property of the input directly to it. Instead, we concatenate the selected values into a single text value by joining them with a comma delimiter. This is then the property that we bind the input value to.

We'll need a number of components methods, starting with the `isChecked` method:

```
methods: {
  isChecked(item) {
    return this.value.some(s => s === item);
  }
  ...
}
```

To determine whether an item is already checked or not, we can use the native JavaScript `Array.some` function on the `value` prop, comparing each item with the `item` argument we've passed in.

Next, we need the `focus` and `blur` methods:

```
focus() {
  this.open = true;
  this.$emit("focus");
},
blur() {
  this.open = false;
  this.$emit("blur");
}
```

These are fairly self-explanatory, but aside from emitting the native events to enable validation from the consuming parent component, we toggle the `open` property so that the items box is displayed while the user has focus in the component's `input` element.

We then need the `keydown` method:

```
keydown(event) {
  event.preventDefault();
}
```

No logic required here at all, as the only thing we need to do is prevent all key presses inside the `input`. This isn't a text box the user should be able to type in, as the value we display inside it is computed from the selections they make from the available items.

Finally, we need the `check` method:

```
check(item) {
  var current = Object.assign([], this.value);
  var index = current.indexOf(item);

  if (index > -1) {
    current.splice(index, 1);
  } else {
    current.push(item);
  }

  this.$emit("input", current);
  this.$refs.input.$el.focus();
}
```

This method is fired each time a user clicks on one of the list group item rows in the items box, regardless of whether the item has already been checked or not. Bearing that in mind, we need to be able to both push new selections to the `items` array, or remove a selection if it already exists. That being said, we also already know that we cannot directly mutate the `value` prop, or Vue will start throwing errors and/or warnings to the browser console. Instead, we create a method level `current` variable, using `Object.assign` to clone the `value` array on to it. We can then use this `current` variable to see whether the `item` argument already exists within it by using the `Array.indexOf` function.

If the item already exists, that is, the index of the item within the array is greater than `-1`, we remove it using the `Array.splice` function, passing the index of the array to start removing from and the number of positions to remove. If it doesn't already exist, we simply push it to the array. With the `current` array now up to date with the state of which items are currently checked, we can notify the parent component that the value has changed by emitting an `input` event, passing the newly updated value as an argument like we did previously. Finally, to prevent the items box disappearing due to a `click` event outside of the `input` element, we focus back on to it using the `ref` we added to the input in the template.

To finish off the component, we need a bit of styling to get things displayed the way we want. Add the following `style` section:

```
<style lang="scss" scoped>
.multi-select {
  position: relative;

  .list-group {
    position: absolute;
    top: 42px;
    right: 0;
    left: 0;
    z-index: 1;
    -webkit-box-shadow: 0px 2px 10px 0px rgba(204, 204, 204, 1);
    -moz-box-shadow: 0px 2px 10px 0px rgba(204, 204, 204, 1);
    box-shadow: 0px 2px 10px 0px rgba(204, 204, 204, 1);
    margin-bottom: 20px;

    .list-group-item {
      cursor: pointer;
      &:hover {
        background: darken(#fff, 5%);
      }
    }

    .custom-control {
      cursor: pointer;
    }
  }
}
```

```
    }  
  }  
}  
</style>
```

Persisting new products to the database

At this point, our client-side changes for the admin panel are near enough complete, and if you run the application now, you should be able to browse to the admin panel, view a list of orders and existing products, and fill in the create product form to see our custom input controls in action. However, if you try to save a product that passes client-side validation, nothing will happen as the API endpoint we told it to send to does not yet exist. We'll fix that now.

Creating a slug generator

In order to support SEO-friendly URLs in our product catalogue, we rely on identifying specific products by their URL `slug` property. For example, when accessing a page with a relative path of `/products/soylent-megafone`, the slug that we extract is `soylent-megafone`. This is what we pass back to our API endpoint to fetch the specific products details. So far, these properties have been hardcoded into our seed data method, so how do we *slugify* a product name so that it becomes URL-friendly?

We could look for a NuGet package to do the job for us, but it seems a little overkill to bring in a third-party dependency for something so minor. Instead, we'll create our own string extension helper method to convert a normal string into a URL-friendly slug. Create a new `Infrastructure/StringExtensions.cs` file, and then add the following static class definition:

```
public static class StringExtensions  
{  
    public static string GenerateSlug(this string phrase)  
    {  
        string str = phrase.RemoveDiacritics().ToLower();  
        // invalid chars  
        str = Regex.Replace(str, @"[^a-z0-9\s-]", "");  
        // convert multiple spaces into one space  
        str = Regex.Replace(str, @"\s+", " ").Trim();  
        // cut and trim  
        str = str.Substring(0, str.Length <= 45 ? str.Length : 45).Trim();  
        str = Regex.Replace(str, @"\s", "-"); // hyphens
```



```
        return str;
    }

    private static string RemoveDiacritics(this string text)
    {
        var s = new string(text.Normalize(NormalizationForm.FormD)
            .Where(c => CharUnicodeInfo.GetUnicodeCategory(c) !=
UnicodeCategory.NonSpacingMark)
            .ToArray());

        return s.Normalize(NormalizationForm.FormC);
    }
}
```

I must admit that I didn't write this code myself, but rather found it online at the personal blog of a developer named Adam Hathcock (<https://adamhathcock.blog/2017/05/04/generating-url-slugs-in-net-core/>).

The comments are pretty clear, but essentially what we're doing here is sanitizing the string to remove any unwanted characters, then replacing the spaces with hyphens to form a URL-friendly version.

Creating the API endpoint

We can now add an additional API endpoint to the `Features/Products/Controller.cs` file. The controller action definition looks like this:

```
[HttpPost, Authorize(Roles = "Admin")]
public async Task<IActionResult> Create([FromBody] CreateProductViewModel
model)
{
    ...
}
```

This endpoint will only accept HTTP POST requests from authenticated users who belong to the `Admin` role. Remembering that our custom typeahead control allows users to enter any arbitrary string value, we first need to check whether what they entered for the brand and operating system already exist in the database or not. We do this as follows:

```
var brand = await _db.Brands.FirstOrDefaultAsync(x => x.Name ==
model.Brand);

if (brand == null)
    brand = new Brand { Name = model.Brand };
```

```
var os = await _db.OS.FirstOrDefaultAsync(x => x.Name == model.OS);

if (os == null)
    os = new OS { Name = model.OS };
```

The typeahead control also automatically matches the case of an existing item, so when we do a string comparison here, we don't need to worry about the casing of either side. If either the `brand` or `os` variables are null, we create new entities, which we'll attach to a new `Product` instance next:

```
var product = new Product
{
    Name = model.Name,
    Slug = model.Name.GenerateSlug(),
    ShortDescription = model.ShortDescription,
    Description = model.Description,
    TalkTime = model.TalkTime,
    StandbyTime = model.StandbyTime,
    ScreenSize = model.ScreenSize,
    Brand = brand,
    OS = os,
    Thumbnail = "/assets/images/thumbnail.jpeg",
    Images = new List<Image>
    {
        new Image { Url = "/assets/images/gallery1.jpeg" },
        new Image { Url = "/assets/images/gallery2.jpeg" },
        new Image { Url = "/assets/images/gallery3.jpeg" },
        new Image { Url = "/assets/images/gallery4.jpeg" },
        new Image { Url = "/assets/images/gallery5.jpeg" },
        new Image { Url = "/assets/images/gallery6.jpeg" }
    }
};
```

Most of the properties of this object are simply copied directly from the view model we received as a parameter to the action method. However, notice that for the `Slug` property, we invoke our newly created `GenerateSlug` string extension on the `model.Name` property. This will take care of making the product name URL-friendly for us, by doing things such as replacing spaces and a few other special characters with dashes, then converting the final string to lowercase. We're also keeping this endpoint simple by hard-coding the product images to those that we've already stored in the `wwwroot/assets/images` folder.

Next, let's look at how to add the list of features the user selected from our custom multi-select control:

```
foreach (var feature in model.Features)
{
    var feat = await _db.Features.SingleAsync(x => x.Name == feature);
    product.ProductFeatures.Add(new ProductFeature { Feature = feat });
}
```

At this point, we know that any value we receive as part of the `model.Features` list must already exist in the database, so we use LINQ's `SingleAsync` method, which will throw an exception if it can't find a match. It is then a simple case of adding a new `ProductFeature` entity to the list, linking it back to the `Feature` entity we just queried the database for.

We need to do a similar process with the product variants, but this time there is a little more logic to perform:

```
foreach (var variant in model.Variants)
{
    var colour = await _db.Colours.FirstOrDefaultAsync(x => x.Name ==
variant.Colour);

    if (colour == null)
        colour = new Colour { Name = variant.Colour };

    var capacity = Convert.ToInt32(variant.Storage.Substring(0,
variant.Storage.IndexOf("GB")));
    var storage = await _db.Storage.FirstOrDefaultAsync(x => x.Capacity ==
capacity);

    if (storage == null)
        storage = new Storage { Capacity = capacity };

    product.ProductVariants.Add(new ProductVariant
    {
        Colour = colour,
        Storage = storage,
        Price = variant.Price
    });
}
```

As we did with the brand and operating system properties, we start by checking whether the `colour` and `storage` properties of each product variant already exist in the database or not. If they do, we link to the existing entities, and if they don't, we create new ones. It is then a simple case of pushing a new `ProductVariant` entity into the `product.ProductVariants` list.

Finally, we need to complete the action method with the following:

```
_db.Products.Add(product);

await _db.SaveChangesAsync();

return Ok();
```

We will add the newly created `product` object to the database, save our changes, then return a 200 OK HTTP response code.

The `CreateProductViewModel` class that we receive from the request body belongs in the same folder, and looks like this:

```
namespace ECommerce.Features.Products
{
    public class CreateProductViewModel
    {
        public string Name { get; set; }
        public string ShortDescription { get; set; }
        public string Description { get; set; }
        public decimal TalkTime { get; set; }
        public decimal StandbyTime { get; set; }
        public decimal ScreenSize { get; set; }
        public string Brand { get; set; }
        public string OS { get; set; }
        public List<string> Features { get; set; }
        public List<CreateProductVariantViewModel> Variants { get; set; }
    }
}
```

And the `CreateProductVariantViewModel` class that it depends on looks like this:

```
namespace ECommerce.Features.Products
{
    public class CreateProductVariantViewModel
    {
        public string Colour { get; set; }
        public string Storage { get; set; }
        public decimal Price { get; set; }
    }
}
```

Remote validation with Vee-Validate

At this point, we can now actually complete the create product form and submit it back to our API to persist a new product into the database. However, if you've remembered that we originally put a unique index on the `Slug` column of the `Products` table, you'll know that it would be very easy for the API call to fail if we send a duplicate product name. You may also have spotted that right at the top of the create product form, we rendered the following input element:

```
<form-input
  label="Name"
  name="name"
  :error="errors.first('name') "
  v-model="product.name"
  v-validate="'required|min:10|uniqueProductName' " />
```

Notice the `uniqueProductName` validation rule. This is obviously not a default rule built into Vee-Validate, as how can they possibly know whether our product name is unique or not? This is a rule that we need to instruct Vee-Validate on how to enforce, and seeing as we've not done this yet, you'll see errors popping up in your browser console.

To remedy this, we're going to extend the `Validator` object that Vee-Validate injects into our components. We could have done this as part of the create product form component, but it's not very SOLID to do so. Instead, create an empty `ClientApp/helpers/validation.js` file, and then add the following code into it:

```
import axios from "axios";
import { Validator } from "vee-validate";

const isUnique = value =>
  ...
});

Validator.extend("uniqueProductName", {
  validate: isUnique,
  getMessage: (field, params, data) => data.message
});
```

As we'll need to hit an API endpoint in order to determine whether a user-entered product name has already been taken or not, we start by importing `axios`. We also need to import the `Validator` object from Vee-Validate so that we can extend it to include our custom validation rule.

We then need to define a function that takes a single value parameter and determines whether that value is valid or not. In this case, we define an `isUnique` function, which we'll complete in just a minute. With this in place, we can invoke the `Validator.extend` function in order to add our own rule to be used on our components. The first argument this function expects is the string name of the rule, which will be used in the `v-validate` directive of an input control, as we did here:

```
v-validate="'required|min:10|uniqueProductName' "
```

The second argument is an object that needs to contain a `validate` function and a `getMessage` function. The `validate` function is used to validate the input, and in our case we simply assign the `isUnique` function we scaffolded out earlier. The `getMessage` function always receives three arguments: `field`, `params`, and `data`. The one we're interested in here is the `data` argument as it contains the returned result from the `validate` function. We're expecting this result to contain a `message` property, which will contain the appropriate error message if validation fails.

The full `isUnique` function needs to look like this:

```
const isUnique = value =>
  new Promise(resolve => {
    const payload = { name: value };

    axios.post("/api/products/validate", payload).then(response => {
      const isValid = response.data;
      if (isValid) {
        return resolve({
          valid: true
        });
      } else {
        return resolve({
          valid: false,
          data: {
            message: "Name is already in use."
          }
        });
      }
    });
  });
```

Vee-Validate validator functions are asynchronous, so we must return a promise from this function. We then start by constructing a `payload` object with a single `name` property to which we assign the `value` that we are validating. We can then use `axios.post` to hit the `/api/products/validate` endpoint, passing the `payload` object as its argument.

When the API call completes, we're expecting to receive a simple `true/false` response as to whether the payload was valid or not. If we receive `true`, the payload was valid and we resolve the promise with an object where its `valid` property is also `true`. If we receive `false` from the API, the payload was invalid and we resolve the promise with an object where its `valid` property is also `false`, but this time including a `data` object. This is the object that gets passed as the third argument to the `getMessage` function we discussed a moment ago. As such, we define a `message` property with the error string we want to be displayed in the UI if validation fails.

Making our app aware of the new custom validation rule

Our validation rule is now complete, but as yet our components cannot use it as our app isn't aware of the new file. To fix this, we simply need to import the new helper file in an appropriate location within the `ClientApp/boot.js` file:

```
import Vue from "vue";
import VueRouter from "vue-router";
import store from "./store";
import BootstrapVue from "bootstrap-vue";
import NProgress from "nprogress";
import VueToastr from "@deveodk/vue-toastr";
import "@deveodk/vue-toastr/dist/@deveodk/vue-toastr.css";
import VeeValidate from "vee-validate";

//plugins
import "./helpers/validation";

Vue.use(VueRouter);
Vue.use(BootstrapVue);
Vue.use(VueToastr, {
  defaultPosition: "toast-top-right"
});
Vue.use(VeeValidate);
```

Ordering matters here, so make sure that this new import statement is placed *before* we call `Vue.use(VeeValidate)`.

Creating the validation API endpoint

The last step is to actually create the API endpoint that's going to validate the product name for us. Open up the `Features/Product/Controller.cs` file and add the following new action method to the bottom:

```
[HttpPost("validate"), Authorize(Roles = "Admin")]
public async Task<IActionResult> Validate([FromBody]
ValidateProductViewModel model)
{
    var valid = await _db.Products.AllAsync(x => x.Name.ToLower() !=
model.Name.ToLower());
    return Ok(valid);
}
```

As with our other admin panel API endpoints, we only allow authenticated users who belong to the `Admin` role, and we listen on the `/api/products/validate` URL. The method body is very simple: we declare a `valid` Boolean variable and then check whether all of the products in the database have different names to the one we're validating. Finally, we return a 200 OK response with the `valid` Boolean value as the response body, just as we expect in our client-side validator function.

The `ValidateProductViewModel` class we expect to receive from the request body looks like this:

```
namespace ECommerce.Features.Products
{
    public class ValidateProductViewModel
    {
        public string Name { get; set; }
    }
}
```

Tidying things up

Fundamentally, our admin panel is now complete, but there are a few things we need to do in order to just clean things up a little and fix a few minor bugs that have crept in as we've changed things.

Linking to the admin panel

For starters, the only way of accessing the admin panel is to manually enter `/admin` as the relative URL path in your browser. Some sites prefer this as an approach, so it's not obvious that such an area of the app even exists, unless you know it's there. However, if this isn't the case, then we need to add a link to it from our main `nav` menu—but only displayed if the current user belongs to the `Admin` role.

Open up the `ClientApp/components/App.vue` file and make the following amendment to the `template` section:

```
<b-collapse is-nav id="nav_collapse">
  <b-navbar-nav>
    <b-nav-item to="/products">Products</b-nav-item>
    <b-nav-item v-if="isAdmin" to="/admin">Admin</b-nav-item>
  </b-navbar-nav>
  <b-navbar-nav class="ml-auto mr-4">
    <cart-summary v-if="isCustomer" />
    <auth-nav-item />
  </b-navbar-nav>
</b-collapse>
```

We'll also need an additional computed property to satisfy the `v-if` directive we just added:

```
computed: {
  showAuthModal() {
    return this.$store.state.showAuthModal;
  },
  isAdmin() {
    return this.$store.getters.isInRole("Admin");
  },
  isCustomer() {
    return (
      this.$store.getters.isInRole("Customer") ||
      !this.$store.getters.isAuthenticated
    );
  }
}
```

Fixing a logout bug

Now that we have nested route definitions, if we log out while viewing one of those nested pages, for example, the create product page, we won't be redirected back to the app home page. This is because our current `logout` method looks like this:

```
logout() {
  this.$store.dispatch("logout").then(() => {
    if (this.$route.meta.requiresAuth) {
      this.$router.push("/");
    }
  });
}
```

We are only checking whether the current route has the `requiresAuth` meta property set to `true`, rather than any route in its parent-child hierarchy. To resolve this, open up the `ClientApp/components/app/AuthNavItem.vue` file and make the following modifications:

```
logout() {
  this.$store.dispatch("logout").then(() => {
    if (this.$route.matched.some(route => route.meta.requiresAuth)) {
      this.$router.push("/");
    }
  });
}
```

Now, the check will traverse up the route definition tree to see whether the current route or any of its ancestors require authentication.

Fixing a bug by selecting a product variant

This one is slightly more involved than the rest. Before we added the ability to create new products, we were assuming that every combination of color and capacity would have a corresponding variant. For example, if we have a phone with two color options and two storage options, we would have four product variants in total. However, when building out the UI to add these on the fly, it became apparent that this isn't the case, as we may not want to sell every combination. In fact, the way our UI works means we only add the specific combinations we want.

For example, suppose we add three product variants to a new product, each of which has a different value for color and storage than the others. We'd have three distinct color values to choose from, and three distinct storage values to choose from. With our current logic, we'd expect nine different product variants, covering every combination of those values. However, we only actually have three variants because that's all we added in the UI. When a customer navigates to a product details page, they will have the ability to specify a combination of color and capacity that may not actually exist!

The fix this, we have a number of changes to make. Starting with the server-side changes, we need to modify the `Get` action method in the `Features/Products/Controller.cs` file:

```
var product = await _db.Products.Select(x => new ProductDetailsViewModel
{
    Id = x.Id,
    Slug = x.Slug,
    Name = x.Name,
    ShortDescription = x.ShortDescription,
    Description = x.Description,
    Price = x.ProductVariants.OrderBy(v => v.Price).First().Price,
    Thumbnail = x.Thumbnail,
    Images = x.Images.Select(i => i.Url),
    Features = x.ProductFeatures.Select(f => f.Feature.Name),
    //Colours = ...,
    //Storage = ...,
    Variants = x.ProductVariants
        .OrderBy(v => v.Colour.Name)
        .ThenBy(v => v.Storage.Capacity)
        .Select(v => new ProductVariantViewModel
        {
            ProductId = x.Id,
            Name = x.Name,
            Thumbnail = x.Thumbnail,
            ColourId = v.ColourId,
            Colour = v.Colour.Name,
            StorageId = v.StorageId,
            Capacity = v.Storage.Capacity,
            Price = v.Price
        })
    })
.FirstOrDefaultAsync(x => x.Slug == slug);
```

We need to remove the `Colours` and `Storage` properties, as we'll compute the valid values for these on the client side instead. We then added some `OrderBy` statements to make sure the product variants list is in the right order by the time it hits the client.

As we've removed the `Colours` and `Storage` properties from the controller, they are also no longer needed in the corresponding `ProductDetailsViewModel` either:

```
namespace ECommerce.Features.Products
{
    public class ProductDetailsViewModel
    {
        public int Id { get; set; }
        public string Slug { get; set; }
        public string Name { get; set; }
        public string ShortDescription { get; set; }
        public string Thumbnail { get; set; }
        public IEnumerable<string> Images { get; set; }
        public IEnumerable<string> Features { get; set; }
        public decimal Price { get; set; }
        public string Description { get; set; }
        //public IEnumerable<SelectListItem> Colours { get; set; }
        //public IEnumerable<SelectListItem> Storage { get; set; }
        public IEnumerable<ProductVariantViewModel> Variants { get; set; }
    }
}
```

Finally, we have a number of changes to make in the `ClientApp/components/product/Details.vue` component. Start by making the following changes to the template section:

```
<h5>Variants</h5>
<b-form-group label="Colour">
  <b-form-select :options="colours" v-model="colour" />
</b-form-group>
<b-form-group label="Capacity">
  <b-form-select :options="storage" v-model="capacity" />
</b-form-group>
```

Rather than binding the `options` props of the two select boxes to the lists we used to have on the model returned from the API, we instead bind to a pair of data properties. To do this, we'll be making use of some of the utility functions in the `lodash` library, which we've already installed. In the `script` section, we need to make sure we import it for use:

```
import _ from "lodash";
import Gallery from "../Gallery.vue";
```

We also need to modify the `created` life cycle hook so that it invokes some methods that we'll be defining shortly:

```
created() {
  this.computeColours();
  this.computeStorage();
  this.computeProductVariant();
}
```

The `computeColours` method will extract the distinct `colour` values from the array of product variants, and then set the currently selected `colour` value to the first item in the computed array. The `computeStorage` method will do a similar job for the storage options, basing them on product variants where the color matches the selected `colour` value that we just set. The `computeProductVariant` method will work out which variant to track based on the currently selected `colour` and `storage` values.

We're going to be moving a number of our properties that are currently computed so that they become plain old data properties instead. Make the following changes to the `data` function:

```
data() {
  return {
    open: false,
    index: 0,
    colours: [],
    colour: null,
    storage: [],
    capacity: null,
    variant: null
  };
}
```

We no longer have any `computed` properties, so the `computed` object can be removed entirely.

We're also going to add a pair of `watch` functions to keep the select boxes in sync with the available variant options, and update the `variant` `data` property to the correct variant based on the current state of those select boxes:

```
watch: {
  colour: {
    handler(value) {
      this.computeStorage();
      this.computeProductVariant();
    }
  },
}
```

```
capacity: {
  handler(value) {
    this.computeProductVariant();
  }
}
```

We will start by watching the `colour` property, and we'll invoke the `computeStorage` and `computeProductVariant` methods every time its value changes. Next, we'll watch the `capacity` value, but this time the only thing we need to do is invoke the `computeProductVariant` method when it changes.

The new `computeColours` method looks like this:

```
computeColours() {
  this.colours = _.uniqBy(
    this.product.variants.map(v => {
      return {
        value: v.colourId,
        text: v.colour
      };
    }),
    "value"
  );
  this.colour = this.colours[0].value;
}
```

We use the `uniqBy` function from `lodash` to ensure we only get distinct items in the array based on a property named `value`. This property comes from the `Array.map` function, which we use on the `product.variants` array, where we return a new array of objects that contain `value` and `text` properties. These are what we need to pass into the select box for it to be able to bind the options properly.

The `computeStorage` method looks like this:

```
computeStorage() {
  this.storage = this.product.variants
    .filter(v => {
      return v.colourId === this.colour;
    })
    .map(v => {
      return {
        value: v.storageId,
        text: v.capacity
      };
    });
}
```

```
    this.capacity = this.storage[0].value;
  }
```

This method is slightly simpler, as we filter the product variants array for only those variants whose color matches the color we've selected in the first select box. We then take that filtered array of variants and run the same `Array.map` function as we did for `colours`, returning a new array of objects containing `value` and `text` properties that can be bound to a select box.

Finally, the `computeProductVariant` method looks like this:

```
computeProductVariant() {
  this.variant = this.product.variants.find(
    v => v.colourId == this.colour && v.storageId == this.capacity
  );
}
```

This is taken directly from the old `variant` computed property, so it should not need any explanation.

Our changes are now complete, so if you run the application again, it should work exactly as it did before, but it will now handle product variants properly, regardless of the combinations we defined when creating the product. This is all very well and good, but why is our implementation so much more complicated now?

The problem is that now that we have the capacity select box items entirely dependent on the selected value of the color select box, the storage items can change. With our previous method of using computed properties for everything, when the color select box changed its value, both the `storage` and `variant` computed properties tried to update themselves simultaneously. This meant there was a very brief split second where the `variant` computed, may not actually find a matching variant, unless we force it to wait until the UI finish refreshing before doing its calculation. This is what we've done by using watchers and methods instead, and we no longer get any console errors while we wait for everything to sync up.

Summary

We've covered a lot again in this chapter, some of which was considerably more complicated than previous chapters. However, we are now entirely feature complete!

We started by adding role-based authorization to both our server-side and client-side code. This included adding more advanced route guards to our client-side route definitions, then hiding a number of UI elements if the user wasn't in an appropriate role to view them.

We then started to build out the UI components that were necessary for our admin panel, which for the first time required us to configure nested route definitions. We refactored the order list component to be usable both in the customer's my account page and in our new admin panel. We then built a product list component and created a product form component. While doing so, we discussed the amount of code duplication we currently had, and decided to build a number of custom input components designed to make our lives easier and reduce that duplication.

To implement our custom input components, we needed to look at Vue component inheritance so that we could abstract common properties and template sections into a base form input component. We then inherited from this component by building custom input, text area, typeahead, and multi-select components.

We finished off the Create Product page by adding an API endpoint for persisting the new products into the database, then added remote validation to ensure only unique product names were sent to the API for processing. Finally, we spent some time tidying up by making a few minor enhancements and bug fixes for issues that cropped up after the changes we've made in this chapter. We also fundamentally changed the way we handle product variants, based on the way we add them in the first place and the fact we don't always sell every combination of colour/capacity for a given phone.

This chapter has completed the base feature set of our application. In the next few chapters, we'll be looking at deploying the application into a cloud environment, followed by some advanced topics to build on what we've already learned and built.

10 Deployment

With the application now feature-complete, we are ready to deploy to a production cloud environment. When deciding which environment to use, we have plenty to choose from, including Microsoft Azure, Google Cloud Platform, and Amazon AWS, to name but a few of the most common options. We'll be making use of Microsoft Azure, since when building ASP.NET applications it makes a lot of sense to stick with Microsoft when choosing a hosting platform. We can also get an application such as proof-of-concept apps up and running quickly and easily, at a low cost.

In this chapter, we will cover the following topics:

- Registering for an Azure portal account
- Setting up an Azure database/app service
- Configuring Azure environment variables and connection strings
- Preparing an application for deployment
- Setting up simple Git deployments in Azure
- Enabling logging in Azure
- Forcing HTTPS connections

Registering for an Azure account

The first thing we'll need to do is head over to <https://portal.azure.com> and register for an account. Most people already have a Microsoft account for one reason or another, but if you don't, you can sign up for a new one.

Once you've either signed in or created a new Microsoft account and signed in, you'll be presented with the Azure portal signup screen:

Microsoft Azure stuartatcliffe88@hotmail.com [Sign out](#)

Azure free account sign up

Start with a £150 credit for 30 days, and keep going for free

1 Identity verification by phone ^

Country code

Phone number

We delivered a code to your phone.

Verification code

2 Identity verification by card v

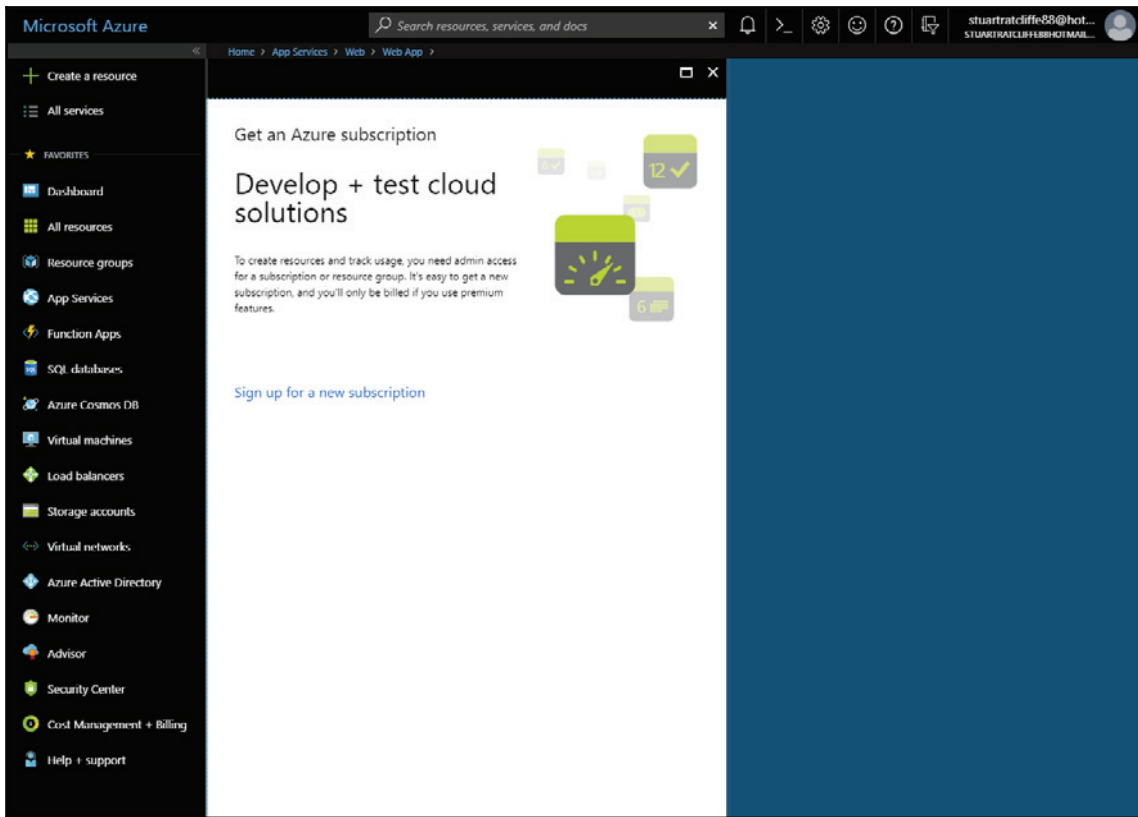
3 Agreement v

English [Privacy & Cookies](#) [Trademarks](#) [Legal](#) [Support](#) [Give us feedback](#) © 2018 Microsoft

Every Azure portal user must go through phone and credit card verification before they're allowed access, but don't worry too much as the only thing that will cost you money is the SQL Server database. This is around £4 GBP a month at the time of writing, but for the first month, everything is free, as you get £150 in credits for signing up.

The registration process is very simple, so it doesn't need any further explanation. Once you've finished registering, you should be presented with a screen that looks something

like this:



At this point, we are ready to start setting up our production environment.

Setting up an Azure environment

Before we start creating any databases or web app services, we need to create a **subscription**, followed by a **resource group**. But what exactly are these things and why do we need them?

Understanding Azure subscriptions and resources

The first thing we need to create, and as such understand, is a subscription. Every Azure user must have at least one subscription, as every resource we create must be associated with a single parent subscription. This is all very well and good, but what actually is a resource within the context of Azure? The answer is essentially any manageable item or service that we can provision. Examples include databases, web apps, virtual machines, and storage accounts.

So, we now know that subscriptions are containers for Azure resources, but they also have two other main purposes: they allow us to control user access to the groups of resources that they own, and they allow us to pay for those resources via a single monthly invoice due to the fact that we get billed for the entire subscription. This is good for any application, but even more so for more complex apps that may utilize the standard web app service and database resources, but also require other resources, such as file storage and push notifications. Rather than having to pay individually for those resources, you pay one monthly bill for everything.

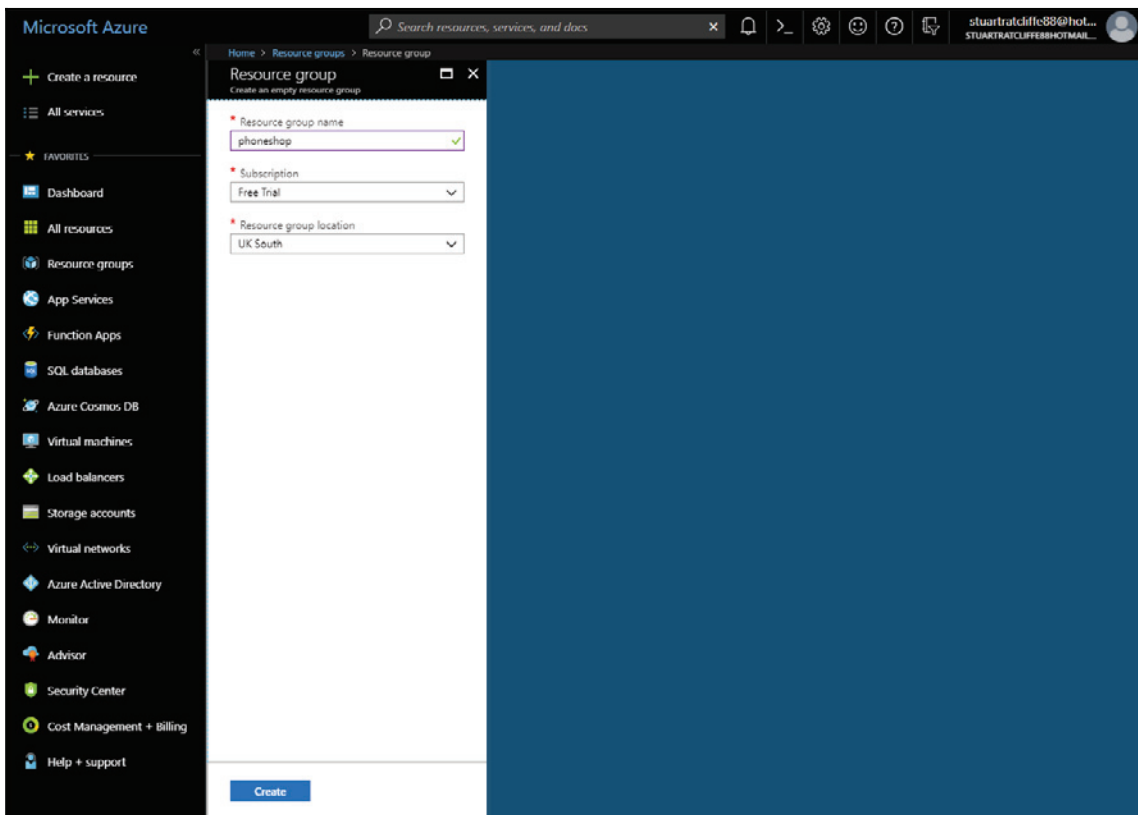
The next thing we need to create is a resource group, which as the name suggests is a logical group of resources. Why do we need a resource group if we already group resources under a subscription, I hear you ask. Again, there are two main purposes that resource groups aim to fulfill. Firstly, you may not want all of your subscription users to have full access to every resource they contain. Instead, you can utilize a second level of user access management at the resource group level rather than the subscription level. Secondly, they allow us to perform actions, for example resource deletion, on multiple resources at a time by specifying a resource group instead of a single resource.

There are many different ways that you can utilize subscriptions and resource groups depending on the size and structure of your organization. Hosting companies are now branching away from offering physical (or even virtualized) servers, instead choosing to offer managed cloud hosting using services such as Azure. They will often go as far as consulting on the best ways of architecting applications within Azure, including partitioning using vLANs on top of resource groups. This might be something to consider if you are a large organization with serious complexity involved with the hosting of your applications, or you simply don't have time to manage your environments yourself. However, for the sake of our sample application, we'll be keeping things simple with a single subscription and resource group to store everything else that we'll need.

Creating a subscription and resource group

Creating a subscription is incredibly easy, as there is a link to do so on the very first screen you see after creating a portal account (see the previous screenshot). Even better is that this first subscription is entirely free for the first month, as long as you don't go over £150 worth of free credits. At the end of that month, you have to opt in to a pay as-you-go subscription, meaning you won't get a nasty bill after the first 30 days unless you've made an explicit choice to continue your service—this isn't one of those free trials that automatically bill you after the first month if you forget to cancel!

Once you've followed the instructions to create your first subscription, we need to make a resource group to hold our database and web app resources. To do so, click on the **Resource groups** link in the main menu on the left, and then click the **Add** button at the top of the screen. You'll be presented with a simple form on the screen that looks like this:

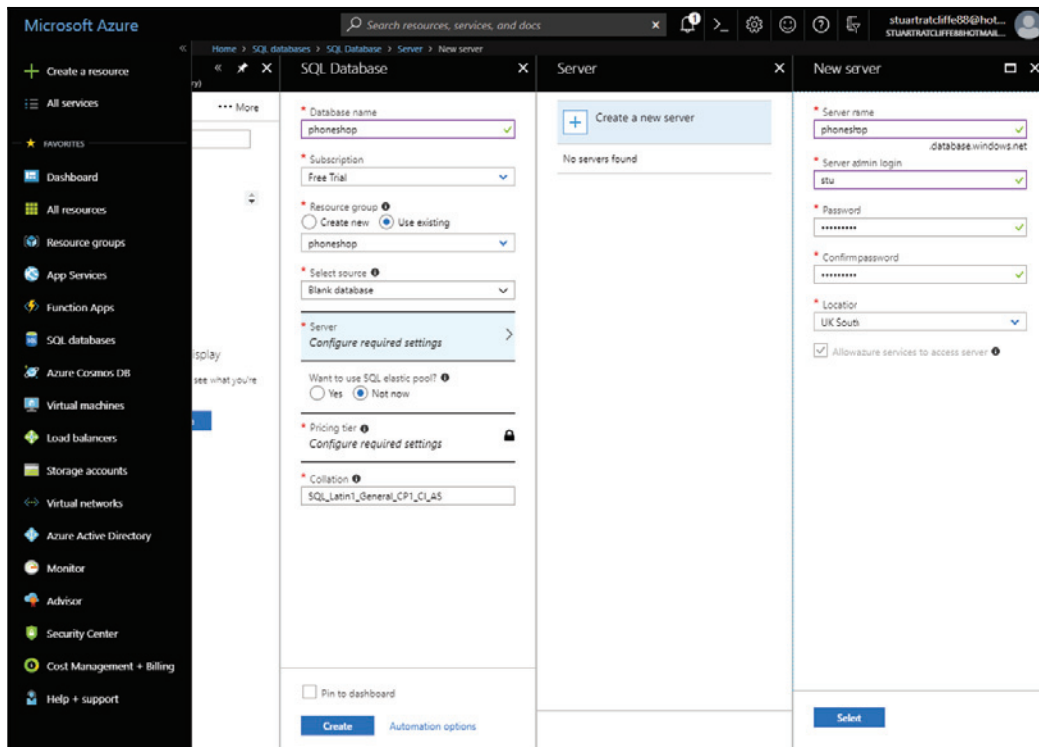


What you call it and which location you choose for it is entirely up to you. I named mine `phoneshop`, and used the **UK South** data center as the location, seeing as I am based in the UK and we're building an online phone shop. With this done, we are ready to start creating our actual resources.

Creating a database

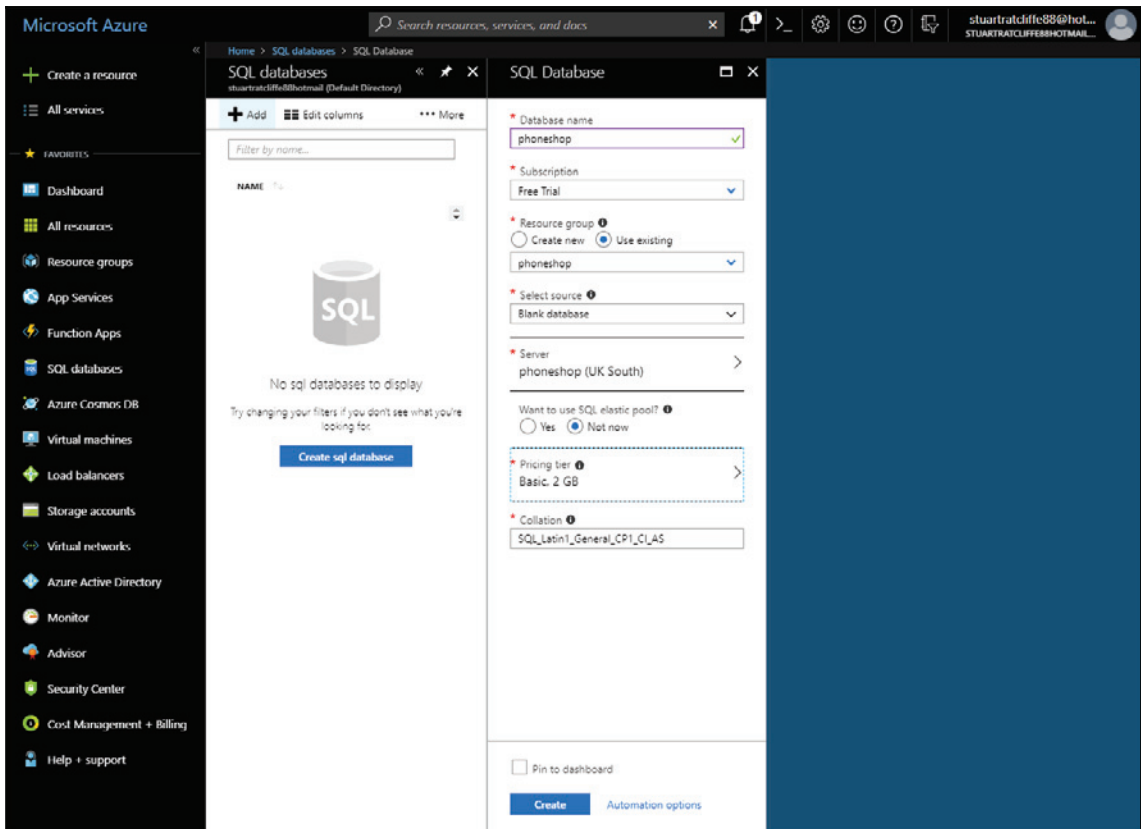
It makes sense to start off with the database, seeing as we'll need the connection string later, when it comes to setting up the web app. There's a couple of parts to this, as in addition to the database itself, we also need to configure the server that will host it. Both are done at the same time, as when creating a new database you can either choose to create a new server or pick an existing one. As this is our first database, it will be the former on this occasion.

Again, in the main menu on the left, click on the **SQL databases** link followed by the **Add** link at the top of the page that follows. Most of the fields are fairly self-explanatory, and by the time you get to the **Server** section, it should look something like this:



I chose to name the database `phoneshop`, left the default subscription selection as my **Free Trial**, and selected the **phoneshop** resource group we just created by checking the **Use existing** radio button. We want the database to be completely empty, so we choose the **Blank database** option for its source. Then, we come to the **Server** section, where our only option is to choose to create a new server seeing as no other servers were found. This opens a secondary form where I named the server `phoneshop`, entered my preferred server admin login credentials, and again chose **UK South** as the data center location.

Having completed the database server form and hitting the **Select** button at the bottom, we can finish off the main form for creating the database:



We don't wish to use SQL elastic pool, so I've left this as the default **Not now** option. Finally, I've specified **Basic, 2 GB** as the pricing tier for this database. It's worth noting at this point that we don't pay for the database server itself; instead, we must pay for each individual database that's hosted on it. *Additional databases mean additional cost.* Once you've finished this form, hit the **Create** button to finalize this. This could take a couple of minutes to complete, so just wait until it's done before we move on to the web app itself.

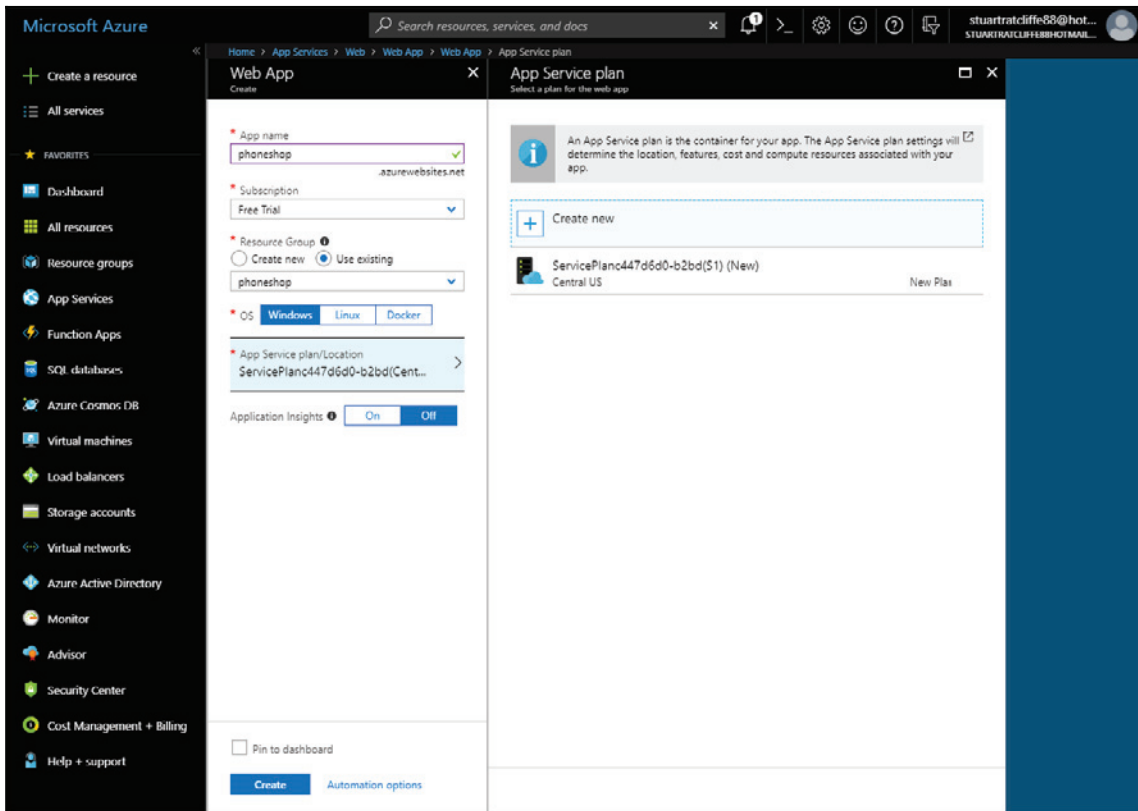
Creating an app service

An **app service** is a self-contained environment for deploying a single web or mobile app within Azure. They support a number of different application frameworks such as ASP.NET, Java, Node.js, and PHP all without the need for us to configure any kind of server infrastructure ourselves. All we need to do is spin one up, then deploy our application to it, and Azure will handle the rest for us. However, if you have more advanced needs, then we have a few different ways of overriding the default configuration, including the use of environment variables and the `web.config` file.

As with creating the database there are actually two parts to the creation of our first app service. This is because app services must always belong to another resource called an **app service plan**, which again we can choose to create at the same time as creating the app service itself. However, unlike with databases and database servers, it's the app service plan that is charged for and, as such, determines the pricing tier of our web app service. Speaking of which, there are a number of different pricing tiers available for app service plans, starting with the *Free* tier, which is what we'll be using for our sample application. The monthly cost of each tier above the free one is calculated by how much you actually use. This isn't an easy cost to estimate, although there are pricing calculators on the Azure website that aim to help you. My recommendation would be to make use of the £150 free credit for your first 30 days and see how much of that gets used up by one of your existing applications.

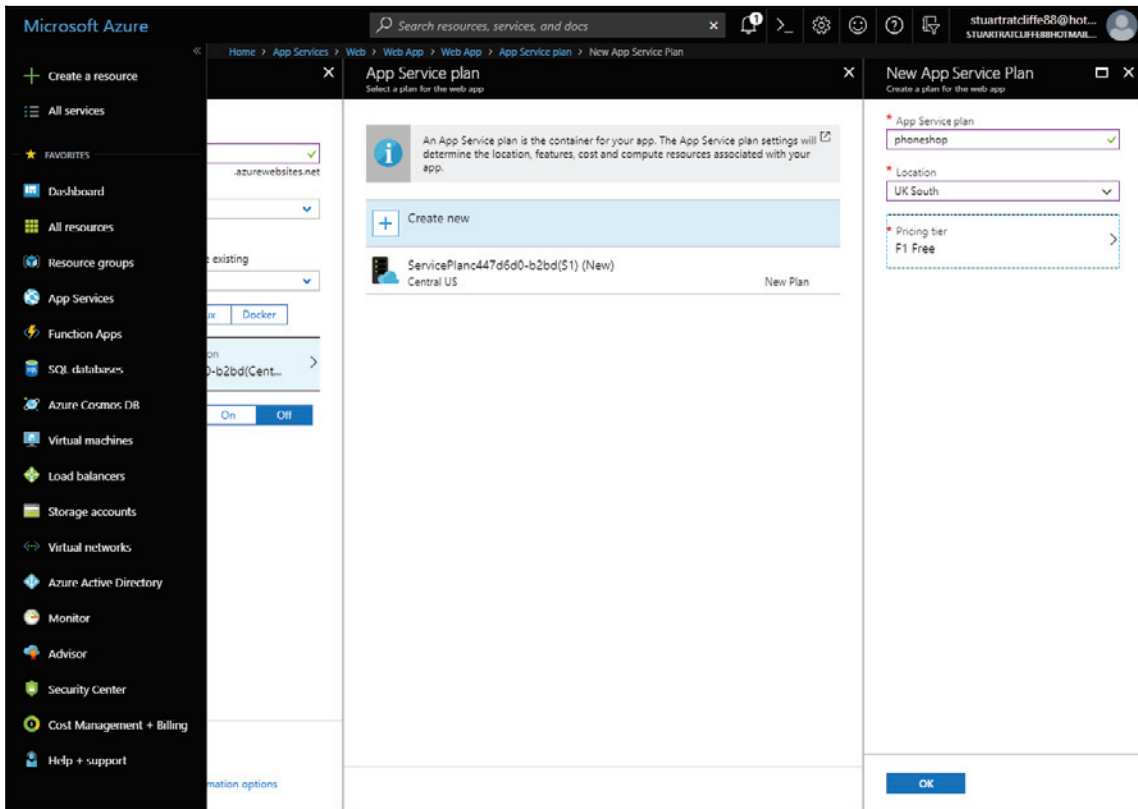
Aside from an increase in price, you also gain additional functionality and available resource limits as you go up the pricing tiers. For example, in the free tier we only get 1 GB of disk space, cannot use a custom domain name, and have no option for autoscaling. However, moving up to the *Standard* tier (the recommended minimum for production apps), we get 50 GB of disk space, custom domains are supported, and we can even set up autoscaling. This means that if we have huge spikes in activity, the application is far less likely to fall over, as Azure will automatically scale up the amount of available resources to deal with the additional requests. On top of this, on a standard tier service plan, we can actually host up to 10 app services, that is, 10 different web applications for the price of one. To really get the best value from Azure App Services, you need to stack up multiple applications under a single service plan.

To create our app service/plan, click on the **App Services** link in the main menu on the left, then click the **Add** button at the top of the screen that follows. You'll be presented with a whole host of different applications you can create, but we just need the first option: **Web App**. On the screen that follows, hit the **Create** button at the bottom, and you'll be presented with the following form to complete:

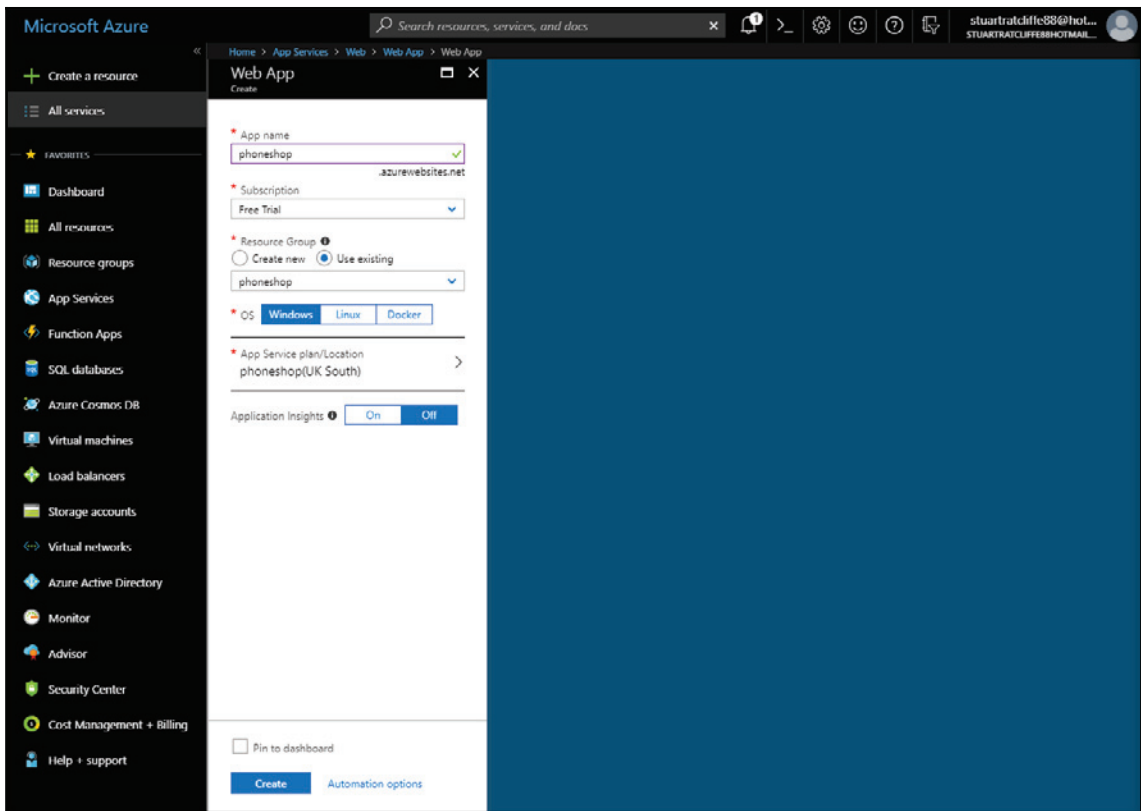


As with our other resources, I've chosen to name the App Service `phoneshop`, left the subscription as the default **Free Trial** option, and chosen to use my existing **phoneshop** resource group. I've left the OS as the default **Windows** option; note how the **App Service plan/Location** field has already been filled in for us. By default, Azure will choose to create a new App Service plan for you with a generic name using the **S1** pricing tier that's located in the **Central US** data center. It is very easy to skip over this field, seeing as it has been pre-completed, and then end up with a higher pricing tier than you'd like, and a database/web app running on different continents. I've certainly done this before, and then spent a good few hours trying to work out why there was around a 500 ms latency on every database call within my application.

Rather than take the default option, we're going to hit the **Create new** button to specify our own service plan details:



I've chosen to name it **phoneshop** again, but selected the **UK South** data center and the **F1 Free** pricing tier. Once you've configured your own service plan, hit the **OK** button at the bottom, which should then leave you with just the **Application Insights** field to finalize our app service:

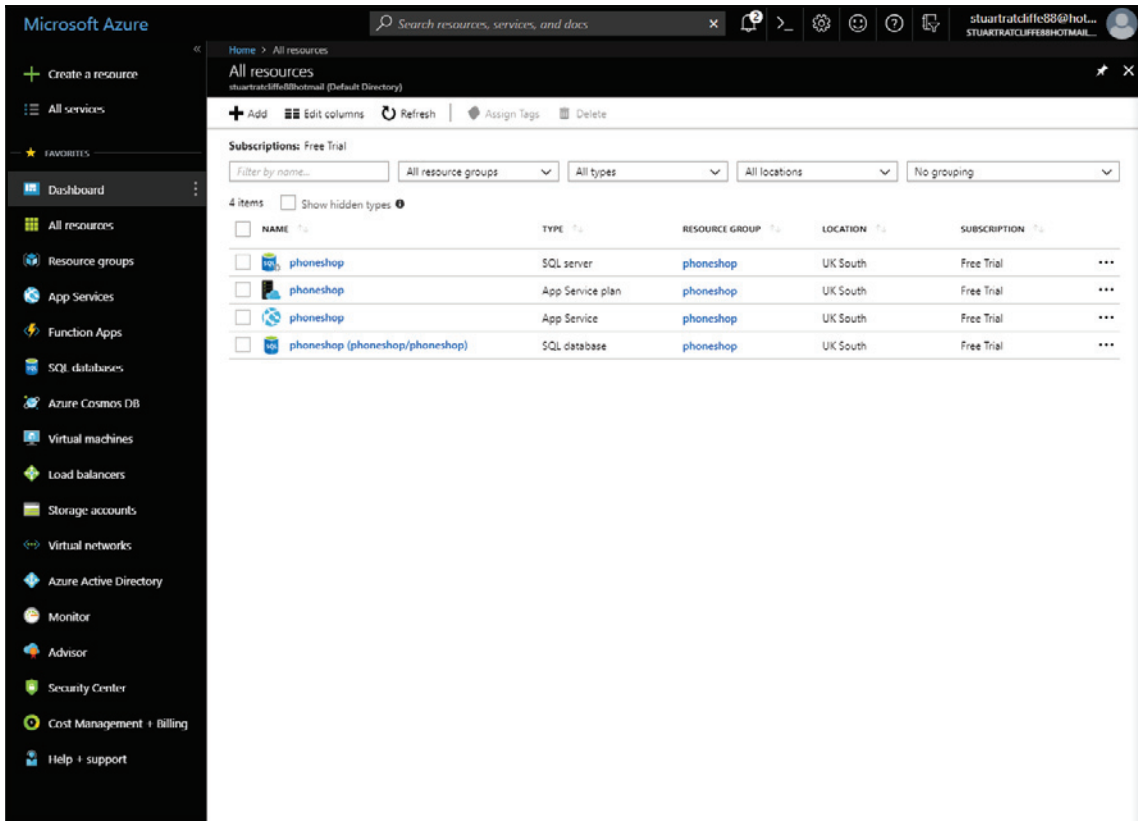


I'm not going to bother using application insights on my application, but feel free to turn it on if you wish. At this point, we can hit the **Create** button at the bottom to finalize the creation of our app service. As with the database, this can take a couple of minutes to complete, so just wait for the notification bar to let you know it's finished.



The notification bar is accessible by clicking the bell icon at the top of all Azure portal screens!

Once the app service finishes deploying, we will have created everything we need to host our e-commerce application in the cloud. To see all the resources we have created and to make sure everything is up and running, click on the **All resources** link in the main menu. You should be greeted with a screen that looks like this:

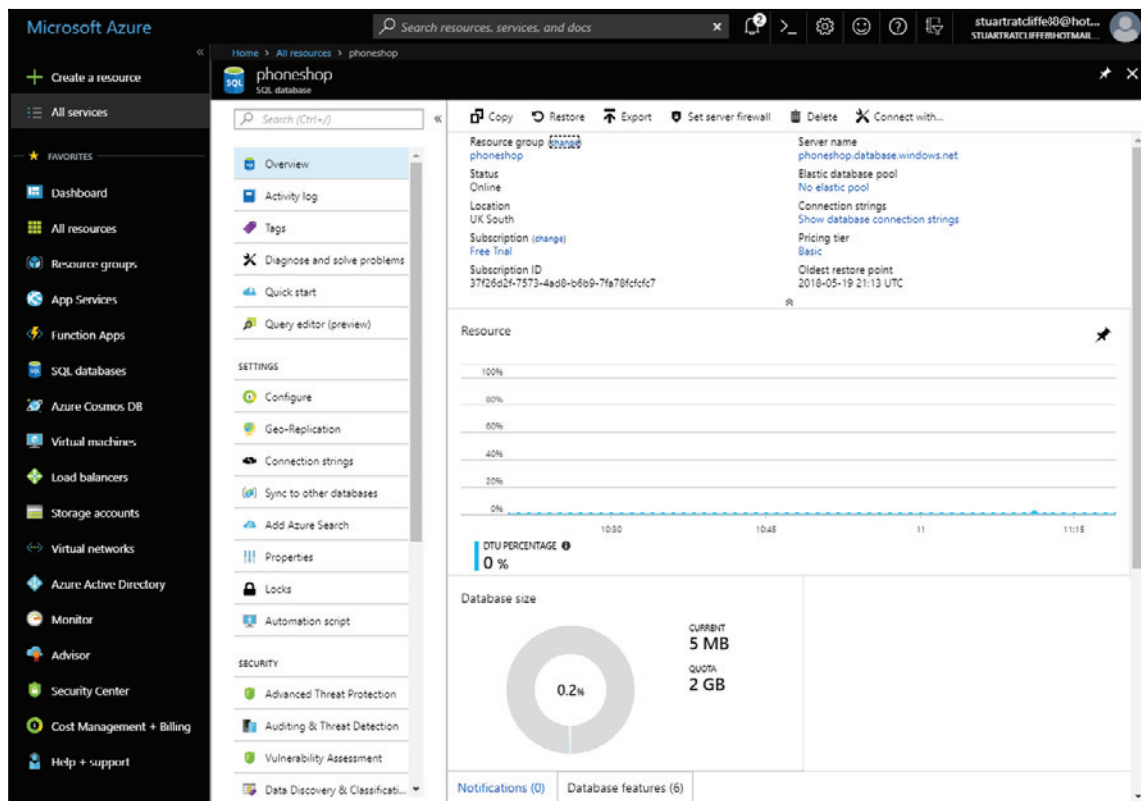


Double-check you have the same four resource types deployed: **SQL server**, **App Service plan**, **App Service**, and **SQL database**. Also ensure that they are all located in the same data center location of your choice.

Configuring environment variables

At the very least, most web applications will need the database connection string to be set as an environment variable. If you've used ASP.NET Core before, you'll know that it is incredibly easy to include environment-specific configuration settings using the `appsettings.json` and `appsettings.{EnvironmentName}.json` files. However, I wouldn't recommend storing sensitive variables such as connection strings and third-party API credentials in these configuration files. Instead, we can add them as environment variables directly within Azure, keeping them well away from source control and the eyes of developers who perhaps shouldn't have access to these settings.

First, we need to find out what our Azure connection string actually is. To do so is very easy, and from the **all resources** page that you should still be viewing, click on the SQL database name to go to the resource details page for our Azure database. This should look something like the following:



The screenshot displays the Microsoft Azure portal interface for an Azure SQL database. The left-hand navigation pane shows various services, with 'SQL databases' selected. The main content area shows the details for the 'phoneshop' SQL database. The 'Overview' tab is active, displaying the following information:

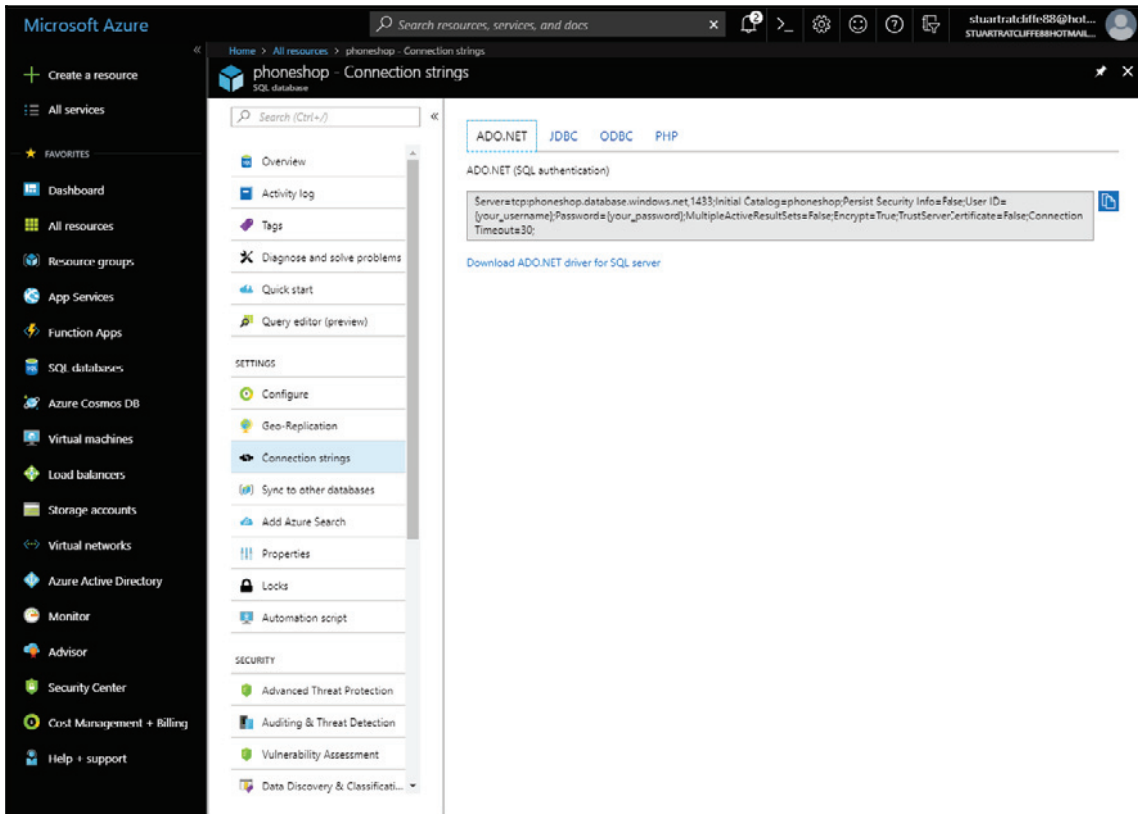
- Resource group: phoneshop
- Status: Online
- Location: UK South
- Subscription: Free Trial
- Subscription ID: 37f26d24-7573-4ad0-b6b9-7fa78f6fc7
- Server name: phoneshop.database.windows.net
- Elastic database pool: No elastic pool
- Connection strings: Show database connection strings
- Pricing tier: Basic
- Oldest restore point: 2018-05-19 21:13 UTC

Below the overview, there are two charts:

- DTU Percentage:** A line chart showing DTU usage over time, with a current value of 0%.
- Database size:** A donut chart showing the current database size is 5 MB, which is 0.2% of the 2 GB quota.

The bottom of the page shows 'Notifications (0)' and 'Database features (6)'.

You'll notice that a secondary menu has now appeared on the left, with a whole host of different items to explore if you are interested. This is where you'll find any kind of configuration that you might want to do, such as geo-replication and syncing with other databases. The one we are interested in here is the **Connection strings** option, but you can also click on the **Show database connection strings** link toward the top of the main window. Either of these links should take you to the following screen:



There are a number of tabs showing connection strings in different formats that are dependent on the application framework in use, but the one we want is the default **ADO.NET** option. Click the **Copy** button to the right of the box to copy this connection string to your clipboard.

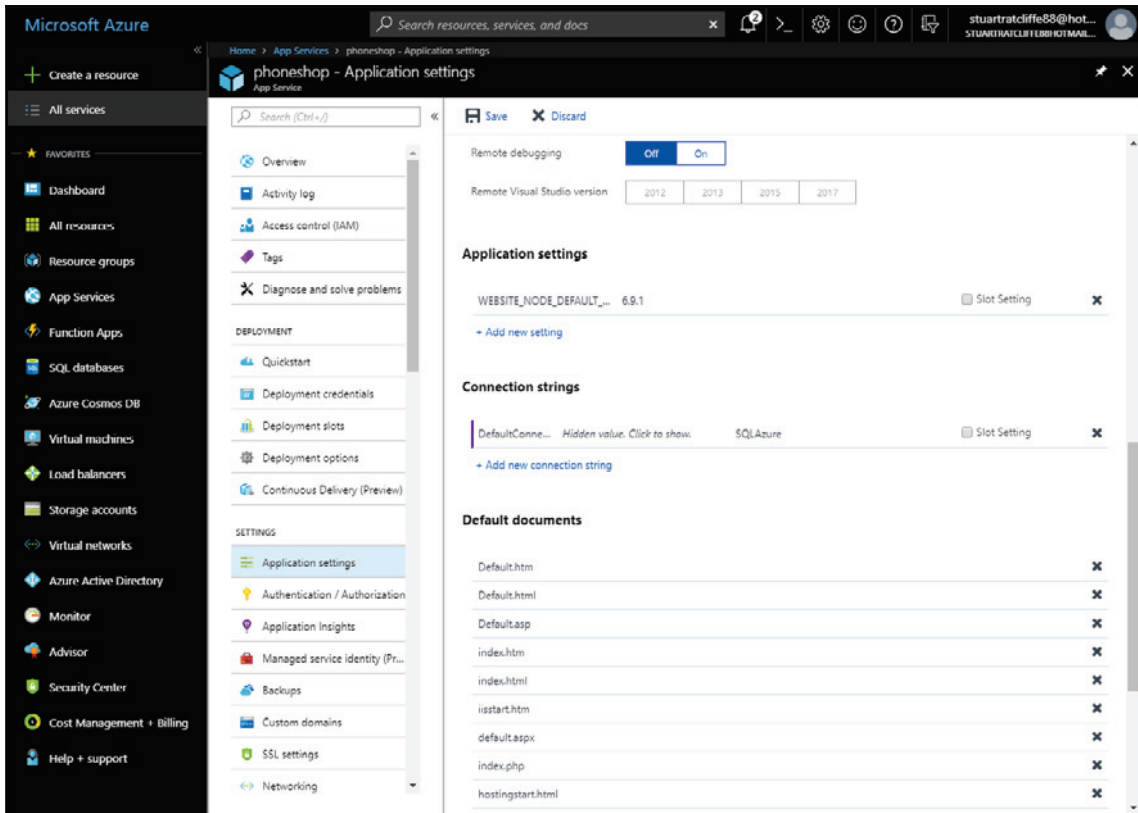
We now need somewhere to add this as an environment variable on our app service. Click on the **App Services** link in the main menu, then on the name of the app service we created earlier. You'll be presented with a similar screen to the one we just saw for the database:

The screenshot displays the Microsoft Azure portal interface for an App Service named 'phoneshop'. The left-hand navigation pane includes sections for 'Create a resource', 'All services', 'FAVORITES', and various resource categories like 'Dashboard', 'All resources', 'Resource groups', 'App Services', 'Function Apps', 'SQL databases', 'Azure Cosmos DB', 'Virtual machines', 'Load balancers', 'Storage accounts', 'Virtual networks', 'Azure Active Directory', 'Monitor', 'Advisor', 'Security Center', 'Cost Management + Billing', and 'Help + support'. The main content area is titled 'phoneshop App Service' and features a search bar and a list of actions: 'Browse', 'Stop', 'Swap', 'Restart', 'Delete', 'Get publish profile', and 'Reset publish profile'. A link to the 'Quickstart guide for deploying code to your app' is provided. Below this, a table lists key properties:

Resource group (change)	phoneshop	URL	https://phoneshop.azurewebsites.net
Status	Running	App Service plan/pricing tier	phoneshop (Free: 0 Small)
Location	UK South	FTP/deployment username	No FTP/deployment user set
Subscription (change)	Free Trial	FTP hostname	ftp://waws-prod-in1-007.fb.azurewebsites.windows.net
Subscription ID	37f26d24-7573-4ad8-b6b9-7fb78fcdcf7	FTPS hostname	ftps://waws-prod-in1-007.tp.azurewebsites.windows.net

The 'DEPLOYMENT' section includes 'Quickstart', 'Deployment credentials', 'Deployment slots', 'Deployment options', and 'Continuous Delivery (Preview)'. The 'SETTINGS' section includes 'Application settings', 'Authentication / Authorization', 'Application Insights', 'Managed service identity (Pr...', 'Backups', 'Custom domains', 'SSL settings', and 'Networking'. Three diagnostic cards are visible: 'Diagnose and solve problems', 'Application Insights', and 'App Service Advisor'. At the bottom, there are three monitoring graphs: 'Http 5xx' (showing 0 errors), 'Data In' (showing 0 B), and 'Data Out' (showing 0 B). The x-axis for all graphs represents time from 10:30 to 11:15.

Again, there is a whole host of options in the secondary menu for configuring the app service. We don't have time to discuss many of them, but most are pretty self-explanatory if you want to have a play around. What we need is the **Application settings** option, which will take you to a screen like this:



Note that, to find the **Application settings** and **Connection strings** sections, you need to scroll down quite a way. Here, I've already added a connection string by clicking on the **Add new connection string** link. If you're using the same setting names, as I've done in the `appsettings.json` file, you need to ensure you call this connection string **DefaultConnection**. Before you paste in the value, you'll also need to replace the `{your_username}` and `{your_password}` sections, using the access credentials you configured when setting up the SQL server earlier.

So, how does this work if we've already got a connection string defined in the `appsettings.json` file? If you open up the `Program.cs` file, you'll see a section at the bottom that looks like this:

```
public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .Build();
```

When we call the `WebHost.CreateDefaultBuilder` method, a number of things happen. First, the `appsettings.json` file is loaded in order to provide the application with our default set of application settings. Next, the `appsettings.{EnvironmentName}.json` file is loaded based on the specific environment we're running under, which loads the environment-specific settings of our application. If any of these settings already exist, that is, they also appear in the root `appsettings.json` file, the environment-specific settings will always take precedence. Finally, there is a call to `config.AddEnvironmentVariables()`, which does a similar job to adding an environment-specific config file. Any environment variables configured on the server will be added to the application's configuration, and environment variables whose name matches a key within the settings files will again take precedence over them. To put it simply, environment variables will always be used over their counterparts in `appsettings.json` files.

At this point, most simple applications can be deployed, and would probably work just fine. However, in our case, we have a few additional environment variables to configure:

The screenshot shows the Azure portal interface for configuring an application. The left sidebar contains navigation options like 'All services', 'Dashboard', and 'App Services'. The main area displays the 'Application settings' for a web app. Key settings include:

- Remote debugging:** A toggle switch set to 'On'.
- Remote Visual Studio version:** A dropdown menu set to '2017'.
- Application settings:** A list of key-value pairs:
 - `WEBSITE_NODE_DEFAULT_VERSION`: 8.9.3 (Slot Setting)
 - `ASPNETCORE_ENVIRONMENT`: Production (Slot Setting)
- Connection strings:** A section with a 'DefaultConnection' string set to 'SQLAzure' (Slot Setting).
- Default documents:** A list of files: Default.htm, Default.html, Default.asp, index.htm, index.html, iisstart.htm, and default.aspx.

By default, Azure app services are set to use Node version 6.9.1. However, for our application to work, you need to raise this version to 8.9.3, just like I've done, by changing the value of the `WEBSITE_NODE_DEFAULT_VERSION` application setting. I've also added a second setting named `ASPNETCORE_ENVIRONMENT` with a value of `Production`. I'm not going to explain this one as I'm assuming most of you are already familiar with ASP.NET Core environments.



Back in Chapter 7, *User Registration and Authentication*, we talked about overriding the `JwtKey` app setting in production. The **Application settings** section is where you can do so if you wish.

Make sure you hit the **Save** button at the top of the page. Our app service and all its required environment variables have been fully configured. However, before we can actually deploy the application, we have some preparation work to do.

Preparing the application for deployment

We're currently using PostgreSQL as our local development database, and although you can set up a Postgres database in Azure, it is far more expensive than running with SQL Server instead. This is why we chose to create a SQL Server and associated database when setting up the environment earlier. However, we now need to configure our application to work with multiple database providers, depending on the environment we are running in. We're also going to tweak the post-publish build steps that came preconfigured with the project template we started from. There are a couple of potential issues that can crop up when using Git to deploy to Azure, so we'll aim to combat them in advance to avoid headaches later on.

Configuring multiple database providers

Working with two different providers based on the environment we're running in might seem like a strange way of doing things. However, it is actually more common than you might think. Many development teams like to run their apps locally with open source database providers such as Postgres or SQLite, but make use of Microsoft SQL Server in their production environments.

We have a couple of changes to make so that we can add in support for SQL Server, starting with how we register the `DbContext` class with the built-in DI container. Open up the `Startup.cs` file, and make the following modifications right at the top:

```
public Startup(IConfiguration configuration, IHostingEnvironment env)
{
    Configuration = configuration;
    _env = env;
}

public IConfiguration Configuration { get; }

private IHostingEnvironment _env { get; }

...
```

Next, at the very top of the `ConfigureServices` method, make the following changes:

```
public void ConfigureServices(IServiceCollection services)
{
    if (_env.EnvironmentName == EnvironmentName.Development)
    {
        services.AddDbContext<EcommerceContext>(options =>
            options.UseNpgsql(Configuration.GetConnectionString(
                "DefaultConnection")));
    }
    else
    {
        services.AddDbContext<EcommerceContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString(
                "DefaultConnection")));
    }
    ...
}
```

Finally, we need to manually tweak the migration files to instruct them on how to set up our table identity columns in SQL Server. As an example, the creation script for the `AspNetUsers` table in the `Initial` migration file currently looks like this:

```
migrationBuilder.CreateTable(
    name: "AspNetRoles",
    columns: table => new
    {
        Id = table.Column<int>(nullable: false)
            .Annotation("Npgsql:ValueGenerationStrategy",
                NpgsqlValueGenerationStrategy.SerialColumn),
        ConcurrencyStamp = table.Column<string>(nullable: true),
        Name = table.Column<string>(maxLength: 256, nullable: true),
        NormalizedName = table.Column<string>(maxLength: 256,
            nullable: true)
    }
    ...
```

Note the `.Annotation("Npgsql:...")` line—in addition to this, we need to chain on a similar statement that specifies the SQL Server-specific evaluation strategy for identity columns. The updated version looks like this:

```
migrationBuilder.CreateTable(
    name: "AspNetRoles",
    columns: table => new
    {
        Id = table.Column<int>(nullable: false)
            .Annotation("Npgsql:ValueGenerationStrategy",
```

```
NpgsqlValueGenerationStrategy.SerialColumn)
    .Annotation("SqlServer:ValueGenerationStrategy",
        SqlServerValueGenerationStrategy.IdentityColumn),
    ConcurrencyStamp = table.Column<string>(nullable: true),
    Name = table.Column<string>(maxLength: 256, nullable: true),
    NormalizedName = table.Column<string>(maxLength: 256,
        nullable: true)
}
...
```

We won't cover everything here, but you'll need to make the same change in every migration file that has a table creation step like the preceding one. This currently includes the `Initial`, `ProductEntity`, `Catalogue`, and `Orders` migration files.

Tweaking the post-publish build steps

If you're entirely new to modern frontend development frameworks, you may be wondering why we keep referring to "building" our frontend code. These days, client-side applications are composed of many different JavaScript files, or more specifically modules. These modules are loosely connected using `import` and `export` statements, as we've seen while building out our sample e-commerce application.

While in development, we use middleware in our ASP.NET application to dynamically build the JavaScript for us, and update those built files as and when we make changes to them. However, in production, we have no need for the bundled JavaScript files to be updated until we do another deployment. As such, as part of the deployment, we want all of our client-side JavaScript to be built into static files, which will be served from the `wwwroot` directory just like any other.

We've also been using certain pieces of syntax that are not yet understood by all web browsers. We can solve this issue by transpiling our modern JavaScript code down into an older version that browsers can understand, similarly to how we compile C# into a form that underlying servers can understand. This process of bundling files together and transpiling them into a common format is what we mean when we refer to the *building* of a Vue.js client application.

As part of the .NET project template we used to scaffold the application back in [Chapter 3, Getting Started with the Project](#), we already have a post-publish task set up that builds our client-side JavaScript bundles and places them within the `wwwroot` folder. This task is at the bottom of the `ECommerce.csproj` file, and looks like this:

```
<Target Name="PublishRunWebpack" AfterTargets="ComputeFilesToPublish">
  <Exec Command="npm install"/>
```

```

<Exec Command="node node_modules/webpack/bin/webpack.js
--config webpack.config.vendor.js --env.prod"/>
<Exec Command="node node_modules/webpack/bin/webpack.js --env.prod"/>
<ItemGroup>
  <DistFiles Include="wwwroot\dist\**"/>
  <ResolvedFileToPublish Include="@ (DistFiles->'%(FullPath) ')"
  Exclude="@ (ResolvedFileToPublish) ">
    <RelativePath>% (DistFiles.Identity)</RelativePath>
    <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
  </ResolvedFileToPublish>
</ItemGroup>
</Target>

```

However, when it comes to running this script inside Azure, it is common to see an npm error thrown to do with our SCSS compilation. The crux of this error is that, for whatever reason, npm believes that our Node and/or npm versions have changed since we ran the npm install command. This simply isn't the case, and there are plenty of other people in similar positions who are posting GitHub issues all over the internet. Thankfully, it is a simple one-line fix:

```

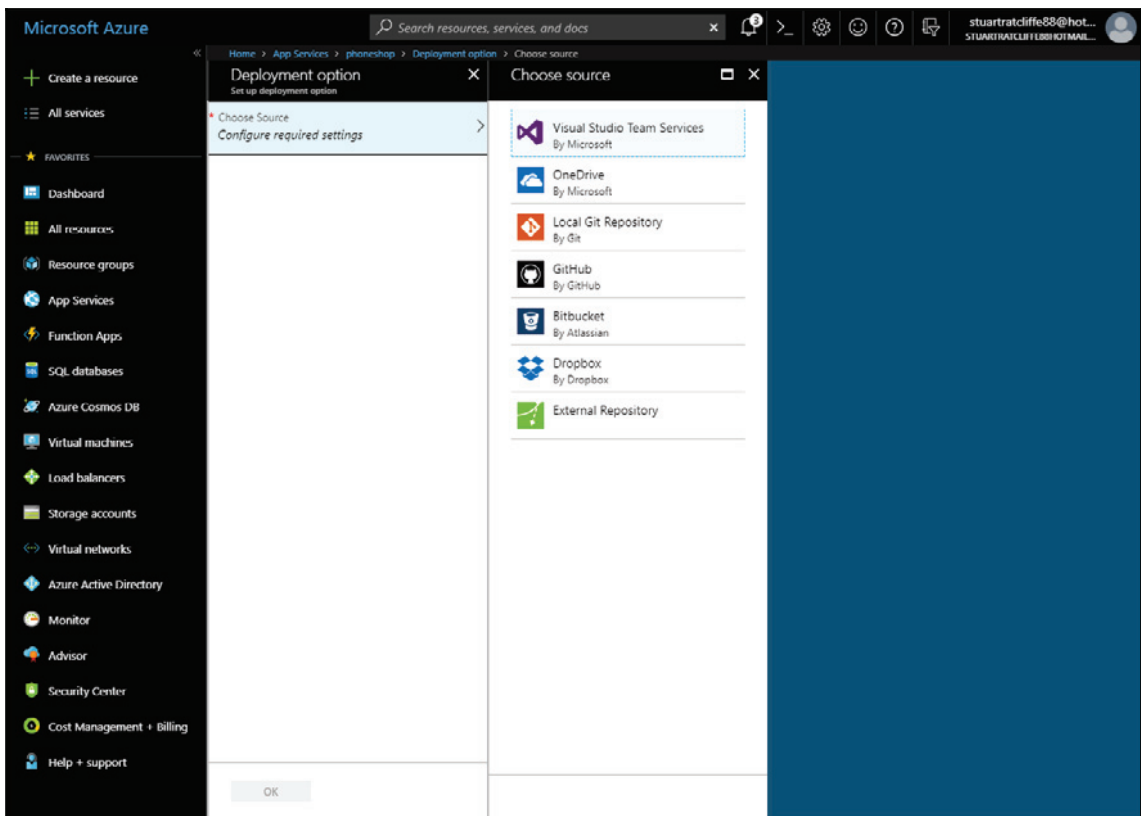
<Target Name="PublishRunWebpack" AfterTargets="ComputeFilesToPublish">
  <Exec Command="npm install"/>
  <Exec Command="npm rebuild node-sass"/>
  <Exec Command="node node_modules/webpack/bin/webpack.js --config
webpack.config.vendor.js --env.prod"/>
  <Exec Command="node node_modules/webpack/bin/webpack.js --env.prod"/>
  <ItemGroup>
    <DistFiles Include="wwwroot\dist\**"/>
    <ResolvedFileToPublish Include="@ (DistFiles->'%(FullPath) ')"
    Exclude="@ (ResolvedFileToPublish) ">
      <RelativePath>% (DistFiles.Identity)</RelativePath>
      <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
    </ResolvedFileToPublish>
  </ItemGroup>
</Target>

```

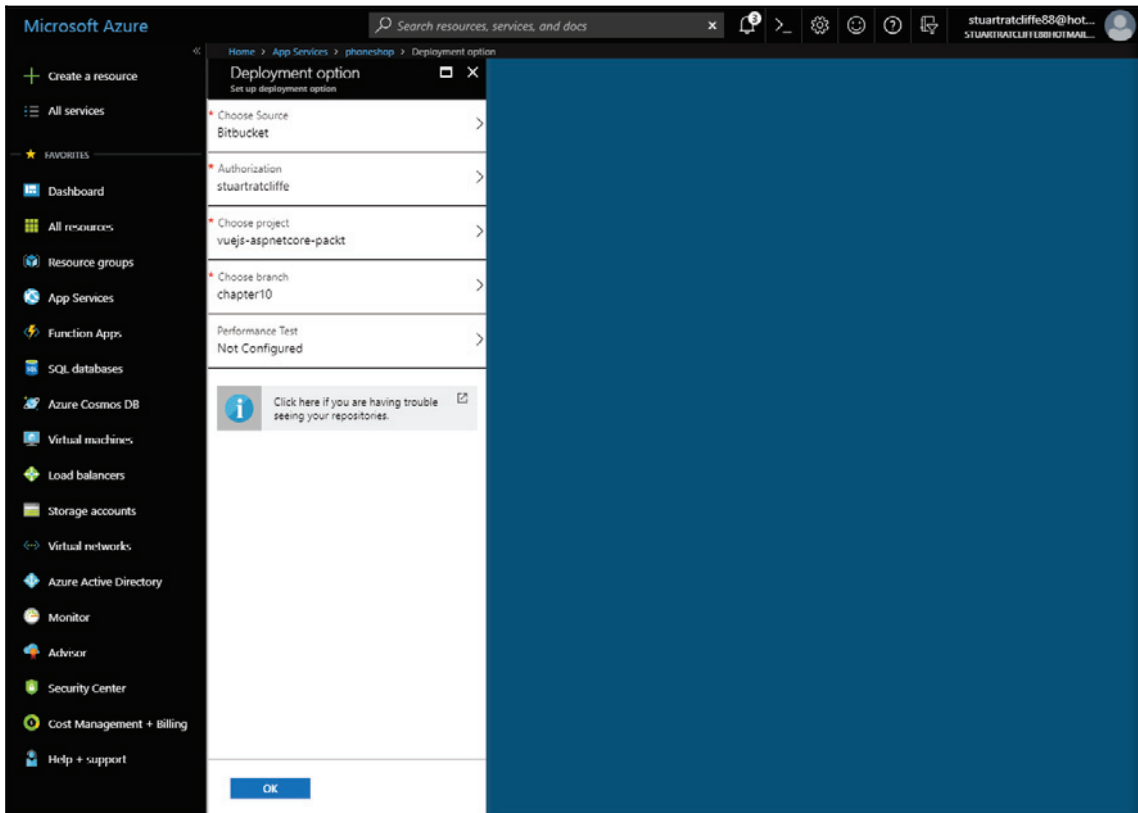
The error itself suggests we run the npm rebuild node-sass command to try to resolve the issue, which is exactly what we've done by adding the additional <Exec /> statement here.

Configuring Git deployments

Configuring Git deployments in Azure is an absolute breeze as long as you are using one of the main source control providers such as GitHub, Bitbucket, or VSTS. Back inside the Azure portal, navigate to the **App Services** page, then click on the name of the app service we created earlier. From the details page that follows, we need to click on the **Deployment options** link from the secondary menu on the left. The screen that follows should look like this:

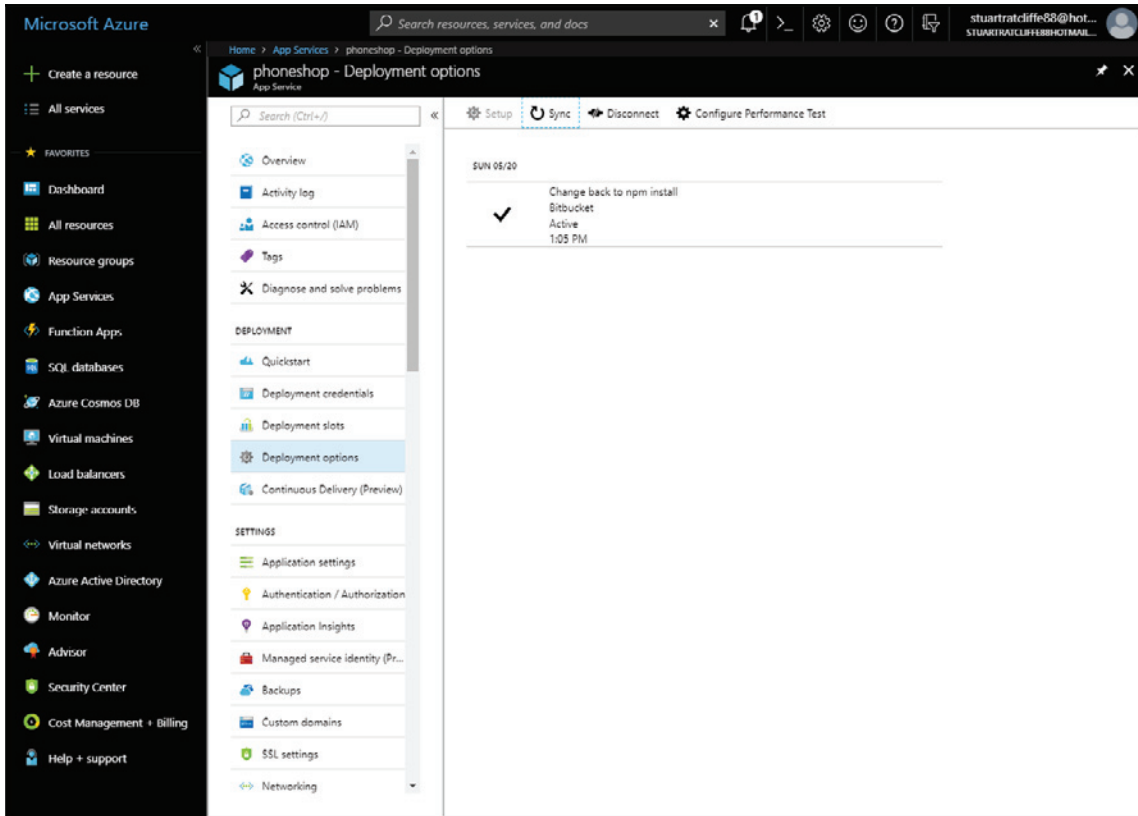


After clicking on the **Configure required settings** button, you will be presented with a number of source control options to choose from. Connecting to these is very simple, and the wizard will guide you through the process very quickly. As an example, I am using Bitbucket, so after selecting the appropriate option, I am presented with the following screen to finalize my deployment options:



You must first complete the **Authorization** section, which will ask you to authenticate with the provider you've chosen, then grant access for Azure to connect to your repositories. Next, you can select a specific project from within your source control account, followed by the branch you wish to deploy from. Optionally, you can also configure performance tests to run after every successful deployment, but to keep things quick and simple, here, I am choosing not to configure them. After hitting the **OK** button, Azure will automatically run the first deployment for you based on the latest version of your code on the branch you specified. As such, make sure the changes we've just made to prepare the application for deployment are committed and pushed to your repository, or the deployment will likely fail.

Click on **Deployment options** again and you should see that the application is currently being deployed. The most recent Git commit message will be shown alongside a spinner to indicate that the deployment is still in progress. After a few minutes, if all is well, the deployment will succeed and you should see a screen like this:



Occasionally, you may see deployments fail due to the following npm error: `npm ERR! code EBUSY`. This does happen from time to time, and the solution is to go to your app service details page and completely stop it and then start it again using the buttons at the top of the **Overview** section.

With the application deployed, you can now check it out by browsing to the Azure URL for your app service, which is based on its name like so: `{appname}.azurewebsites.net`. In my case, I can head to `phoneshop.azurewebsites.net` to see my application up and running. From now on, every `git push` to the remote repository will trigger a newly automated deployment to the environment we just set up and configured.

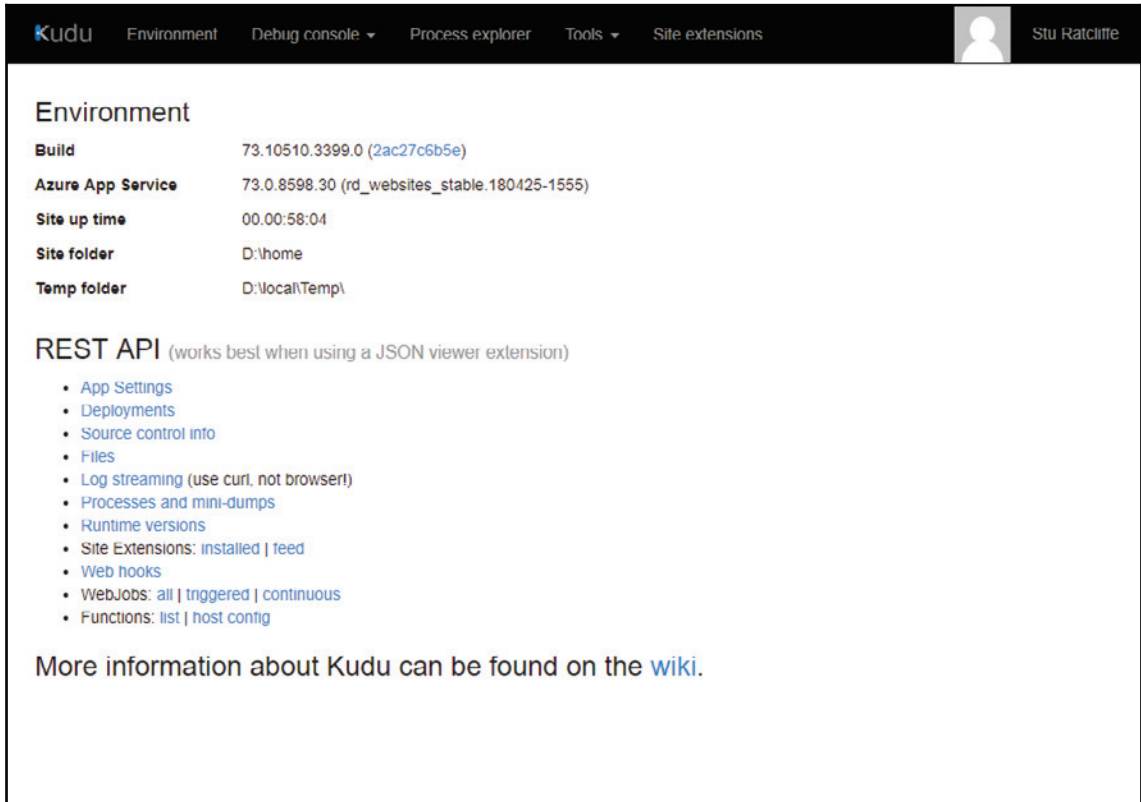
Finalizing the apps configuration

Our app is now deployed, but there are a couple of further enhancements we can make if we wish. First of all, it is incredibly hard to debug live issues without some kind of logging enabled, and second of all, most applications these days should be running on HTTPS—especially those with payment processing, such as ours. In this section, we'll take a look at how to enable logging within Azure, and then how to make the most of the free SSL certificate we have access to while using an Azure website's subdomain as we are doing now.

Enabling logging in Azure

Enabling logging in Azure is probably the most difficult aspect of configuring an app service. While viewing the app service details page, you can scroll right down on the secondary menu to find the **Diagnostics logs** section, where you can configure all kinds of logging options. However, what we'd be interested in is the **Application Logging (Filesystem)** option, but if you hover over the more information icon, you'll notice that it only remains enabled for 12 hours before being automatically disabled again. This is because this feature is intended for use with the live log stream functionality, which can be accessed from the **Log stream** secondary menu item. This won't help us with historic log messages, as you need to be watching the log stream in real time to get any benefit from it.

Instead, we're going to access the filesystem of our app service directly, and enable the standard out log file in the `web.config` file. To do so, find the **Advanced Tools** secondary menu item, then click the **Go** link in the screen that follows. A new browser tab should open, which looks like this:



The screenshot shows the Kudu web interface. At the top, there is a navigation bar with the following items: "Kudu", "Environment", "Debug console", "Process explorer", "Tools", "Site extensions", and a user profile icon for "Stu Ratcliffe".

The main content area is titled "Environment" and contains the following details:

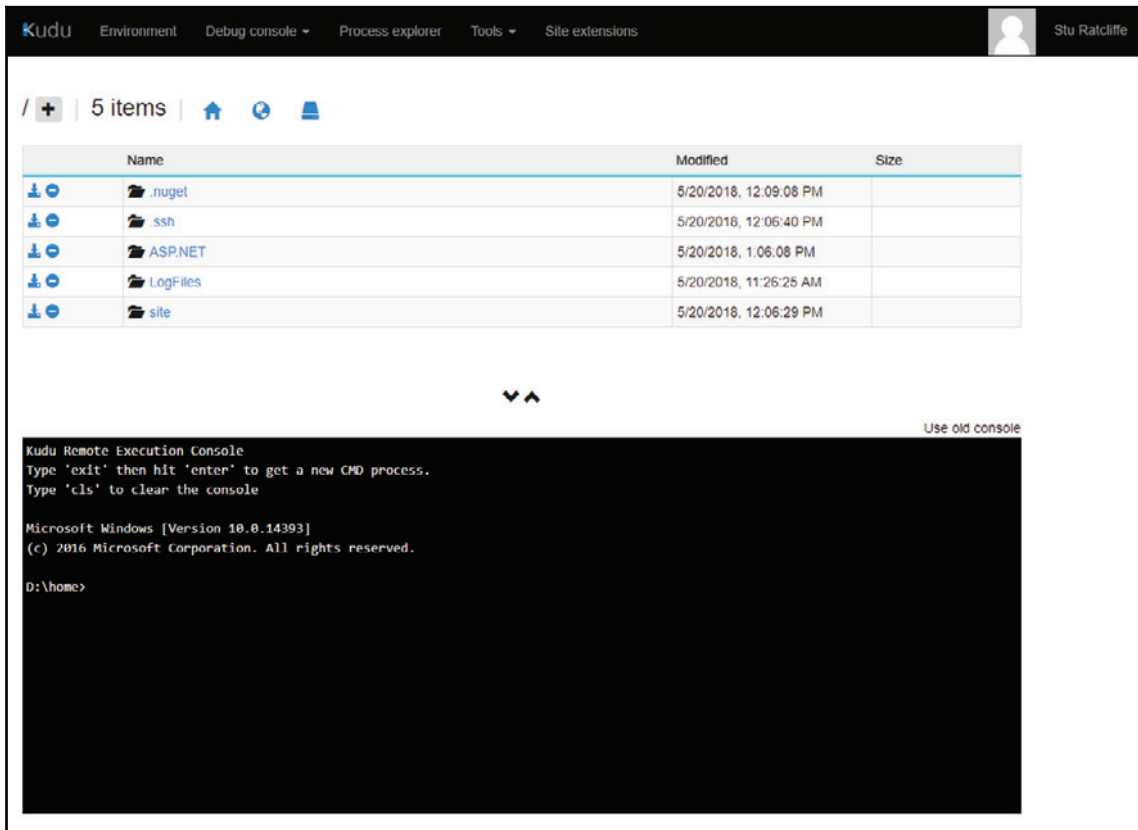
Build	73.10510.3399.0 (2ac27c6b5e)
Azure App Service	73.0.8598.30 (rd_websites_stable.180425-1555)
Site up time	00.00:58:04
Site folder	D:\home
Temp folder	D:\local\Temp\

Below the environment details is a section titled "REST API (works best when using a JSON viewer extension)" with a list of links:

- [App Settings](#)
- [Deployments](#)
- [Source control info](#)
- [Files](#)
- [Log streaming \(use curl, not browser!\)](#)
- [Processes and mini-dumps](#)
- [Runtime versions](#)
- [Site Extensions: installed | feed](#)
- [Web hooks](#)
- [WebJobs: all | triggered | continuous](#)
- [Functions: list | host config](#)






At the bottom of the page, there is a link: "More information about Kudu can be found on the [wiki](#)."

From the main menu at the top of the page, select **Debug console** followed by **CMD**, which should present you with the following screen:



The screenshot shows the Kudu Remote Execution Console interface. At the top, there is a navigation bar with the following items: Kudu, Environment, Debug console (with a dropdown arrow), Process explorer, Tools (with a dropdown arrow), and Site extensions. On the right side of the navigation bar, there is a user profile icon and the name "Stu Ratcliffe".

Below the navigation bar, there is a breadcrumb navigation area showing "/ + | 5 items |" followed by icons for home, refresh, and a bell. Below this is a table listing the contents of the current directory:

Name	Modified	Size
 .nuget	5/20/2018, 12:09:08 PM	
 ssh	5/20/2018, 12:06:40 PM	
 ASP.NET	5/20/2018, 1:06:08 PM	
 LogFiles	5/20/2018, 11:26:25 AM	
 site	5/20/2018, 12:06:29 PM	

Below the table, there are two small arrows pointing up and down. At the bottom of the console, there is a terminal window titled "Kudu Remote Execution Console" with the following text:

```
Kudu Remote Execution Console
Type 'exit' then hit 'enter' to get a new CMD process.
Type 'cls' to clear the console

Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

D:\home>
```

On the right side of the terminal window, there is a link that says "Use old console".

From here, we have full access to the console of our app service, as well as the preceding GUI to browse the filesystem. We need to navigate to the `site/wwwroot` directory, which will display all of the files that we deployed using Git to the app service earlier:

The screenshot shows the Kudu interface for an Azure App Service. The top navigation bar includes 'Kudu', 'Environment', 'Debug console', 'Process explorer', 'Tools', and 'Site extensions'. The user profile 'Stu Ratcliffe' is visible in the top right. The main area displays a file explorer for the directory `... / wwwroot` containing 15 items. The files listed are:

File Name	Last Modified	Size
<code>ECommerce.PrecompiledViews.pdb</code>	5/20/2018, 12:45:47 PM	28 KB
<code>ECommerce.runtimeconfig.json</code>	5/20/2018, 1:05:06 PM	1 KB
<code>Npgsql.dll</code>	7/25/2017, 2:33:52 PM	607 KB
<code>Npgsql.EntityFrameworkCore.PostgreSQL.dll</code>	8/17/2017, 12:03:48 AM	147 KB
<code>package-lock.json</code>	5/20/2018, 1:03:08 PM	236 KB
<code>package.json</code>	5/20/2018, 1:03:07 PM	2 KB
<code>Stripe.net.dll</code>	4/6/2018, 7:39:12 PM	388 KB
<code>web.config</code>	5/20/2018, 1:05:07 PM	1 KB

Below the file explorer is a 'Kudu Remote Execution Console' showing a Windows command prompt session:

```

Kudu Remote Execution Console
Type 'exit' then hit 'enter' to get a new CMD process.
Type 'cls' to clear the console

Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

D:\home>
D:\home\site>
D:\home\site\wwwroot>
  
```

The one we're interested in should be right at the bottom of the list, named `web.config`. Click on the pencil icon to the left to edit this file, after which the contents should look like this:

```

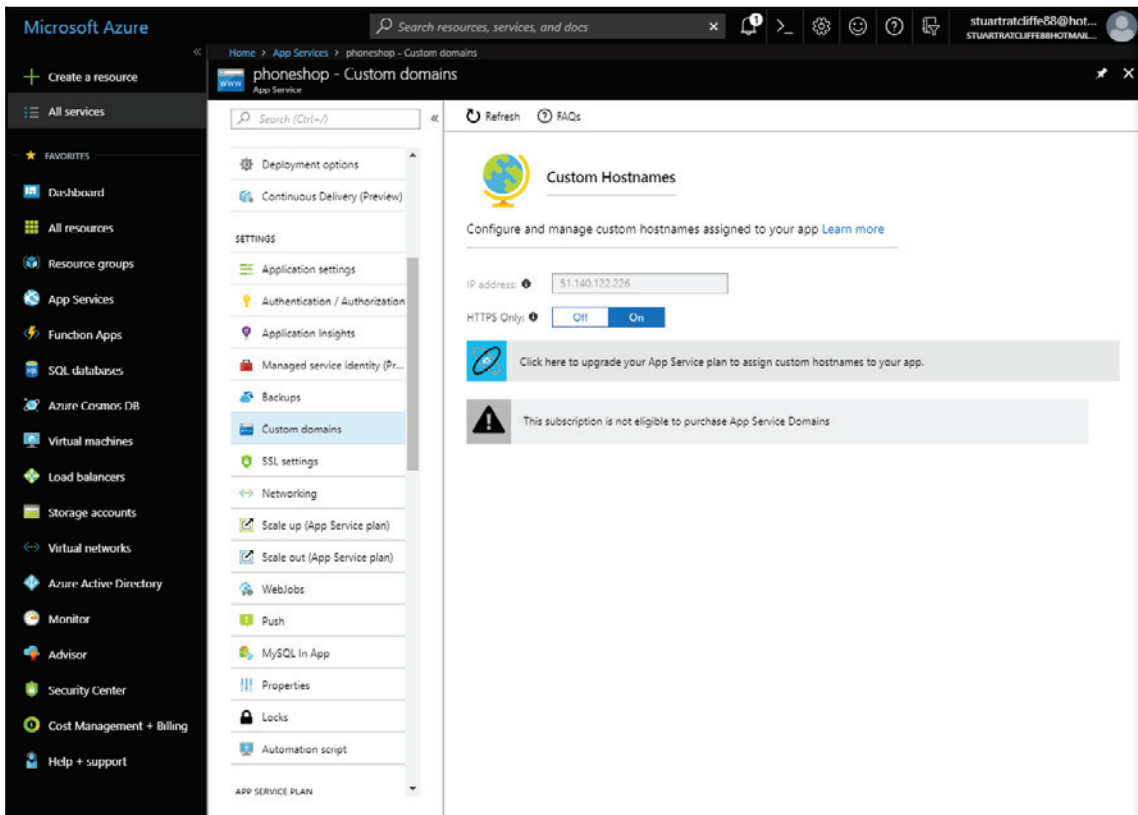
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*"
        modules="AspNetCoreModule" resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath="dotnet" arguments=".\ECommerce.dll"
      stdoutLogEnabled="true" stdoutLogFile="\\?\%home%\LogFiles
  
```

```
\stdout" />
</system.webServer>
</configuration>
```

The only thing we've changed here is to set the `stdoutLogEnabled` setting to `true`. Once you've made this change, hit the **Save** button at the top of the screen to commit the changes. From now on, any unhandled exceptions from our application will be logged to a file in the `home/LogFiles` directory, which you can find by following the same steps as earlier, but changing the directory you navigate to.

Forcing HTTPS connections only

Forcing HTTPS on Azure App Services could not be easier. While viewing the details page for your app service resource, click on the **Custom domains** secondary menu item, then simply select **On**, which is next to the **HTTPS Only** option. After doing so, the page should look like this:



Summary

Azure may not be the cheapest option when deploying small personal projects to the cloud, but for businesses with many different applications, it's highly competitive in comparison with the other major cloud hosting providers. We've only scratched the surface of what you can do with the Azure portal, so I strongly encourage you to make the most of the free 30-day trial and make use of as many different resources as you can.

We started out by getting to grips with the concepts of Azure subscriptions, resources, and resource groups. Once familiar with those, we started to deploy the resources we needed for our application, including a single resource group, a database server, a SQL database, an app service plan, and an app service. We then made some changes to the application itself in order to prepare it for deployment into Azure. The main change we needed to make was to enable support for multiple database providers so that we could utilize a full-blown SQL server in our production environment, while remaining cross-platform-compliant in development.

We then configured our application to deploy automatically from every `git push` to a source control provider of our choice. Finally, we added some additional configuration to enable logging, and forced the use of HTTPS on our newly deployed application.

In the next few chapters, we're going to cover some more advanced topics such as refresh token authentication flows, server-side rendering, and Continuous Integration and deployment using VSTS.

11

Authentication and Refresh Token Flow

In the last chapter, we looked at how to set up a production-ready Azure environment and configured our application so that we can deploy it automatically using Git deployments. As the application is now feature-complete and deployed to our chosen hosting environment, it's time to start building on some of our existing features using some more advanced concepts. In this chapter, we're going to extend our current authentication mechanism to include *refresh* tokens as well as the access tokens we're already using. In summary, we'll cover the following topics in this chapter:

- Refresh tokens: What are they and why would we use them?
- Adding refresh token support to the backend
- Handling refresh token flow on the frontend using axios interceptors

Understanding refresh tokens

Before we can implement them, we need to understand what refresh tokens are used for, what they actually are, and why we would want to use them on top of our current **JSON Web Token (JWT)**-based access tokens.

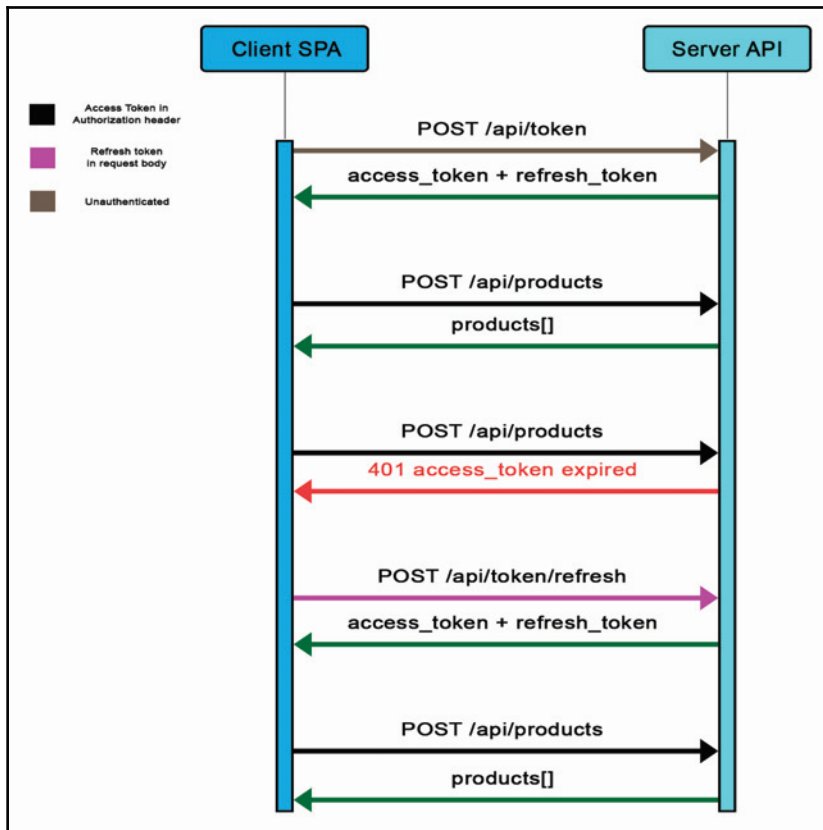
What are refresh tokens used for?

Let's start by understanding the basics of what we're actually going to use refresh tokens for, before diving into the nitty gritty of what they are actually composed of. As the name implies, a refresh token is used for refreshing an existing token—or more specifically, refreshing an existing *access* token.

Refresh tokens are used to obtain a new access token as and when your existing access token expires. What's more, this doesn't require any input from the user; it is all done silently in the background without them even knowing. Essentially, this enables users to remain logged in indefinitely. At this point, you might be wondering about the security aspect of this, but don't worry, we're actually *increasing* the security of our application! More on that later in this chapter.

So, how does this actually work? When an access token expires, we return a 401 HTTP status code to signify that the user is no longer authenticated. With some modifications to our frontend application code, we can listen for this status code on every API request we make, and attempt to refresh the token if we catch one. If the refresh succeeds, we can retry the original request and carry on as if nothing happened, but if it fails, we simply redirect the user to the login page so that they can authenticate fully again.

The following diagram shows a simple refresh token flow between a **Client SPA** and a **Server API**:



What are refresh tokens?

Plainly and simply, a refresh token is an arbitrary string of text that we assign to each user within the database as and when they authenticate successfully. The contents of this string of text do not really matter, as long it is a unique value; when we try and refresh the user's access token, we need to be able to uniquely identify them based on their refresh token.

So, why do we bother if the contents of the string don't matter, seeing as we already have a unique identifier in the form of the user's database ID value? There are two aspects to this question: firstly, why don't the contents of the string matter; and secondly, why can't we use their database ID as the refresh token if it already uniquely identifies them?

Starting with the latter, we can't use their database ID value because that value should never change once initially created. Every time a user authenticates or refreshes their existing access token, a new refresh token is generated and stored in the database. By changing the refresh token so often, it makes it far less likely that malicious users who have somehow gained access to another user's refresh token can use it to obtain an access token.

So, now that we know we can't use their existing ID, and must use a new unique string value, why does it not matter what the content is? The answer is simple: the refresh token is not trying to fully describe the user it's been assigned to—this is the job of an access token, which has a whole bunch of claims encoded within it. The refresh token simply needs to be able to uniquely identify which user in the database is trying to refresh their existing access token. The obvious choices for globally unique strings are GUIDs, but to make it slightly shorter and more readable, we'll be converting a GUID to a Base64-encoded string instead.



If you aren't familiar with the term **GUID**, it stands for **Globally Unique Identifier**, and describes a value that is virtually guaranteed to be unique, even across multiple databases and/or application boundaries.

Why use refresh tokens?

So, why would we actually bother using refresh tokens? After all, our access tokens are already configured to be valid for 30 days from the point of login, which is quite a long time anyway.

The main answer to this question is to increase the security of our application. Once a JWT access token has been generated, it is 100% valid until it expires based on the expiration date we set for it when we created it. There is absolutely nothing that we can do to prevent it from being used to successfully authenticate a user. This is because JWT-based authentication does not consult with the database when an existing access token is passed to an API request; everything the server needs to know about a user to authorize them is encoded into the token itself. The problem, then, is that the longer an access token is valid for, the more time a malicious user has to make use of that access token if they manage to get hold of it.

On the other hand, refresh tokens are stored in the database, and as such the database must be queried in order for a refresh token to be validated. If we remove the refresh token from the database, a subsequent API request to refresh the user's access token would then fail. This means that they can have a much longer expiration due to the fact that we have a way of invalidating them if we suspect a malicious user has gained access to the tokens of one of our customers.

Refresh tokens allow us to have very short access token expiration dates set, usually measured in minutes rather than days. This minimizes the window that access tokens can be abused for if they are compromised, while still leaving users logged in for longer periods due to the ability to refresh the token until we manually invalidate their refresh token as well.

Adding refresh token support to the backend

We've covered enough theory for now, so let's crack on and see how refresh tokens actually work. We can't do anything on the frontend of the app until the backend supports refresh tokens, so that's where we're going to start.

Extending the AppUser model

First up, we need a place to store the refresh token as and when we generate it. As previously discussed, this token is unique to each user, so it belongs in our `Data/Entities/AppUser` entity model:

```
namespace ECommerce.Data.Entities
{
    public class AppUser : IdentityUser<int>
```

```

{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string RefreshToken { get; set; }

    [NotMapped]
    public string FullName
    {
        get { return $"{FirstName} {LastName}"; }
    }

    public List<Order> Orders { get; set; } = new List<Order>();
}

```

With this property in place, we also need to make sure it is unique. As this requires an index to be placed on the database field, it also serves the purpose of speeding up the query of finding a user based on their refresh token—something we'll be doing quite frequently. Open up the `Data/EcommerceContext.cs` file, locate the overridden `OnModelCreating` method, and update it as follows:

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    modelBuilder.Entity<Product>()
        .HasIndex(b => b.Slug)
        .IsUnique();

    modelBuilder.Entity<ProductFeature>()
        .HasKey(x => new { x.ProductId, x.FeatureId });

    modelBuilder.Entity<ProductVariant>()
        .HasKey(x => new { x.ProductId, x.ColourId, x.StorageId });

    modelBuilder.Entity<Order>()
        .OwnsOne(x => x.DeliveryAddress);

    modelBuilder.Entity<AppUser>()
        .HasIndex(x => x.RefreshToken)
        .IsUnique();
}

```

And finally, we can create a migration to update our database accordingly. Open a Terminal at the root of your project and run the following:

```
dotnet ef migrations add RefreshTokens
```

Generating refresh tokens

Now that we have a place to store the refresh token, we actually need to generate one. We already have the ideal place to do so, which is in the `GenerateToken` method of the `Features/Authentication/Controller.cs` class:

```
private async Task<TokenViewModel> GenerateToken(AppUser user)
{
    ...

    var refreshToken =
        Convert.ToBase64String(Guid.NewGuid().ToByteArray());

    user.RefreshToken = refreshToken;
    await _userManager.UpdateAsync(user);

    return new TokenViewModel
    {
        AccessToken = new JwtSecurityTokenHandler().WriteToken(token),
        AccessTokenExpiration = expires,
        RefreshToken = refreshToken,
        FirstName = user.FirstName,
        LastName = user.LastName,
        Roles = roles
    };
}
```

Now, every time a user authenticates using their email and password, a new refresh token will be generated, stored in the database, and returned, along with their access token. You'll probably have noticed that the `RefreshToken` property doesn't actually exist on the `TokenViewModel` class, so let's add it now:

```
namespace ECommerce.Features.Authentication
{
    public class TokenViewModel
    {
        [JsonProperty("access_token")]
        public string AccessToken { get; set; }
        [JsonProperty("access_token_expiration")]
        public DateTime AccessTokenExpiration { get; set; }
        [JsonProperty("refresh_token")]
        public string RefreshToken { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public IEnumerable<string> Roles { get; set; }
    }
}
```

Refreshing JWT access tokens

All that's left now is to provide a means for the client-side application to actually refresh an access token when it expires. We can do so by adding a simple controller action to the `Features/Authentication/Controller.cs` class:

```
[HttpPost("refresh")]
public async Task<IActionResult> RefreshToken([FromBody]
RefreshTokenViewModel model)
{
    if (!ModelState.IsValid)
        return BadRequest();

    var user = await _db.Users.SingleOrDefaultAsync(x => x.RefreshToken ==
model.RefreshToken);

    if (user == null)
        return BadRequest();

    var token = await GenerateToken(user);

    return Ok(token);
}
```

By now, this should be fairly self-explanatory, so in summary what we are doing here is the following:

1. Defining a HTTP POST action listening on the `/api/token/refresh` route
2. Returning a plain `BadRequest` result if validation fails
3. Querying the database for the single user whose `RefreshToken` matches that of the model being posted from the client
4. Returning a plain `BadRequest` result if no matching user is found
5. Generating new access/refresh tokens using the existing `GenerateToken` method
6. Returning the token model within an `Ok` result

Notice how we've not bothered to supply any kind of error information with the `BadRequest` results being returned here. As previously discussed, our client-side app will use this endpoint without the end user ever knowing about it. As such, we don't really need to supply any kind of error information, unless you wish to for debugging purposes.

The RefreshTokenViewModel class that we're accepting on this action method needs to look like this:

```
namespace ECommerce.Features.Authentication
{
    public class RefreshTokenViewModel
    {
        [Required, JsonProperty("refresh_token")]
        public string RefreshToken { get; set; }
    }
}
```

If you hadn't already noticed, we're also accessing the DbContext directly using an _db property that does not yet exist. We need to add the additional dependency to the top of this controller like so:

```
namespace ECommerce.Features.Authentication
{
    [Route("api/[controller]")]
    public class TokenController : Controller
    {
        private readonly EcommerceContext _db;
        private readonly SignInManager<AppUser> _signInManager;
        private readonly UserManager<AppUser> _userManager;
        private readonly IConfiguration _configuration;

        public TokenController(
            EcommerceContext db,
            SignInManager<AppUser> signInManager,
            UserManager<AppUser> userManager,
            IConfiguration configuration)
        {
            _db = db;
            _signInManager = signInManager;
            _userManager = userManager;
            _configuration = configuration;
        }

        ...
    }
}
```

With that, we have everything we need to issue and validate refresh tokens from our server-side API.

Finishing up

When we first implemented authentication back in [Chapter 7, User Registration and Authentication](#), we were returning the expiry date of the access token so that the client could use it to determine whether the token was still valid without needing to make an API call. However, now that our refresh token functionality will automatically retrieve new access tokens for us, we no longer care about the access token's expiry date. As such, we can remove it from the model that's returned to the client.

Start by opening the `Features/Authentication/TokenViewModel.cs` file and removing the lines highlighted in the following code snippet:

```
namespace ECommerce.Features.Authentication
{
    public class TokenViewModel
    {
        [JsonProperty("access_token")]
        public string AccessToken { get; set; }
        //[[JsonProperty("access_token_expiration")]
        //public DateTime AccessTokenExpiration { get; set; }
        [JsonProperty("refresh_token")]
        public string RefreshToken { get; set; }
        public IEnumerable<string> Roles { get; set; }
    }
}
```

Next, at the bottom of the `GenerateToken` method of the `Features/Authentication/Controller.cs` class, modify the returned object by removing the access token's expiry field:

```
return new TokenViewModel
{
    AccessToken = new JwtSecurityTokenHandler().WriteToken(token)
    //AccessTokenExpiration = expires,
    RefreshToken = refreshToken,
    Roles = roles
};
```

Finally, we're currently taking a numeric value from the `appsettings.json` file and using it as the number of days an access token is valid for. We now want this to be the number of *minutes* it is valid for instead. In the same controller, find the following line in the `GenerateToken` method and change it like so:

```
var expires =
    DateTime.Now.AddMinutes(Convert.ToDouble(_configuration["Authentication:Jwt
    ExpireMins"]));
```


We now also need to rename this setting within the `appsettings.json` file itself:

```
"Authentication": {  
  "JwtKey": "ECOMMERCE_SUPER_SECRET_KEY",  
  "JwtIssuer": "http://localhost:5000",  
  "JwtAudience": "http://localhost:5000",  
  "JwtExpireMins": 15  
}
```

This completes our entire server-side changes. Let's see what's involved on the client side.

Adding refresh token support to the frontend

We have now discussed the basic flow of how refresh tokens work, as well as implementing a means of performing that flow by adding an additional API endpoint to our server. All that remains is deciding how we actually modify the frontend of the application to automatically handle our refresh token flow without letting the user know what's happening.

We know that we need to watch every API call for a 401 HTTP status code response and attempt to obtain a new access token by hitting the refresh token API endpoint. We also know that we don't want to be doing this manually on every API request in the system, as it would be completely impractical to try and maintain them if the logic ever changed. We need a way of defining this logic in a single place and have it automatically work any time we add new API requests in the future.

Luckily, all of these requirements are handled very easily by using an axios **interceptor** function. Axios interceptor functions quite literally *intercept* our API requests and/or responses and provide a means for us to run our custom logic on every request in the system with ease. We can define such a function to look for a 401 response status code on every API call, check whether we have a refresh token available, and use it to make a follow-up API call to the refresh token endpoint if we do. This will start to make a lot more sense when we look at the code later!

The only problem is that the only place we can really put this function right now is in the `ClientApp/boot.js` file, which is already getting pretty bloated. The bulk of this file is currently router-specific code, which we can certainly extract into separate files in order to thin it down a bit. The other benefit this will give us is that we'll be able to import the `router` object into other files so that we can make use of it; this is exactly what we need for our refresh token handling.

Extracting router configuration into separate files

Start by creating a new `ClientApp/router` directory, which will ultimately contain two files: `index.js` and `routes.js`. Starting with `routes.js`, we need to extract our page component import statements out of `ClientApp/boot.js` and drop them at the top of this new file:

```
//import page components
import Catalogue from "../pages/Catalogue.vue";
import Product from "../pages/Product.vue";
import Cart from "../pages/Cart.vue";
import Checkout from "../pages/Checkout.vue";
import Account from "../pages/Account.vue";

//import admin pages
import AdminIndex from "../pages/admin/Index.vue";
import AdminOrders from "../pages/admin/Orders.vue";
import AdminProducts from "../pages/admin/Products.vue";
import AdminCreateProduct from "../pages/admin/CreateProduct.vue";
```

Next, we can extract the `routes` array from `ClientApp/boot.js` as well:

```
const routes = [
  { path: "/products", component: Catalogue },
  { path: "/products/:slug", component: Product },
  { path: "/cart", component: Cart, meta: { role: "Customer" } },
  {
    path: "/checkout",
    component: Checkout,
    meta: { requiresAuth: true, role: "Customer" }
  },
  {
    path: "/account",
    component: Account,
    meta: { requiresAuth: true, role: "Customer" }
  },
  {
    path: "/admin",
    component: AdminIndex,
    meta: { requiresAuth: true, role: "Admin" },
    redirect: "/admin/orders",
    children: [
      {
        path: "orders",
        component: AdminOrders
      },
    ],
  }
];
```

```
    path: "products",
    component: AdminProducts
  },
  {
    path: "products/create",
    component: AdminCreateProduct
  }
]
},
{ path: "*", redirect: "/products" }
];
```

Finally, we simply need to export the `routes` array so it can be imported by the main router definition file:

```
export default routes;
```

The main router definition will now belong in the `ClientApp/router/index.js` file, and again we can start by extracting some of the `import` statements that used to reside in `ClientApp/boot.js`:

```
import Vue from "vue";
import VueRouter from "vue-router";
import NProgress from "nprogress";

import routes from "../routes";
import store from "../store";

Vue.use(VueRouter);
```

Note that the `Vue` import line needs to remain in both the `ClientApp/router/index.js` and `ClientApp/boot.js` files, but `VueRouter` and `NProgress` can be removed from the boot file. We can then import the `routes` array from the other file we just created, as well as import the `store` object. Again, the `store` object needs to remain in both files. Finally, we've extracted the `Vue.use(VueRouter);` line, which can now be removed from the boot file.

Next, we can extract the actual `router` object declaration from the boot file:

```
const router = new VueRouter({ mode: "history", routes: routes });
```

This is followed by the `router.beforeEach` hook:

```
router.beforeEach((to, from, next) => {
  NProgress.start();

  if (to.matched.some(route => route.meta.requiresAuth)) {
```

```
if (!store.getters.isAuthenticated) {
  store.commit("showAuthModal");
  next({ path: from.path, query: { redirect: to.path } });
} else {
  if (
    to.matched.some(
      route => route.meta.role &&
      store.getters.isInRole(route.meta.role)
    )
  ) {
    next();
  } else if (!to.matched.some(route => route.meta.role)) {
    next();
  } else {
    next({ path: "/" });
  }
}
} else {
  if (
    to.matched.some(
      route =>
        route.meta.role &&
        (!store.getters.isAuthenticated ||
          store.getters.isInRole(route.meta.role))
    )
  ) {
    next();
  } else {
    if (to.matched.some(route => route.meta.role)) {
      next({ path: "/" });
    }

    next();
  }
}
});
```

After that, here is the `router.afterEach` hook:

```
router.afterEach((to, from) => {
  NProgress.done()
});
```

Finally, we need to export the `router` object so that it can be imported elsewhere:

```
export default router;
```

Just to make sure everything is still looking OK in the `ClientApp/boot.js` file, the `import` statements at the top should currently look like this:

```
import Vue from "vue";
import router from "./router";
import store from "./store";
import BootstrapVue from "bootstrap-vue";

import VueToastr from "@deveodk/vue-toastr";
import "@deveodk/vue-toastr/dist/@deveodk/vue-toastr.css";
import VeeValidate from "vee-validate";

//helpers
import "./helpers/validation";
```

Note that we needed to import the `router` object from our newly created `router` directory. Next, the Vue plugin installation section should look like this:

```
Vue.use(BootstrapVue);
Vue.use(VueToastr, {
  defaultPosition: "toast-top-right"
});
Vue.use(VeeValidate);

// filters
import { currency, date } from "./filters";

Vue.filter("currency", currency);
Vue.filter("date", date);
```

This is followed by our store initialization from the `localStorage` section:

```
import axios from "axios";
const initialState = localStorage.getItem("store");

if (initialStore) {
  store.commit("initialise", JSON.parse(initialStore));
  if (store.getters.isAuthenticated) {
    axios.defaults.headers.common["Authorization"] = `Bearer ${
      store.state.auth.access_token
   }`;
  }
}
```

And finally, we have our Vue instantiation:

```
new Vue({
  el: "#app-root",
  router,
  store,
  render: h => h(require("./components/App.vue"))
});
```

Refreshing access tokens with axios interceptors

Axios interceptor functions can be thought of like action filters or middleware in ASP.NET MVC. They are invoked on every single request that we make using `axios`, and we can hook into the request both before it has run and after. This makes them ideal for our refresh token requirements, as we can check every response that we get back from an API request and perform some custom logic if we detect an authentication failure.

In this case, we'll be looking for 401 response status codes, checking whether we have a refresh token to use, and then attempting to obtain a new access token using the new API endpoint we created earlier on. If we get one, we re-trigger the original request that failed, and the user is none the wiser that any of this has happened, unless they're watching the network tab in their browser devtools.

This will make more sense when we get our interceptor in place, so let's create a new `ClientApp/helpers/interceptors.js` file and start by importing the modules we'll need, like so:

```
import axios from "axios";
import store from "../store";
import router from "../router";
```

We obviously need `axios`, seeing as this is an `axios` interceptor function, but we'll also need a reference to our `store` and `router` objects so that we can persist the new access token to the store, or redirect to the home page if the refresh token fails. Next, we need to define the interceptor function itself, which currently has the body of the function omitted for brevity:

```
axios.interceptors.response.use(undefined, function(error) {  
  ...  
})
```

As we are only interested in checking the *response* of an `axios` call, we use `axios.interceptors.response`. However, if we wanted to perform some logic *before* an API request, we could just as easily have used `axios.interceptors.request` as well. For example, we could have defined an `axios` interceptor function that applies our access token to the request if it exists in local storage, rather than our current implementation of adding it to the default headers.

The `use` function that we invoke in the preceding code takes two optional arguments, both of which are callback functions. The first is a function to invoke if the response returns a successful HTTP response code, and the second is a function to invoke if an error code is returned. All we care about are 401 (Unauthorized) codes, so we can pass `undefined` as our success code callback function. All of our logic belongs inside the error code callback function, which receives a parameter that we've aptly named `error` containing the details of the request and why it failed.

The first thing we need to do inside this function is get hold of the original request that just failed so that we can retry it if we successfully manage to refresh the access token:

```
const originalRequest = error.config;  
if (  
  error.response.status === 401 &&  
  !originalRequest._retry &&  
  store.state.auth.refresh_token  
) {  
  ...  
}  
  
return Promise.reject(error);
```

We can obtain the object model of the original request from the `error.config` variable, which in this case we reference using a new `originalRequest` variable. Next, we check whether the response code returned was equal to 401, that the request was not a retry, and that we have a refresh token in our store state. If any of these checks fail, we don't do anything other than reject the promise, as this isn't a response that we care about intercepting with this function.



All axios interceptor functions return promises by default.

However, if all of these checks pass, we drop down inside the `if` block, where we start doing the following:

```
originalRequest._retry = true;

const payload = {
  refresh_token: store.state.auth.refresh_token
};
```

At this point, we can mark our `originalRequest` object as a retry by setting the `_retry` property to `true`. We then construct a `payload` object, which is what we'll be sending to the API in order to attempt an access token refresh. The API request itself is next, which looks like this:

```
return axios
  .post("/api/token/refresh", payload)
  .then(response => {
    const auth = response.data;
    axios.defaults.headers.common["Authorization"] = `Bearer ${
      auth.access_token
   }`;
    originalRequest.headers["Authorization"] = `Bearer ${
      auth.access_token
   }`;
    store.commit("loginSuccess", auth);
    return axios(originalRequest);
  })
  .catch(error => {
    store.commit("logout");
    router.push({ path: "/" });
    delete axios.defaults.headers.common["Authorization"];
    return Promise.reject(error);
  });
```


We perform a HTTP POST request, passing the `payload` object that we just created, then checking the response and performing some logic depending on whether it succeeded or failed. If we successfully managed to refresh the access token, we drop into the `then` block and do the following:

1. Create a new `auth` variable referencing the `response.data` property
2. Set the global `axios` default authorization header based on the `auth.access_token` property
3. Set the `originalRequest` authorization header in the same way
4. Commit the `loginSuccess` mutation in order to persist the new tokens to the store
5. Retry the original request using `return axios(originalRequest);`

You may be wondering why we need to set the authorization header on both the global `axios` defaults and the `originalRequest` object. Unfortunately, the `originalRequest` object would have contained the expired access token taken from the global `axios` defaults at the point the request was originally sent. Even by overriding the defaults at this point, the `originalRequest` object will remain unchanged, as when we call `axios(originalRequest)`, it quite literally sends the request as is, with no modifications from the global defaults.

If we fail to refresh the access token, we then drop down inside the `catch` block and do the following:

1. Commit the `logout` mutation to clear the `auth` state from the store
2. Redirect to the home page using the `router` object we imported earlier
3. Delete the default `axios` authorization header
4. Reject the promise returned from the interceptor function

At this point, it should make more sense as to why our router configuration refactoring was worthwhile. As we need access to the `router` object here, the only place it was available was in the `ClientApp/boot.js` file, or any of our Vue components due to the fact that the router is injected into them all using the `$router` property. This function doesn't belong in either of those places, so it makes sense to extract the router config into a file that we could export it from, meaning we could then import it in files such as this one.

Our interceptor function is now complete, but we still need to import it into our entry boot file or it will never be invoked. Open up the `ClientApp/boot.js` file and add the following import line:

```
//helpers
import "../helpers/validation";
import "../helpers/interceptors";
```

Our interceptor function will now be globally used by all `axios` requests, meaning our refresh token handling is now complete. However, we have one final change to make to ensure things work as we expect.

Finishing up

If you remember, at the end of our server-side changes, we removed the access token expiration date from the response returned from a successful authentication request. We don't care about when the token expires anymore, as our refresh token interceptor will handle our need to obtain a new access token every 15 minutes anyway. As such, we can modify our authenticated getter function to omit the expiry date check. Open up the `ClientApp/store/getters.js` file and modify the `isAuthenticated` function like so:

```
export const isAuthenticated = state => {
  return state.auth !== null && state.auth.access_token !== null;
};
```

We can now test that everything is working properly by monitoring the **Network** tab within Chrome's DevTools. Rather than waiting 15 minutes for your access token to expire, you can change the expiration in the `appsettings.json` file to a much shorter interval. You'll then need to log in, navigate to a page that requires authentication, wait for the access token to expire, and hit the refresh button. You should see output similar to the following:

The screenshot displays a web application interface on the left and the Chrome DevTools Network tab on the right. The web application shows a list of orders for a customer named 'Stu Ratcliffe'. The Network tab shows several requests, with three highlighted in blue: an 'orders' request (401 status), a 'refresh' request (200 status), and another 'orders' request (200 status).

Order #	Customer	Placed	Items	Total	Payment status
1	Stu Ratcliffe	10/04/2018	3	£457.00	Paid
2	Stu Ratcliffe	14/04/2018	5	£735.00	Paid
3	Stu Ratcliffe	14/04/2018	5	£735.00	Paid
4	Stu Ratcliffe	14/04/2018	1	£139.00	Paid
5	Stu Ratcliffe	14/04/2018	1	£149.00	Paid
6	Stu Ratcliffe	15/04/2018	1	£599.00	Paid

Name	Status	Type	Initial	Size	Time	Waterfall	As
orders	200	Other	88...	139 ms			
refresh	200	xhr	82...	349 ms			
orders	401	xhr	19...	72 ms			
orders	200	xhr	66...	63 ms			

Summary

The refresh token flow is a great way of increasing the security of a JWT-based authentication system, and is heavily used in OAuth2 and Open ID Connect-based applications. We've seen how easy it was to implement on the server, and just one additional file was needed on the client to tie everything together. The benefits of using refresh tokens certainly outweigh the very minor overhead of the time taken to implement them.

In the next chapter, we're going to look at another more advanced concept in Vue in order to enhance the SEO of our application: **server-side rendering (SSR)**.

12

Server-Side Rendering

In the last chapter, we increased the security of our application by implementing a more advanced refresh token authentication flow. This enabled us to minimize the amount of time an *access* token is valid for, and as such vulnerable for, without compromising on the length of time a user can remain logged in for.

The next, more advanced, topic on our feature list is **Server-Side Rendering (SSR)**. This is by far one of the most complex features to properly set up and configure in a Vue.js application, particularly when you have password-protected sections as we do. In summary, in this chapter, we're going to look at the following topics:

- Why use SSR in the first place?
- The easy way: Nuxt.js
- Additional npm packages required for SSR
- More advanced webpack configuration
- How to boot a Vue.js application on both the client and server
- How to pre-fetch component data on the server
- How to access protected API routes from the server
- How to conditionally hide specific components on the server
- How to validate that SSR is working

Before we can do anything else, we have a few npm packages to install and some fairly major code refactors to perform before we can render our application on the server.

Why use SSR in the first place?

When SPA frameworks such as Vue, React, and Angular came about, one of the benefits of using them was to move the task of rendering your application away from the server and into the client. This meant that servers were only required to render a minimal HTML file consisting of some kind of root element that the application would be mounted into, and the asset references of the JavaScript and CSS files needed by the application. For example, if we run the application now and look at its source using a web browser, this is all we receive from the server:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8" />
5   <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6   <title>Home Page - ECommerce</title>
7   <base href="/" />
8
9   <link rel="stylesheet" href="/dist/vendor.css?v=Di_qY8daTe8dfcrQ10FFMT8eLhn5xT5DAp8K_IEDOE" />
10
11  <link href="https://use.fontawesome.com/releases/v5.0.6/css/all.css" rel="stylesheet" />
12 </head>
13 <body>
14
15 <div id='app-root'>Loading...</div>
16
17
18
19 <script src="https://js.stripe.com/v3/"></script>
20 <script src="/dist/vendor.js?v=xt2W6pS6B9f2HCFAsryk9U750J9RaR68Pk_vINhcLAK"></script>
21
22 <script src="/dist/main.js?v=k0yYqqI-6ECv_KbEz1D-KSDsHd2MdggmE7XpdNmNqyQ"></script>
23
24 </body>
25 </html>
26
```

Inspecting the existing server-rendered HTML

Notice the only elements rendered within the `body` tags are our `app-root` element and JavaScript references. Once the JavaScript assets are loaded, they then render the application and mount it into the DOM using the root element that we rendered as a placeholder. So, why are we now suddenly making the decision to move that concern back to the server? We're not going to go into a lot of detail, but it's worth at least exploring the fundamentals of why SSR is important.

Search engine optimization

The first and foremost benefit is **Search Engine Optimization (SEO)** purposes. SEO is the process of optimizing a web application in order to promote it as far up the rankings of search engines as possible. SEO is important in a lot of web applications, but in an e-commerce application, it is particularly important for new customers to be able to find our shop when they search for the items they wish to buy.

If we render our application on the server, it is guaranteed to be treated by the search engine crawlers in exactly the same way as any other standard HTML web page, with absolutely no negative impact on its rankings. However, although there is an argument that most search engines now support the crawling of JavaScript, generally speaking, I don't believe it is 100% reliable. It is more than possible that our search engine rankings can in fact be seriously impacted by *not* rendering on the server, ultimately *decreasing* our sales potential and profitability.

Performance

The other main benefit to SSR is performance, or at least *perceived* performance. But what exactly does that mean? SPA frameworks make web applications much more interactive with rich client-side UIs and animations. However, they also drastically increase the amount of JavaScript that needs to be downloaded when a user visits your web page. If we only render the application on the client, unless we specifically cater for it, the user will be greeted with a blank white screen until the JavaScript has been loaded and kicks in to render the application.

On a modern PC and browser combination, this happens so fast that it's barely noticeable. However, we still need to think about those on older machines, or slower connections such as unreliable mobile data connections. The delay will be far more noticeable and may even deter users from sticking around to wait. But, this doesn't actually answer the question of what we mean by *perceived* performance.

When we render the application on the server, the browser is sent the full HTML of the specific page they requested. This means that they potentially see a fully-rendered web page far sooner than if we were to render the same page on the client instead. However, even though the page looks to be fully rendered, it won't be *interactive* until all of the JavaScript has been loaded and initialized. Unfortunately, SSR isn't a silver bullet that solves all of our problems in that respect! The concept of perceived performance is that users are almost tricked into believing that the website is loading faster than it actually is, because they see something in their browser sooner, even if they can't actually click around the app for another few milliseconds.

How does SSR work?

SSR adds a lot of complexity to our application. As it stands, with our application being rendered on the client, all the server does on the initial web request is render an empty HTML shell containing script references to our application's JavaScript files. Once the browser renders this empty shell, it starts to download the script files and executes them when they finish. At this point, our Vue application will be initialized, and will start to trigger the API requests that are necessary to fetch the data required to render the full HTML and CSS of the application. Once the application finishes rendering, it is immediately ready for the user to start interacting with it. This means that the application is both viewable and interactable at the exact same time, right at the end of the application startup cycle.

In comparison, when we start using SSR, the initial request to the server results in a fully rendered HTML page being returned and rendered by the browser. To do so, the server must initialize our Vue application, including the triggering of any API requests necessary to populate the store and complete the initial render. Once the browser receives this HTML and renders it, the application is viewable by the user. However, the browser still needs to download the application script references and initialize the Vue application *again* on the client. This means that although the application is viewable much sooner in the startup process, there is a slight delay before the client initializes the application and it becomes interactable.

On the face of it, this process doesn't seem too scary, but there are a number of complications that this brings about. We'll cover these in much more detail as we start to implement SSR, but in summary, they include the following:

- API requests performed during a server render may well require authentication, and as such need access to the user's JWT token. This is currently an issue due to the fact that the token is stored in local storage, which isn't accessible on the server.
- As the Vue application is initialized first on the server and then again on the client, we don't want the API requests performed during the server render to be duplicated by the client's initialization.
- Data fetched during the initial server render is lost after the server finishes processing the request, meaning that our store will be empty on the client unless we *hydrate* it from the copy we created during the server render.
- Not all of our client-side components and libraries can be initialized or rendered on the server; for example, any code that relies on the `window` or `document` APIs must be ignored during the initial server render.

In essence, the way we initialize the application is very different depending on whether we are rendering on the client or the server, and we have to provide very strict instructions as to how we want the application to boot on each.

The easy way – Nuxt.js

Nuxt.js is a framework built on top of **Vue.js**, aimed specifically at the use case of building universal or server-rendered web applications. It makes it incredibly easy too, as it does a fantastic job of abstracting away the infrastructure code necessary to render on both the client and server. It leaves us to focus purely on building our application UI, and even provides a number of additional features that really help us do that. Things such as, but not limited to, additional component hooks for fetching asynchronous data, middleware, and shared component layouts are all included on top of the default functionality we get with plain **Vue.js**.

However, there is one downside, and unfortunately for us it's a major one. **Nuxt.js** has a dependency on being hosted directly within a **Node.js** server. It can't be directly linked with an **ASP.NET Core** web application as we've done with our client-side application so far, even through the use of the **JavaScriptServices** middleware. That being said, there is a valid argument that large applications should have their frontends and backends developed and deployed completely independently of one another anyway. I've certainly built applications where I hosted a **Nuxt.js** frontend application in a **Docker** container running a **Node.js** server on **Linux**, as well as a completely separate **ASP.NET Core** web API hosted in a **Docker** container running either **Linux** or **Windows**.

This client/server separation has a number of pros and cons, but the most prominent of each are probably flexibility and complexity, respectively. We increase the complexity of our hosting environment and deployment pipeline, but gain the benefit of increased flexibility in how we can scale the frontend and backend independently, based on the specific needs of the application. On top of this, if you have separate frontend and backend development teams, then splitting your application like this can also be a benefit to you by helping those teams work completely independently from one another—if that's how you prefer to work, of course.

I would strongly recommend looking into **Nuxt.js** and whether splitting the frontend and backend of your application could work for you or your organization, especially if you'll be implementing **SSR** at any point. However, we won't be going down this route, and instead we'll be looking at what's involved with implementing **SSR** ourselves. I also think it's important to understand how **SSR** works under the hood, and just how much effort is involved with implementing it. This way, you fully appreciate what you're gaining by using a framework such as **Nuxt.js** for future projects!

Preparing the application for SSR

Aside from the additional npm packages we need, we also need to make some fundamental changes to the way we're fetching the data for each page, as well as the way we're storing the user's access token once authenticated. SSR is made far easier if we make use of Vuex for all of our app's data requirements. We're already using Vuex for things like shopping cart data and authentication state, but we've not used it for our product catalog, user account, or admin panel sections. We'll need to refactor these areas to make use of Vuex before we can even think about enabling SSR for the entire application.

The reason Vuex makes things easier is that even though the application will originally be rendered on the server, there is still the process of initializing the app on the client side as well. After the initial server render, the client needs to take control of any future page renders until the point that we hit the server again. To do so, it needs to be made aware of any application state that was handled by the server and *hydrate* the client-side state to match. This prevents the client side needing to duplicate the API calls that already took place on the server. As our application state is centralized in one place with Vuex, this client-side hydration is made far easier than if we leave state scattered among the different pages of our application.

Aside from the use of Vuex, we also need to refactor the way we store the user's authentication state. Currently, we persist the entire Vuex store to the browser's *local storage*, including the `auth` object containing their tokens. However, for the password-protected sections of our app, in order to enable SSR, we'll need access to these tokens for making the API calls necessary to pre-fetch the data on the server. While the app is rendering on the server, it has no access to the browser's local storage, meaning that it will proceed as if the user is unauthenticated and simply kick them back out to the login modal on the home page. To resolve this, we'll need to persist the access tokens into a storage mechanism that is available on both the client and server—and cookies are the perfect solution.

Installing npm packages required for SSR

There are quite a few additional npm packages to install, so open up your Terminal at the root of your project and run the following commands:

```
yarn add aspNet-prerendering
yarn add vue-server-renderer
yarn add webpack-merge
yarn add vuex-router-sync
yarn add vue-no-ssr
yarn add js-cookie
```

Alternatively, you could install all of these in one command like so:

```
yarn add aspnet-prerendering vue-server-renderer webpack-merge  
vuex-router-sync vue-no-ssr js-cookie
```

The first three of these packages are all required whenever we wish to render a Vue.js application on the server with ASP.NET Core. The `aspnet-prerendering` package is one that the ASP.NET Core `Microsoft.AspNetCore.SpaServices` NuGet package needs to perform SSR on any client-side, framework-based project, including Vue, as well as both React and Angular. The `vue-server-renderer` package is specific to Vue.js, and is responsible for actually generating the HTML of our application on the server. Finally, `webpack-merge` is used to literally merge multiple webpack configurations into one. This makes it easier for us to define different configurations for each environment, client, and server, and then merge them into one.

The other three packages defined here are included as part of the refactoring we're going to need to do, and are there to make our lives much easier in the process. The `vuex-router-sync` package, once configured, will automatically persist the values of the currently active `Vue-Router` route into our Vuex store. We'll need this when we move the product catalog data into the store, as the API calls we make require parameters from the query string. The `vue-no-ssr` package is a simple helper component that prevents certain portions of our app from attempting to render on the server. Some custom UI components, which rely on the `window` or `document` browser-only objects, will cause errors if we try and render them on the server. As such, we'll simply prevent them from rendering until we reach the client. Finally, the `js-cookie` package gives us a super simple API for getting and setting browser cookie values—perfect for when we move the authentication tokens out of local storage.

Adding Vuex actions and mutations for all API requests

We currently have a number of pages/components that fetch their own data directly using Axios. As previously discussed, we need to refactor these to use our Vuex store instead.

Defining additional Vuex actions

To start with, we need a set of new Vuex actions, which all belong in the `ClientApp/store/actions.js` file. The first of these is responsible for retrieving the list of products from our API:

```
export const fetchProducts = ({ commit }, query) => {
  return axios.get("/api/products", { params: query }).then(response => {
    {
      commit("setProducts", response.data);
    }
  });
};
```

We use `axios` to perform a HTTP GET request to the `/api/products` endpoint. This action will be used by both the product catalog page and the admin panel. In the case of the catalog, we need to pass the `query` parameters to the API to perform our server-side filtering of results. To do so, we've defined a `query` parameter for this action, which we assign to the `params` object in the `axios` call. Once the API call resolves, we simply commit the `setProducts` mutation.

The next action we need is the `fetchFilters` action, which looks like this:

```
export const fetchFilters = ({ commit }) => {
  return axios.get("/api/filters").then(response => {
    {
      commit("setFilters", response.data);
    }
  });
};
```

This is virtually identical to the previous one, save for the change of API endpoint and mutation that we commit. As such, we won't discuss it again and move straight on to the next `fetchProduct` action:

```
export const fetchProduct = ({ commit }, slug) => {
  return axios.get(`/api/products/${slug}`).then(response => {
    {
      commit("setProduct", response.data);
    }
  });
};
```

Again, this is almost identical, but the one thing to notice is the `slug` parameter, which we expect so that we can identify a single product in the subsequent API call. The final action we need is the `fetchOrders` action:

```
export const fetchOrders = ({ commit }) => {
  return axios.get("/api/orders").then(response => {
    commit("setOrders", response.data);
  });
};
```

Defining the additional Vuex mutations

The next thing we'll need is a matching set of mutations to actually set the store data when the actions we've just defined are invoked. These belong in the `ClientApp/store/mutations.js` file, and look like this:

```
export const setProducts = (state, products) => {
  state.products = products;
};

export const setFilters = (state, filters) => {
  state.filters = filters;
};

export const setProduct = (state, product) => {
  state.product = product;
};

export const setOrders = (state, orders) => {
  state.orders = orders;
};
```

These are incredibly simple and shouldn't need any explanation at this point, as we simply take the associated parameter and assign it to the store's `state` object.

Defining the additional store state properties

For the mutations we've just defined to work, we'll need a number of additional properties on our Vuex `state` object. Actually, to be completely accurate, we technically don't need to add these properties, but best practices recommend that we do in order to be explicit in what we expect our store to keep track of. Open up the `ClientApp/store/index.js` file and add the following properties:

```
const store = new Vuex.Store({
  strict: true,
  actions,
  mutations,
  getters,
  state: {
    auth: null,
    showAuthModal: false,
    loading: false,
    cart: [],
    products: [],
    filters: [],
    product: null,
    orders: []
  }
});
```

Updating existing pages to use Vuex

Now that we have a set of Vuex actions and mutations for retrieving and storing all of our page-level data, we need to update our components to use them. We are currently making use of the `beforeRouteEnter` hook to trigger the API requests necessary to fetch the data for each page. However, if you read the official documentation on SSR with Vue (<https://ssr.vuejs.org/>), the recommended approach is to define a custom `asyncData` function on any component that has data requirements which must be fulfilled before it can be fully rendered. That way, we can configure the initial server render of the application to wait for all `asyncData` functions to complete before completing and returning the resulting HTML to the browser. We can also modify the client-side routing logic of the application to look for these `asyncData` functions and wait for them to complete before allowing the page to change, in much the same way as the `beforeRouterEnter` hook works currently.

The biggest change we need to make is in the `ClientApp/pages/Catalogue.vue` page, so that's where we'll start.

Refactoring the catalog page

In the `script` section, we first need to delete the `data` function and `methods` object entirely. The `products` and `filters` that we used to store locally in this component have now been moved to the store. Next, in the `computed` object, we also need to delete the `sorted` computed property. We now need to make a change to the existing `sortedProducts` computed property, as well as add a new `filters` computed property. These changes look like this:

```
computed: {
  sortedProducts() {
    return this.$store.getters.sortedProducts;
  },
  filters() {
    return this.$store.state.filters;
  }
}
```

Now that the `products` array is in the store, we can also move the sorting logic there as well, which makes this component far simpler. We don't need to do any processing on the `filters` property, so we simply extract the raw value straight from the `state` object. The `sortedProducts` getter is not yet defined, but belongs in the `ClientApp/store/getters.js` file and looks like this:

```
export const sortedProducts = state => {
  switch (state.route.query.sort || 0) {
    case 1:
      return state.products.slice().sort((a, b) => {
        return b.price > a.price;
      });
      break;
    case 2:
      return state.products.slice().sort((a, b) => {
        return a.name > b.name;
      });
      break;
    case 3:
      return state.products.slice().sort((a, b) => {
        return b.name > a.name;
      });
      break;
    default:
      return state.products.slice().sort((a, b) => {
        return a.price > b.price;
      });
  }
}
```

```
};
```

This is largely a case of cutting and pasting the old logic straight from the component, but we have had to change things ever so slightly. For starters, we now decide which sorting rule to use by evaluating the `state.route.query.sort` value if it exists, or defaulting to zero if it doesn't. Next, in each of the switch cases, we also call `.slice()` on the `products` array to ensure that our sorting is performed on a new copy of the array rather than the original. If we omit this, then we'd be greeted by a host of console warning messages stating that we shouldn't be directly mutating the state outside of our mutations. Aside from these changes, the rest of the logic is exactly the same as before, so no more needs to be said at this point.

Back in the catalog page component, we now need to update our data fetching logic to trigger the new Vuex actions we created earlier, rather than perform the API calls directly within the component as we're doing currently. The `beforeRouteEnter` hook that we currently have can be deleted entirely, and then replaced with the following custom component method:

```
asyncData({ store, route }) {
  return Promise.all([
    store.dispatch("fetchProducts", route.query),
    store.dispatch("fetchFilters")
  ]);
}
```

Note that this belongs at the same level as the hook it replaces, and does not belong inside the `methods` object. As previously discussed, this function is not a standard function that Vue knows what to do with, but we'll see how to invoke it shortly. At this point, it is worth noting that we *must* return a promise from this function if we want to wait for it to complete before allowing the page to change or the initial server render to complete. In this case, we use the `Promise.all` function, which accepts an array of functions and returns a single promise to represent them all. Here, we dispatch both the `fetchProducts` and `fetchFilters` actions that we created earlier, passing the current `route.query` object to the former. Finally, the `beforeRouteUpdate` hook also needs to be updated to dispatch one of these actions as well:

```
beforeRouteUpdate(to, from, next) {
  this.$store.dispatch("fetchProducts", to.query).then(() => {
    next();
  });
}
```


We simply dispatch the `fetchProducts` action again, wait for it to resolve, then allow the router to finish navigation by invoking the `next` callback as we did before. Now that we've finished these changes to the catalog page, we can quickly perform the same refactoring process on the other pages as well. However, these changes will be very much the same as we've just done, so we'll skim over them pretty quickly.

Refactoring the product details page

Open up the `ClientApp/pages/Product.vue` file, locate the `script` section, then remove the `data` function and `methods` object entirely. In their place, add the following computed object:

```
computed: {
  product() {
    return this.$store.state.product;
  }
}
```

Next, replace the `beforeRouteEnter` hook with the following `asyncData` function:

```
asyncData({ store, route }) {
  return store.dispatch("fetchProduct", route.params.slug);
}
```

Refactoring the account page

Open up the `ClientApp/pages/Account.vue` file, locate the `script` section, then remove the `data` function and `methods` object entirely. In their place, add the following computed object:

```
computed: {
  orders() {
    return this.$store.state.orders;
  }
}
```

Next, replace the `beforeRouteEnter` hook with the following `asyncData` function:

```
asyncData({ store }) {
  return store.dispatch("fetchOrders");
}
```

Refactoring the orders admin page

Open up the `ClientApp/pages/admin/Orders.vue` file, locate the `script` section, then remove the `data` function and `methods` object entirely. In their place, add the following computed object:

```
computed: {
  orders() {
    return this.$store.state.orders;
  }
}
```

Next, replace the `beforeRouteEnter` hook with the following `asyncData` function:

```
asyncData({ store }) {
  return store.dispatch("fetchOrders");
}
```

Refactoring the products admin page

Open up the `ClientApp/pages/admin/Products.vue` file, locate the `script` section, then remove the `data` function and `methods` object entirely. In their place, add the following computed object:

```
computed: {
  products() {
    return this.$store.state.products;
  }
}
```

Next, replace the `beforeRouteEnter` hook with the following `asyncData` function:

```
asyncData({ store }) {
  return store.dispatch("fetchProducts");
}
```

Refactoring the create product admin page

Open up the `ClientApp/pages/admin/CreateProduct.vue` file, locate the `script` section, then remove the following properties from the object returned from the `data` function:

```
brands: []
os: []
features: []
colours: []
storage: []
```

Note that this time we cannot delete the `data` function entirely, as there are other properties that we still need in this component. We can't delete the `methods` object entirely either, but we can delete just the `setData` function as it's no longer required. Next, add the following computed object:

```
computed: {
  brands() {
    return this.$store.state.filters.brands;
  },
  os() {
    return this.$store.state.filters.os;
  },
  features() {
    return this.$store.state.filters.features;
  },
  colours() {
    return this.$store.state.filters.colours;
  },
  storage() {
    return this.$store.state.filters.storage;
  }
}
```

Finally, replace the `beforeRouteEnter` hook with the following `asyncData` function:

```
asyncData({ store }) {
  return store.dispatch("fetchFilters");
}
```

This completes the changes we need to make in the way we fetch our page-level data.

Changing the way we persist user authentication state

As previously discussed, when we enable SSR, any initial API calls made during a page load will be triggered from the server, rather than the client as they are now. We're currently storing user access tokens in the browser's local storage, which isn't available during a server render, so we need to move them to a cookie instead. On top of this, we've actually kept things simple by keeping the full state object synced in local storage, rather than just the small bits we need. However, now that we've moved all of our app's data requirements into the global store, we don't want to keep persisting it all. If we left things as they are now, the entire store would be persisted to local storage and then copied back into the store once the application initializes on the client. This means that any store data we fetch on the server will be overwritten by the copy persisted in local storage, which is not at all what we want to happen! As such, we're only going to persist the shopping cart data into local storage, and the authentication-related data into a cookie. The rest will be stored in-memory only.

Changing our approach of persisting state to local storage

The first thing we need to do is alter the `store.subscribe` call that we make in the `ClientApp/store/index.js` file. It now needs to look like this:

```
store.subscribe((mutation, state) => {
  const cartMutations = [
    "addProductToCart",
    "updateProductQuantity",
    "removeProductFromCart",
    "setProductQuantity",
    "clearCartItems"
  ];

  if (cartMutations.indexOf(mutation.type) >= 0) {
    localStorage.setItem("cart", JSON.stringify(state.cart));
  }
});
```

Remember that this function will be invoked each time a mutation is committed to the store. We no longer wish to subscribe to every mutation as we did before, so here we start by defining an array of mutation names that we actually do care about listening for. This list contains the name of any mutation that manipulates the `state.cart` array in any way. We then check to see whether the mutation that is currently being handled is present in our list, and if it is, we persist the current value of the `cart` array into local storage.

Now that we've handled the persistence of cart items, we also need to change how we read this data when the app starts up. Open up the `ClientApp/boot.js` file and locate the following section:

```
const initialStore = localStorage.getItem("store");

if (initialStore) {
  store.commit("initialise", JSON.parse(initialStore));
  if (store.getters.isAuthenticated) {
    axios.defaults.headers.common["Authorization"] = `Bearer ${
      store.state.auth.access_token
   }`;
  }
}
```

This section needs to be replaced entirely with the following, much simpler, alternative:

```
const cartItems = localStorage.getItem("cart");
if (cartItems) {
  store.commit("setCartItems", JSON.parse(cartItems));
}
```

As we're now only storing the cart items, we changed the local storage key to `cart`, so this is now what we use when checking whether the user has already added anything to their cart in a previous browsing session. If they have, we commit the `setCartItems` mutation, which will ultimately replace the `initialise` mutation that we used to commit before. This does not yet exist, so open up the `ClientApp/store/mutations.js` file, delete the `initialise` mutation declaration, and replace it with the following:

```
export const setCartItems = (state, items) => {
  state.cart = items;
};
```

This takes care of persisting cart data between sessions, but user authentication state will now be lost each time the user refreshes a page or closes their browser.

Storing authentication state in cookies

There are a few ways we could go about doing this, one of which is to do what we've done with the cart data and subscribe to the mutations that manipulate the `auth` object in the `store` state. However, to demonstrate the other obvious choice, we're going to modify the Vuex actions for logging in/out instead. Open up the `ClientApp/store/actions.js` file, then right at the top, add the following line to import the `js-cookie` library that we installed earlier:

```
import Cookie from "js-cookie";
```

With this in place, we can scroll down a bit and look for the `login` action. We need to set a cookie if the `axios` request succeeds, or remove the cookie if it fails. The updated action looks like this:

```
export const login = ({ commit }, payload) => {
  return new Promise((resolve, reject) => {
    commit("loginRequest");
    axios
      .post("/api/token", payload)
      .then(response => {
        const auth = response.data;
        axios.defaults.headers.common["Authorization"] =
          `Bearer ${
            auth.access_token
          }`;
        commit("loginSuccess", auth);
        commit("hideAuthModal");
        Cookie.set("AUTH", JSON.stringify(auth));
        resolve(response);
      })
      .catch(error => {
        commit("loginError");
        delete axios.defaults.headers.common["Authorization"];
        Cookie.remove("AUTH");
        reject(error.response);
      });
  });
};
```

It's incredibly simple, thanks to the `js-cookie` library. Similarly, we'll need to modify the `logout` action to remove the cookie as well:

```
export const logout = ({ commit }) => {
  commit("logout");
  delete axios.defaults.headers.common["Authorization"];
```

```
    Cookie.remove("AUTH");  
  };
```

Setting up and configuring SSR

We've finished all of the refactoring that we needed to do in order to prepare the application to be rendered on the server. We now need to look at how we configure our application to be booted on both the client and the server, as we already determined that we need to be very specific about what happens when and where. This also includes enhancing our existing webpack configuration so that it too understands how our application needs to work on both the client and server.

Let's start with how we actually boot our application. Currently, we have a single `ClientApp/boot.js` file, which handles this task for us nicely as we only ever boot the application on the client. However, we now need to split this file into three separate parts: `app.js`, `client.js`, and `server.js`. As you've probably already guessed, `client.js` and `server.js` are used to instruct Vue how our application should be booted on the client and server, respectively. To save on duplication, we also specify `app.js`, which will contain any functionality that should be shared between both client and server files. This can be likened to `app.js` acting as a base class, and `client.js` and `server.js` acting as subclasses in C#.

Defining the shared boot logic

Start by creating a new `ClientApp/app.js` file, then adding the following `import` statements at the top to get things started:

```
import Vue from "vue";  
import VeeValidate from "vee-validate";  
import router from "./router";  
import store from "./store";  
import { currency, date } from "./filters";  
import { sync } from "vuex-router-sync";  
import Cookie from "js-cookie";  
import axios from "axios";  
import "./helpers/interceptors";  
import "./helpers/validation";  
import App from "./components/App.vue";
```

The vast majority of these can be copied and pasted directly from the old `ClientApp/boot.js` file, but just be sure not to miss the few additions that we've added here. In fact, the majority of all three of the new files we're creating at the minute can be copied and pasted from our old boot file.

Remember that this is the shared boot file, and as such anything we import here will be imported on both client and server renders. If any of the files or third-party libraries that we import here are not compatible with SSR, this can completely prevent the app from loading as it will fail to render on the server. As an example, if we were to import the `Velocity` library either here or in the `ClientApp/server.js` file, the app would fail to load due to the dependency `Velocity` has on the `window` and `document` APIs, which are not available on the server.

Next, we need to copy across a few lines to install the `VeeValidate` plugin, as well as register our two custom filters, `currency` and `date`:

```
Vue.use(VeeValidate);
Vue.filter("currency", currency);
Vue.filter("date", date);
```

Remember that everything we're adding to this file are pieces that we need when rendering on both the client and the server. If we didn't include these statements, our app would fail to render completely. The next thing we need is brand new in this file, as we're about to make use of the `vuex-router-sync` package that we installed earlier:

```
sync(store, router);
```

Under the hood, this method invocation is configuring a number of mutations, which will automatically add the `route` object to our store state, then keep it up to date whenever the current route changes. We've already made use of this `route` object in some of our refactoring of API requests into `Vuex` actions/mutations.

Until this point, we've managed to avoid the need for any kind of environment-specific configuration variables within the client-side Vue portion of our app. However, now that we're looking to enable SSR, `axios` will be unable to determine the base URL path of our API requests when rendering on the server. This isn't an issue when rendering on the client because it has access to the `window` object, which is enough to determine the URL that the app is currently running on. As such, from now on, we'll need to explicitly set the base URL of our `axios` requests. To do so, add the following statement just below the preceding `sync` method invocation:

```
axios.defaults.baseURL =
  process.env.NODE_ENV === "production"
    ? "https://phoneshop.azurewebsites.net"
    : "http://localhost:5000";
```

We interrogate the `NODE_ENV` environment variable to determine whether we're running in production or not. If we are, we set the base URL to our Azure environment URL—make sure to swap this out for your own URL—and if not, we default back to our localhost URL that we've been using while developing the app. With this configuration in place, we can make an API request to a path such as this:

```
/api/products
```

When we do so, Axios will automatically prefix this with our base URL, like this:

```
http://localhost:5000/api/products
```

This then means that when rendering on the server, our API calls will still succeed, as they have been doing in the browser so far. Next up, we can now check whether the current user is already authenticated or not by checking for the `AUTH` cookie we set in our Vuex actions earlier:

```
const auth = Cookie.get("AUTH");
if (auth) {
  store.commit("loginSuccess", JSON.parse(auth));
  axios.defaults.headers.common["Authorization"] = `Bearer ${
    store.state.auth.access_token
 }`;
}
```

Again, you won't yet find this code anywhere else so make sure to copy it down carefully. It's also important that we check the user's authentication state in this file, or we'll find that any conditional checks that we do in router hooks or component templates will fail to detect that the user has authenticated when the app is initially rendered. This will make more sense when we add the next part:

```
const app = new Vue({
  router,
  store,
  render: h => h(App)
});
```

As we check the authentication state *before* initializing our root `Vue` instance, we ensure that the store holds accurate data for the first render of the application. Finally, we can export the objects we'll need to import in the client- and server-specific boot files:

```
export { app, router, store };
```

Defining the client-specific boot logic

We can now build on our base `app.js` file by defining our client-specific boot file. Create a new `ClientApp/client.js` file, then add the following `import` statements at the top:

```
import Vue from "vue";
import { app, router, store } from "../app";
import BootstrapVue from "bootstrap-vue";
import VueToastr from "@deveodk/vue-toastr";
import "@deveodk/vue-toastr/dist/@deveodk/vue-toastr.css";
```

Notice how most of these statements are importing presentation-only components, such as CSS style sheets and libraries, for client-side-only features, such as toast notifications. The server will only be responsible for rendering the HTML markup of our application, not making it fully interactive for the user. Applying any CSS styles and fully initializing the application so it becomes interactable is still the job of the browser, which means we only need to import these files in our client-side boot file.

We can now install the `BootstrapVue` and `VueToastr` libraries as we've done before:

```
Vue.use(BootstrapVue);
Vue.use(VueToastr, {
  defaultPosition: "toast-top-right"
});
```

Hydrating the client-side store

Next, we have another new piece of functionality that you won't have seen before:

```
if (window.__INITIAL_STATE__) {  
  store.replaceState(__INITIAL_STATE__)  
}
```

As this file will not be run until we hit the user's browser, we have access to the `window` object, and here we check to see whether a property called `__INITIAL_STATE__` has been set on it. If it has, we call the `store.replaceState` function and pass it as an argument to completely replace our client-side store state with the value of this object. This process is known as **hydrating** the client-side store from the store we created while server rendering. Again, the server is only responsible for rendering static HTML, just like any typical server-side development framework such as ASP.NET MVC or Ruby on Rails. The application does not become fully interactive until all of our client-side JavaScript has been loaded and initialized. That being said, our Vuex actions are still run on the server in order to pre-fetch any data required to actually render the HTML; this means that we must have had an active Vuex store on the server. Rather than trigger those API requests again when we hit the server, instead we simply hydrate the client-side store with any data that we already fetched on the server.

Loading shopping cart data from local storage

The only data in our application that the server will never have any knowledge of is the shopping cart items we're currently storing in the browser's local storage. Remembering that this file is for client-specific boot logic, it's the ideal place to query for existing cart items and commit them to the store if we find any:

```
const cartItems = localStorage.getItem("cart");  
if (cartItems) {  
  store.commit("setCartItems", JSON.parse(cartItems));  
}
```

This is exactly the same as how we retrieved the cart items in the old boot file, so we won't go into any more detail than that, especially as the final section in this file is far more involved.

We now need to configure the client-side router to scan through the components that are about to be rendered on the next page, and look for any that define the `asyncData` function that we talked about earlier. If we find any, we need to wait for the promises they return to resolve before allowing the navigation to continue. This is far more complicated than when we were using the `beforeRouterEnter` hook previously, but it does allow us to reuse the same data fetching logic on both the client and server, seeing as the `beforeRouterEnter` hook is not available during server renders.

The official SSR documentation already has a recommended approach to achieve what we want, so I'll be following their direction with the following:

```
router.onReady(() => {
  router.beforeResolve((to, from, next) => {
    const matched = router.getMatchedComponents(to);
    const prevMatched = router.getMatchedComponents(from);

    let diffed = false;
    const activated = matched.filter((c, i) => {
      return diffed || (diffed = prevMatched[i] !== c);
    });

    if (!activated.length) {
      return next();
    }

    Promise.all(
      activated.map(c => {
        if (c.asyncData) {
          return c.asyncData({ store, route: to });
        }
      })
    )
      .then(() => {
        next();
      })
      .catch(next);
  });

  app.$mount("#app-root");
});
```

This is almost a direct copy of the methods suggested by the documentation, and it already does a fantastic job of explaining how this works, so I strongly suggest you check it out by going here: <https://ssr.vuejs.org/guide/data.html#client-data-fetching>.

Pre-fetching component data

Essentially, what we are doing here is implementing another `VueRouter` hook, which runs before the page transition has resolved and creates a list of components that we haven't already rendered:

```
const matched = router.getMatchedComponents(to);
const prevMatched = router.getMatchedComponents(from);

let diffed = false;
const activated = matched.filter((c, i) => {
  return diffed || (diffed = prevMatched[i] !== c);
});
```

If we don't find any components that we actually care about fetching data for, we simply return early:

```
if (!activated.length) {
  return next();
}
```

However, if we do have some components that we care about, we make use of the `Promise.all` function again to make sure that the hook doesn't finish running until all components' data has been loaded:

```
Promise.all(
  activated.map(c => {
    if (c.asyncData) {
      return c.asyncData({ store, route: to });
    }
  })
)
.then(() => {
  next();
})
.catch(next);
});
```

We loop over all of the components in the `activated` array using the standard JavaScript `Array.map` function, before checking whether each component has defined an `asyncData` method. If they have, we invoke it and pass in the `store` and `to` parameters. As a reminder, this is to fulfill the needs of the `asyncData` methods we defined on components such as this one:

```
asyncData({ store, route }) {
  return store.dispatch("fetchProduct", route.params.slug);
}
```

Finally, once all component data requirements have been fulfilled, we mount our app onto the DOM:

```
app.$mount("#app-root");
```

Remembering our promises

At this point, I need to stress how important it is to always remember the `return` statement when dealing with promises in JavaScript. The whole point of the pretty complicated code we've just talked about is to enable us to *wait* for our API calls to complete before moving on to the next page or completing a server render of the application. We do that using the following code that we just discussed:

```
Promise.all(
  activated.map(c => {
    if (c.asyncData) {
      return c.asyncData({ store, route: to });
    }
  })
)
  .then(() => {
    next();
  })
  .catch(next);
});
```

Again, we're using the `Promise.all` function to wait for all of the component `asyncData` methods that we find by using the `return` statement as we invoke them. If we omit the `return` statement, the `asyncData` function will still be invoked, but we simply won't wait for it to finish before moving on with the `next` callback.

This rule applies all the way down the chain if the logic you use within an `asyncData` function contains any further asynchronous function calls. Take the following, for example:

```
asyncData({ store, route }) {
  return store.dispatch("fetchProduct", route.params.slug);
}
```

The `store.dispatch` function is asynchronous, so again if we omitted the `return` statement here, the `asyncData` function would not wait for the dispatch call to complete before notifying its caller that it has completed its processing. This would yet again mean the page navigation would complete before the API call had the chance to return any data.

Defining the server-specific boot logic

With our shared and client-specific boot logic in place, the last piece of the puzzle is defining how our app boots on the server. Create another new `ClientApp/server.js` file, then add the following `import` statements at the top:

```
import { app, router, store } from "../app";
import axios from "axios";
```

Next, we need to export a default function, which expects a `context` parameter and returns a promise:

```
export default context => {
  return new Promise((resolve, reject) => {
    ...
  })
}
```

We'll see where the `context` parameter originates from in a moment, but for now you just need to know that it contains all of the information we'll need to render the app, and is somewhat similar to the `HttpContext` object in an ASP.NET request. It includes things like the URL being accessed and any cookies that were passed up as part of the original HTTP request.

Inside this function, we first need to check for the `AUTH` cookie:

```
const auth = context.cookies.find(c => c.key === "AUTH");
if (auth) {
  store.commit("loginSuccess", JSON.parse(auth.value));
  if (store.getters.isAuthenticated) {
    axios.defaults.headers.common["Authorization"] = `Bearer ${
      store.state.auth.access_token
    }`;
  }
}
```

As we aren't inside the browser at this point, the `js-cookie` library we used before will not work. Instead, we can access the request cookies using the `context.cookies` property, which contains an array of key/value pairs. More specifically, we'll use the `Array.find` function to look for a cookie where the key is equal to `AUTH`. If we find one, we'll do exactly the same as we've done before by committing the `loginSuccess` mutation, then setting the default authorization header for any future `axios` HTTP requests.

Next, we need to push the URL path being rendered into our `router` object:

```
router.push(context.url);
```

Finally, we need to pre-fetch the data needed by any of the components matched by that URL in a similar way to what we did in the client boot file. We do this by looking for `asyncData` functions again and waiting for them to resolve before allowing the render to complete. If we allow the render to complete before we finish fetching the required data, the HTML returned to the client would not contain anything that relies on that data. For fairly obvious reasons, this would potentially make SSR a completely pointless feature.

Again, the official SSR documentation already suggests the ideal way of doing this, so I'll be taking their lead and using the code they provide as a base:

```
router.onReady(() => {
  const matchedComponents = router.getMatchedComponents();
  if (!matchedComponents.length) {
    return reject(new Error({ code: 404 }));
  }
  Promise.all(
    matchedComponents.map(Component => {
      if (Component.asyncData) {
        return Component.asyncData({ store, route:
          router.currentRoute });
      }
    })
  )
  .then(() => {
    context.state = store.state;
    resolve(app);
  })
  .catch(reject);
}, reject);
```

This is a direct copy of the sample code for server-side data fetching, which you can find [here](https://ssr.vuejs.org/guide/data.html#server-data-fetching): <https://ssr.vuejs.org/guide/data.html#server-data-fetching>.

It is strongly recommended to have a read through this, as it already explains these concepts incredibly well. However, in summary, what we're doing here is very similar to what we did on the client. If any components are matched to the currently rendered URL, we loop over each one and look for an `asyncData` function. If we find any, we invoke them all under the umbrella of a single `Promise.all` call, ensuring that we don't finish the render until all data has been fetched. Once this singular promise is resolved, we set the `context.state` property to the current `state` value within the `store` object, then resolve the top-level promise, which wraps the whole `router.onReady` hook.

This really does now show why the `asyncData` approach is worthwhile. It serves both the purpose of waiting for server-side data fetching for SSR and client-side data fetching between page changes.

Deleting the old boot file

At this point, we've finished with our new boot file structure, and as such we can delete the original `ClientApp/boot.js` file entirely. This will then introduce a new problem, in that our current webpack configuration is looking for that specific file. Let's fix that now.

Making webpack aware of the client/server boot files

With our updated boot files ready to go, we need to modify our existing webpack configuration to make it aware of those new files. We're going to go through a similar process, albeit much shorter, to split the webpack configuration into a shared client/server-specific object.

Defining a shared webpack configuration object

Luckily for us, the necessary changes here are minimal. Our existing configuration object will, save for a minor tweak, become the shared configuration object that we'll later merge with the client- and server-specific objects. Open up the `webpack.config.js` file, then add the following additional property near the top of the file:

```
const path = require("path");
const webpack = require("webpack");
const ExtractTextPlugin = require("extract-text-webpack-plugin");
const bundleOutputDir = "./wwwroot/dist";
const UglifyJSPlugin = require("uglifyjs-webpack-plugin");
const merge = require("webpack-merge");
```

Next, around the line 10 mark, you should have a single `return` statement, which returns an array like this:

```
return [
  {
    stats: { modules: false },
    context: __dirname,
    resolve: { extensions: [".js"] },
```

```
    entry: { main: "./ClientApp/boot.js" },
    module: {
      ...
    }
    ...
  }
};
```

Rather than returning the array here, we're going to define a function that returns a `sharedConfig` object instead:

```
const sharedConfig = () => ({
  stats: { modules: false },
  context: __dirname,
  resolve: { extensions: [".js"] },
  module: {
    ...
  }
  ...
});
```

Also notice that we've removed the `entry` property. Now that we're rendering on both the client and server, we'll need to specify environment-specific `entry` properties instead. The only other change we need to make to this shared configuration object is to remove one of the plugins we've been using up until this point. If you look a little further down in this file, you'll see a `plugins` array like this:

```
plugins: [
  new webpack.DefinePlugin({
    "process.env": {
      NODE_ENV: JSON.stringify(isDevBuild ? "development" :
        "production")
    }
  })
  // new webpack.DllReferencePlugin({
  //   context: __dirname,
  //   manifest: require("./wwwroot/dist/vendor-
  //     manifest.json")
  // })
]
```

For some reason, when we leave this `DllReferencePlugin` in place, our application fails to render on the server. However, by removing it, all we are doing is disabling the webpack bundle-splitting feature, which separates the vendor bundle from our application bundle. The application will work perfectly without it, albeit with slightly slower reload times when running in development.

This seems to be an issue with webpack itself, or at least with `DllReferencePlugin`, and despite having several issues open on GitHub, I've so far failed to work out a resolution. As soon as I do, I'll ensure that the sample code is updated to reflect the change.

Defining client- and server-specific webpack configuration objects

With the shared configuration function in place, we can now use the `webpack-merge` library we imported earlier to create both client- and server-specific objects. Directly beneath the `sharedConfig` function, add the following:

```
const clientBundleConfig = merge(sharedConfig(), {
  entry: { main: "./ClientApp/client.js" },
  output: {
    path: path.join(__dirname, "wwwroot/dist")
  }
});

const serverBundleConfig = merge(sharedConfig(), {
  target: "node",
  entry: { "main-server": "./ClientApp/server.js" },
  output: {
    libraryTarget: "commonjs2",
    path: path.join(__dirname, "wwwroot/dist")
  }
});
```

Both of these objects are merged with the `shared` object using the `merge` function from `webpack-merge`. Also, in both cases, we needed to specify the `entry` property to instruct webpack where to start loading and bundling our files from, as well as the `output` property to determine where to place the bundle file after it's finished processing it. You'll notice that we point the entry properties at the `ClientApp/client.js` and `ClientApp/server.js` files we just created, and output them both to the `wwwroot/dist` folder. The only other thing we needed to add to the server-specific configuration was that we're targeting Node.js rather than the browser, and our `output.libraryTarget` property should be `commonjs2`.

With these in place, we simply need to return a new array containing both configuration objects:

```
return [clientBundleConfig, serverBundleConfig];
```

Updating the vendor webpack configuration to include SSR libraries

Before we're done with webpack configuration, we have one very small change to make in the `webpack.config.vendor.js` file. Open it up and add the following additional library to the `vendor` array:

```
vendor: [
  "event-source-polyfill",
  "axios",
  "vue",
  "vue-router",
  "vuex",
  "bootstrap/dist/css/bootstrap.min.css",
  "bootstrap-vue",
  "nprogress/nprogress.css",
  "aspnet-prerendering"
]
```

This is the npm module that ASP.NET Core uses under the hood to render SPAs on the server. As always, now that we've modified this file, we need to regenerate our vendor bundle using the following Terminal command:

```
yarn webpack
```

Enabling SSR

We finally have all of the configuration files in place that we'll need in order to render our app on the server. We now need a file that will be invoked directly from our MVC view and will be responsible for actually rendering the application for us. Create a new `ClientApp/renderOnServer.js` file, then start it off with the following:

```
process.env.VUE_ENV = "server";

const fs = require("fs");
const path = require("path");

const filePath = path.join(__dirname, "../wwwroot/dist/main-server.js");
const code = fs.readFileSync(filePath, "utf8");

const bundleRenderer = require("vue-server-renderer").createBundleRenderer(
  code
);
```

The important part here is how we physically read the `wwwroot/dist/main-server.js` file, then create a **bundle renderer** using the contents of it. The `main-server.js` file is the product of our server-side webpack configuration, bundling our application based on the contents of the `server.js` file we defined earlier. The bundle renderer we create from this file is ultimately what renders the application as a string so that we can render it into the DOM.

Next, we need the following:

```
var prerendering = require("aspnet-prerendering");
module.exports = prerendering.createServerRenderer(function(params) {
  return new Promise(function(resolve, reject) {
    ...
  });
});
```

Here, we make use of the `aspnet-prerendering` npm module to create a server renderer. This acts as the interface between the ASP.NET HTTP request and our server-side rendering functionality. The `params` argument is used to pass HTTP parameters from the original ASP.NET request down into the promise function within. Using these, we can create the `context` object that we expect within the `ClientApp/server.js` file:

```
const context = {
  url: params.url,
  absoluteUrl: params.absoluteUrl,
  baseUrl: params.baseUrl,
  data: params.data,
  domainTasks: params.domainTasks,
  location: params.location,
  origin: params.origin,
  cookies: params.data.cookies
};
```

Most examples you'll find on documentation sites do not include the original HTTP request cookies as we've done here. In our case, these are crucial for us to be able to use the user's access token to perform any API requests.

Finally, we use the bundle renderer we created earlier to render the app as a string:

```
bundleRenderer.renderToString(context, (err, _html) => {
  if (err) {
    reject(err.message);
  }
  resolve({
    globals: {
      html: _html,
    }
  });
});
```

```
        __INITIAL_STATE__: context.state
    }
  });
});
```

The `renderToString` function we're using here accepts two parameters: the `context` object we wish to pass to the server boot file and a callback function to invoke once the rendering has finished. This callback function has its own two parameters: an `err` property, which will be null if everything was OK, and the HTML string that's been generated. In this case, if we find that the `err` property has a value assigned to it, we know that the request failed, so we reject the promise being returned from the preceding `createServerRenderer` call. If there is no value in the `err` property, we resolve the promise and pass an object with the `_html` string that was just generated. We also take the `context` object, which has just been processed by the server, and assign its `state` value to an `__INITIAL_STATE__` property that will be used to hydrate the application once it reaches the client. Because we attached the `html` and `__INITIAL_STATE__` properties to the `globals` object, they will be automatically assigned to the `window` object once we hit the browser.

The last step is to make our MVC view aware of this file using an ASP.NET Core **taghelper**, which comes pre-installed with the `SpaServices` NuGet package. Open up the `Features/Home/Index.cshtml` view and modify the `app-root` component's `div` element as follows:

```
<div
  id="app-root"
  asp-prerender-module="ClientApp/renderOnServer"
  asp-prerender-data="new { cookies =
    ViewContext.HttpContext.Request.Cookies }">
</div>
```

We use the `asp-prerender-module` taghelper to point to the `renderOnServer.js` file we just created. As previously mentioned, we're completely reliant on the HTTP request cookies being sent to this file as well, which is why we also use the `asp-prerender-data` taghelper to pass in a `cookies` object, which we retrieve from the `ViewContext.HttpContext.Request.Cookies` parameter.

And with that, we have successfully set up and configured SSR in our application. However, due to some of the functionality we have in our components, if we try to run the app now, it would fail to render on the server. We'll fix that in the final section.

Conditionally rendering elements that rely on the browser

Some of our components are entirely reliant on being used in the browser due to their coupling with the `window`, `document`, or `localStorage` APIs. If we don't prevent these components from rendering on the server, we'll get all kinds of errors that aren't particularly easy to debug.

As an example, at the top of the `script` section in the `ClientApp/components/catalogue/FilterAccordion.vue` component, we are importing the `Velocity` library that we're using for animations:

```
import * as Velocity from "velocity-animate";
```

`Velocity` is one of those libraries that cannot run outside of the browser, and simply importing it like this is enough to cause our SSR to fail. This is a particularly tricky situation, as using the `import` syntax as we are means that any `import` lines have to be at the root level of the `script` tag. That is, we can't simply do this:

```
if (typeof window !== "undefined") {  
  import * as Velocity from "velocity-animate";  
}
```

Even if we could, we'd still get errors because the `Velocity` object wouldn't even exist on the server, and we expect it to exist further down in the same file. This is how we actually fix it:

```
let Velocity;  
if (typeof window !== "undefined") {  
  Velocity = require("velocity-animate");  
} else {  
  Velocity = function() {  
    return Promise().resolve(true);  
  };  
}
```

We start by declaring an empty `Velocity` variable, then determining whether we are rendering on the client based on whether the `window` object is `undefined` or not. If it isn't, we are on the client and can safely use the `require` syntax to load `Velocity` rather than the `import` syntax. If the `window` object *is* `undefined`, we still need to assign a value to the `Velocity` variable, so we use a simple function that always returns a promise which resolves. The real `Velocity` function also returns a promise, so we need to mimic that behavior.

Including the example we just looked at, there are a handful of different places that we'll need to make a similar change to. Unfortunately, none of these are exactly the same, so each needs a fairly specific fix to be applied.

Fixing the range filter component

In the `ClientApp/components/catalogue/RangeFilter.vue` file, we'll be using the third-party `vue-slider` component. This is another example of a component that isn't compatible with SSR, so we'll need to hide it while rendering on the server. In the `template` section, make the following changes:

```
<no-ssr>
  <vue-slider
    :value="value"
    :formatter="formatter"
    :min="min"
    :max="max"
    :interval="interval || 1"
    :lazy=true
    width="90%"
    @callback="filter">
  </vue-slider>
</no-ssr>
```

All we've done is wrap the `vue-slider` component with another third-party component that we installed earlier the `no-ssr` component. As the name implies, it prevents its contents from being rendered on the server. However, we can't stop here as we're still importing the `vue-slider` component in the `script` section. As with the preceding `Velocity` example, simply importing this component is enough to prevent the app rendering on the server.

At the top of the `script` section, make the following changes:

```
import noSsr from "vue-no-ssr";

let vueSlider = {};
if (typeof window !== "undefined") {
  vueSlider = require("vue-slider-component");
}

export default {
  name: "range-filter",
  components: {
    vueSlider,
  }
}
```



```
    noSsr
  },
  ...
}
```

We first need to import the `no-ssr` component, then do a similar browser check to the one we did in the `Velocity` example. We declare a `vueSlider` variable with the default value of an empty object, then override it with the actual `vueSlider` object by loading it using the `require` syntax, as long as we're in the browser (that is, the `window` object is not undefined). Finally, we add the `no-ssr` component to the list of child components.

Fixing the checkout form component

In the `ClientApp/components/checkout/CheckoutForm.vue` component, we use the `Stripe` library to initialize the credit card input field. `Stripe` cannot be used on the server, so we'll need to prevent this initialization from happening until we hit the browser.

In the `script` section, we simply need to move the initialization logic into the `mounted` life cycle hook like so:

```
let card = null;

export default {
  name: "checkout-form",
  mounted() {
    let stripe = Stripe(`pk_test_NNMExLrT99IPhWHmPdu3xuXo`),
        elements = stripe.elements(),
        style = {
          base: {
            lineHeight: "24px"
          }
        };

    card = elements.create("card", { style: style });
    card.mount(this.$refs.card);
  }
  ...
}
```

The `mounted` hook is not invoked at all during a server render, so this is enough to fix our issues in this component.

Fixing page transition animations in the router

In the `ClientApp/router/index.js` file, we configured a `beforeEach` hook to trigger a transition animation on every page change. To do so, we use the `NProgress` library, which is yet another one that isn't compatible with SSR. This time, the fix is incredibly simple, as we simply don't bother starting the animation if we're on the server:

```
router.beforeEach((to, from, next) => {
  if (typeof window !== "undefined") {
    NProgress.start();
  }
  ...
})
```

Fixing the store subscription to persist cart items to local storage

In the `ClientApp/store/index.js` file, we have a subscription that runs on every mutation and persists the user's shopping cart contents if the mutation manipulated them. However, during SSR, we have no access to local storage, so again we need to prevent this code from running if we are on the server:

```
if (typeof window !== "undefined") {
  store.subscribe((mutation, state) => {
    const cartMutations = [
      "addProductToCart",
      "updateProductQuantity",
      "removeProductFromCart",
      "setProductQuantity",
      "clearCartItems"
    ];

    if (cartMutations.indexOf(mutation.type) >= 0) {
      localStorage.setItem("cart", JSON.stringify(state.cart));
    }
  });
}
```


Summary

SSR is a complicated feature to properly utilize, and seeing as there are definite drawbacks to doing so, it should always be evaluated to determine whether it's really necessary. We started out by looking at a few of the reasons to implement SSR, and hopefully after finishing this chapter, you'll also understand some of the drawbacks. As we've seen towards the end, there are a lot of hoops to jump through depending on what kind of functionality your app has and how many third-party libraries you use. As such, implementing SSR should never be a decision that's taken lightly.

We then started preparing our application for SSR by refactoring some of our existing implementation details to make our lives easier when rendering on the server. We then started adding the additional configuration files we needed, such as client- and server-specific boot files and webpack configurations. This then enabled us to tie things together by looking at how to actually trigger a server render using ASP.NET Core taghelpers and a custom JavaScript module, which is invoked by the `SpaServices` middleware.

Finally, we fixed a number of issues that resulted from rendering our application on the server. Specifically, we needed to conditionally render certain parts of the UI, depending on whether we were in the browser or not, due to tight coupling to browser-based APIs such as `window`, `document`, and `localStorage`.

The next chapter is the final one, where we'll be looking at building on our existing deployment pipeline and adding continuous integration and deployment functionality using **Visual Studio Team Services (VSTS)**.

13

Continuous Integration and Continuous Deployment

In the previous chapter, we implemented one of the more complicated features of building a modern SPA frontend: server-side rendering. We had to make quite a lot of changes to the application to prepare it for SSR, as well as fix a number of bugs that cropped up when attempting to render browser-reliant components on the server. However, the application itself is now as feature-complete as it's going to be for the purposes of this book.

In this final chapter, we're going to improve our current deployment mechanism by implementing a **Continuous Integration (CI)** and **Continuous Deployment (CD)** pipeline using **Visual Studio Team Services (VSTS)**. VSTS isn't the only option available to us, as there are many different ways that we can build a CI/CD pipeline, but as we're hosting within Azure, it makes a lot of sense to stick with a Microsoft ecosystem due to how seamlessly they work together.

In summary, in this chapter, we're going to cover the following topics:

- CI/CD: Why bother?
- Disabling Azure app service Git deployments
- Getting started with VSTS
- Setting up a CI build
- Setting up a CD release

CI/CD – why bother?

Before we go much further, we need to be clear that CI and CD are entirely separate concepts, and it is perfectly acceptable to do CI without doing CD. However, it is not possible to do CD without CI! So, aside from giving us more control than our current approach, why should we bother?

Continuous integration

We may not have any automated tests in our application currently, but they are a fundamental part of ensuring success in most production-grade applications. CI not only gives us peace of mind that the application still *builds* after every code push, but also that it still *works*. Automated tests are always run as part of the CI process, giving us immediate feedback if something breaks after we push any changes to source control.

We also ensure that the application builds on a completely separate environment to that of the developers who built it. *It works on my machine* is a common phrase in many dev teams, which becomes completely irrelevant if something goes wrong on a properly configured and reliable build server.

Continuous deployment

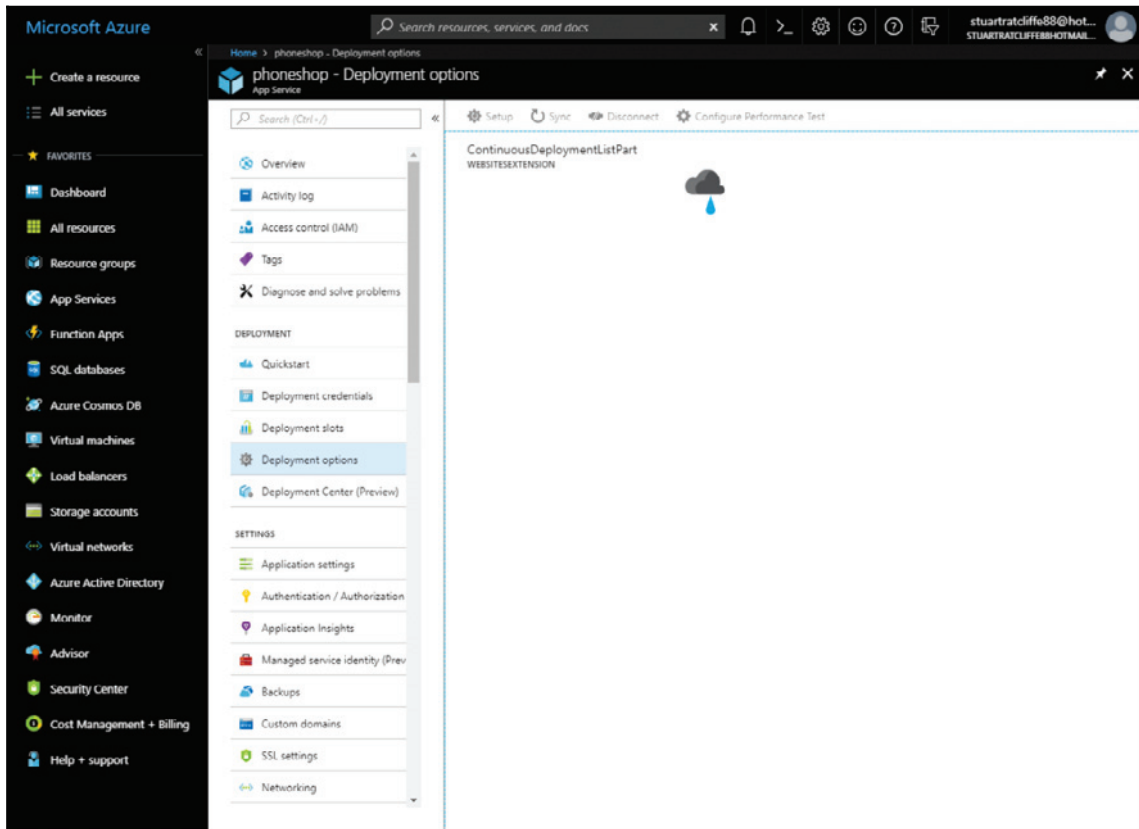
CD is one way of removing human error from the list of possible problems that can occur when deploying an application to production. A well-tested deployment pipeline is far more reliable than a human, who may forget to perform a crucial step of the process.

We often deploy far more frequently when using a CI/CD pipeline as well, meaning new versions are much smaller, and as such there is less chance of bugs creeping into the live environment. However, if the worst does happen and we need to roll back to a previous release, the smaller size benefits us here as well, as there is less to do to get back to the last known working version.

Disabling Azure app service Git deployments

Back in [Chapter 10, Deployment](#), we configured the application to deploy on every push to the master branch using the built-in Azure app service Git deployment feature. However, using this method, we get very limited control over the deployment process, and it is not uncommon to get some pretty strange errors preventing the application from deploying until you simply retry at a later time. As we'll be moving the responsibility for deploying the application over to VSTS, we need to disable the Azure Git deployment feature.

Log in to the Azure portal, then navigate to your web app service that we created earlier. In the secondary navigation menu on the left, head to the **Deployment options** page, then hit the **Disconnect** button at the top of the page that loads. You may need to wait a couple of minutes for the notification to confirm that the deployment has been disconnected, but once this is done, if you refresh the page, it should look like this:



Disabling Azure Git deployments

Getting started with VSTS

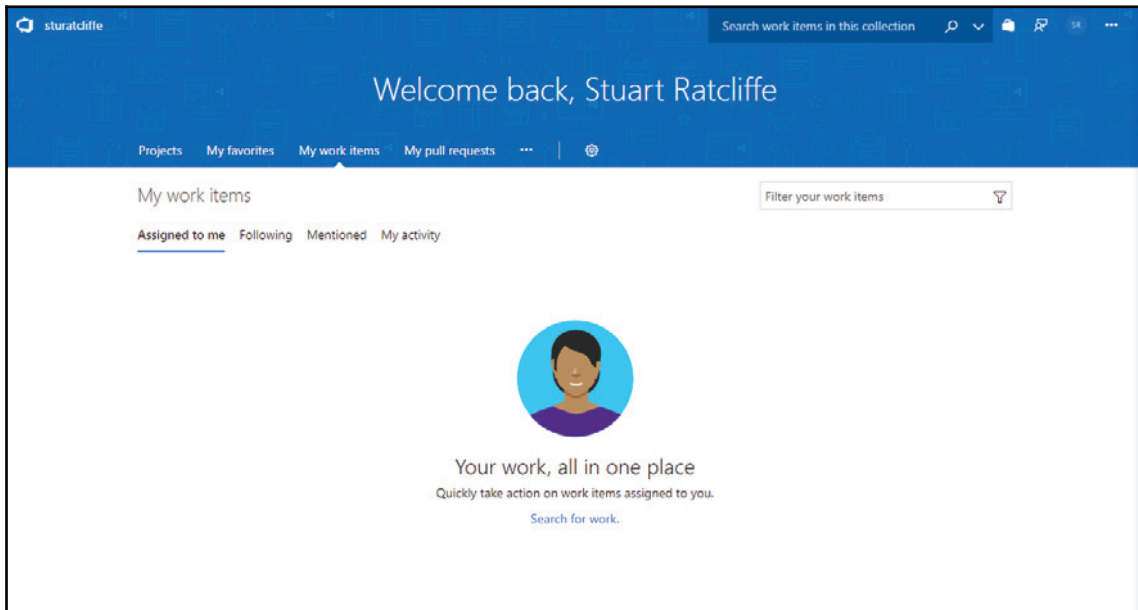
If you've already got a VSTS account and have configured a project to use for source control purposes, feel free to skip this section and move on to the next one, where we'll be configuring the CI/CD pipeline. If not, continue on to see how you can get started with a new VSTS account and project.

Creating a VSTS account

First things first, you'll need to sign up for a VSTS account if you don't already have one. Head over to the following URL in your browser, then hit the **Get started for free** button to begin the account registration process: <https://www.visualstudio.com/team-services/>.

Once you finish the registration process, you'll be able to access your account dashboard using a custom URL based on the username you provided when you signed up. For example, mine is the following: <https://sturatcliffe.visualstudio.com/>.

And when you first visit the page, you should see something like this:

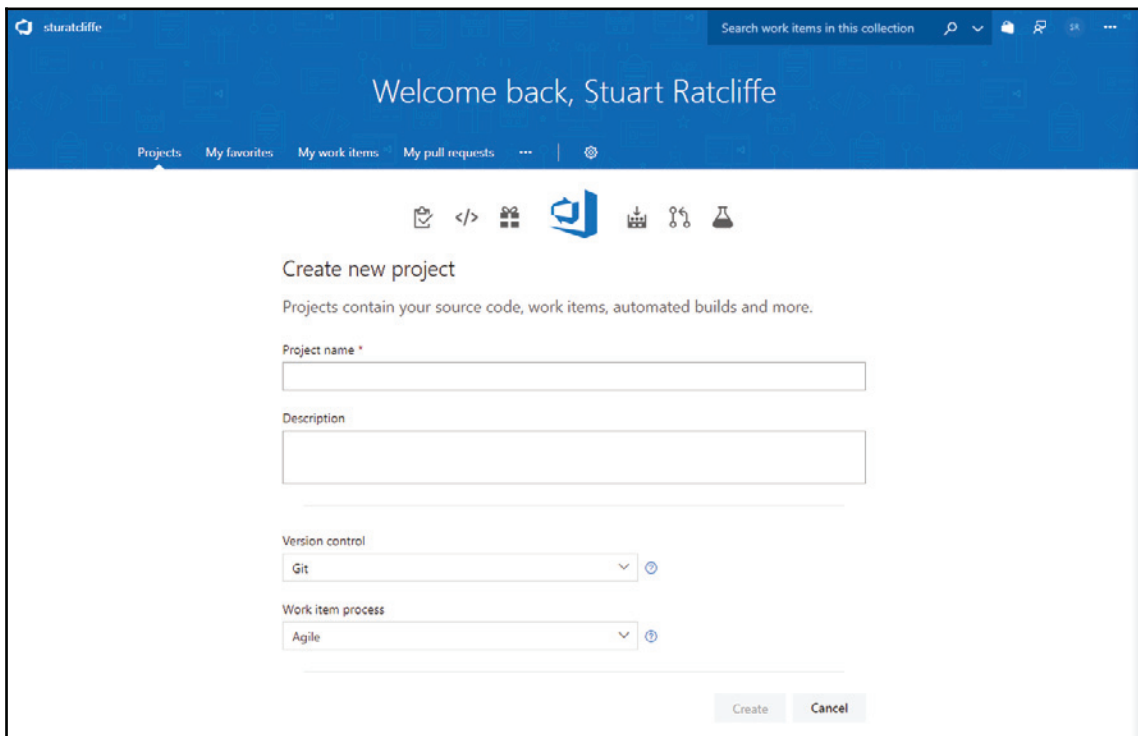


Setting up a team services project

With your account all set up, the next step is to create a project. A project in VSTS can be used to manage your application's life cycle all the way from planning through to deployment. You can manage backlogs of work items and bugs using a rich Kanban board-style UI, house your application source code using Git repositories, and create CI/CD pipelines using the concepts of builds and releases.

When it comes to the latter, which after all is what we are most interested in for the purposes of this chapter, we aren't restricted to using VSTS to house our source code. My code is currently stored in a Bitbucket account, but VSTS can use OAuth to connect to many different cloud-based Git repositories to fetch your code and build/release it. However, if you're looking for a place to store your code in private repositories, VSTS is a great option due to the fact that you get unlimited private repositories for up to five users—far more generous than many other providers!

After creating your account and landing on the screen displayed earlier, you should see a **Projects** menu item, which will take you to a page that looks like this:



stuartcliffe Search work items in this collection

Welcome back, Stuart Ratcliffe

Projects My favorites My work items My pull requests

Create new project

Projects contain your source code, work items, automated builds and more.

Project name *

Description

Version control
Git

Work item process
Agile

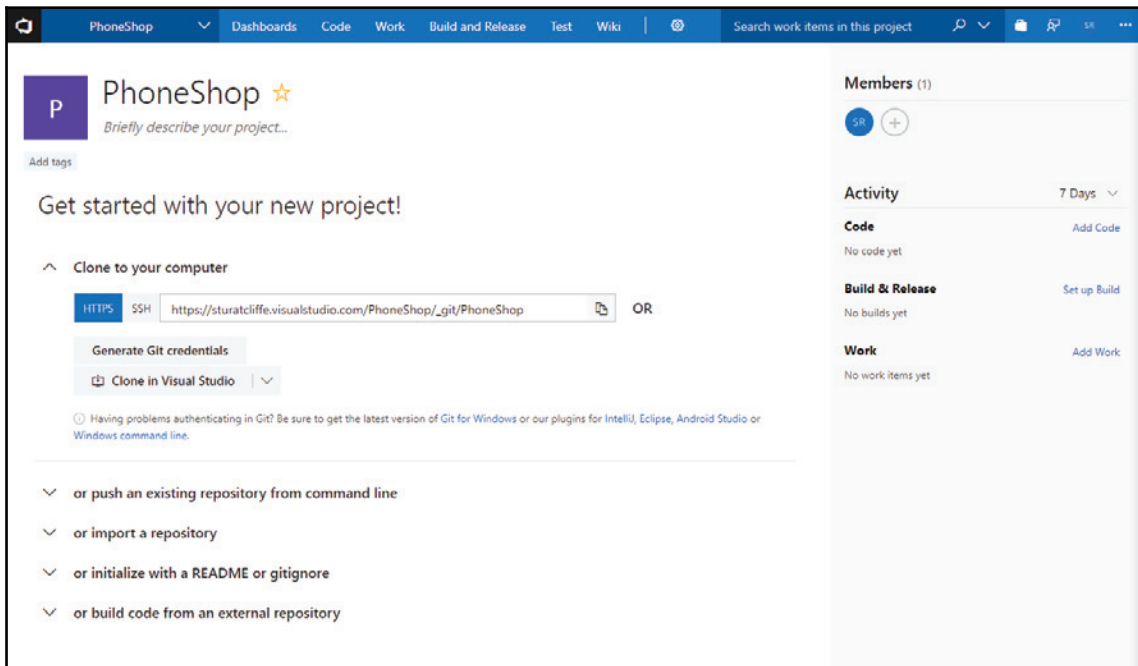
Create Cancel



Note that if you are using an existing VSTS account and already have a project created, you'll see your list of projects instead. There should be a button to create a new project if this is the case.

We can use this form to create an appropriately named project for our application; in my case, I chose to name it `PhoneShop`. You can enter whatever you like for the description, but as it isn't mandatory, I left mine blank. If you followed along in *Chapter 10, Deployment*, it is safe to assume that you already have your code stored in some kind of Git repository. Therefore, leaving the **Version control** option as **Git** should be fairly self-explanatory. We won't be discussing work item processing, so you can also leave that as the default **Agile** option.

After completing this form, you should be greeted with the default project dashboard screen, which looks something like this:



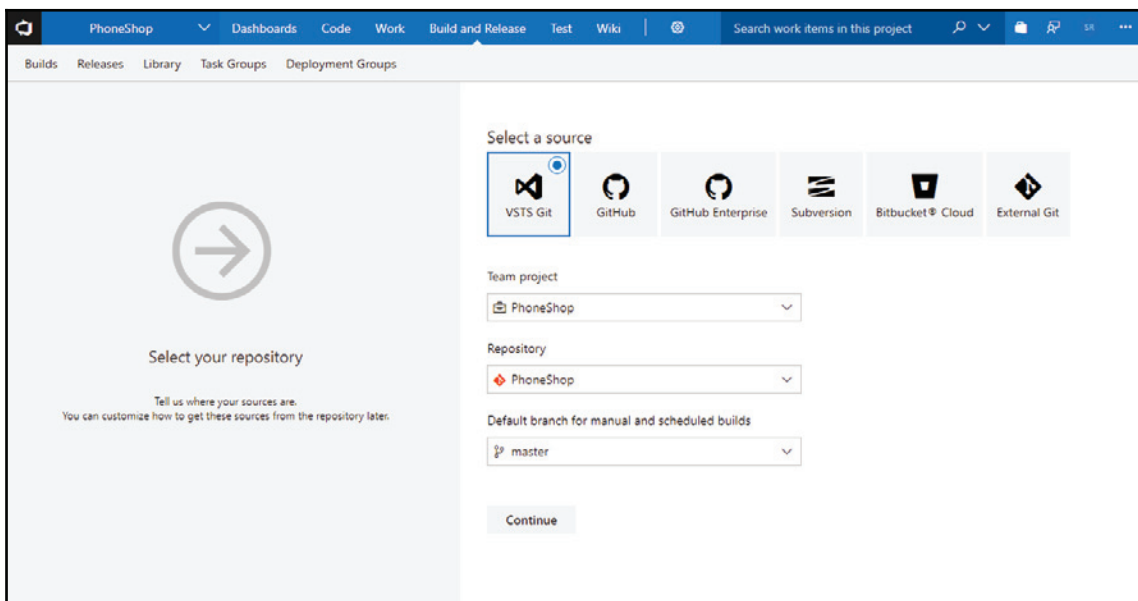
From here, you can manage everything there is to manage about your new project, but for now all we are interested in is getting started with our CI/CD pipeline.

Building a CI/CD pipeline

With our VSTS project set up and ready to go, we can finally start building our CI/CD pipeline, starting with a VSTS build.

Setting up a VSTS build

From the preceding project dashboard page, on the right-hand side, there is a **Set up Build** link next to the **Build and Release** heading. Clicking on this link will take you to a page that looks like this:

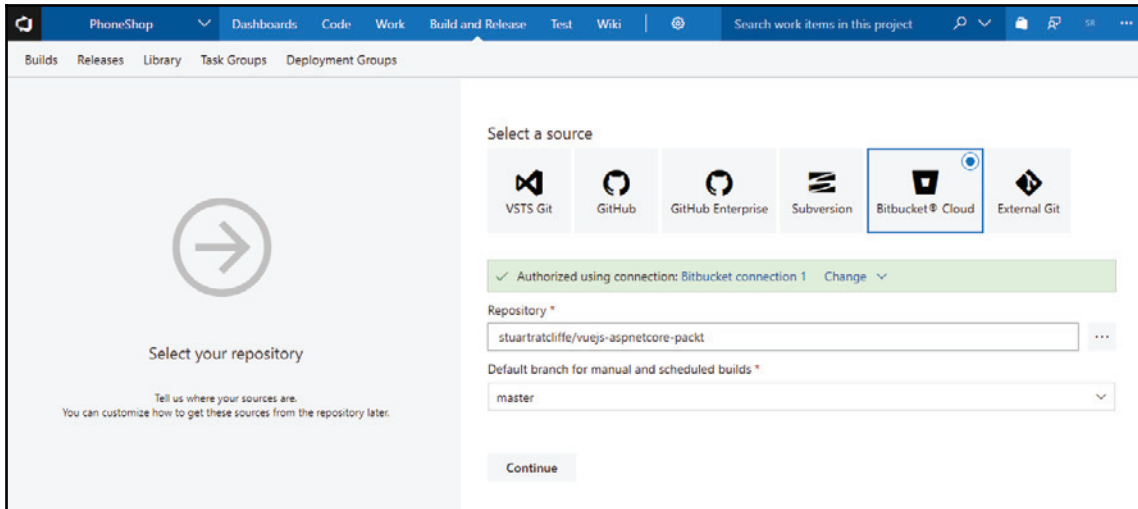


If you're using VSTS for source control, then selecting your code repository is as simple as specifying the team project, repository, and default branch you wish to build from. In this case, a default **PhoneShop** repository and **master** branch was created for me when I set up the team project earlier. However, in my case, I need to select the **Bitbucket Cloud** option, followed by the **Authorize using OAuth** button to connect to my Bitbucket account. At this point, you'll need to select your own source control provider, then follow the instructions to allow VSTS to connect to it so that it can access your source code.

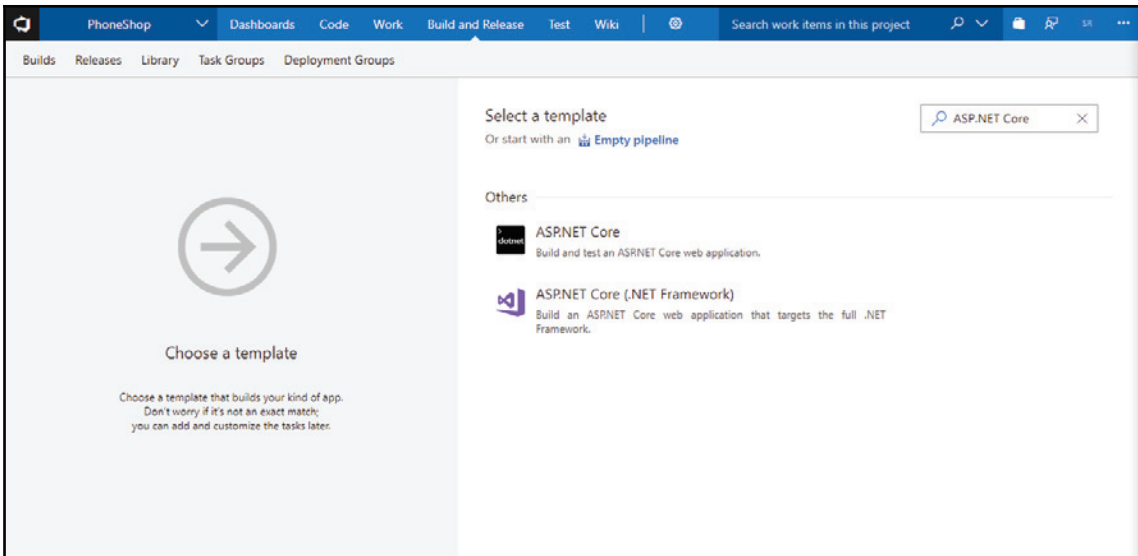


Make sure your browser has popup blocking disabled for the VSTS website. When connecting to other source control providers using OAuth, VSTS will try and open the login pages in a new window.

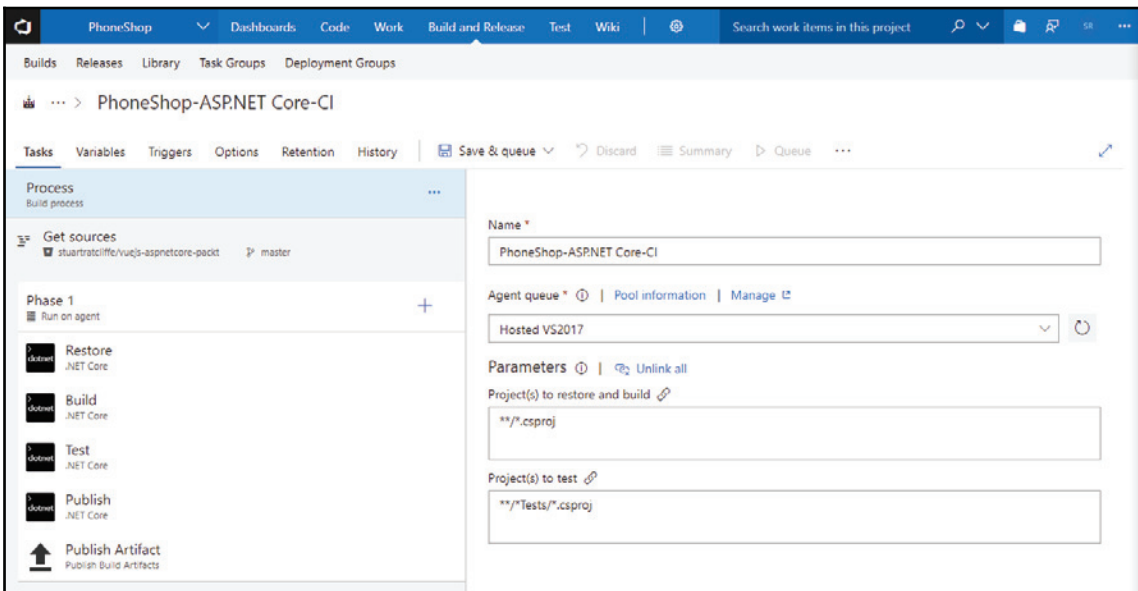
After authorizing access to your source control provider, you can then select the Git repository and branch you wish to build:



Upon clicking the **Continue** button, you'll be taken to a screen to select a build template. You can either scroll down the list to look for the **ASP.NET Core** option, or alternatively search for it as I've done here:



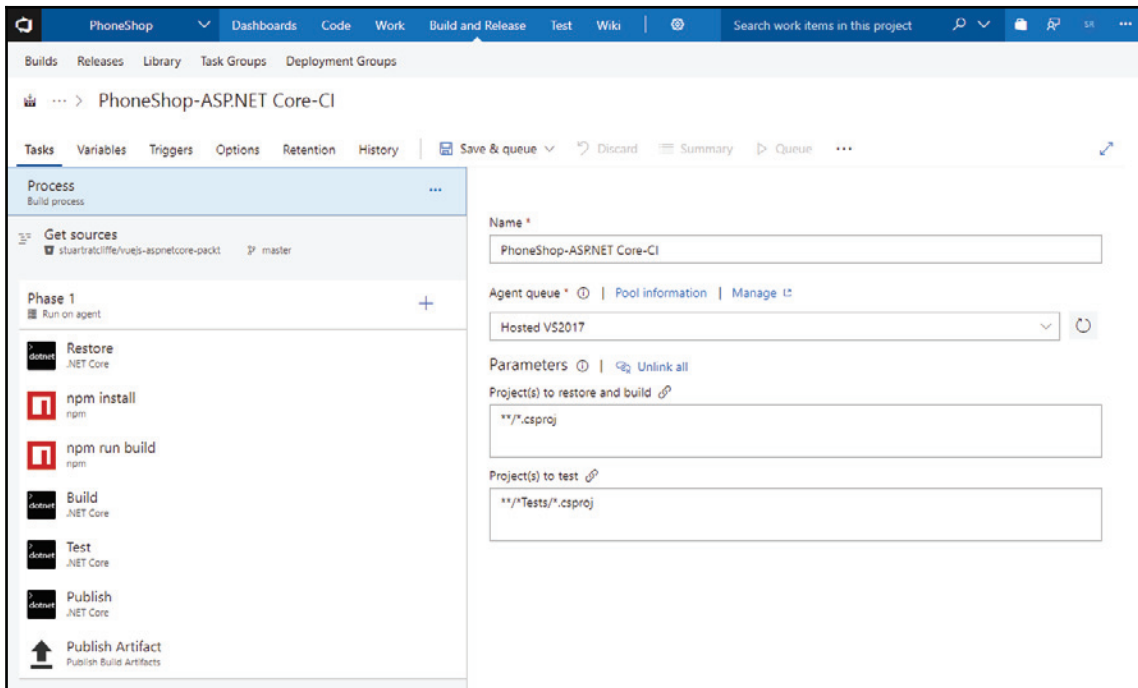
The one we're interested in is the first option displayed here, not the **(.NET Framework)** version. Upon selecting it, you'll be presented with the following build definition screen where we can make any necessary modifications to the default template:



By default, the build is configured to **Get sources**, that is, to download our source code from Git and then run the **.NET Restore**, **Build**, **Test**, and **Publish** commands in that order. If any of these steps fail, the whole build will fail. Finally, if the build succeeds, the **Publish Artifact** step uploads a ZIP file containing our packaged application so that it can be used in a release that we'll configure shortly.

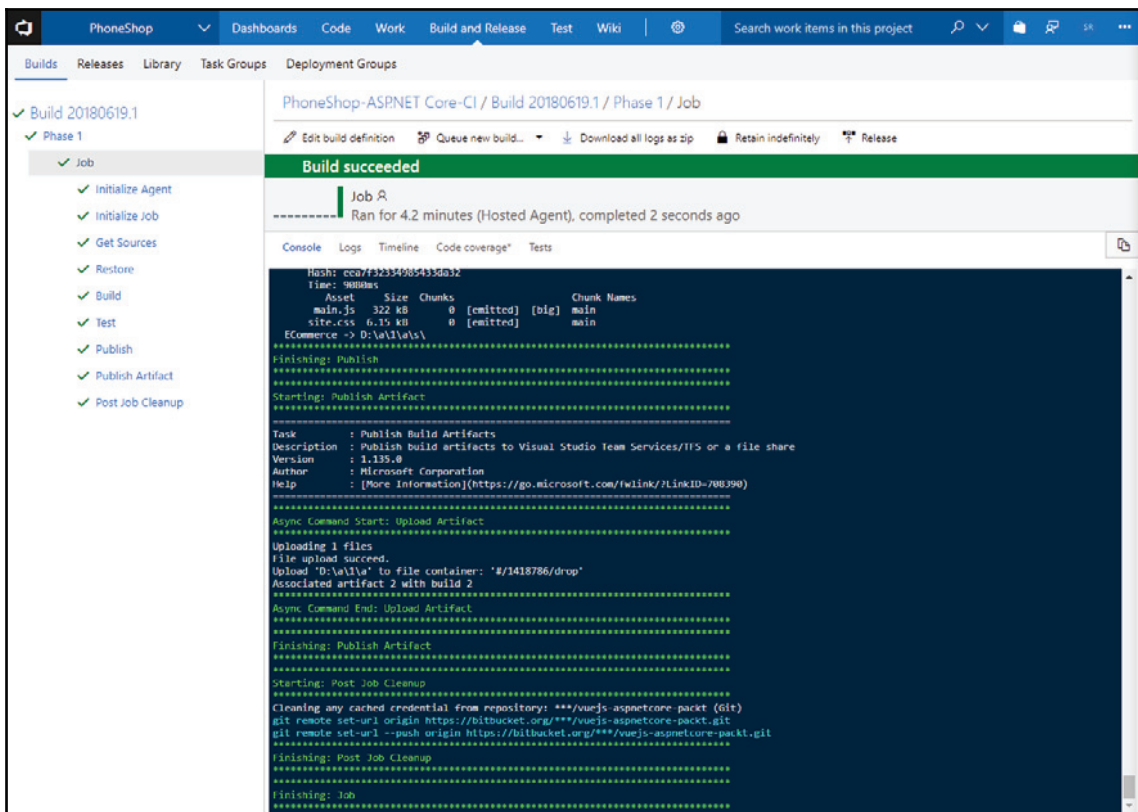
The way we're building our frontend Vue code is already part of the **.NET Publish** command, so we actually don't need to modify this default template at all. The only thing we could do if we wanted is remove the **Test** step, as we haven't written any tests. However, the build won't fail if we don't, as it just won't find any tests to run. Therefore, it's worth leaving it in place in case we were to add some tests in the future.

You can customize this build process to do pretty much anything you can think of with regards to building and packaging an application. For example, if we weren't already building our client-side application as part of the `.NET publish` command, we could configure the build step here instead. To do so, we could add a couple of `npm` commands to run as part of this build process:



However, in our case, this is completely unnecessary so we'll just leave the template in its default state. We can now test our build by clicking the **Save & queue** button at the top of the page. In the modal window that follows, leave the default values for all fields and hit the **Save & queue** button at the bottom to kick off our first build. You'll see a pale green banner displayed at the top of the page to confirm that the build was triggered, including a link to click on, which will take you to the build page itself. From here, you can monitor the progress of the build, which usually takes around 5 minutes to complete depending on how many npm and NuGet packages need to be downloaded.

When the build finishes, you should see a screen that looks something like this:

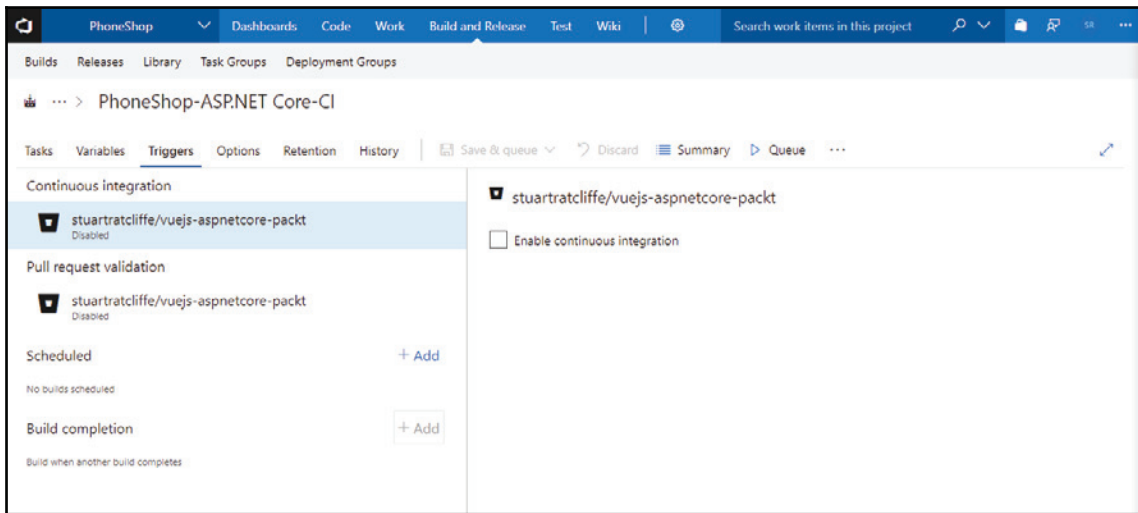


You'll also receive an email to notify you whether or not the build succeeds, meaning you don't need to sit and watch every time you push new changes to your repository.

Enabling CI

At this point, our build is set up and tested so we know that it works, but as it stands we need to manually trigger builds as and when they are required. If we want to automate these builds to run on every push to the master branch, we need to enable CI.

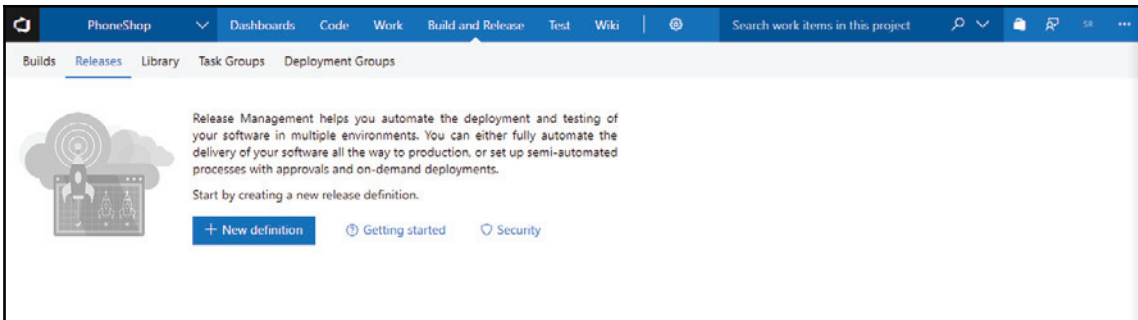
From the preceding build page, click the **Edit build definition** link to be taken back to the page where we had the option of modifying the default template. From here, click on the **Triggers** link at the top of the page, which will present you with the following screen, where you can enable CI:



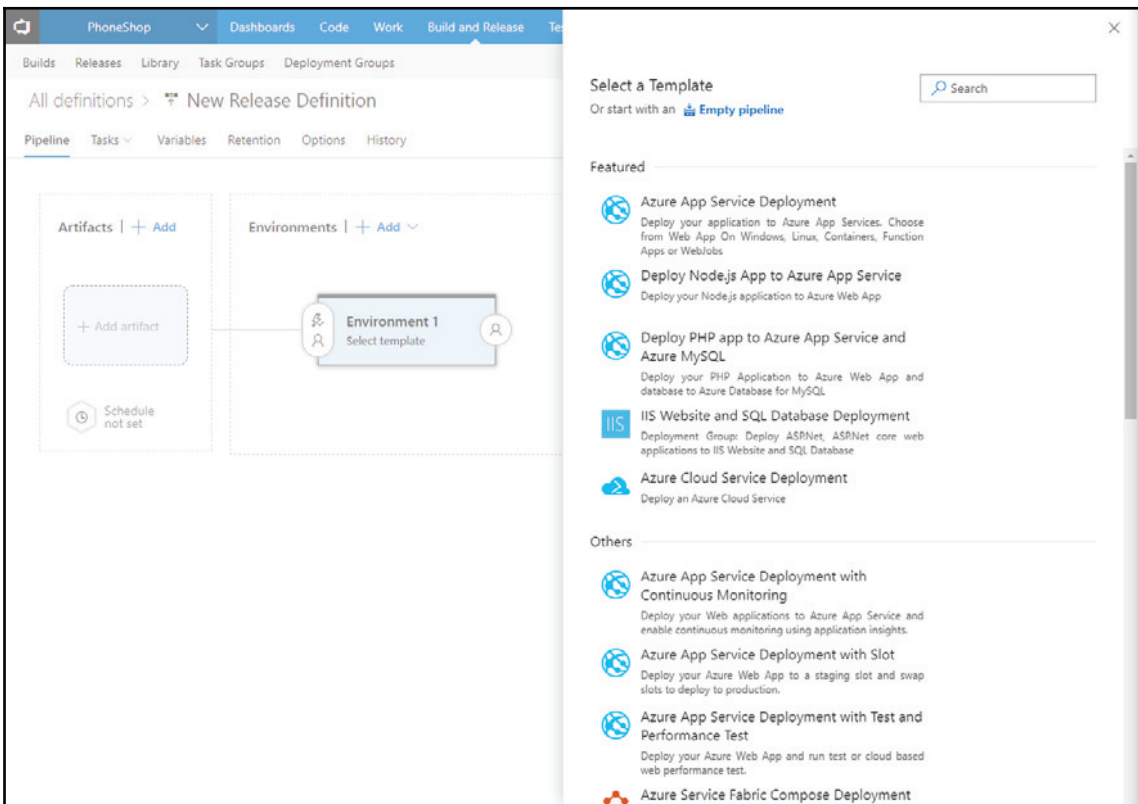
After checking the **Enable continuous integration** checkbox, you can save the build by hovering over the **Save & queue** menu item, then clicking the **Save** option. Now, every time we push new changes to the master branch, our build will run automatically.

Setting up a VSTS release

With our CI build in place, we can now create a VSTS release to actually deploy the packaged application to our Azure app service. We can do so by following the **Releases** link at the top of the screen, which will bring you to the following page:



We don't have any releases yet, so let's go ahead and click the **New definition** button to create one. As with the build definition, you'll be presented with a list of templates to choose from:

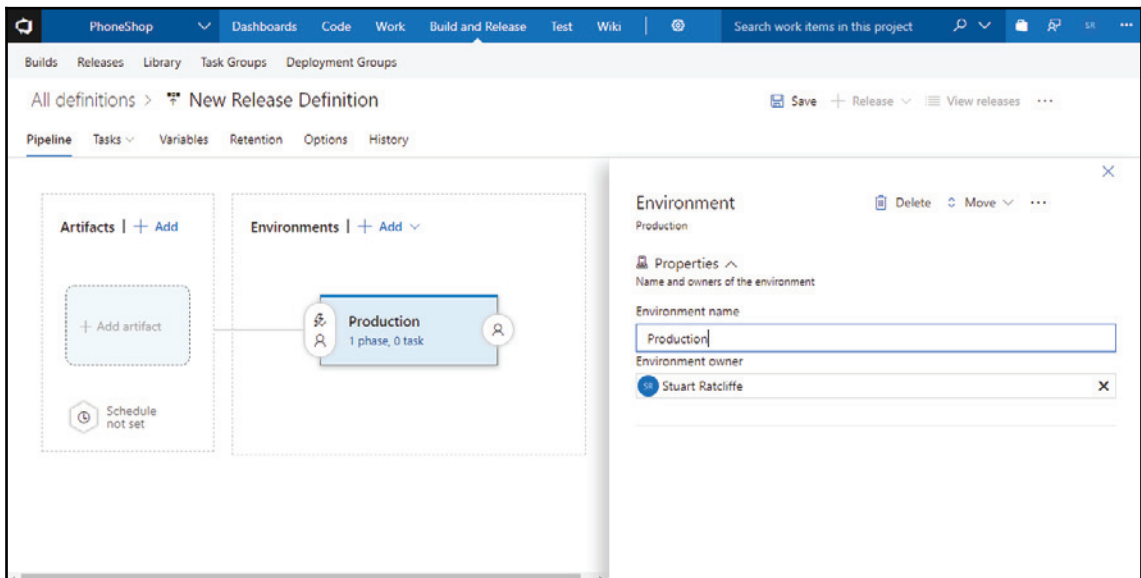


Rather than start with a predefined template, this time we'll use the **Empty pipeline** link at the top and create our own from scratch. By building our own, you'll see just how easy it is, even without the nice-to-have templates that Microsoft provide for us out of the box.

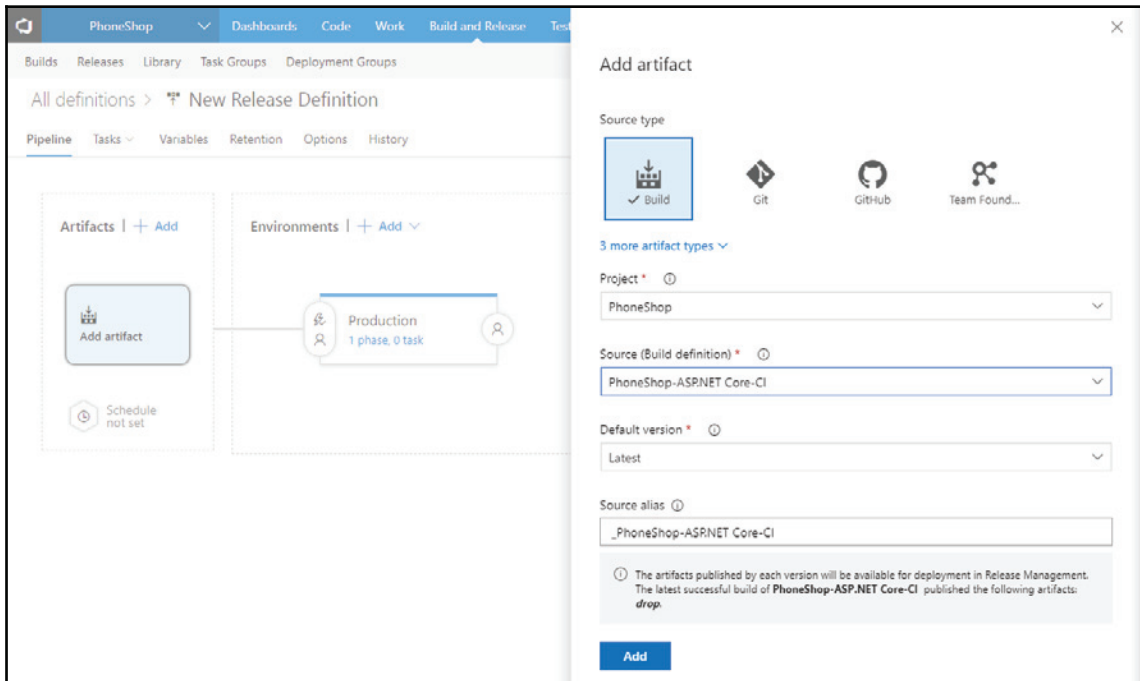


If you'd rather use a template, the Azure App Service Deployment template will work fine after you configure it to point at your own environment!

After selecting the template, we then need to give the environment we're releasing to a name. I've entered `Production` for mine:

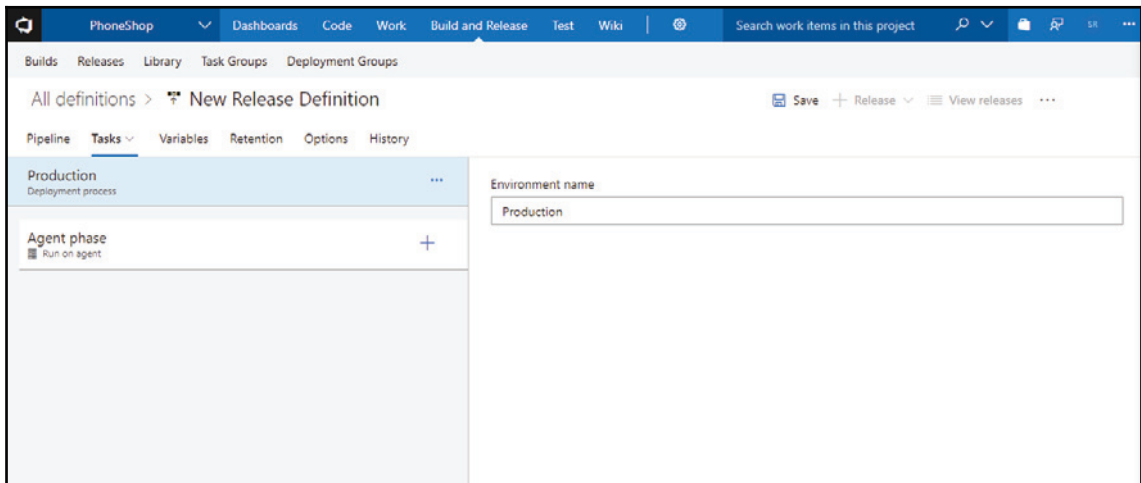


Once you've given the environment a name, close the popup as we now need to add an artifact for VSTS to actually release for us. An artifact is a packaged application that we need to push to our Azure app service, which in our case is the ZIP file that we ended up with at the end of our CI build. We can do so by clicking on the **Add artifact** section to the left of the **Pipeline** tab. Another modal will open from the right-hand side, which looks like this:

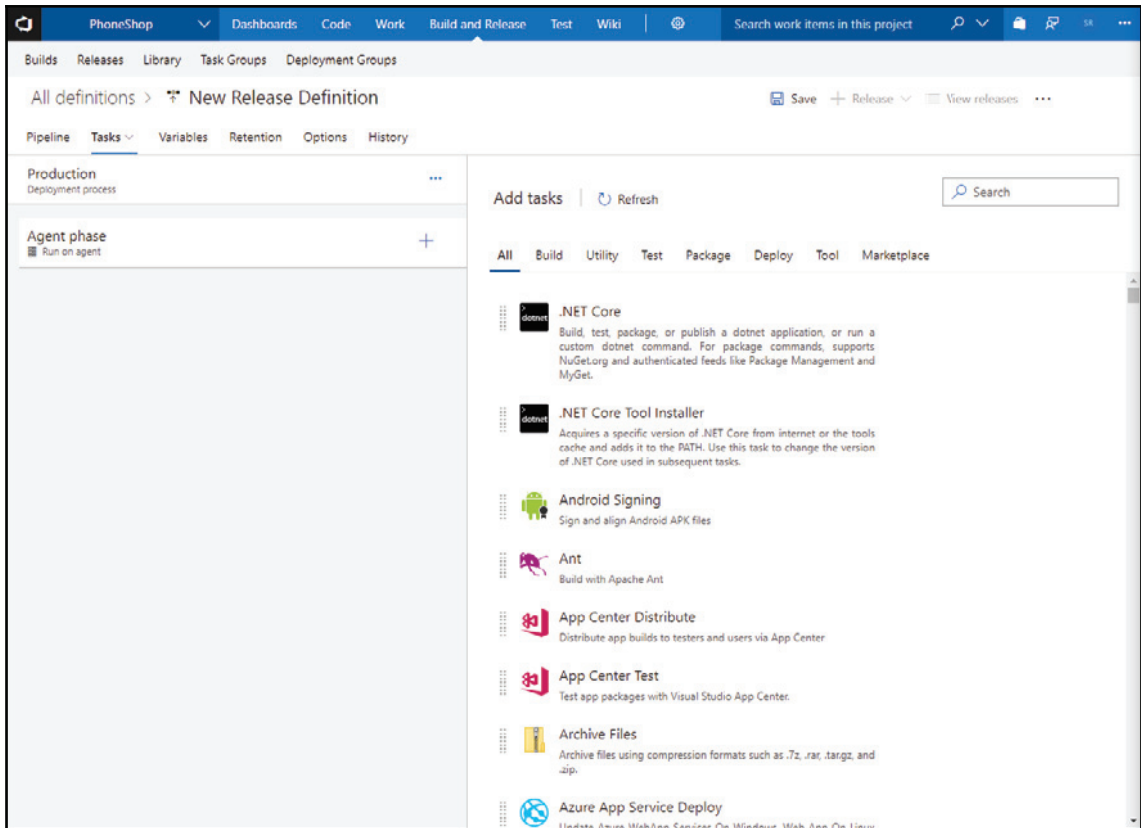


We need to leave the source type as the default **Build** option, then specify the build that we wish to use in the **Source** drop-down menu. In my case, I left the default build name of **PhoneShop-ASP.NET Core-CI**, but if you renamed yours, it will display your specific build name here instead. After selecting the source, the **Default version** dropdown should auto populate to **Latest**, which is what we want. Click the **Add** button at the bottom to complete the artifact selection form.

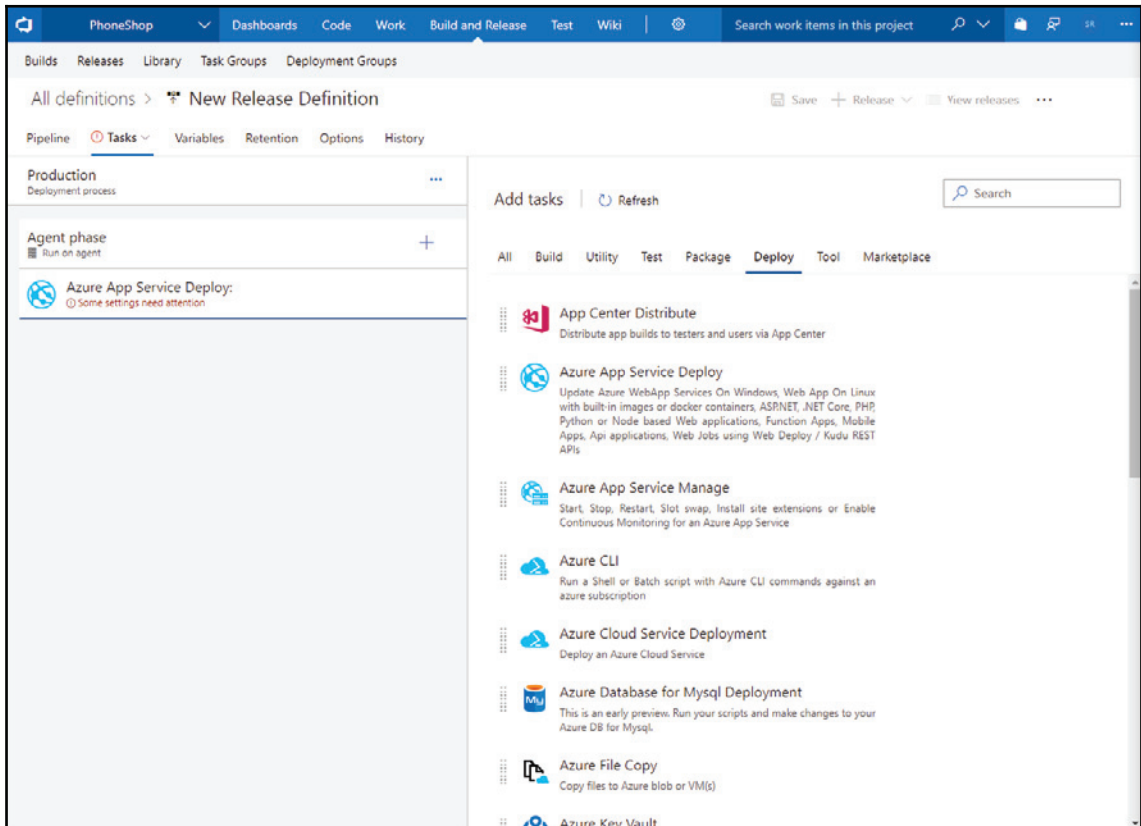
The next step is to tell the release definition what we actually want it to do with the build artifact. Click on the **Tasks** tab and you'll see the following screen:



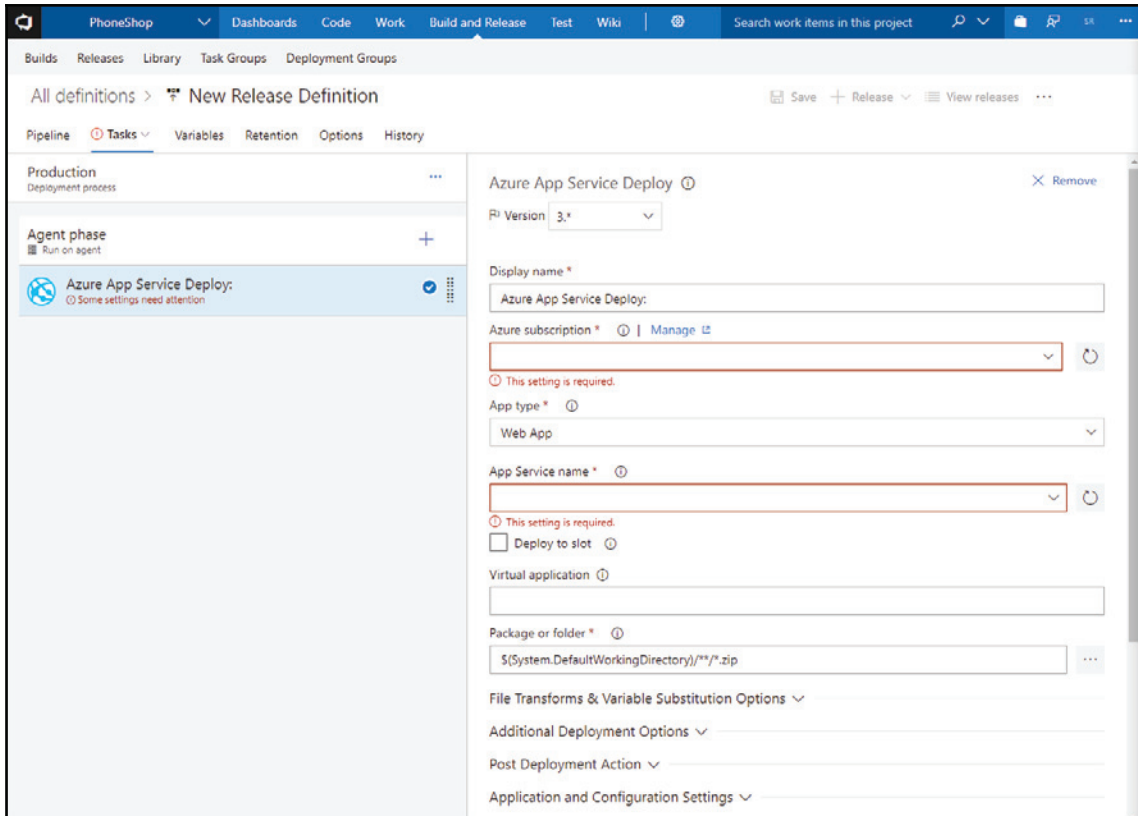
We need to add a task that pushes the build artifact to our Azure app service. This is where it really pays to use VSTS for our CI/CD pipeline, as it provides default tasks for doing just that. Click the + symbol to the right of the **Agent phase** box, and you'll see the following screen with a whole host of available tasks to choose from:



Under the **Deploy** tab, you should be able to find a task named **Azure App Service Deploy**. Click the **Add** button next to it to add the task to the release:

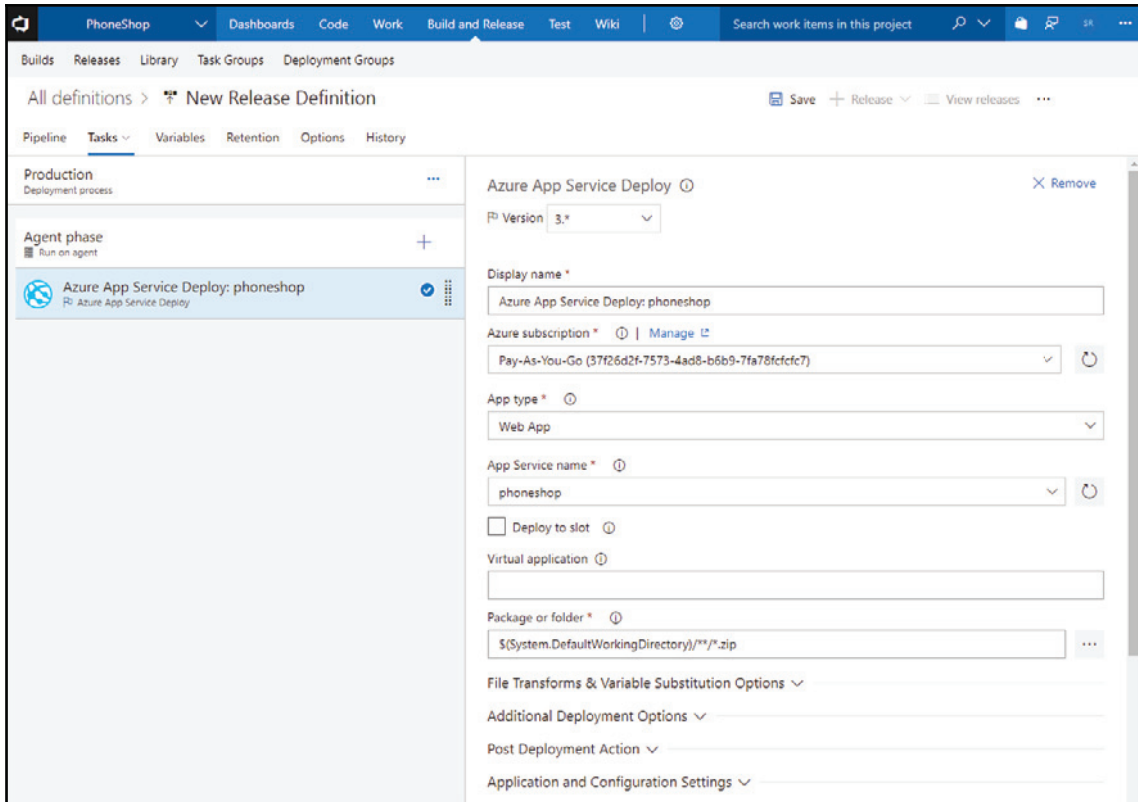


Notice that as soon as you add it, you'll see a red exclamation warning that states some settings need attention. We need to set the Azure subscription and specific app service we want to deploy to, which we can do by clicking on the new **Azure App Service Deploy** task:



At the time of writing this book, the **Version** dropdown had a **4.*** (preview) option available. It is advisable to stick to the latest non-preview version to avoid issues.

As long as you used the same Microsoft account for both your Azure and VSTS accounts, you should see your Azure subscription in the associated dropdown. Once selected, click the **Authorize** button to allow VSTS to access the subscription by creating a service endpoint for you. After doing so, you should be able to select your app service as well:

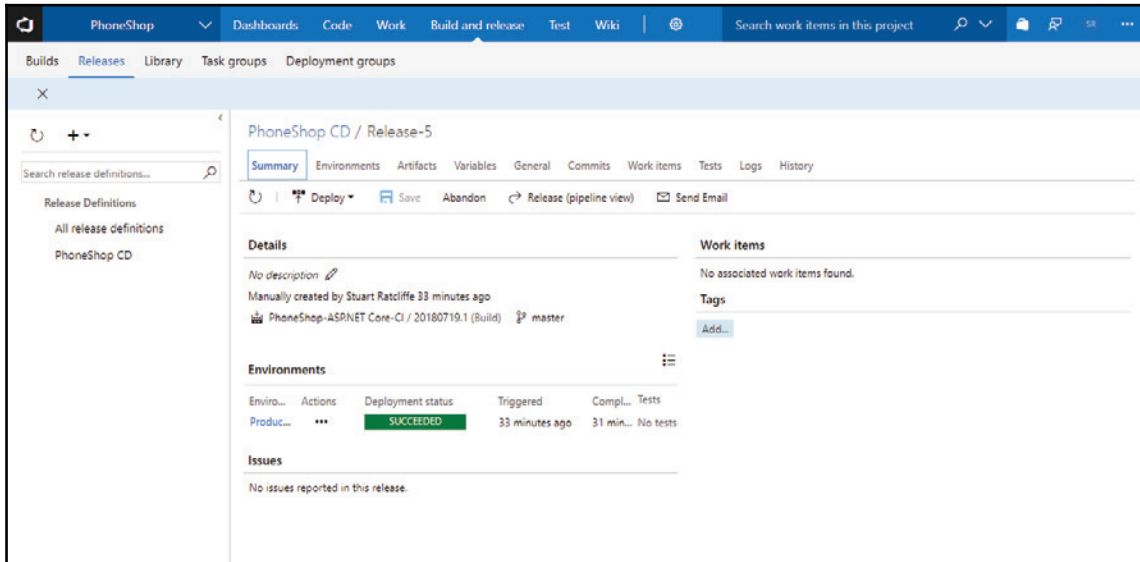


Everything else can remain as is, with the default options selected. However, unless you wish to leave the default **New Release Definition** name, you can change it using the pencil icon that appears when you hover near the name at the very top of the screen. Whether you change the name or not, you must remember to hit the **Save** button at the top to ensure the release definition is saved successfully.



VSTS releases are much more robust than Azure Git deployments, but occasionally you'll still see the **ERROR_FILE_IN_USE** message on a failed deployment. Usually, this is resolved by manually kicking off the release again, but on rare occasions you may need to restart your App Service from the Azure portal.

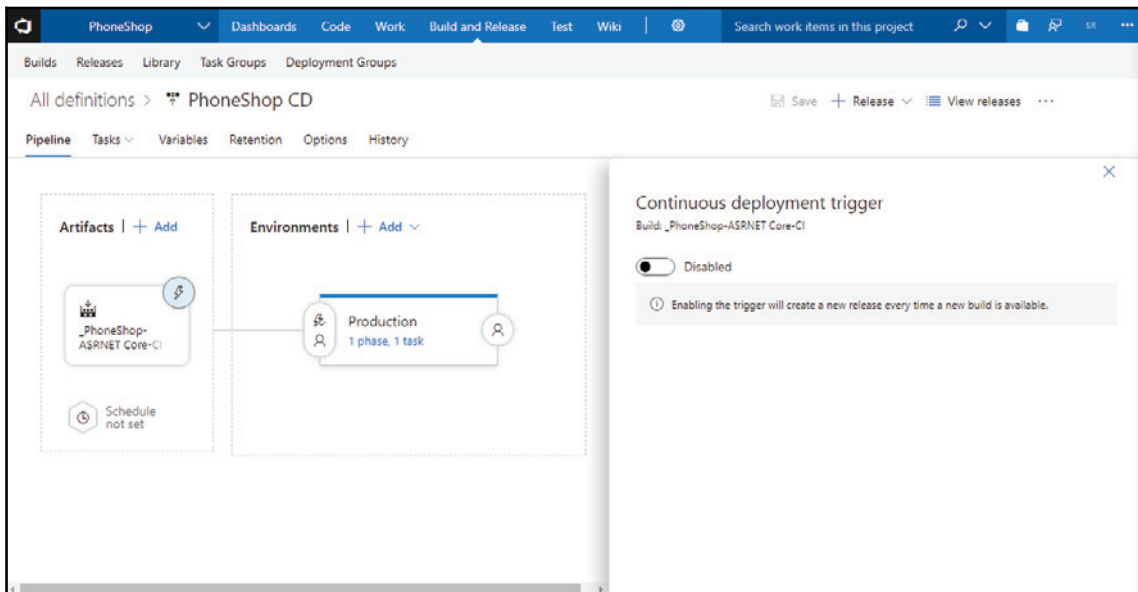
If you manually trigger a release at this point, if all goes well, you'll see a screen that looks something like this once it completes:



Enabling CD

Similar to when we initially created the build definition, as it stands, the release will only run when we trigger it manually. This may be what you want, depending on the preferences of you or your business, but if you'd rather have your releases triggered automatically after every successful build, then we need to enable CD.

To do so, head back to the **Pipeline** tab of our release definition, where you'll see a lightning bolt button in the **Artifacts** section. This will open another modal from the right of the screen, where you can set the **Continuous deployment trigger**:



Flip the switch here to **Enabled**, then remember to hit the **Save** button again to ensure that these changes are persisted. Now, every push to our master Git branch will trigger a new build, which if successful will in turn trigger a release to our production Azure environment.

Summary

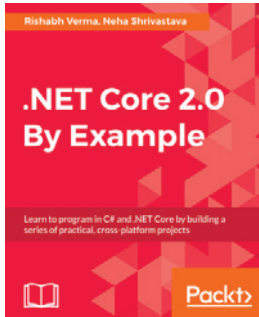
In this chapter, we've created a much more robust deployment mechanism by building a CI/CD pipeline in VSTS. We started by disabling our original Git deployment mechanism in Azure itself, before creating a VSTS build to replace it. We saw that we have much more control over how our application is published if we need it, but also how simple it is to create a build if your requirements are fairly simple, like ours are.

We then enabled CI so that every push to the master Git branch of our source control repository triggers a new build in VSTS. Next, we created a release definition to connect to our Azure subscription and push the build artifact to our production environment app service.

We saw how easy it is to integrate VSTS releases with Azure app services, due to the built-in tasks that Microsoft provide. Finally, we optionally enabled CD so that every time a new build is completed successful, our application is automatically released for us.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

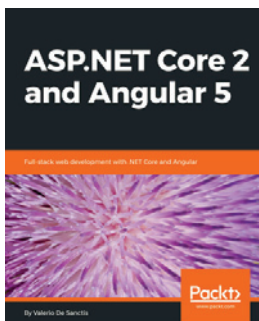


.NET Core 2.0 By Example

Rishabh Verma, Neha Shrivastava

ISBN: 978-1-78839-509-0

- Build cross-platform applications with ASP.NET Core 2.0 and its tools
- Integrate, host, and deploy web apps with the cloud (Microsoft Azure)
- Leverage the ncurses native library to extend console capabilities in .NET Core on Linux and interop with native code. .NET Core on Linux and learn how to interop with existing native code
- Reuse existing .NET Framework and Mono assemblies from .NET Core 2.0 applications
- Develop real-time web applications using ASP.NET Core



ASP.NET Core 2 and Angular 5

Valerio De Sanctis

ISBN: 978-1-78829-360-0

- Use ASP.NET Core to its full extent to create a versatile backend layer based on RESTful APIs
- Consume backend APIs with the brand new Angular 5 HttpClient and use RxJS Observers to feed the frontend UI asynchronously
- Implement an authentication and authorization layer using ASP.NET Identity to support user login with integrated and third-party OAuth 2 providers
- Configure a web application in order to accept user-defined data and persist it into the database using server-side APIs
- Secure your application against threats and vulnerabilities in a time efficient way

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

- access tokens
 - refreshing, with axios interceptor 446, 448, 450
 - removing 450
- accordion component
 - accordion behavior, defining 151
 - building 150
 - styling 153
- account page
 - building 311
 - dates, formatting with reusable date filter 315
 - linking 316
 - OrderList component, building 313
- actions 216
- admin panel components
 - add variant modal component, creating 362, 363, 364
 - building 342, 343
 - nested route definitions, configuring 343, 344
 - product form component, creating 351, 352, 353, 354, 356, 357, 359, 360
 - product list component 349, 350
 - refactoring, for re-use 345, 346, 347
- admin panel
 - accessing 393
 - bug, fixing by product variant selection 394, 396, 398
 - logout bug, fixing 394
- API requests
 - debouncing 195
- API
 - data, fetching 108, 110
- app service plan
 - about 408
 - creating 409, 410, 412
- app service
 - creating 408
- application
 - preparing, for deployment 419
- apps configuration
 - finalizing 426
- argument destructuring 223
- ASP.NET Core
 - about 11
 - application startup 41, 42
 - dependency injection (DI) 42, 43
 - features 39
 - installing 53
 - middleware pipeline 39, 41
 - reference 53
- authentication endpoint
 - extending, with user roles 335, 336
- axios interceptor
 - used, for refreshing access tokens 446, 448, 450
- Azure account
 - registering for 401, 402
- Azure app service Git deployments
 - disabling 495
- Azure App Services
 - HTTPS, forcing on 430
- Azure environment
 - setting up 403
- Azure portal
 - reference 401
- Azure resources 404
- Azure subscriptions
 - about 404
 - creating 405
- Azure
 - database, creating 406, 407
 - environment variables, configuring 413, 414, 415, 416, 417

- logging, enabling 426, 429
- resource group, creating 405

B

- backend dependencies
 - installing 52
- backend setup, Vue application
 - feature folder structure, refactoring to 70
 - refactoring 68
- backend tools
 - ASP.NET Core, installing 53
 - IDE, installing 54
 - IDE, selecting 54
 - installing 52
 - PostgreSQL, installing 53
- Bootstrap 115, 116, 120
- Bootstrap-Vue 116, 120
- browser reliant elements
 - checkout form component, fixing 489
 - page transition animations, fixing in router 490
 - range filter component, fixing 488
 - rendering 487
 - store subscription, fixing to persist cart items to local storage 490
- Bulma
 - reference 116

C

- cart summary component
 - building 295
- catalog page
 - components, tidying 172
 - filter behavior, adding 168, 171
 - filtering logic, testing 174
 - filters component, adding to 167
 - template, updating 168
- checkout components
 - building 293
 - cart summary component, building 295
 - checkout form component, building 297
 - checkout success component, building 309
- checkout form component
 - basic Bootstrap styling, adding to Stripe elements 309
 - building 297
 - client-side validation 298
 - delivery address form fields, finishing 300
 - form input state, validating 303
 - order, submitting to API 306
 - payment details, verifying with Stripe 305
 - payment information, capturing 301
 - Stripe elements 302
- CI/CD pipeline
 - building 500
 - CI, enabling 505
 - VSTS build setup 500, 502, 503, 504
 - VSTS release build 513
 - VSTS release setup 505, 507, 509, 510, 511, 513
- CLI tools
 - versus SPA templates 59
- client app
 - auth navigation item component 281, 283
 - authentication 267
 - authentication modal component 272, 274
 - login form component 275, 277
 - register form component 278, 279, 280
 - user registration 267
 - Vuex authentication getters 269
 - Vuex login 270
 - Vuex logout actions 270
 - Vuex mutations, for managing authentication state 268
 - Vuex register 270
 - Vuex state properties, for authentication 267
- client-side role-based authorization 336, 337
- client-side routes
 - role checks, adding 337, 338, 339
- client-side routing 101, 104, 106, 107
- client-side sorting
 - about 185
 - sort component, adding to catalog page 188, 189
 - sort component, building 186, 187
- client-side validation 292
- client-specific boot logic, SSR
 - client-side store, hydrating 475
 - component data, pre-fetching 477
 - defining 474
 - promises, dealing with 478

- shopping cart data, loading from local storage 475
- client
 - accordion component, building 150
 - filtering 147
 - filters component, adding to catalog page 167
 - filters component, building 154
 - filters component, refactoring 175
 - libraries, installing 147
- component behavior
 - about 19
 - computed properties 23, 25
 - lifecycle hooks 26, 28
 - methods 23
 - props 21, 22
 - state 20, 21
 - watchers 25, 26
- component composition 91, 92, 95, 98, 100
- component presentation
 - about 28
 - directives 28
 - parent-child component, communication 37
- components
 - about 11, 12, 13
 - accordion template structure, defining 150
 - UI, composing 14, 15
- conditional rendering 89, 91
- ConEmu
 - about 57
 - reference 57
- continuous deployment
 - about 495
 - enabling 515
- continuous integration 495
- currency filter
 - creating 234, 235, 236
- custom input controls
 - building 372
- custom type ahead control
 - building 372, 373, 374, 376, 378

D

- data model
 - applications seed data, updating 140, 141, 143
 - database, dropping 133

- DbContext class, updating 139
- extending 133
- migration, creating to reflect model changes 139
- new entities, adding 134, 135, 137
- updated entities, adding 134, 135, 138
- data
 - fetching, from API 108, 110, 113
- database
 - context, creating 71
 - context, registering for DI 73
 - creating 74, 75
 - creating, on application startup 78, 80
 - initial migration, creating 76, 77
 - seeding, on application startup 78, 80
 - setting up 71
- DbContext 439
- dependency injection (DI) 42
- deployment
 - application, preparing for 419
- directives
 - about 28
 - attribute, binding with v-bind 29, 31
 - conditional display, with v-show 31
 - control flow, with v-if and v-else 31
 - event handling, with v-on 34, 35
 - form input binding, with v-model 36, 37
 - lists, rendering with v-for 33
- dotnet CLI
 - project, scaffolding 63

E

- EF Core
 - about 43
 - compiled queries 45
 - global query filters 44
 - in-memory provider, for testing 45, 46
 - relationships, configuring 43
- Entity Framework (EF) 42

F

- filters component
 - adding, to catalog page 167
 - behavior, scaffolding 159
 - brand filter, adding 155
 - building 154

- color filters, adding 158
- common multi-select filter component, extracting 176, 179
- common range filter component, extracting 180, 181
- computed properties, defining 160
- duplication, highlighting 175
- feature filters, adding 158
- methods, defining 161, 163
- multi-select components, rendering 182
- OS filters, adding 158
- price filter, adding 156
- range filter components, rendering 182
- refactoring 175
- screen size filter, adding 157
- styling 167
- template, scaffolding 155
- testing 185

Font Awesome 147

frontend dependencies

- installing 49

frontend setup, Vue application

- default components, replacing 66, 68
- refactoring 64
- TypeScript, removing 65

frontend tools

- installing 49
- Node Package Manager (npm), installing 49, 50
- Node, installing 49, 50
- npm, versus Yarn 51
- Vue, installing 51

G

- getters 216
- Git deployments
 - configuring 423, 424, 425
- Google Chrome
 - reference 49

H

- Hot Module Replacement 62
- HTTPS
 - forcing, on Azure App Services 430
- hydrating 475

I

- IDE
 - installing 54
 - selecting 54
- interceptor function 441

J

- JavaScriptServices middleware 457
- JSON Web Token (JWT) 432
- JWT authentication
 - adding, to API 249
 - configuring 251, 253
 - issuing 254, 257, 258
 - testing 260, 263, 264
 - user registration 264, 266
 - user role support, adding 258, 260
 - using 250

L

- libraries
 - Font Awesome, installing 148
 - installing 147
 - npm packages, installing 148
- lifecycle hooks 26, 28
- lodash 195
- logging
 - enabling, in Azure 426, 429

M

- multi-select control
 - building 378, 380, 382
- multiple database providers
 - configuring 419
- mutation 216

N

- Navbar component 122
- Node Package Manager (npm)
 - installing 49, 50
 - versus Yarn 51
- Node
 - installing 49, 50
 - reference 49
- npm packages

installing, for SSR 458

NProgress 120

Nuxt.js 457

O

order list API endpoint

adding 330

orders and payments

order object, persisting 326

payment, processing with Stripe 328

processing 325

total order price, calculating 327

owned entity types

about 321

configuring 322

defining 321

in EF Core 2.0 321

need for 321

P

page, Vuex

account page, refactoring 465

catalog page, refactoring 463, 464

create product admin page, refactoring 467

orders admin page, refactoring 466

product details page, refactoring 465

products admin page, refactoring 466

updating 462

parent-child component

communication 37

post-publish build steps

tweaking 421

PostgreSQL

installing 53

reference 54

product details component

variant, adding 211

variants, adding 212, 213

product details page

finishing 200, 201, 202, 203, 205

gallery component, creating 206, 207, 208, 209, 211

productivity tools

about 55

Terminal Emulator, installing on Windows 57

VS Code extensions, installing 55, 56, 57

Vue.js Chrome devtools extension, installing 57

products list

displaying 84, 86, 89

products, adding to cart

about 221

actions, creating 223, 224, 227, 228

mutations, creating 221, 222

products, persisting to database

about 384

API endpoint, creating 385, 386, 387

slug generator, creating 384

products

removing, from cart 236, 237, 238

props 21, 22

R

refresh tokens

about 432

adding, to backend 435

adding, to frontend 441

AppUser model, extending 435

finishing up 440

generating 437

JWT access tokens, refreshing 438

need for 434

router configuration, extracting into files 442, 443, 445

register form component

fixing 316

remote validation, with Vee-Validate

about 389

app, making aware of custom validation rule 391

validation API endpoint, creating 392

role checks

adding, to client-side routes 337, 338, 339

S

search bar component

API requests triggering, watchers used 194

creating 190, 191, 194

server-rendered application

testing 491, 492

server-side payment processing

about 318

- orders and payments, processing 325
- orders migration, creating 323
- orders, adding to data model 318
- owned entity types, in EF Core 2.0 321
- Stripe.net NuGet package, configuring 323
- Stripe.net NuGet package, installing 323
- server-side rendering (SSR) 59
- Server-Side Rendering (SSR)
 - about 453
 - application, preparing 458
 - client-specific boot logic, defining 474
 - configuring 471
 - enabling 484, 485, 486
 - existing pages, updating for Vuex 462
 - functions 456
 - mutations, adding for API requests 459
 - need for 454
 - npm packages, installing 458
 - performance 455
 - search engine optimization (SEO) 455
 - server-specific boot logic, defining 479, 480, 481
 - setting up 471
 - shared boot logic, defining 471, 472, 474
 - user authentication state, persisting 468
 - Vuex actions, adding 459
 - webpack, configuring for client/server boot files 481
- server-side role-based authorization 340
- server-specific boot logic, SSR
 - defining 479
 - deleting 481
- server
 - controller actions, updating 144
 - filtering 143
 - filtering logic, testing 146
- shadow property 322
- shopping cart page
 - cart summary component, creating 241, 242, 244
 - CartItem component, creating 229, 230
 - creating 229
 - list of cart items, displaying 232, 233
 - UX, improving with add to cart feedback 247, 248
- shopping cart

- building, options 198
- getter, adding to display cart total 240, 241
- items, updating 238, 239
- persisting to database approach 199
- persisting to local storage 200
- persisting to session state approach 199
- persisting, to local storage 244, 245, 246
- products, removing from 236, 237, 238
- single-file components (SFCs) 12, 59
- Single-Page Application (SPA) 10
- source maps 63
- SPA templates
 - versus CLI tools 59
- Stripe.net NuGet package
 - configuring 323, 324
 - installing 323
- Stripe
 - account, registering 292
 - dashboard 291
 - initiating 292
 - integration 291
 - JavaScript library 292
 - PCI compliance 291
 - using 290
 - VeeValidate npm package, installing for client-side validation 293

T

- table splitting 321
- taghelper 486
- Terminal Emulator
 - installing, on Windows 57

U

- UI elements
 - hiding, based on role 340, 341
- UI state 20
- UI
 - composing, with components 14, 15
- user authentication state, SSR
 - persisting 468
 - persisting, to local state 469
 - persisting, to local storage 468
 - storing, in cookies 470
- User Experience (UX)

- about 115
- application wide layout elements, adding 121
- application wide styles, adding 123
- Bootstrap-Vue 116
- data, fetching 127, 128
- framework, selecting 116
- improving 116
- npm modules, installing 118
- page loading indicator, adding 129
- product details components, styling 123, 126
- product list, styling 123, 126
- transition, adding on page change 131
- vendor bundle, building 120
- webpack configuration, modifying to SASS 118
- webpack vendor configuration, updating 119
- user roles
 - authentication endpoint, extending with 335, 336

V

- Vee-Validate
 - remote validation 389
- Vetur 55
- VS Code extensions
 - installing 55, 56
- VS Code
 - about 54
 - reference 54
- VSTS account
 - creating 497
- VSTS
 - about 497
 - team services project setup 498, 499
- Vue component inheritance
 - about 364, 366
 - form input base component, defining 366, 367
 - inheritance, from base component 368
- Vue instance
 - about 16
 - attaching, to DOM 16
 - component trees, building 18, 19
 - data properties, defining 16
 - data, rendering into DOM with expressions 17, 18
- Vue.js Chrome devtools extension
 - installing 57

- Vue.js
 - about 10
 - basics 16
 - reactivity 19
 - Vue instance 16
- Vue
 - installing 51
- Vuex actions, SSR
 - defining 460
- Vuex documentation
 - reference 214
- Vuex mutations, SSR
 - defining 461
 - store state properties, defining 462
- Vuex state properties
 - for authentication 267
- Vuex store
 - defining 217, 218
- Vuex
 - about 213
 - actions 216
 - configuring 219, 221
 - features 214
 - getters 216
 - installing 219, 221
 - mutations 216
 - working 215

W

- watchers 25, 26
- web browser
 - selecting 48
- webpack, SSR
 - client- and server-specific webpack configuration
 - objects, defining 483
 - configuring, for client/server boot files 481
 - shared webpack configuration object, defining 481
 - vendor webpack configuration, updating for SSR
 - libraries 484
- webpack
 - about 51, 60
 - bundle, splitting 62
 - configuration 61, 62
 - production bundles 63

working 60
Windows
Terminal Emulator, installing 57

Yarn
installing 52
reference 51
versus Node Package Manager (npm) 51

Y