Building Real-World Web Applications with Vue.js 3

Build a portfolio of Vue.js and TypeScript web applications to advance your career in web development

JORAN QUINTEN

Building Real-World
Web Applications with Vue.js 3

Build a portfolio of Vue.js and TypeScript web applications
to advance your career in web development

Joran Quinten

Building Real-World Web Applications with Vue.js 3

To my wife, Lydia, for encouraging me to try new things, supporting me, and giving me the space to grow and pursue ambitious goals. To my kids, Joep and Lena, for your regular distractions, for your general curiosity, and for teaching me to see the world through different eyes.

– Joran Quinten

Contributors

About the author

Joran Quinten has a passion for getting people to love technology and getting technology to play nice. With over a decade of experience in developing and designing software, he has built up a broad background in development and architecture. He loves sharing knowledge and has been invited to speak at several tech events. Joran graduated from Fontys University of Applied Sciences, Eindhoven, in 2010. Currently, he works for Jumbo Supermarkten, a national grocery chain, where he is the technical lead of the design system team and acts as an ambassador for the tech department. He is passionate about mentoring and coaching junior developers. Joran lives close to Eindhoven with his wife, son, and daughter.

I want to express gratitude to my supportive team of editors and reviewers who have been of great help in helping me to find the right tone and in offering encouragement throughout the writing process. Special thanks to my wife, Lydia, and my kids, who teach me to take up and experience new challenges. And finally, thanks to family, friends, and colleagues who have been interested in and supportive of this journey.

About the reviewers

Abdallah Idriss Lutaaya is a software engineer with more than four years of experience in full stack development. He holds a master's in computer science from the International University of Applied Sciences, Germany, and a bachelor's from Makerere University, Uganda.

Abdallah excels in crafting scalable web applications and building robust backend systems for seamless integration.

His language repertoire includes Python, JavaScript, Dart, Node.js, TypeScript, SQL, and Dart, with mastery in React.js, Vue.js, Flutter, Django, and Flask. Abdallah brings a unique mix of academic rigor and practical experience.

Javier Suarez is an experienced software developer with over eight years of experience in building web applications. He possesses a strong proficiency in a wide range of technologies, including Vue.js, Quasar Framework, Nuxt, Pinia, Vite, JavaScript, Django REST Framework, Python, Git, and Linux.

Javier worked as a frontend developer at Housettel in Malaga. During his time there, he made significant contributions to the implementation of property management systems, leveraging his skills in Vue.js and Quasar Framework with TypeScript. His efforts resulted in a reduction in the time required for real estate listings.

He also served as a frontend developer at CodigoJS in Seville. In this role, he played a key role in developing a web application for managing neighborhood associations. Javier utilized his expertise in Django/Python, Vue.js, and Quasar Framework to create an efficient and user-friendly solution.

Since 2022, Javier has been working as a frontend developer at Mind Movies LLC in San Diego, CA. In this position, he has developed a web application using Vue.js, JavaScript, and Quasar Framework. He successfully deployed the application to AWS as a **Progressive Web App** (**PWA**). Collaborating with a team of engineers, he has contributed to the design and implementation of various features within the application.

Pyae Sone Win is an experienced full stack developer with over five years of professional experience in web application and website development. His expertise lies in utilizing programming languages such as PHP, Laravel, and JavaScript to build efficient and effective solutions. He is well versed in using MVC frameworks and data models to design and develop web applications that are both functional and user-friendly. He is also skilled in automated scripts with webpack and single-page applications with Vue.js and React.js.

In addition to his web development skills, Pyae is also experienced in mobile application development for retail e-commerce and education platforms. He has a strong understanding of both iOS and Android development, allowing him to deliver high-quality solutions for both platforms.

He is familiar with the AWS production environment and has experience maintaining and deploying customer-facing applications on the platform. He is also well versed in the best development practices, such as TDD and Scrum, ensuring that he is able to deliver high-quality solutions that meet the needs of his clients.

He is passionate about his work and is always looking for new challenges and opportunities to improve his skills. He is a team player who can easily adapt to new technologies and thrive in fast-paced, deadline-driven environments. He is confident that his skills and experience make him an excellent candidate for any web development or mobile development project.

Table of Contents

Part 3: Advanced Applications

8

Building an Interactive Quiz App   193

9

Experimental Object Recognition with TensorFlow   223

Part 4: Wrapping Up

10

Preface

Welcome to the Vue.js community, which is one of the most friendly frontend communities. Vue.js is a frontend framework that allows you to build performant interactive web applications with ease. Vue.js has a shallow learning curve – getting started is easy! This book guides you on your first steps to creating Vue.js applications with increasing complexity and size.

Apart from showing you the technology and teaching you the best practices with Vue.js, the chapters of this book are set up to teach you about general development practices as well. The book lets you experience different approaches when dealing with new technology, implementing third-party solutions, or orchestrating more complex application structures.

There are many resources that focus on very specific use cases or even single components. The aim of this book is to offer a realistic and broad view of Vue.js developer responsibilities and expectations. Every chapter will result in a functional application. Every application introduces a new concept to familiarize yourself with.

I have designed the chapters to progress in a natural way, where we increase complexity with every chapter. I will enforce concepts iteratively over the course of multiple chapters. I am experienced in using Vue.js in a complex, enterprise-level environment where my experience in adopting useful practices and working with third-party solutions has shaped the focal points of the structure and chapters.

As a mentor and coach, I have tried to show and guide you through regular development processes – not writing perfect code from the start but embracing refactoring steps and improving software while an application grows.

The main goal of this book is to not only allow you to both learn and understand Vue.js and its ecosystem, but also prepare you to land a Vue.js developer job by growing a portfolio that showcases your capabilities as a professional web developer.

Using technologies such as Vue.js, Nuxt, Pinia, and Vite is what enables us to build the wildest applications. This is only possible because of the combined and relentless efforts of core maintainers and the numerous contributors that build and publish open source software. Please consider donating or participating to show your support as well. Any contribution to these frameworks or libraries is welcomed and much needed to keep maintaining and developing software that benefits us all.

Who this book is for

This book is aimed at software engineers with an affinity for web-based technologies. Anyone with a software engineering background should be able to quickly pick up the concepts in this book.

Primarily, the book aims to guide beginner or junior developers to familiarize themselves with Vue.js and frontend technologies and practices. This book helps them build up experience in a broad range of topics, which helps in more successfully applying for a position as a Vue.js developer.

While not the primary focus, any software engineer who is curious about the Vue.js ecosystem can very well progress through the chapters, building up a broad sense of the possibilities of using Vue.js as a framework to build applications.

If you have an affinity with one of the currently popular frontend frameworks or libraries such as React, Angular, Svelte, or Qwik, you will have a head start in grasping concepts such as reactivity, testing, and fetching data from APIs. If you are looking to transition to a Vue.js-orientated position, this book will get you up to speed with the Vue.js approach to common practices.

What this book covers

Chapter 1, *Introduction to Vue.js*, explains the core concepts we need to build Vue.js applications. It will help you set up a development environment with the recommended settings for Vue.js development.

Chapter 2, *Creating a Todo List App*, builds upon the core concepts and explains a key concept in creating interactive web applications – reactivity. It also introduces the development and debugging tools as important tools to maintain and inspect applications.

Chapter 3, *Building a Local Weather App*, explores external data as a resource of a web application. It will add a testing framework to the application, which we can use to increase the robustness of the application by identifying both happy and unhappy user flows and how to deal with those.

Chapter 4, *Building the Marvel Explorer App*, leans heavily on interacting with a great volume of external data and connecting to public APIs to retrieve the correct data for a user. It uses composables (a concept for working with stateful logic) to use and reuse functions and compose more complex behavior. By leveraging the default router, we will introduce multiple views and routes to an application.

Chapter 5, *Building a Recipe App with Vuetify*, teaches you how to use third-party libraries, such as Vuetify, to quickly build an interface. It will strengthen the concept of working with APIs by iteration and introduce Pinia, the default state management library for Vue.js. With Pinia, you will learn to persist state in the browser. You will learn refactoring techniques and experience dealing with changing features and requirements.

Chapter 6, *Creating a Fitness Tracker with Data Visualization* continues the topic of persisting state and teaches you how to store data in an external database, as well as adding entry-level authentication to a web application. It demonstrates trade-offs between abstraction and a more pragmatic approach when building a feature, revisiting refactoring strategies.

Chapter 7, *Building a Multi-Platform Expense Tracker Using Quasar*, sidesteps in a different direction, where you learn how to use web technologies to build applications for one or even multiple non-web platforms using a framework. It continues to solidify previously learned topics by revisiting similar tech stacks but with different goals and features in mind.

Chapter 8, *Building an Interactive Quiz App*, dives deep into developing an app from backend to frontend and includes architectural concepts and decisions as well. It introduces Nuxt, which is one of the most popular and developer-friendly meta frameworks for the Vue.js ecosystem. You will interact with WebSockets and see how to create real-time interactivity between multiple clients at the same time.

Chapter 9, *Experimental Object Recognition with TensorFlow*, teaches you the prototyping practices and how to familiarize yourself with new technologies by experimenting in an isolated environment. It also touches upon developing and testing early on a specific target. Most importantly, it teaches you to have a bit of fun when building new projects, keeping you interested and motivated in your own continuous development.

Chapter 10, *Creating a Portfolio with Nuxt.js and Storyblok*, circles back to Nuxt as a framework to generate code, instead of acting as a real-time web server. This chapter allows you to create a personal project, where you can showcase your talents as a developer while progressing through this book. It connects the application to a headless **content management system** (**CMS**) and teaches you how to automate tasks such as deployments.

To get the most out of this book

As the bare minimum to understand the concepts we will cover, experience with HTML, CSS, and JavaScript and basic knowledge of web engineering are highly recommended.

Software/hardware covered in the book	Operating system requirements
Vue.js 3	Windows, macOS, or Linux
Vue Test Utils	
Nuxt 3	
Pinia	
Vuetify	
Tailwind	
Chart.js	
Quasar	
Express.js	
Socket.io	
TensorFlow.js	

The libraries mentioned will have installation instructions in the sections where they are introduced and used. To follow along with the code, you can find links to the lengthier code blocks in the text, which point to the repository that is part of this book.

The completed code for each chapter can also be downloaded from the repository, or simply viewed only as a reference. In some cases, the repository itself may need additional setup with third-party providers for it to work. These instructions are all listed in the relevant sections.

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

If you encounter issues while building the projects, you can always refer to the source code on the GitHub repository. Each chapter includes a `TROUBLESHOOTING.md` *file to help you on your way if you run into problems.*

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3`. If there's an update to the code, it will be updated in the GitHub repository.

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

Code in Action

The *Code in Action* videos for this book can be viewed at `http://bit.ly/2OQfDum`.

Conventions used

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

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The `v-for` directive repeats the `<li>` item with a `<ListItem />` component enclosed."

A block of code is set as follows:

```
<style scoped>
ul {
  list-style: none;
}
```

```
li {
  margin: 0.4rem 0;
}
</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:

```
<ListItem :is-checked='item.checked' v-on:click.
prevent="updateItem(item)">{{ item.title }}</ListItem>
```

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

```
npm init vue@latest
```

Bold: Indicates a new term, an important word, or words that you see on screen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Click **Save** to close the foreign key property."

> **Tips or important notes**
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

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

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

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

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

Share Your Thoughts

Once you've read Building Real-World Web Applications with Vue.js 3, we'd love to hear your thoughts! Please visit `https://packt.link/r/1837630399` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781837630394`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: Getting Started with Vue.js Development

In this part, you will familiarize yourself with the core fundamentals of the Vue.js framework and learn about applying these fundamentals to create interactive web applications. You will learn the best practices in how to set up projects and ensure stability with automated testing. Lastly, we will learn how to work with external data and build a multi-page application.

This part has the following chapters:

- *Chapter 1, Introduction to Vue.js*
- *Chapter 2, Creating a Todo List App*
- *Chapter 3, Building a Local Weather App*
- *Chapter 4, Building the Marvel Explorer App*

1
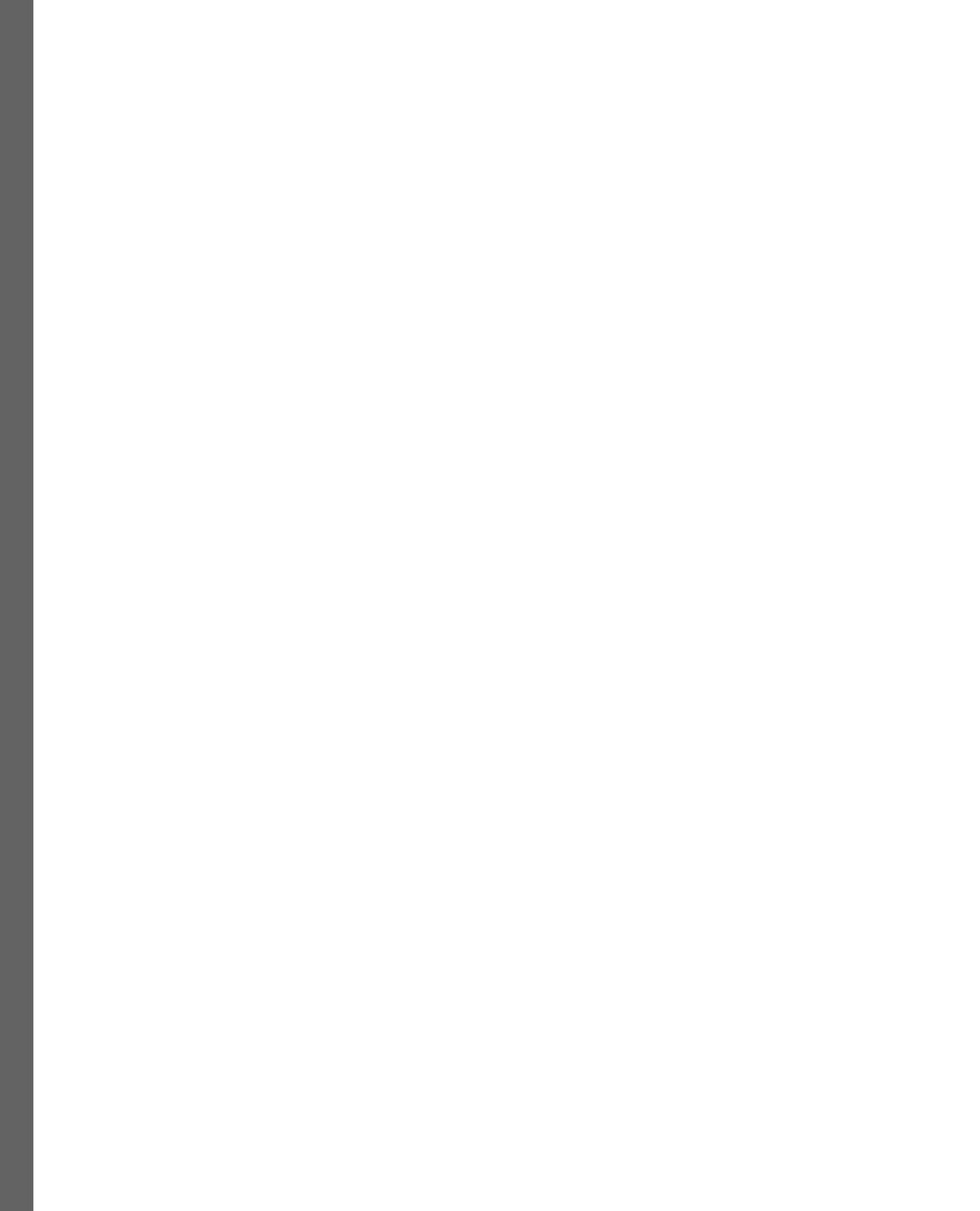
Introduction to Vue.js

This book will get you familiar with one of the most popular modern frontend frameworks at this time: **Vue.js**. Modern frameworks make it easy to add interactivity to static web pages and can help you build entire web applications!

The latter is exactly what you will be learning while going through this book. You will learn different techniques that fit specific use cases, and this will result in a collection of portfolio-ready projects.

Before you start building projects, we will take a look at the context in which we will build those projects. In this chapter, we will get you set up to start any Vue.js project using the best practices.

We will cover the following topics in this chapter:

- The need for Vue.js
- Requirements and tooling
- Setting up the development environment
- My first app

Technical requirements

Before we get started, we need to make sure we've met all requirements to install and run Vue.js, and to also develop applications. Familiarity with HTML, CSS, and JavaScript with TypeScript is required to understand the concepts that are built on top of these technologies.

In order for us to run Vue.js locally, we need to install Node.js (`https://nodejs.org/en/download`). This is a runner for JavaScript on your development machine.

The need for Vue.js

There are a number of frontend frameworks available, and this is an ever evolving and changing landscape. The Vue.js ecosystem is, at this point in time, a very mature landscape, which offers numerous plugins and tooling built on top of it.

Vue.js is a community-driven open source project maintained and developed by people from all over the world. It originated as a personal side project by Evan You and has grown to a framework with high adoption by all types of organizations, such as NASA, Apple, Google, and Microsoft. Being funded by sponsorships from both companies as individuals, Vue.js is an independent framework.

Vue.js is currently on version 3, a breaking change from the previous version, although most patterns are still applicable. Vue.js 3 supports all ES2015-compatible browsers.

Let's look at some reasons to choose Vue.js to build web applications:

- It's performant because it was built to be optimized from the ground up.

- It's lightweight, tree-shakeable, and ships only the code that is needed to run your application. The minimal code (after being optimized in a build step) is about 16 KB in size.

- It's highly scalable, using preferred patterns such as Single File Components and the Composition API, which makes it suitable for enterprise applications.

  Single File Components are part of the Vue.js philosophy where the template, script, and styling of a component are encapsulated in a single file, with the goal of improving the organization, readability, and maintainability of code.

  The **Composition API** allows better organization and reuse of code throughout the application, which makes code more modular and easy to maintain.

On top of all these benefits, the learning curve is very accessible for entry-level developers. With syntax that is similar to classic HTML, JavaScript, and CSS notation, it's easy to get started and find your way around.

In this chapter, we will guide you through the initial setup and go through the steps and setup that you can use as boilerplate for all future Vue.js projects. We will be adopting the recommended setup to make sure you will learn and apply best practices from the start.

We'll first make sure that you have the developer environment set up so that you can start creating interactive web applications!

Requirements and tooling

In order to get started with Vue.js development efficiently, we will need to make sure you can actually run and edit the code sensibly. While you could technically run the code using the library from a **Content Delivery Network** (**CDN**), this is not recommended for real-world applications. As the official

docs (`https://vuejs.org/guide/introduction.html`) also state, there is no build setup involved, and the drawback is that this setup does not support the Single File Component syntax and leaves you with little control over applications' optimizations such as compiling, minification, and lazy loading.

In this book, we will make use of the Vue.js `npm` package and then use it to scaffold starter projects to build upon. We will start all of our projects using the command line. To use the `npm` package, you will need to install Node.js (a JavaScript runtime environment). Make sure to install at least Node.js version 18. npm is a public repository (`https://www.npmjs.com/`) where developers publish, share, and use JavaScript packages.

Node.js can be downloaded from `https://nodejs.org/en`. You can confirm the correct installation of the version by opening a **Command-Line Interface** (**CLI**) and typing the following:

```
node -v
```

It should return something like this:

```
v18.0.0
```

It is always possible to develop or experiment without any local installations. We can turn to web-based environments for that. These environments offer sandboxed environments with sensible presets out of the box. This means they often are configured to use the recommended settings from the official documentation. They offer less control and are somewhat limited in more specific use cases. They are highly valuable, though, if you want to experiment with some of the concepts.

Online resources

Vue.js provides an online development playground, `https://sfc.vuejs.org/`, but I would like to point out **StackBlitz** (`https://stackblitz.com/`), where you can instantiate complete development environments that run in the browser. While it's not useful for deploying applications, this is an excellent way of testing a proof of concept or just using it as a small playground.

You can just register, start a new project, and choose a **Vue.js 3** template to get started. Code examples will be available on GitHub, where you can clone or fork the repository to validate your own code against the working example.

For future reference, the Vue.js docs (`https://vuejs.org/guide/introduction.html`) are very accessible and offer a step-by-step explanation of all the possible contexts. I would certainly recommend checking them out for a more in-depth understanding of the topics we will be covering.

Once you get involved in the Vue.js community, you will find that it is a very helpful, supportive, and friendly community. Again, the official Vue.js website provides some guidance (`https://vuejs.org/about/community-guide.html`).

Welcome to the community, now let's get started!

Setting up the development environment

There are many ways of writing and editing code, and in time, you will find a flow that works best for you. For this book, we will get started using a generally recommended setup. Feel free to make changes that work for you.

Vue.js development takes place in an environment that allows you to write code efficiently by highlighting correct code and helping you to catch errors before you save your changes. Debugging and validating code can take place in various stages, but for the scope of this book, we'll use the development interface as well as the browser environment.

Let's start by installing a widely adopted development interface.

Integrated development environment

An **Integrated Development Environment** (**IDE**) helps you write and edit code by supporting helpers such as syntax highlighting, formatting, and plugins that tie in with the framework of choice. Any modern editor will do, but in this book, we will use Microsoft **Visual Studio Code** (**VSCode**), which is free to use and provides a good developer experience; it can be downloaded from `https://code.visualstudio.com/`.

On top of the installation of the IDE, I recommend the following plugins, which make the developer experience much more pleasant:

- **Vue Language Features** (**Volar**): Supports the markup of Vue.js 3 snippets and highlighting
- **Vue Volar extension pack**: Adds some recommended plugins to help automate some chores while coding
- **Better comments**: For better markup of comments in the code
- **Indent-rainbow**: Applies color to indented blocks of code, to quickly identify levels of indentation

Vue.js can be developed using many other IDEs, other IDEs, such as WebStorm, Sublime Text, Vim/NeoVim, and Emacs. Choose what suits you, bear in mind that screenshots will be shown using the recommended VSCode setup, as described earlier.

Vue.js DevTools

Today's browsers come with built-in tools that allow web developers to inspect and manipulate the HTML, CSS, and JavaScript code of web pages, test and debug their code, measure page performance, and simulate various device and network conditions.

macOS users can open the DevTools using *Cmd + Option + I*. Windows or Linux users can use *Ctrl + Shift + I*.

It is good to note that when you're inspecting an element in the browser, the elements you will see are the elements that are rendered by Vue.js! If you inspect the source code of the browser, you will see just the mounting point of the app. This is the virtual **Document Object Model** (**DOM**) in action, and we will clarify that a bit later on.

Since Vue.js runs (typically) in a browser environment, using DevTools is a skill that is just as valuable as writing clean code. For Chromium-based browsers and Firefox, Vue.js offers a standard plugin.

The Vue.js DevTools plugin helps you with inspecting and manipulating Vue.js components while they are running in the browser. This will help with pinpointing bugs and getting a better understanding of how the state of the app is translated to the **User Interface** (**UI**).

> **Note**
>
> You can find more information and install the plugin here: `https://vuejs.org/guide/scaling-up/tooling.html#browser-devtools`.

We will take an in-depth look at the Vue.js DevTools at a later stage. At this point, we've met all of the requirements of starting out with any Vue.js app, either small or large scale. They all meet the same basic requirements.

At this point you must be eager to dive in and start with the first project, so let's create a small app to familiarize ourselves with development.

My first app

Let's put our acquired tools and knowledge to the test by creating our very first Vue.js application, shall we?

You would usually start by opening your CLI and navigating to a folder where you want to start your projects. Typing the following command creates a new empty project using the official `create-vue` tool:

```
npm init vue@latest
```

Hit *y* to proceed, choose `my-first-vue` as the project name, and select the options shown in the following figure:


```
Vue.js - The Progressive JavaScript Framework

✓ Project name: … my-first-vue
✓ Add TypeScript? … No / Yes
✓ Add JSX Support? … No / Yes
✓ Add Vue Router for Single Page Application development? … No / Yes
✓ Add Pinia for state management? … No / Yes
✓ Add Vitest for Unit Testing? … No / Yes
✓ Add an End-to-End Testing Solution? › No
✓ Add ESLint for code quality? … No / Yes
✓ Add Prettier for code formatting? … No / Yes

Scaffolding project in /Users/jquinten/Projects/packt/my-first-vue...

Done. Now run:

  cd my-first-vue
  npm install
  npm run format
  npm run dev
```

Figure 1.1 – Using the Vue CLI to scaffold out an app with presets

We've selected TypeScript as a superset of JavaScript, which adds static typing. We've also enabled ESLint and Prettier. ESLint checks for syntax errors, formatting issues, and code style inconsistencies, and can even integrate with your IDE to visually mark problematic code. Prettier is used to enforce a consistent code style. These three options enhance the developer experience by highlighting potential issues before you run your code.

Then, following the instructions, you can move into the created folder and type `npm install` to install the required dependencies. This will download the required package files from the `npm` registry and install them in the `node_modules` subfolder of your project.

If you run `npm run dev` the project will spin up a development server, which you can access with your browser. Usually, the local address will be something similar to `http://127.0.0.1:5173/`.

If you open that URL in the browser, you should see your first Vue.js application! The default is an empty starter that holds many of the pointers and links that we've already covered at this point, but it is a good starting point for any developer starting with Vue.js.
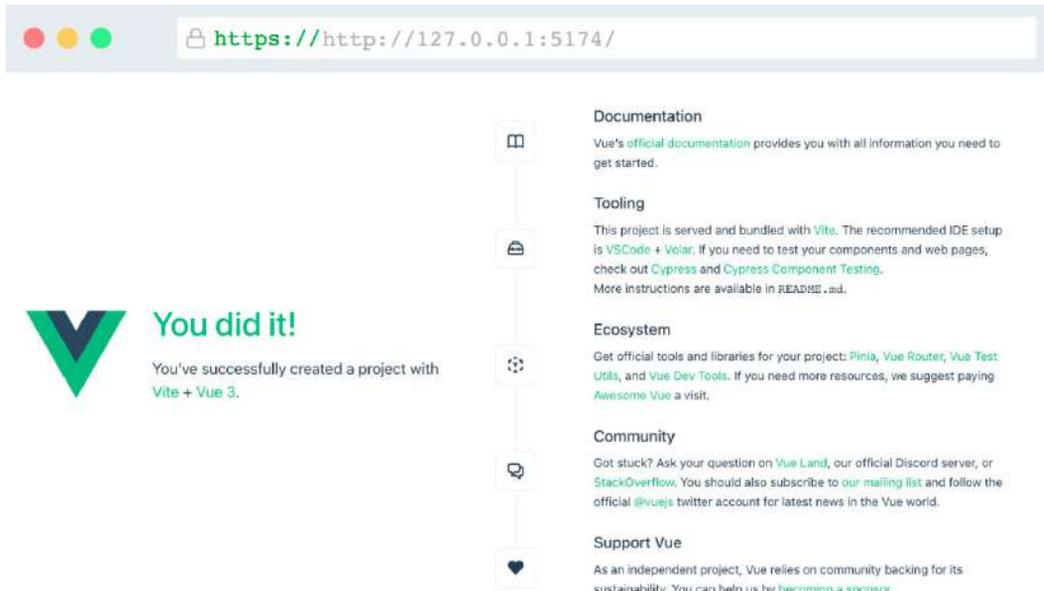
Figure 1.2 – Your very first Vue.js app!

With successful installation, we can take a closer look at what has actually been installed. Let's dive into the installation files!

The project in the IDE

Now, if you open up the project in your IDE of choice, you will notice a predetermined structure. This will be applicable to all the projects that are scaffolded out this way. Let's take a quick look at the structure:
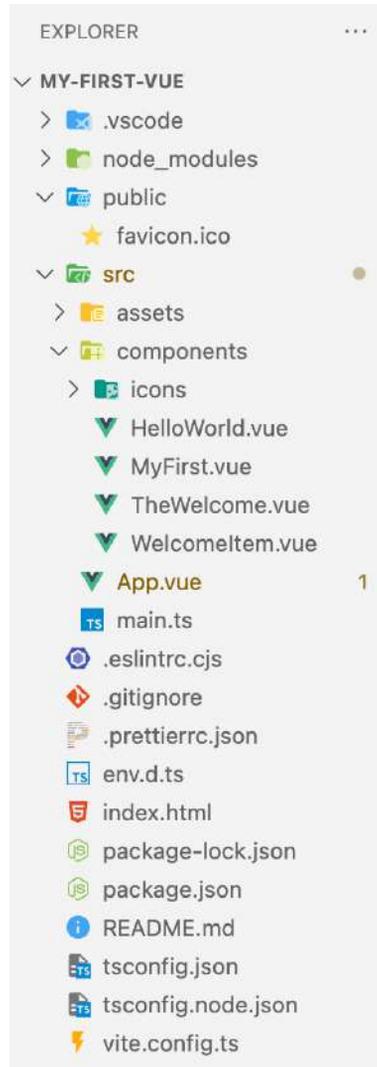
Figure 1.3 – The expanded folder structure of the starter app

At the root of the project, you'll find different types of files that are specific to configuring the project. The main files here are `index.html` and `package.json`. The `index.html` file is the entry point of the app. It is a lightweight HTML template with a `div` element that has the `id` app will be the mounting point of the application.

The `package.json` file is a file that describes the project as a package, defines node scripts that can be executed, and also holds the references to all packages that the project depends on. The `node_modules` folder is the folder that holds all of the installed packages from the `package.json` file. This can be considered a read-only folder for our purposes.

Then we have the `public` and `src` folders. The `public` folder contains static assets, such as fonts, images, and icons that do not have to be processed by the build system. In the starter project, you will find a default `favicon.ico` present.

Lastly, the `src` (short for source) folder is the folder in which we will be making the most changes. It contains two root files at this point. The `main.ts` file registers the Vue application and the styles and mounts it to the HTML template.

The `App.vue` file is the entry point of the Vue.js application. If you open it, you might find some familiar syntaxes mixed in a single file, such as script tags, HTML, and CSS. We will get to that later.

It also contains an `assets` folder similar to the `public` folder, with the difference that these folders can and will be processed by the build system. And finally, there is the `components` folder, where you can place the components that make up the application. If you adopt Single File Components, each will fulfill a specific role and encapsulate the template, script, and styles. You can already see a couple of components, which make up the default starting page.

Your first coding steps

Let's create the first component and add it to the app:

1. Create a new file in the `components` folder called `MyFirst.vue`.

    A Vue.js component is preferably named using at least two camel-cased words and typically consists of a `script`, `template`, and `style` block. None of these are mandatory (although a `style` block without context would be of little value).

2. Let's create a small HTML snippet:

    ```
    <template>
        <div>My first <span>Vue.js</span> component!</div>
    </template>
    ```

3. In `App.vue`, you can use this as a Vue.js component already! If you open it up, you will see a `script` tag with `import` statements. You can remove the `TheWelcome` import line and replace it with this:

    ```
    import MyFirst from './components/MyFirst.vue'
    ```

4. Next, in the `template` tag, you can remove the `<TheWelcome />` HTML-like tag and replace it with the `<MyFirst />` HTML notation.

    If you were still running the code, you would have noticed that the browser updated itself to reflect the changes. This is called hot reloading and makes for a smooth development flow. If you have stopped the process, you can restart it and revisit the page in the browser.

    You should see the component that you've created!

5. Let's add a styling block to add some CSS to the component and see the hot reloading in action. In the `MyFirst.vue` file, add the following code below the `template` block:

```
<style scoped>
div {
  color: #35495f;
  font-size: 1.6rem;
}

span {
  color: #41b883;
  font-weight: 700;
}
</style>
```

    The contents of the style block will be processed like a normal CSS file. The `scoped` attribute means that the `div` and `span` style definitions are scoped down to only this component. Vue adds a unique data attribute to the virtual DOM and attaches the CSS rules to that attribute. In `App.vue`, you can see that global styles are also supported.

In the browser, you will see the component update itself with the new styling! Now that we're familiar with the development environment, we'll start to create a more interactive component in the next chapter.

Summary

At this point, you are ready to get started with Vue.js development. We've worked on setting up our local environment, and we used the recommended way of scaffolding out a new starter project.

In the next chapter, we'll take a closer look at some of the Vue.js concepts (such as reactivity) and learn to inspect our app using official tools. Each lesson will introduce new concepts to apply in development.

2
Creating a Todo List App

Now that we have a development environment set up, we will get started with writing a small first application. In this chapter, we will create a Todo list app, which will teach us how the reactivity of Vue.js and the virtual DOM works. You can use the Todo list app as a guide to track progress in this book!

Let's make it an assignment with some practical requirements:

- We will make sure that you see a list of items

- Each item will have a checkbox

- The list will be sorted by unchecked items first and checked items second

- The status of an item should be preserved by the browser on future visits

There are different ways of writing a valid Vue.js component. Currently, the Composition API is preferred over the Options API. The Options API uses an object-oriented approach, while the Composition API allows for a more reusable way of writing and organizing your code.

In this book, we will use the Composition API notation with shorthand for the setup function unless noted otherwise. This way of writing code removes lots of noise and repetitive operations from your components and is a very efficient way of working. We'll also use the TypeScript variant since it is supported out of the box and offers a better **Developer eXperience** (**DX**) by facilitating strict typing.

> **Note**
>
> You can read more about the syntax here: `https://vuejs.org/api/sfc-script-setup.html#script-setup`. More about defining components using TypeScript can be found here: `https://vuejs.org/guide/typescript/composition-api.html#using-script-setup`.

We will cover the following topics in this chapter:

- Using the CLI tool to create a custom environment for an app

- Vue.js reactivity concept

- Styling with CSS

- First glance at Vue.js DevTools

We have no technical requirements other than what we've covered in the previous chapter, so we can get started right away!

A new project

Let's start by scaffolding out a new project, using the CLI commands from the previous chapter. Open a Terminal window in your projects folder and use the following instructions:

```
npm init vue@latest
```

Hit *y* to proceed, use vue-todo-list as the project name, and select the options shown in the following screenshot:
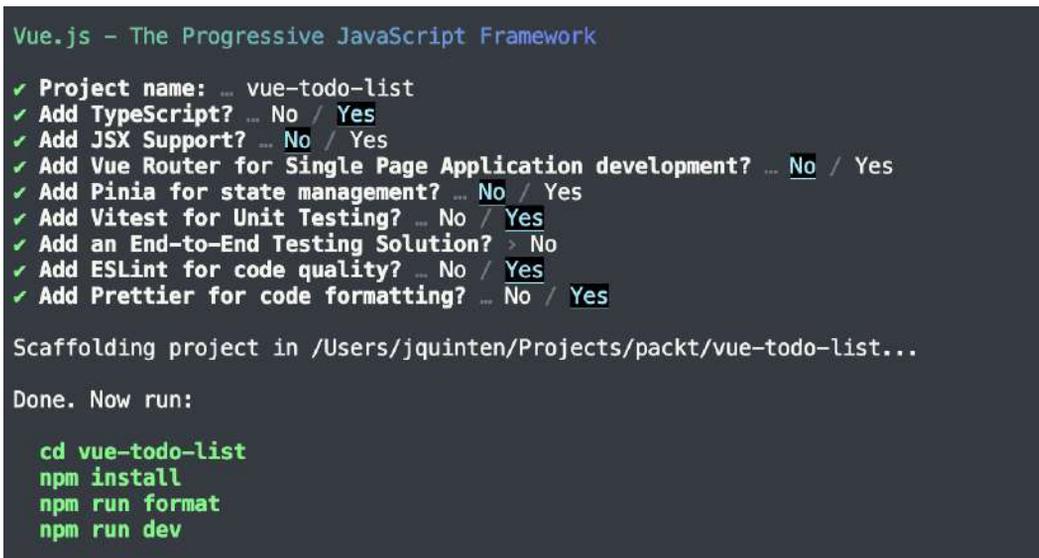
```
Vue.js - The Progressive JavaScript Framework

✔ Project name: … vue-todo-list
✔ Add TypeScript? … No / Yes
✔ Add JSX Support? … No / Yes
✔ Add Vue Router for Single Page Application development? … No / Yes
✔ Add Pinia for state management? … No / Yes
✔ Add Vitest for Unit Testing? … No / Yes
✔ Add an End-to-End Testing Solution? › No
✔ Add ESLint for code quality? … No / Yes
✔ Add Prettier for code formatting? … No / Yes

Scaffolding project in /Users/jquinten/Projects/packt/vue-todo-list...

Done. Now run:

  cd vue-todo-list
  npm install
  npm run format
  npm run dev
```

Figure 2.1 – The setup configuration for the Todo list app

Go ahead and follow the given instructions to install the dependencies and open your favorite IDE to get started.

> **Tip**
>
> npm offers a shorthand for installing, by typing npm i instead of npm install. Read more about npm commands here: https://docs.npmjs.com/cli/v6/commands.

Cleaning up the default installation

Let's first clean up the components folder by removing HelloWorld.vue, TheWelcome.vue, WelcomeItem.vue, and the icons folder. Then we'll remove the references from App.vue and clean up the template.

You will see a __tests__ folder in the components folder, which is added by installing Vitest. You can ignore it for now. Otherwise, the components folder should be empty.

The App.vue file should look like this:

```
<script setup lang="ts">
</script>

<template>
</template>

<style scoped>
… (truncated, unchanged)
</style>
```

The changes will result in a blank page since we've removed all the default elements! Now we can start to build our own application from the ground up.

Building up the app

In this chapter, we will add a couple of components and compose the Todo app to follow the requirements, as listed at the beginning of this chapter (see *Technical requirements*). We'll add the features step by step.

Let's start simply, with an AppHeader component. Create the AppHeader.vue (remember: Vue. js recommends a filename that consists of at least two camel-cased words) file in the components folder. This will just be a static component with a template and css block:

```
<template>
  <header>
    <h1><span class="icon" aria-hidden="true">✅</span> To do</h1>
    <p>Building Real-world Web Applications with Vue.js 3</p>
  </header>
</template>
```

```
<style scoped>
header {
  border-bottom: #333 1px solid;
  background-color: #fff;
}

header::after {
  content: "";
  display: block;
  height: 1px;
  box-shadow: 0px 0px 10px 0px rgba(0, 0, 0, 0.75);
}

h1 {
  font-size: 2rem;
}

h1 .icon {
  font-size: 1rem;
  vertical-align: middle;
}
</style>
```

A Vue component usually consists of template, script, and style blocks. Not all are mandatory (we're not adding any scripts to this component), but between those three, we can define every aspect of our component. In this book, we'll see numerous examples of this pattern in practice.

The template is just a title representation, and we'll use the `scoped` style block to apply CSS rules to the markup. Note the `scoped` attribute, which makes sure that our CSS doesn't affect other components in the application.

Using scoped CSS, we can write clean, readable rules. For single file components, this approach should be the default.

Let's continue building our app by creating a list component.

Creating the ListItem component

We'll create the `ListItem.vue` component in the same folder. It will be the representation of an individual item on the list, and it looks like this:

```
<script lang="ts" setup>
defineProps<{
  isChecked?: boolean | false
```

```
}>()
</script>

<template>
  <label :class="{ 'checked': isChecked }">
    <input type="checkbox" :checked="isChecked" />
    <slot></slot>
  </label>
</template>

<style scoped>
label {
  cursor: pointer;
}

.checked {
  text-decoration: line-through;
}
</style>
```

We define the props that will be passed to the component. Props are the properties that can be controlled from outside of the component. These are usually values that are being processed by the component and determine the unique characteristics of the component in that state. In rare cases, you could pass down a function as well.

By using the `defineProps` method, we're using the Vue.js API to declare our props properly.

The second bit is the way that the component should render HTML to the virtual DOM. Vue.js uses a syntax that is HTML-based. You can read more about it here: `https://vuejs.org/guide/essentials/template-syntax.html#template-syntax`.

We're marking up an HTML `<label>` tag with a dynamic class name: it will render as `class="checked"` when the `isChecked` property evaluates as `true`. In the label, we'll add a checkbox that has a dynamic `checked` attribute: it too is connected to the `isChecked` prop. The `<slot></slot>` tag is Vue.js-specific and it allows us to put any content in that spot, from the parent component.

Lastly, we define the CSS rules for this component, similar to what we did with `AppHeader.vue`.

Creating the list

With the `ListItem` component available to us, we can start generating the list. We'll create a new component for this, which will hold the list information and use it to render all the individual items on the list as well as to provide the interactive features:

1. Let's create a simple file called `TodoList.vue`, with the following contents:

    ```
    <script lang="ts" setup>
    import ListItem from './ListItem.vue'
    </script>


    <template>
    <ul>
      <ListItem :is-checked="false">This is the slotted content</
    ListItem>
    </ul>
    </template>
    ```

2. Before we continue, we want to be able to display our app while we're working on it. So, on the `App.vue` file, follow a similar approach of importing the `AppHeader.vue` and `TodoList.vue` files and adding the components to the template:

    ```
    <script setup lang="ts">
    import AppHeader from './components/AppHeader.vue';
    import TodoList from './components/TodoList.vue';
    </script>


    <template>
      <AppHeader />
      <TodoList />
    </template>


    <style scoped>
    … (truncated)
    </style>
    ```

Now that we can see what we're doing, it is a good time to start the development server. If you're using **Visual Studio Code** (**VSCode**), there's actually a terminal built into the IDE:

- **macOS users**: ^ + `
- **Windows users**: *Ctrl* + `
- **Linux users**: *Ctrl* + *Shift* + `

If you run the `npm run dev` command, it will start the development server and provide you with a local preview URL.
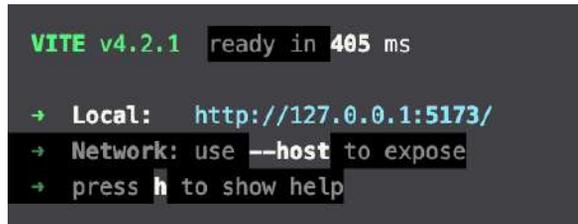
Figure 2.2 – Output of the npm run dev command

Since we don't have a functional app right now, we need to work on its core feature: the list. You can leave the development server running since it will automatically update (this is called hot reloading) with the newly written code.

Making a list

The actual functionality resides in the `TodoList.vue` component, which we'll now create. We'll start small and add the more complex features in steps. Let's start with a static list that has multiple list statuses.

Let's look at the `script` block first. Apart from importing the `ListItem` component, we're defining type for `Item`, which consists of the `title` property as a string and the optional property `checked` as a Boolean. TypeScript lets us define a `Type` alias, which our IDE can plug into when interacting with `Type`.

In this example, when accessing the properties on `item` in the `ListItem` template, the IDE already recognizes the `title` and optional `checked` properties:

```
<script lang='ts' setup>
import ListItem from './ListItem.vue'

type Item = {
  title: string,
  checked?: boolean
}

const listItems: Item[] = [
  { title: 'Make a todo list app', checked: true },
  { title: 'Predict the weather', checked: false },
  { title: 'Play some tunes', checked: false },
  { title: 'Let\'s get cooking', checked: false },
```

```
    { title: 'Pump some iron', checked: false },
    { title: 'Track my expenses', checked: false },
    { title: 'Organize a game night', checked: false },
    { title: 'Learn a new language', checked: false },
    { title: 'Publish my work' }
  ]
</script>
```

When constructing the `ListItems` array, we assign `Type` as an array of that type using the `[]` symbols. We immediately fill the `ListItems` array with a list of items. This means that TypeScript could also infer the types, but it is a better practice to explicitly set the types where possible.

In the template, we are creating an unordered list element and use the `v-for` directive to iterate over the items in the array:

```
<template>
  <ul>
    <li
      :key='key'
      v-for='(item, key) in listItems'
    >
      <ListItem :is-checked='item.checked'>{{ item.title }}</ListItem>
    </li>
  </ul>
</template>
```

The `v-for` directive is used to loop over collections and repeat the template that marks the collection. For every item, the current value is assigned to the first argument (`item`) and optionally provides an index of the collection as the second argument (`key`).

The `v-for` directive repeats the `<li>` item with a `<ListItem />` component enclosed. For every item, we fill the `<ListItem />` component with the `is-checked` and `title` properties for that item.

The `key` attribute helps Vue.js in keeping track of the changes that are being made so that it can update the virtual DOM more efficiently.

Lastly, we have added a `scoped` style block to stylize the elements for the browser. There's not much going on here:

```
<style scoped>
ul {
  list-style: none;
}

li {
```

```
  margin: 0.4rem 0;
}
</style>
```

We now have a non-interactive Todo list app and have met the first two requirements already. Let's take a look at how we can add some interactivity.

Reactivity explained

If you have opened the app and clicked an item, you can toggle the checkbox, but on refreshing the page, nothing happens. Also, if you've looked closely at the CSS of the `<ListItem />` component, you may have noticed that strikethrough styling should be applied on a checked item. This is only the case on the first item.

The toggling of the checkbox is, in fact, native browser behavior and doesn't signify anything in the context of the state of the Todo list!

We need to wire the changes in the UI to the state of the application. In order to get started, we need to import some utilities from the Vue.js package. Add these two lines to the top of the `<script>` block:

```
import { ref } from 'vue'
import type { Ref } from 'vue'
```

The `ref` function is used to add reactivity and track updates to certain parts of the code. The value of `ref` is automatically inferred by TypeScript, but for complex types, we can specify the type.

> **Note**
>
> Vue.js also offers a `reactive` utility to mark reactivity. There are slight differences between the two, where `ref` can be used to track primitives and objects and `reactive` can only be initialized with an object. In general, you can opt for consistency in code by choosing `ref` over `reactive`. The only *downside* is that you have to access the value of the reactive item by a `.value` property in the script block. When using the variable in the `template` block, it is automatically unwrapped by Vue.js. It's therefore a small concession for being able to use `ref` consistently.

Now that we have imported the utility, we can mark the `listItems` to be tracked by wrapping the contents in the `ref` function:

```
const listItems: Ref<Item[]> = ref([
  { title: 'Make a todo list app', checked: true },
  { title: 'Predict the weather', checked: false },
  { title: 'Play some tunes', checked: false },
  { title: 'Let\'s get cooking', checked: false },
```

```
    { title: 'Pump some iron', checked: false },
    { title: 'Track my expenses', checked: false },
    { title: 'Organise a game night', checked: false },
    { title: 'Learn a new language', checked: false },
    { title: 'Publish my work' }
])
```

Note that capitalized `Ref` is used to type the value and lowercase `ref` is used as a wrapper of the array of items. If we now want to access the values in the script block, we need to access them by using `listItems.value`.

Now that the `listItems` are reactive, the virtual DOM will respond automatically to changes on the variables. We can add a method that changes an item so that it will be reflected in the user interface.

Let's add the following function to the `script` block:

```
const updateItem = (item: Item): void => {
  const updatedItem = findItemInList(item)
  toggleItemChecked(updatedItem)
}

const findItemInList = (item: Item): Item | undefined => {
  return listItems.value.find(
    (itemInList: Item) => itemInList.title === item.title
  )
}

const toggleItemChecked = (item: Item): void => {
  item.checked = !item.checked
}
```

Adopting Robert C Martin's Clean Code philosophy, I split the instruction into separate functions with their own clear intent. When calling `updateItem` with `item` as an argument, it will try to find it in the `itemList` and toggle the `checked` property on the object.

We can see that TypeScript is guiding us to a slightly better solution: because `findItemInList` could return an `undefined` value and `toggleItemChecked` expects a parameter, the argument of calling the `toggleItemChecked` function gets a squiggly line.

Figure 2.3 – TypeScript hinting at a possible problem in our code

We can fix this by adding a statement surrounding the call of the `toggleItemChecked` function:

```
const updateItem = (item: Item): void => {
  const updatedItem = findItemInList(item)
  if (updatedItem) {
    toggleItemChecked(updatedItem)
  }
}
```

With the `script` block changes complete, we can attach the interactivity to the user interface in the `template` block. We want the visitor to be able to click on a `ListItem` to mark it as complete. Vue.js has a built-in directive for this: `v-on`. This acts as an event handler and supports a couple of modifiers as well. For more information, see `https://vuejs.org/api/built-in-directives.html#v-on`.

We can add it to the template like this:

```
<ListItem :is-checked='item.checked' v-on:click.
prevent="updateItem(item)">{{ item.title }}</ListItem>
```

We've also added a `.prevent` modifier to prevent the default behavior of the checkbox mechanics. This is all the code that's needed to call the method!

There's even a shorthand for `v-on:click`, by using `@click`. You will see examples of both directives in resources, so it's good to understand they are the same.

Under the hood, Vue.js uses the `ref` function to register a series of watchers on the values. The template engine is used to generate a virtual DOM (a node tree representation of the elements that make up the component). Once a reactive value is changed, the virtual DOM node where that value is used is also changed.

Vue.js compares the changes to the DOM and updates only the necessary elements in the real DOM to reflect the state. Being able to granularly update the DOM very accurately is what makes Vue.js 3 a highly performant framework, since it doesn't have to traverse whole branches of virtual DOM nodes!

Let's use the list that we have as input for our next step, where we'll look at sorting.

Sorting the list

We're now perfectly capable of showing variables in the template. There are cases, however, when you have a need for more advanced expressions, such as, in our example, the requirement of sorting the list. For variables that have no side effects and include reactive data, you can use the Vue.js `computed` function.

Typically, you would use `computed` for filtering data, format expressions, displaying calculations, or Boolean conditionals. Let's apply it to sort the list with completed items at the bottom.

First, we'll import `computed` to the `TodoList` component. We can add it to the import where we also import the `ref` function:

```
import { ref, computed } from 'vue'
```

The `computed` function is very similar to `ref`, in the sense that it follows the same reactivity in updating the DOM when the value changes and you can even access the value using the `.value` property in the `script` block! The main difference is that a `computed` value is cached and only updates when one of the inputs changes.

For sorting the list, we can use the `listItems` as input and apply a simple JavaScript sorting function on the array. We can just add this line to define the `computed` value:

```
const sortedList = computed(() =>
    [...listItems.value].sort((a, b) => (a.checked ? 1 : 0) -
(b.checked ? 1 : 0))
)
```

As you can see, `computed` is a function that gets called on a change of the reactive value. In this case, `listItems.value`. We'll simply apply a `sort` function to the collection.

In the template, we can now swap out `listItems` for the `sortedList` variable and you will see that checked items will be placed below the unchecked items.

Preserving changes to the list

We have a final requirement to achieve now, and that is to preserve the state of the list on reloading and revisiting the app. We'll keep it as simple as we can for now and use the `localStorage` API of the web browser to store and retrieve the state of the list.

We'll first add the functions that we can use to write to `localStorage` and retrieve from `localStorage`:

```
const setToStorage = (items: Item[]): void => {
  localStorage.setItem('list-items', JSON.stringify(items))
}

const getFromStorage = (): Item[] | [] => {
  const stored = localStorage.getItem('list-items')
  if (stored) {
    return JSON.parse(stored)
  }
  return []
}
```

These two functions interface with browsers' abilities to store a string of data, so we need to stringify and parse that object. We're storing the data on the `list-items` key.

Now we need to make sure we try and retrieve the data when the component gets loaded. There's a function for it, called `onMounted` and it is part of the Vue.js core, so we can import it in a similar fashion to the `ref` and `computed` functions.

The `onMounted` function is what we call a life cycle hook. They are functions that get called at certain points in the *life cycle* of a component. The main life cycle events are triggered when a component gets mounted (or before), gets updated (or before), gets unmounted (or before), and gives an error. More information can be found here: `https://vuejs.org/api/composition-api-lifecycle. html#composition-api-lifecycle-hooks`.

In our case, we want the list to be retrieved in the browser when the component gets rendered (it would otherwise have no access to `localStorage`). So, we'll import the function:

```
import { ref, onMounted, computed } from 'vue'
```

And we need to create a reactive variable to hold the items:

```
const storageItems: Ref<Item[]> = ref([])
```

We'll also create a function (`initListItems`) that will run once when mounted, and move the initialization of the `listItems` there. We'll also make a change to the declaration of the `listItems` by wrapping it with a check on the existence of `storageItems`. If they do not exist, we will use the `listItems` as default and write the contents to `localStorage`:

```
const initListItems = (): void => {
  if (storageItems.value?.length === 0) {
    const listItems = [
      { title: 'Make a todo list app', checked: true },
      { title: 'Predict the weather', checked: false },
      { title: 'Read some comics', checked: false },
      { title: 'Let\'s get cooking', checked: false },
      { title: 'Pump some iron', checked: false },
      { title: 'Track my expenses', checked: false },
      { title: 'Organise a game night', checked: false },
      { title: 'Learn a new language', checked: false },
      { title: 'Publish my work' }
    ]

    setToStorage(listItems)
    storageItems.value = listItems
  }
}
```

Now, we add the following functions to retrieve any locally stored list items:

```
onMounted(() => {
  initListItems()
  storageItems.value = getFromStorage()
})
```

In order to keep the changes in sync, we can now modify the `findItemInList` function to look in the `storageItems` collection rather than `listItems` and also write the change to the storage after the item has been updated. We'll modify the `updateItem` and `findItemInList` functions as follows:

```
const updateItem = (item: Item): void => {
  const updatedItem = findItemInList(item)
  if (updatedItem) {
    toggleItemChecked(updatedItem)
    setToStorage(storageItems.value)
  }
}

const findItemInList = (item: Item): Item | undefined => {
  return storageItems.value.find(
    (itemInList: Item) => itemInList.title === item.title
  )
}
```

Now, in the template, we're using a computed value, so we should update the computed function too in order to see `localStorage` as the input for our data:

```
const sortedList = computed(() =>
  [...storageItems.value].sort((a, b) => (a.checked ? 1 : 0) -
(b.checked ? 1 : 0))
)
```

We've seen how we can use different components with specific uses to build a simple reactive app and how we can organize our code with readability and maintainability in mind. Vue.js encourages using Single File Components to structure your code.

Single File Components

The way that we have organized the app, with individual components having a single feature to fulfill is referred to as the **Single File Components** (**SFC**) philosophy.

This approach is designed to enhance code readability, maintenance, and reusability. With SFC, you can create reusable and modular components that can be easily shared and reused across different projects.

To be fair, we did cut some corners with the `TodoList.vue` component, since we could have abstracted the getting and setting of the `listItems` to a different component. For the sake of this example, however, it illustrates the capabilities in an acceptable way. There are no strict rules or guidelines for how you structure your components.

Note that you can structure or restructure the contents of the script block in a way that makes sense to you. You have the freedom to group related sets together, which makes for very readable code that's easy to refactor.

The Vue.js DevTools

If you've not yet installed the Vue.js DevTools, please refer back to the *Vue.js DevTools* section in *Chapter 1*, *Introduction to Vue.js*, to follow the instructions. We will take a close look at the DevTools using our Todo list application for reference.

If you have the browser plugin installed and you visit a website where Vue.js is detected, the icon in the toolbar will indicate that Vue.js is detected on that particular URL:
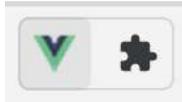
Figure 2.4 – Screenshot of Vue DevTools in the browser's toolbar

If you click it, it will refer you to opening the browser's DevTools, where a tab dedicated to Vue is added.

The **Vue.js** tab offers a lot of ways of drilling down into a certain aspect of the rendered code and some time travel inspection methods. It offers an accessible representation of the inputs and outputs of a component, which can help you visualize how a component is rendered.

So, let's zoom in on a particular element, by using the inspect mode.

Inspecting a component

Let's see if we can inspect a `ListItem` component. We have several ways of doing this: we can drill down into the DOM tree in the Vue.js panel, we can filter for the component name, and we can use the crosshair button to point out the component on the page.

Figure 2.5 – Drilling down into the DOM tree in the Vue.js panel

In *Figure 2.6*, we'll use the filtering option to type the name of the component we want filtered. This works well when you're not exactly sure what the structure of the application looks like.
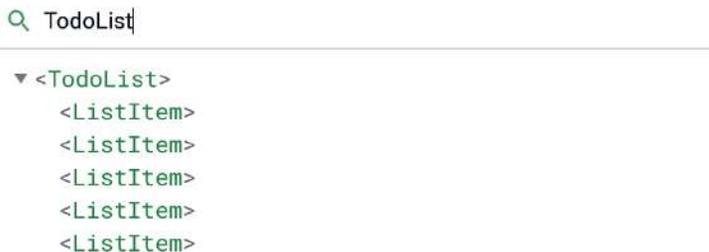
Figure 2.6 – Filtering for the component name

In *Figure 2.7*, we use the crosshair icon to select the element from the browser's viewport. This works very well when you have a strong visual reference to a component!
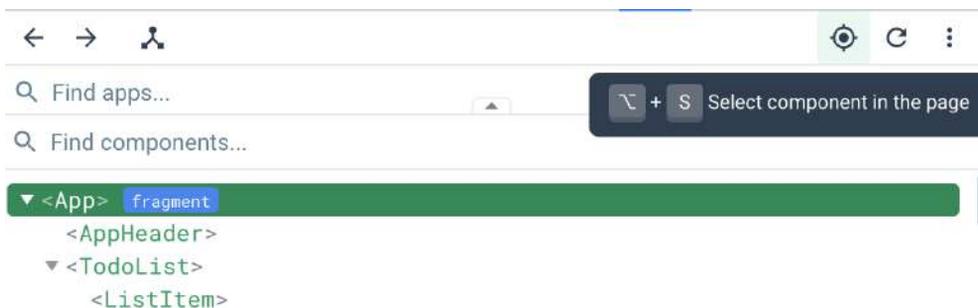
Figure 2.7 – Using the crosshair to point out a component on the page

Depending on your use case, you may prefer one method over the other. For this example, feel free to try all of them out to see their effect.

When you click in the component, you will see additional details, such as the props, extract of the setup function, event listeners, and the `onClick` event we registered with the `v-on` (or `@`) directive.

Apart from inspecting the props that the component was given, we can use the control buttons to scroll to and highlight the component on the page, inspect the render function for that component, highlight the generated DOM code, and even open the source file in the code editor!

Figure 2.8 – The various controls of Vue DevTools in the browser extension

Those several ways of inspecting components are useful tools when debugging the state of a component. What makes them especially powerful is that you're looking at the component from within the browser's environment, which is also how users of your app experience and interact with your application!

Manipulating a component

Apart from inspecting, we can also manipulate the state of a component. We can't modify the properties of a `ListItem`, since it's read-only. Let's take a look at the `TodoList` component.

If you inspect it, you'll see two collections that power the list: the `sortedList` and `storageList` variables.

Figure 2.9 – The collections that power the contents of the list

Again, `sortedList` is a computed property and cannot be manipulated.
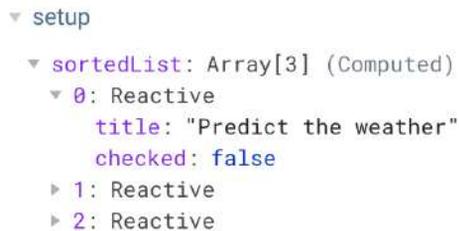
Figure 2.10 – The values of the computed sortedList items cannot be modified

When we look at `storageList` and expand the collection, we see some modifiers. We can toggle the `checked` property and update the `title` property. Those changes even propagate to the values of the corresponding `sortedList`!

```
▼ setup
  ▼ sortedList: Array[3] (Computed)
    ▶ 0: Reactive
    ▶ 1: Reactive
    ▼ 2: Reactive
        title: "Predict the weather"
        checked: true
  ▼ storageItems: Array[3] (Ref)
    ▶ 0: Object
    ▼ 1: Object            Quick edit
        checked: true ✏ ☑      🗑  ⋮
        title: "Predict the weather"
    ▶ 2: Object
```

Figure 2.11 – The values of the storageItems items can be modified
and propagate to the corresponding sortedList values

With the browser being dependent on the computed value, it means you see the effect in the browser as well. This is very useful for debugging different variants of the state of the user interface. You can also see the methods that are used in the component, available for inspection.

In other scenarios, we will touch upon different uses of Vue DevTools so you will slowly get more familiar with using them to more accurately debug or inspect the applications you build. When debugging any application state that affects the browser, Vue DevTools offers a very good set of features to help you analyze what is happening with the rendering of the application.

Taking a look at the application, you'll notice that the first item on our Todo list is checked, which now accurately represents the progress we've made. Let's work on checking off the next items on the list!

Summary

At this point, we've used the Vue CLI tool to create and customize our app boilerplate settings. We've been using two-way data binding, which translates to the reactivity in our applications. Using and applying the Single File Components philosophy, we can now apply this to build applications that are maintainable at any scale.

With Vue DevTools, we have learned a means of inspecting components and can apply this to debug our applications.

In the next chapter, we'll connect our application with external APIs, giving it real-time data to work with.

3

Building a Local Weather App

Now that we can build a small app, we can add a bit more complexity. In this case, we'll take a look at including another browser API and combining it with an external data source for our app. We will build a small weather app that returns the current weather.

We will start applying a different means of styling, using Tailwind as our CSS framework of choice, and in order to provide some additional robustness, we will also take a look at including some tests in our application.

We'll cover the following topics in this chapter:

- Working with external data coming from different types of APIs

- How to handle asynchronous data

- Applying Tailwind to quickly style any application

- Ensuring stability by adding unit tests for features

Let's see what requirements we have to fulfil to get our application up and running.

Technical requirements

For this chapter, we are going to use a third-party API to provide us with actual data. We need to register an account at `https://www.weatherapi.com/` and retrieve the API keys for usage in our app.

We'll add Tailwind CSS to apply styling to our app. The `https://tailwindcss.com/` website offers extensive documentation as well as an installation guide.

For our unit test, we'll use the Vitest framework: `https://vitest.dev/`.

You can find the source code here: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/tree/main/03.weather`.

Initializing the app

Let's begin by starting with a slightly different configuration for the Vue.js CLI starter:

```
npm init vue@latest
```

Hit y to proceed, choose vue-local-weather as the project name, and select the options shown in the following screenshot:
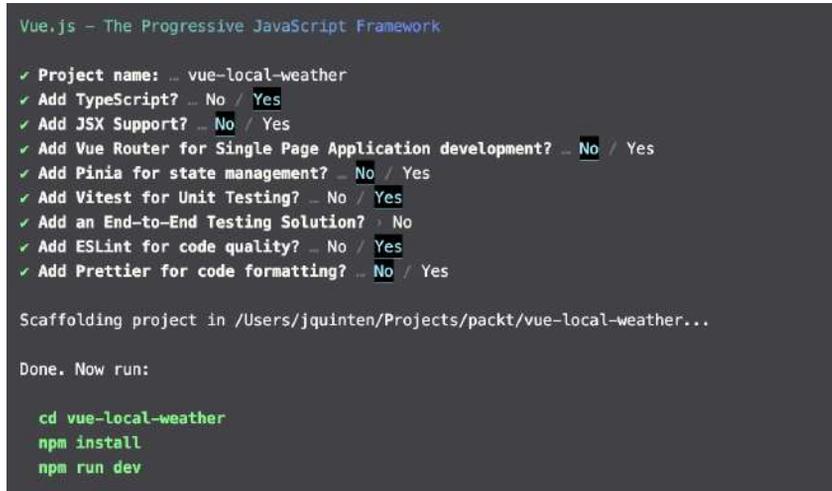
Figure 3.1 – The setup configuration for the local weather app

After following the instructions to install the dependencies and cleaning up the default files, we can get to work!

Working with different types of APIs

In order to retrieve local weather, we need a way to retrieve a location. The weather service we will be using accepts different sorts of location data, but we'll go with latitude and longitude for this example.

It's convenient that the browser's geolocation API can provide us with just that! Let's start by building a component that requests this information and displays it to the user interface.

Let's create a file in the components folder, called GetLocation.vue. We'll start in the script tag by importing the utilities from Vue.js and define the data that's expected to be available:

```
<script lang="ts" setup>
import { ref } from "vue";
import type { Ref } from "vue";
```

```
type Geolocation = {
  latitude: number;
  longitude: number;
};

const coords: Ref<Geolocation | undefined>= ref();
</script>
```

Now, we're saying that we will expect the reactive property `coords` to contain a latitude and longitude. Nothing fancy. Let's write a function that retrieves the data from the geolocation API (`https://developer.mozilla.org/en-US/docs/Web/API/Navigator/geolocation`). Now, bear in mind that a user can deny access to this API, so we need a fallback as well.

We'll add a reactive Boolean property, `geolocationBlockedByUser`, to keep track of the success of calling the API and a function that does the actual calling:

```
<script lang="ts" setup>
import { ref } from "vue";
import type { Ref } from "vue";

type Geolocation = {
  latitude: number;
  longitude: number;
};

const coords: Ref<Geolocation | undefined> = ref();
const geolocationBlockedByUser: Ref<boolean> = ref(false);

const getGeolocation = async (): Promise<void> => {
  await navigator.geolocation.getCurrentPosition(
    () => {},
    (error: { message: string }) => {
      geolocationBlockedByUser.value = true;
      console.error(error.message);
    }
  );
};
</script>
```

There's a couple of things going on here. The `getGeolocation` function is being defined and, because it is dependent on user input, it is an asynchronous function by default. The promise it returns is empty because we use `successCallback` to update our reactive property.

That bit is empty right now, but we'll add it in the next step. The `errorCallback` function only gets called if the geolocation cannot be retrieved, and for now, we'll just assume that the user declined usage. So, we set the `geolocationBlockedByUser` value to true and log the error to the console.

Looking at the documentation (`https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API/Using_the_Geolocation_API#getting_the_current_position`), we see that `getCurrentPosition` returns an object (position) that holds latitude and longitude on a `coords` property. Since our `coords` reactive property expects a latitude and longitude, we process the data from the API as follows:

```ts
const getGeolocation = async (): Promise<void> => {
  await navigator.geolocation.getCurrentPosition(
    async (position: { coords: Geolocation }) => {
      coords.value = position.coords;
    },
    (error: { message: string }) => {
      geolocationBlockedByUser.value = true;
      console.error(error.message);
    }
  );
};
```

This all works now, but the function hasn't been executed yet. Like in the previous chapter, we'll use the `onMounted` hook to execute the function once the component gets mounted on the DOM. The entire `script` tag should now look like this:

```ts
<script lang="ts" setup>
import { ref, onMounted } from "vue";
import type { Ref } from "vue";

type Geolocation = {
  latitude: number;
  longitude: number;
};

const coords: Ref<Geolocation | undefined> = ref();
const geolocationBlockedByUser: Ref<boolean> = ref(false);

const getGeolocation = async (): Promise<void> => {
  await navigator.geolocation.getCurrentPosition(
    async (position: { coords: Geolocation }) => {
      coords.value = position.coords;
    },
    (error: { message: string }) => {
```

```
      geolocationBlockedByUser.value = true;
      console.error(error.message);
    }
  );
};

onMounted(async () => {
  await getGeolocation();
});
</script>
```

Let's quickly add a template block that renders the output like this:

```
<template>
  <div v-if="coords && !geolocationBlockedByUser">{{ coords.latitude
}} {{ coords.longitude }}</div>
  <div v-if="geolocationBlockedByUser">User denied access</div>
</template>
```

Adding the component to App.vue is a matter of importing the component and rendering it on the template:

```
<script setup lang="ts">
import GetLocation from "./components/GetLocation.vue";
</script>

<template>
  <GetLocation />
</template>
```

Start our development server with npm run dev. Now, if you open the app in the browser, you should expect to see a browser popup asking for permission to share your location. If you grant access, you should see coordinates that your browser determined as your location (results may vary). If you have denied access, you should see the message stating that you did.

> **Note**
>
> If you inspect the console closely, you may notice a warning: **Only request geolocation information in response to a user gesture**. It is typically considered an anti-pattern or rude to immediately attempt to gather geo information. Not all browsers will always show a confirmation dialog and that could lead to users disclosing information without them knowing!

The correct approach would be to add a button to the template that executes the getGeolocation function using the onClick directive. That way, a user actively initiates the request for geolocation.

Handling data from a third-party API

Now that we have our coordinates, we can start to request localized weather data. For this example, we'll make use of a public weather API (`https://www.weatherapi.com/`). In order to make use of the service, we'll need to request an API key. If you sign up for an account, the free tier will allow you to make 1,000,000 requests per month, which should be more than enough!

It is common practice to store these sorts of access keys or secrets in a local environment variables file. This practice allows our build processes to detach local development operations from our production environments. It keeps those variables in one place, rather than being spread throughout your application.

For now, we'll store the API key in a file at the root of your project called `.env`, with the following contents:

```
VITE_APP_WEATHER_API_KEY=Replace this with the key
```

The `VITE_APP_` prefix makes sure that Vite automatically exposes the variable to the application.

> **Note**
>
> With a client-based web application, the key will be exposed by default, since it will be attached to API calls that you can inspect via your browser's network requests. For our purposes, this is fine. In a production-like environment, you would likely proxy the request via your own backend to obfuscate any secrets from the public.

Constructing the API call

Having our token on hand, we can start to make the call. Let's discover how we need to construct the endpoint address to retrieve our relevant data.

Using the API explorer (`https://www.weatherapi.com/api-explorer.aspx`), we see that we can fetch data from the service using a location. While the explorer shows a place name, if we dig into the request parameters (`https://www.weatherapi.com/docs/#intro-request`), we see that the `q` parameter also accepts a latitude and longitude in decimal degrees, such as `q=48.8567,2.3508`. This is something we can use!

Looking at the docs, we need something like this:

```
https://api.weatherapi.com/v1/current.json?key=OUR_SECRET_KEY&q=OUR_
LATITUDE_AND_LONGITUDE
```

We can manually call this by pasting this endpoint as a URL in the browser, replacing the variables with our actual data. You should see a formatted JSON object with weather data for your location! Now that we're assured that everything works, we can move the logic to a Vue component to include it in our app.

Let's create a component called `WeatherReport.vue` in the `component` folder. We'll start with the `script` block, and start by describing two types that we will be working with and defining the props that this component needs (`coords`):

```
<script lang="ts" setup>
type WeatherData = {
  location: {
    localtime: Date;
    name: string;
    region: string;
  };
  current: {
    temp_c: number;
    temp_f: number;
    precip_mm: number;
    condition: {
      text: string;
      icon: string;
    };
    wind_degree: number;
    wind_kph: number;
    wind_mph: number;
  };
};

type Coords = { latitude: number; longitude: number }

interface Props {
  coords: Coords;
}
const props = defineProps<Props>();
</script>
```

For the `WeatherData` type, I've taken a look at what the API returns to us, only describing the properties that we're interested in. Feel free to pick metric or imperial units when implementing! The `Coords` type is as simple as holding numerical values for latitude and longitude and we can reuse that type within our `script` block, for instance, to describe the `coords` component property.

If we want to use the response from the endpoint, we need to make it reactive. We can do this using `ref` and we'll map it to a data constant:

```
<script lang="ts" setup>
import { ref } from "vue";
import type { Ref } from 'vue'
```

```
type WeatherData = {
  …
};

type Coords = { latitude: number; longitude: number }

interface Props {
  coords: Coords;
}
const props = defineProps<Props>();

const data: Ref<WeatherData | undefined> = ref();
</script>
```

With this in place, we're ready to define the caller function, using the fetch API. In the function, we'll take in a parameter representing the coordinates for our request. We'll return the data so we can later map it to the data property we've just created:

```
<script lang="ts" setup>
import { ref, onMounted } from "vue";
import type { Ref } from 'vue'

type WeatherData = {
  …
};

type Coords = { latitude: number; longitude: number }

interface Props {
  coords: Coords;
}
const props = defineProps<Props>();

const data: Ref<WeatherData | undefined> = ref();

const fetchWeather = async (coords: Coords): Promise<WeatherData> => {

  const { latitude, longitude } = coords;

  const q = ${latitude},${longitude};
  const res = await fetch(
    `https://api.weatherapi.com/v1/current.json?key=${
      import.meta.env.VITE_APP_WEATHER_API_KEY
    }&q=${q}`
```

```
  );
  const data = await res && res.json();
  return data;
};

onMounted(async () => {
  const { latitude, longitude } = props.coords;
  const weatherResponse = await fetchWeather({latitude, longitude});
  data.value = weatherResponse;
});
</script>
```

As you can see, we're describing a `fetchWeather` function that returns a promise in the shape of the `WeatherData` type. We use the `coords` parameter to construct the URL, combining it with the secret key. On fulfilling the request with a response, we transform it to JSON and return the value.

Similar to our `GetLocation` component, we want to fetch the data immediately, so we've used the `onMount` hook in a similar fashion. We've passed the component props to the `fetchWeather` function and mapped the response to the reactive data variable.

Now that we have our data, we can mark up the template to show the information! We're dealing with asynchronous data, so we have a UI state where data is still loading.

Let's start with adding the two states to the `WeatherReport.vue` file:

```
<template>
  <div>
    <article
      v-if="data && data.current">
      {{ data.current }}
    </article>
    <div v-else>Loading...</div>
  </div>
</template>
```

Provided you have a quick response from the server, the data should show up almost instantly.

Now let's take a look at how we can build our interface with style!

Styling with Tailwind

Tailwind CSS is a popular utility-based CSS framework that we can use to build and style user interfaces by using and combining predefined classes. Tailwind is very scalable due to abstracting the writing of CSS rules, which provides consistency and maintainability where it's used. Let's take a look at how we can apply Tailwind CSS to our little application.

The installation guide (`https://tailwindcss.com/docs/guides/vite`) covers all of the steps we need to execute:

1. First, we will have to add the dependencies to the project:

    ```
    npm install -D tailwindcss postcss autoprefixer
    ```

    We're installing Tailwind, but also the tooling to allow Vite to process the stylesheet using PostCSS. PostCSS is a powerful JS tool for transforming CSS using JavaScript (`https://postcss.org/`).

2. Next, we'll initialize the default configuration for Tailwind:

    ```
    npx tailwindcss init -p
    ```

    The command will have generated two configuration files for us (a PostCSS config file and a Tailwind config file). Out of the box, they will already help us, but let's add a good practice to the `tailwind.config.js` file:

    ```js
    /** @type {import('tailwindcss').Config} */
    export default {
      purge: ['./index.html', './src/**/*.{vue,js,ts,jsx,tsx}'],
      content: ['./src/**/*.{vue,js,ts,jsx,tsx}'],
      theme: {
        extend: {},
      },
      plugins: [],
    }
    ```

    The line that we added will remove all unused styles from the production build of our application! The pattern in the `content` property tells the plugin where Tailwind should be applied. The next step is exposing the utility classes of Tailwind to the application.

3. Create a `style.css` file in the `./src` folder and add the following lines to import Tailwind CSS utility classes in your development and build steps:

    ```css
    @tailwind base;
    @tailwind components;
    @tailwind utilities;
    ```

4. And finally, open the `./src/main.ts` file to import the CSS file into the app:

    ```ts
    import { createApp } from 'vue'
    import './style.css'
    import App from './App.vue'

    createApp(App).mount('#app')
    ```

Bear in mind that an imported CSS file differs from how you would traditionally link a CSS stylesheet to an HTML file. By importing it, the CSS file will be part of the development and build pipelines, which allows us to perform advanced operations on the styles before they are output.

In our case, we're importing references to Tailwind CSS, adding browser-specific prefixes, and removing unused classes from the stylesheet. Having this amount of control and power is very convenient for building advanced apps!

With our setup in place, we can start to apply Tailwind CSS to our application. Tailwind uses utility classes to define the styling of an element.

Utility classes

The approach of utility-class-based CSS frameworks is built around the notion of having CSS tightly coupled with a user interface while abstracting underlying rules and definitions. Instead of adding a class name to an element and then applying CSS styles within that class, you now add multiple class names to an element that describe the CSS behavior.

Having a tightly coupled relationship between an element and how it's styled has the benefit of it being very maintainable, with few hidden rules or side effects that affect how an element will be rendered. The base CSS file remains the same size; it's just the list of all utility classes.

We can even remove unused styles since we know what classes we're using in the markup. This has a higher level of predictability than more traditional CSS, where the relation between a style definition and the element that depended on the style is a lot less clear. Especially for rapid prototyping, the utility-based approach really shines, so let's put it into practice.

Let's make a small change to the `index.html` file to see Tailwind CSS in action. We're going to add a list of classes to the `<div/>` element where our app is mounted:

```
bg-gradient-to-b from-indigo-500 via-purple-500 to-pink-500 w-full
h-screen flex items-center justify-center
```

One of the strengths of Tailwind CSS is its readability. From the markup, we can visualize how the component will render in the browser. In this case, a multi-colored gradient as a background and the content at the horizontal and vertical center of the page. To learn more about the available Tailwind utility classes, the official docs provide a comprehensive list of all available classes: `https://tailwindcss.com/docs`.

The file looks like this:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
```

```
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <title>Vite + Vue + TS</title>
  </head>
  <body>
    <div id="app" class="bg-gradient-to-b from-indigo-500 via-
purple-500 to-pink-500 w-full h-screen flex items-center justify-
center"></div>
    <script type="module" src="/src/main.ts"></script>
  </body>
</html>
```

If we look at the application in the browser now, we can see the result of this change. Combining more of the utility classes, we'll stylize the data we get from the weather API.

Let's go over to `WeatherReport.vue` and add some styling and HTML elements:

```
<template>
  <div>
    <article
      v-if="data && data.current"
      class="max-w-md w-96 rounded-lg shadow-lg p-4 flex bg-white
text-black"
    >
      <div class="basis-1/4 text-left">
        <img :src="data.current.condition.icon" class="h-16 w-16" />
      </div>
      <div class="basis-3/4 text-left">
        <h1 class="text-3xl font-bold">
          {{ data.current.condition.text }}
          <span class="text-2xl block">{{ data.current.temp_c
}}&#8451;</span>
        </h1>
        <p>{{ data.location.name }} {{ data.location.region }}</p>
        <p>Precipitation: {{ data.current.precip_mm }}mm</p>
      </div>
    </article>
    <div v-else>Loading...</div>
  </div>
</template>
```

This is already looking good! See how we're combining Tailwind classes to determine how an element should be styled? You can experiment with all sorts of different ways of presenting the data.

We have a couple of predefined properties not yet mapped to the template. Let's take a look at them because they require some extra attention.

Formatting data

Let's take a look at the timestamp from the service. We're receiving it as a `datetime` string. We can format it to show up as a bit more readable information. We can just use a formatting function to format any date string into something we like. We'll create the function in the `<script>` block of the component:

```
const formatDate = (dateString: Date): string => {
  const date = new Date(dateString);
  return new Intl.DateTimeFormat("default", {
    dateStyle: "long",
    timeStyle: "short",
  }).format(date);
};
```

We're using the browser's built-in `Intl` namespace (`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl`) to interpret date information and returning it to the user based on the browser's settings. Now we can simply call the method from the template:

```
<template>
  <div>
    <article
      v-if="data && data.current"
      class="max-w-md w-96 rounded-lg shadow-lg p-4 flex bg-white
text-black"
    >
      <div class="basis-1/4 text-left">
        <img :src="data.current.condition.icon" class="h-16 w-16" />
      </div>
      <div class="basis-3/4 text-left">
        <h1 class="text-3xl font-bold">
          {{ data.current.condition.text }}
          <span class="text-2xl block">{{ data.current.temp_c
}}&#8451;</span>
        </h1>
        <p>{{ data.location.name }} {{ data.location.region }}</p>
        <p>Precipitation: {{ data.current.precip_mm }}mm</p>
        <p>{{ formatDate(data.location.localtime) }}</p>
      </div>
    </article>
    <div v-else>Loading...</div>
  </div>
</template>
```

Vue.js interprets the information between the brackets as an expression, so it will just evaluate the output of the `formatDate` method as the rendered information.

Custom style use cases

Now we're left with information regarding wind: speed and direction. Representing the wind speed is straightforward: either pick the metric value or the imperial value and render it in the template. Representing the wind direction, we can make it a bit more user-friendly by indicating direction using an arrow that points in a certain direction.

For such a specialized operation, it is best to create a new component. Let's create one called `WindDirection.vue` for the moment. It will receive a numerical property called `degrees`. We will use the property to create a dynamic (computed) style, depending on the input:

```ts
<script lang="ts" setup>
import { computed } from "vue";
interface Props {
  degrees: number;
}
const props = defineProps<Props>();

const windStyle = computed(() => ({
  transform: "rotate(" + props.degrees + "deg)",
}));
</script>
<template>
  <span
    ><span aria-hidden="true" class="inline-block"
:style="windStyle">↓</span
    ><span class="sr-only">Wind direction: {{ degrees }} degrees</span></span
  >
</template>
```

In the file, there are two interesting things going on. For starters, the `degrees` property gets used to create a computed value called `windStyle`. That value holds a dynamic CSS property and thus will be responsive to any prop it will get. We apply the style by binding it to the `:style` property of the `<span>` element, which simply holds an arrow.

The second thing to point out is the `<span>` element with the `sr-only` class. This is a technique to make the content more accessible and it is something you should always consider. The arrow and its rotation only make sense when you can see the component being rendered. Not everybody can rely on sight when using the web. Some people use tools such as screen readers to process information on a given page.

Tailwind offers a special class utility to mark content for screen readers only, which means its contents are hidden by default browser behavior. A screen reader will likely read out the contents of the element. In this case, we're describing the meaning of the arrow with its rotation.

Please be mindful of providing tools to make any website or application accessible for all by default.

Let's now add this component to our application to wrap it all up:

```ts
<script lang="ts" setup>
import { ref, onMounted } from "vue";
import type { Ref } from 'vue'
import WindDirection from "./WindDirection.vue";
// abbreviated…
</script>
<template>
  <div>
    <article
      v-if="data && data.current"
      class="max-w-md w-96 rounded-lg shadow-lg p-4 flex bg-white
text-black"
    >
      <!-- abbreviated -->
      </div>
      <div class="basis-3/4 text-left">
        <h1 class="text-3xl font-bold">
          <!-- abbreviated -->
        </h1>
        <p>{{ data.location.name }} {{ data.location.region }}</p>
        <p>Precipitation: {{ data.current.precip_mm }}mm</p>
        <p>{{ formatDate(data.location.localtime) }}</p>
        <p>
          Wind: {{ data.current.wind_kph }} kph
          <wind-direction :degrees="data.current.wind_degree" />
        </p>
      </div>
    </article>
    <div v-else>Loading...</div>
  </div>
</template>
```

That's it! That's our weather app! You can style it any way you want. If you want to add or remove properties, please finish the last part first, because it will help you maintain stability throughout your changes.

In our next section, we'll add unit tests for specific features of the app. Unit tests take in small bits (units) of the code and validate that these parts have the same output when given certain conditions. Let's see how that works.

Ensuring stability with Vitest

Now that we have a working app, adding or removing a property can be done with ease. Just update the file and you're ready. This is, however, not always a desirable situation. Having the ability to remove properties that easily could result in unwanted bugs in your application!

We can add more control to our code by describing its behavior using tests. In this part, we'll take a look at adding unit tests with Vitest and Vue Test Utils, to demonstrate where unit tests can help you (and where they cannot).

Vue Test Utils

The official testing library for Vue.js projects is **Vue Test Utils** (`https://test-utils.vuejs.org/`). A testing framework is a series of tools and functions that you can use to create isolated instances of a component and manipulate it to assert certain behaviors.

The purpose of unit testing is to validate that each unit (or component) of the software is working as expected and meets the specified requirements. In our case, we could write tests for our Vue components, but also for JavaScript files that just export functions.

With our selected preset, we've already included the test tooling in our app. It even added a test script as part of the `package.json` file to run our tests with, so we can get started with writing some tests. There are several ways of organizing and structuring code. I prefer to have tests next to the component that they are testing. Test files are identified by a `.spec.ts` suffix in the filename, so you can always spot them with ease.

There are a couple of things to consider when it comes to writing good unit tests:

- You should be able to test each functionality in isolation

- Each unit test should be independent of all other tests, and not rely on the state of other tests or functions

- Tests support your documentation, so use logical and descriptive names to help understand what a test covers

We'll discover what I mean while we are writing different sets of tests or both Vue components as plain JavaScript files.

Let's start with something simple. Our entry point of the application is the `./src/app.vue` component. Let's quickly open the file:

```
<script setup lang="ts">
import GetLocation from "./components/GetLocation.vue";
</script>

<template>
  <GetLocation />
</template>
```

Now, in terms of functionality, little can go wrong. It imports the `GetLocation` component and mounts it to the template. If we consider this from a documentation point of view, you could describe the functionality of the component as *it should render the GetLocation component*. We can assert this, and this is exactly how we will write our first test.

Create a file called `App.spec.ts` with the following content:

```
import { describe, it, expect } from 'vitest'
import { shallowMount } from "@vue/test-utils";
import GetLocation from "./components/GetLocation.vue";
import App from "./App.vue";

describe("App", ():void  => {
  it("renders the GetLocation component", ():void => {
    const wrapper = shallowMount<App>(App);
    expect(wrapper.findComponent(GetLocation).exists()).toBe(true);
  });
});
```

In this example, we import some tools to write our tests and assertions from Vitest. We also import a method to mount a component in our test and import the `GetLocation` and `App` components. You can execute the unit tests with this command:

```
npm run test:unit
```

This command will execute all of the recognized test files and keep a watcher running to rerun any changed tests. The script automatically generates a report of the tests in the console.

A (shallow) mounted component will behave just as it would when being rendered by a browser, but we run this in isolation. With Vue Test Utils, you can either `mount` or `shallowMount` a component, with the difference being that a mount also attempts to render any children. A `shallowMount` function creates stubs for any children, which reduces side effects and focuses directly on the component itself. As a rule of thumb, I tend to always use `shallowMount` unless I need to assert specific parent-child behavior.

We scope the test with a `describe` block and make sure that we describe what the subject of our test is.

Then we write our test: *it renders the GetLocation component*. This simply mounts the `App` component and then asserts whether the stub of the `GetLocation` component was found in the render tree.

Global test functions

We will use three functions in all our tests to create our test assertions. These functions are used to create clear and organized testing code, with the `describe` function used to group tests into logical test suites, the `it` function used to define individual test cases, and the `expect` function used to define the expected behavior of the code being tested.

We can make a small modification in our project files that removes the need for manually importing those much used functions.

Open the `vite.config.ts` file and look for the `test` property. If we add a new property called `globals` with the value `true` as a child of the `test` property, you should end up with something similar to this:

```
export default defineConfig({
  plugins: [vue()],
  test: {
    globals: true,
    environment: "jsdom",
  }
})
```

Now we can use the functions, but our IDE is not aware of those functions, so we need to configure some additional settings. First, we need to install some types by running the following command in the CLI:

```
npm i --save-dev @types/jest
```

We'll update the `tsconfig.vitest.json` file with the following addition to `types`:

```
{
  "extends": "./tsconfig.app.json",
  "exclude": [],
  "compilerOptions": {
    "composite": true,
    "lib": [],
    "types": ["node", "jsdom", "jest"]
  }
}
```

Now our IDE is perfectly okay with directly using the `describe`, `it`, and `expect` functions. And that's all you need to do in order to register them as globally available functions. You can remove that line from our first test file if you like.

A simple component test

Let's work our way on to more complex components now. Let's start with the `WindDirection` component. We can assert a couple of things here that the component renders, but also that it adds the correct style to the direction indicator and that the screen reader text reflects the same value.

First, we'll create the `WindDirection.spec.ts` file, with the following contents:

```
import { shallowMount } from "@vue/test-utils";
import WindDirection from "./WindDirection.vue";

describe("WindDirection", () => {
  it("renders without crashing", (): void => {
    const wrapper = shallowMount(WindDirection, {
      props: {
        degrees: 90,
      },
    });
    expect(wrapper).toBeTruthy();
  });
});
```

This only asserts that the component should not error when provided with the minimum requirement of properties. `toBeTruthy` is a loose assertion that asserts if the value is any expression or value that evaluates to true.

When we take into consideration that we want to isolate tests as much as possible, we can add a test for rendering the wind direction arrow with the appropriate style. For that, we want to retrieve an element – the `span` element the computed style is applied to. It is common practice in these cases to add a specific attribute called `data-testid` to the element. Modify the `span` element to add the `data-testid` attribute:

```
<span aria-hidden="true" class="inline-block" data-testid="direction"
:style="windStyle">↓</span>
```

Now we have something to have our tests point at. The benefit of adding a specific attribute for testing over targeting by class name or hierarchical structure is that this is far less likely to be subject to changes over time, which makes your tests more robust:

```
import { shallowMount } from "@vue/test-utils";
import WindDirection from "./WindDirection.vue";
```

```
describe("WindDirection", () => {
  it("renders without crashing", (): void => {
    const wrapper = shallowMount(WindDirection, {
      props: {
        degrees: 90,
      },
    });
    expect(wrapper).toBeTruthy();
  });

  it("renders the indicator with the correct wind direction", (): void
=> {
    const wrapper = shallowMount(WindDirection, {
      props: {
        degrees: 90,
      },
    });
    const direction = wrapper.find("[data-testid=direction]");
    expect(direction.attributes("style")).toContain("rotate(90deg)");
    expect(direction.html()).toContain("↓");
  });
});
```

As you can see, we localized the element using the [data-testid=direction] query and then asserted its style. By mapping it to the contents (a downward-pointing arrow), the combination of rotation with the arrow provides meaningful context. If we were to replace the downward-facing arrow with any other content, the component would lose its meaning and the test would rightfully fail.

We can add the final assertion, aimed at screen reader usage. First, we'll add another data-testid attribute to the component. In this case, to the screen-reader-related element:

```
<span class="sr-only" data-testid="direction-sr">Wind direction: {{
degrees }} degrees</span>
```

This value has to be unique in order for us to target it from the test file, which now looks like this:

```
import { shallowMount } from "@vue/test-utils";
import WindDirection from "./WindDirection.vue";

describe("WindDirection", () => {
  it("renders without crashing", (): void => {
    const wrapper = shallowMount(WindDirection, {
      props: {
        degrees: 90,
      },
```

```
    });
    expect(wrapper).toBeTruthy();
  });

  it("renders with the correct wind direction", (): void => {
    const wrapper = shallowMount(WindDirection, {
      props: {
        degrees: 90,
      },
    });
    const direction = wrapper.find("[data-testid=direction]");
    expect(direction.attributes("style")).toContain("rotate(90deg)");
    expect(direction.html()).toContain("↓");
  });

  it("renders the correct wind direction for screen readers", (): void
=> {
    const wrapper = shallowMount(WindDirection, {
      props: {
        degrees: 270,
      },
    });
    const srOnly = wrapper.find("[data-testid=direction-sr]");
    expect(srOnly.classes()).toContain('sr-only')
    expect(srOnly.html()).toContain("Wind direction: 270 degrees");
  });
});
```

In terms of isolation, we are executing this test on a new instance of the mounted component, and we are only looking at the contents of the `direction-sr` attributed component. We do this because, if a test fails, we should be able to immediately see what the cause and effect is.

We could add the `expect` lines to the previous test block, but if any assertion failed, we would not be able to see the direct cause. In small code bases, it wouldn't be a big concern, but you can imagine the complexity when you're dealing with a code base made up of hundreds of components with their unit tests. That is why isolation and simplicity are key.

Mocking external sources

The previous component only dealt with its own state. But we've also incorporated external sources in our application. We can't test what we can't control, so we don't have to test those external sources. We do have to test the way that our component interacts with external sources. To make that more predictable, we can use mocks to control the output.

A good example is the browser API we're using to retrieve the geolocation of a user. Let's create our `GetLocation.spec.ts` file to test the component!

```
import { shallowMount } from "@vue/test-utils";
import GetLocation from "./GetLocation.vue";

describe("GetLocation", () => {
  it("should render the component without crashing", (): void => {
    const wrapper = shallowMount(GetLocation);
    expect(wrapper).toBeTruthy();
  });
});
```

If we run our test now, it will fail. We need to fix two things here, to be honest. First of all, the test is an asynchronous test, since the retrieval of geolocation is a promise:

```
import { shallowMount } from "@vue/test-utils";
import GetLocation from "./GetLocation.vue";

describe("GetLocation", () => {
  it("should render the component without crashing", async ():
Promise<void> => {
    const wrapper = await shallowMount(GetLocation);
    expect(wrapper).toBeTruthy();
  });
});
```

Unfortunately, this still doesn't work. This is because the tests are not executed in an actual browser, but in jsdom, which is a JavaScript-based browser environment. It doesn't support (all of the) native browser APIs.

The component tries to access the `navigator.geolocation.getCurrentPosition` API, but it doesn't exist! We need to mock it to allow our component to render. Mocking can be a bit abstract, but it is really about controlling the environment that affects our components. In our case, we can use a very straightforward implementation:

```
it("should render the component without crashing", async ():
Promise<void> => {
  global.navigator.geolocation = {
    getCurrentPosition: () => {},
  };
  const wrapper = await shallowMount(GetLocation);
  expect(wrapper).toBeTruthy();
});
```

In this case, we're simply providing a method called `getCurrentPosition` so that it exists in the *browser* when we're executing our test. This service doesn't return any valid or useful information, but that's not what we're interested in here. We just want our component to be able to render.

Also bear in mind that this test highlights a flaw in our application: it needs `navigator.geolocation.getCurrentPosition` to be present; otherwise, it will fail!

Mocking for success

To extend our testing scenario, we need to assert that our component is able to return a successfully resolved geolocation. We are going to create a new test case, because of isolation, and improve upon our mocked navigator API. We'll use Vitest's `vi.fn()` function for this.

The `vi.fn()` function (`https://vitest.dev/api/vi.html#vi-fn`) is a Vitest function that creates a spy on the function. This means it stores all call arguments, returns, and instances. By storing it in `mockGeoLocation`, we can assert its properties more easily. The function takes in an argument, which is a callable mock instance.

At the top of the test file, we'll import the `vi` function like this:

```
import { vi } from 'vitest'
```

Let's have a look at the test:

```
it("displays when geolocation resolved successfully", async ():
Promise<void> => {
  const mockGeoLocation = vi.fn((successCallback: Function) => {
    const position = {
      coords: {
        latitude: 51.5074,
        longitude: -0.1278,
      },
    };
    successCallback(position);
  });
  global.navigator.geolocation = {
    getCurrentPosition: mockGeoLocation,
  };

  const wrapper = await shallowMount<GetLocation>(GetLocation);
  expect(wrapper.vm.coords).toEqual({
    latitude: 51.5074,
    longitude: -0.1278,
  });
});
```

In this case, instead of just having an empty function on the `navigator.geolocation.getCurrentPostition` method, we've created a mock of what a successful resolution would look like. We can find out the specifications of the `getCurrentPosition` API (https://w3c.github.io/geolocation-api/#dom-geolocation-getcurrentposition) in order to have our mock match the expected behavior.

We're providing a `successCallback` function, which returns coordinates just as the browser API would, and we're immediately invoking it to simulate a user granting access to the geolocation data.

Having a successful resolution, we can assert that the component received the same `coords` object that came from the browser.

Unhappy path

Having tested a successful resolution, the last thing to test is what would happen if the user declined access to the location data. We will use a very similar approach, but instead of the successful callback, we will provide a secondary callback for failure. Again, this is according to the specification:

```
it("displays a message when user denied access", async ():
Promise<void> => {
  const mockGeoLocation = vi.fn((successCallback: Function,
errorCallback: Function) => {
    const error = new Error("User denied geolocation access");
    errorCallback(error);
  });
  global.navigator.geolocation = {
    getCurrentPosition: mockGeoLocation,
  };

  const wrapper = await shallowMount<GetLocation>(GetLocation);
  expect(wrapper.vm.geolocationBlockedByUser).toEqual(true);
  expect(wrapper.html()).toContain("User denied access");
});
```

As you can see, in this case, we're completely ignoring `successCallback` and instead defining and invoking `errorCallback`. As the component dictates, the reactive `geolocationBlockedByUser` property will be set to true and we will show an error message.

The complete test file now looks like this, where we assert that the component renders, and it can resolve a successful query and handle a denied request:

https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/03.weather/.notes/2.1-GetLocation.spec.ts

We've made sure we've tested both the happy and the unhappy path. Let's see how we can apply testing to components dealing with external data from an endpoint.

Testing with APIs

Our final component also has an external dependency. This is not much different from mocking a browser API, as we will discover in this section. If we look at our component, it has the following features: shows the loading state when no data is retrieved and displays the response from the service in a nicely formatted way.

Let's start with assessing that the component can render in a file called `WeatherRepost.spec.ts`:

```
import { shallowMount } from '@vue/test-utils'
import WeatherReport from './WeatherReport.vue'

describe('WeatherReport', () => {
  it("should render the component without crashing", (): void => {
    global.fetch = vi.fn() as any
    const wrapper = shallowMount<WeatherReport>(WeatherReport, {
        props: {
            coords: {
                latitude: 0,
                longitude: 0
            }
        }
    });
    expect(wrapper).toBeTruthy();
  });
});
```

In this case, you see that instead of mocking a `navigator` property, we're mocking a global `fetch` property. We don't need to return anything, so we're keeping this test as simple as possible.

On to testing the loading state. In fact, we're actually cheating a little bit here. There is simply a state of not having data or having data. We're treating not having data as a loading state for the sake of simplicity:

```
it('displays loading message when data is undefined', (): void => {
  global.fetch = vi.fn(() => Promise.resolve({
    json: () => Promise.resolve()
  })) as any

  const wrapper = shallowMount(WeatherReport, {
    props: {
      coords: {
        latitude: 0,
```

```
        longitude: 0
      }
    }
  });

  expect(wrapper.text()).toContain('Loading...')
});
```

For this test, we're just resolving the data as nothing. That means there's no data available to render, which will keep the component in its loading state. We can assert that without data, the **Loading…** text stays visible.

We can also assert the situation when we did receive data from the service. This involves a similar approach, with the difference that instead of resolving nothing, we resolve with a mock weather report:

```
it('displays weather data when data is defined', async () => {
  const mockData = {
    // ...abbreviated
  }
   global.fetch = vi.fn(() => Promise.resolve({
     json: () => Promise.resolve(mockData)
  })) as any

  const wrapper = shallowMount(WeatherReport, {
    props: {
      coords: {
        latitude: 0,
        longitude: 0
      }
    }
  })

  expect(wrapper.text()).toContain(mockData.current.condition.text)
  expect(wrapper.text()).toContain(mockData.current.temp_c)
  expect(wrapper.text()).toContain(mockData.location.name)
  expect(wrapper.text()).toContain(mockData.location.region)
  expect(wrapper.text()).toContain(mockData.current.wind_kph)
  expect(wrapper.text()).toContain(mockData.current.wind_degree)
});
```

We assert that the `mockData` properties are being mapped to the wrapper. There is a problem, though: the assertions are failing! We have two problems, in fact. The `shallowMount` function is flattening the HTML structure a bit and we have to wait for the promises to be resolved.

Luckily, Vue Test Utils has a useful utility for dealing with the promises: `flushPromises` is a utility function that makes sure all pending promises are resolved. We can import it at the top of our file together with our `mount` functions:

```
import { mount, shallowMount, flushPromises } from '@vue/test-utils'
```

If we rerun our test, it will succeed:

```
it('displays weather data when data is defined', async () => {
  const mockData = {
// abbreviated
  }

  global.fetch = vi.fn(() => Promise.resolve({
    json: () => Promise.resolve(mockData)
  })) as any

  const wrapper = mount(WeatherReport, {
    props: {
      coords: {
        latitude: 0,
        longitude: 0
      }
    }
  })

  await flushPromises();

  expect(wrapper.text()).toContain(mockData.current.condition.text)
  expect(wrapper.text()).toContain(mockData.current.temp_c)
  expect(wrapper.text()).toContain(mockData.location.name)
  expect(wrapper.text()).toContain(mockData.location.region)
  expect(wrapper.text()).toContain(mockData.current.wind_kph)
  expect(wrapper.text()).toContain(mockData.current.wind_degree)
});
```

There's one final check since we have a formatter for our timestamp. Let's add a `data-testid` attribute to the element:

```
<p data-testid="localtime">{{ formatDate(data.location.localtime) }}</
p>
```

For testing purposes, we can simplify the mockData object because we are only interested in one property (and not crashing the component):

```
it('displays formats the datetime to a locale format', async () => {
  const mockData = {
    location: {
      localtime: new Date(),
    },
    current: {
      condition: {},
    }
  }

  global.fetch = vi.fn(() => Promise.resolve({
    json: () => Promise.resolve(mockData)
  })) as any

  const wrapper = mount(WeatherReport, {
    props: {
      coords: {
        latitude: 0,
        longitude: 0
      }
    }
  })

  await flushPromises();

  const localtime = wrapper.find("[data-testid=localtime]");
  expect(localtime.text()).toEqual('January 31, 2001 at 11:45 AM')
});
```

Now this will only succeed once because the new Date() function will constantly be refreshed with the date and time of executing the test! Again, we have an external factor that we need to mock in order to isolate our test.

Vitest offers tooling to manipulate dates, times, and even the passing of timers (https://vitest. dev/api/vi.html#vi-setsystemtime). We can modify our test so that the test always assumes the exact same date and time. This way, we can assert the outcome based on a fixed value.

The updated version will look like this:

```
it('displays formats the datetime to a locale format', async () => {
  const mockDateTime = new Date(2000, 12, 31, 11, 45, 0, 0)
  vi.setSystemTime(mockDateTime)
```

```
  const mockData = {
    location: {
      localtime: new Date(),
    },
    current: {
      condition: {},
    }
  }

  global.fetch = vi.fn(() => Promise.resolve({
    json: () => Promise.resolve(mockData)
  })) as any

  const wrapper = mount(WeatherReport, {
    props: {
      coords: {
        latitude: 0,
        longitude: 0
      }
    }
  })

  await flushPromises();

  const localtime = wrapper.find("[data-testid=localtime]");

  expect(localtime.text()).toEqual('January 31, 2001 at 11:45 AM')
  vi.useRealTimers()
});
```

This makes it safe to assert the date based on a static value. I tend to pick my date and time values in such a way that months, days, hours, and minutes are easily identifiable regardless of the notation.

Summary

At this point, we've added a bit more complexity to our app, any external resource calls for additional error handling, and we've learned how to deal with asynchronous data using a loading state. We've been able to quickly style our app using the utility style CSS framework Tailwind. With the unit test, we've made sure that we can assert that our application's core features will continue to work as expected or alarm us if the output changes in any way.

In the next chapter, we'll focus on connecting more extensively with a third-party API, by combining multiple endpoints from an API into a single app.

Creating the Marvel Explorer App

Let's build an app that's a bit more advanced, using a third-party API to feed it with data. I hope you like comics, because we will build an explorer on the Marvel Comics API, and I will try to squeeze in some heroic puns. We'll explore adding different routes and add a bit of abstraction to make better use of writing our code.

We'll cover the following topics in this chapter:

- Getting started
- Retrieving data from the API
- Routing in a single-page application
- Writing and using composables
- Searching for and handling data
- User-friendly error handling

Technical requirements

In this chapter, we'll replace **node package manager** (**npm**) with **performant npm** (**pnpm**): `https://pnpm.io`.

We need to register at `https://developer.marvel.com/` to retrieve an API key. We'll add **Tailwind CSS** (`https://tailwindcss.com/`) to apply styling to this app as well.

In this chapter we introduce routes using the official router for Vue.js applications: `https://router.vuejs.org/`.

The complete code for this chapter is available at `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/tree/main/04.marvel`.

Getting started with our new project

In order to get started, we need an API key. If you go to `https://developer.marvel.com/` and select **Get a Key** from the menu, you will need to register for a free account. Afterward, you will be redirected to the developer portal, where you create a key to interact with the API. Make sure to note the public and private keys.

For our example, we will access the API from localhost, so you need to add `localhost` and `127.0.0.1` to the list of authorized referrers.

> **Note**
>
> If you want to deploy this app to the web, you will need to make sure to add the corresponding URL of the app's address there as well, but the deployment step is not covered in this chapter.

I'd like to point out the documentation, which you'll find under **Interactive Documentation**. I recommend playing around with it for a bit to get a sense of our data provider.

Let's start a new project!

```
npm init vue@latest
```

Hit `y` to proceed, choose `vue-marvel-explorer` as the project name, and select the options shown in the following image:

Figure 4.1 – The setup configuration for the Marvel Explorer app

After following the instructions to install the dependencies we can get to work!

Let's install our project's dependencies using pnpm this time. This is an alternative package manager for node that has some benefits over npm, such as better storage management of packages, which results in increased installation speed and reduced network requests. If your internet connection is less than optimal, pnpm has got your back! You can read the installation guide here (`https://pnpm.io/installation`). The commands are similar to npm, so it should be very easy to get up to speed.

Navigate to your project's folder and type `pnpm install` (instead of `npm install`). The neat thing is that every future installation of the same package will reference the already installed local cache, which saves a ton of bandwidth and time.

We'll install tailwind on the project as well, revisiting the steps from the Weather app, using pnpm, an alternative to npm, aimed at optimizing `node_modules` management:

```
pnpm install -D tailwindcss postcss autoprefixer
pnpm dlx tailwindcss init -p
```

Let's update the `tailwind.config.js` file:

```
/** @type {import('tailwindcss').Config} */
export default {
  purge: ['./index.html', './src/**/*.{vue,js,ts,jsx,tsx}'],
  content: ['./src/**/*.{vue,js,ts,jsx,tsx}'],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Create a `style.css` file in the `./src` folder:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

And finally, open the `./src/main.ts` file to import the CSS file into the app (note that this file contains the router initialization):

```
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
import './style.css'

const app = createApp(App)
```

```
app.use(router)
app.mount('#app')
```

If you run the development server with `pnpm run dev`, you'll see that the demo app added an example route, allowing you to navigate between a Home and About view. Close the dev server, and let's open the project in the code editor.

You can remove the components that the default Vue installation added to clean up your project a bit. If you are unsure, you can always refer to the GitHub repository. The link can be found at the *Technical requirements* section.

Rename the `.env.example` file to .env and make sure to insert the secrets from the Marvel Developer portal:

```
VITE_APP_MARVEL_API_PUBLIC=YOUR_PUBLIC_KEY_HERE
VITE_APP_MARVEL_API_SECRET=YOUR_SECRET_HERE
```

Variables from a `.env` file that are prefixed with `VITE_APP_` are automatically passed to and available in your application on a pre-defined `import.meta.env` object.

> **Note**
>
> I want to stress again that sharing secrets is not a best practice in production-like environments. You'd normally use something like an authorization broker to make sure that the API only receives trusted requests. In a way, we've done this by defining the request domains in the Marvel API configuration. Typically, `localhost` or its equivalent, `127.0.0.1`, is not something you would see in a production environment either!

This concludes our environment set up. We'll now move forward with connecting these settings to our application.

The Superhero connection

We want to retrieve data from the Marvel Comics API from different components in our application. A good pattern for doing this is by creating a Vue composable. A Vue composable is a proven pattern for using and reusing logic throughout your application. We'll create a folder called `composables` in the `src` folder and create a file called `marvelApi.ts`.

You can import the types from the example repository (`https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/04.marvel/src/types/marvel.ts`).

These types are mainly contracts with the API. Feel free to take a look at them. I created them by ingesting the results from the API and defining the types.

We'll start with an asynchronous function that fetches the data from the API from the comics endpoint and returns a promise of the response. We're going to expand the functionality gradually. Add a new composable function called `useComics` to the file and don't forget to import the type:

```ts
import type { Comics } from '@/types/marvel'

export const useComics = async (): Promise<Comics> => {

  const apiKey = import.meta.env.VITE_APP_MARVEL_API_PUBLIC;

  const MARVEL_API = `//gateway.marvel.com/v1/public/
  const API_SIGN = apikey=${apiKey}`

  const requestURI = `${MARVEL_API}/comics?${API_SIGN}`

  const res = await fetch(requestURI);
  const jsonRes = await res.json();

  return jsonRes.data;

}
```

Now we can wire the API call to the user interface. We are going to create our component to display the data from the endpoint. Create a new Vue component in the `src/components` folder called `ComicsOverview.vue`. We'll start with the `script` tag contents:

```ts
<script lang="ts" setup>
import { ref, onMounted } from "vue";
import type { Ref } from "vue";

import { useComics } from "@/composables/marvelApi";
import type { Comic } from "@/types/marvel";

const isLoading: Ref<boolean> = ref(false);
const data: Ref<Comic[] | undefined> = ref();

const getComics = async () => {
  isLoading.value = true;
  const comics = await useComics();

  data.value = comics.results;
  isLoading.value = false;
};
```

```
onMounted(async () => {
  await getComics();
});
</script>
```

The `script` block is very similar to the weather app from *Chapter 4*. We're requesting data when the component gets mounted, and we'll track the state using the `isLoading` variable.

In the template of the same file, we'll add the following contents:

```
<template>
  <div>
    <div v-if="isLoading"><p>Loading comics…</p></div>
    <div v-if="data && !isLoading">
      <div
        class="grid grid-flow-row grid-cols-1 gap-4 md:grid-cols-2
lg:grid-cols-4"
      >
        <div :key="comic.id" v-for="comic in data">{{ comic.title }}</
div>
      </div>
    </div>
  </div>
</template>
```

You can quickly see the result by temporarily importing the component into `App.vue` and loading it in the template. A slight difference here is that we've abstracted the actual fetch to the composition, which makes the code of the component a bit cleaner and makes the fetch more reusable.

Now that our data is coming in, we're going to polish the component a bit. Let's create a `LoadingIndicator.vue` component:

```
<script setup lang="ts">
const props = defineProps<{
  text?: string;
}>();
</script>
<template>
  <div
    class="flex flex-col items-center justify-center p-4 pt-16 min-h-
min min-w-screen"
  >
    <div v-if="text" class="mb-4">
      {{ text }}
    </div>
    <div class="flex space-x-2 animate-pulse">
```

```
        <div class="w-3 h-3 bg-gray-500 rounded-full"></div>
        <div class="w-3 h-3 bg-gray-500 rounded-full"></div>
        <div class="w-3 h-3 bg-gray-500 rounded-full"></div>
      </div>
    </div>
</template>
```

We can import it to the component and then use it to replace the `<div v-if="isLoading"><p>Loading comics…</p></div>` element, like so:

```
<script lang="ts" setup>
import { ref, onMounted } from "vue";
import type { Ref } from "vue";

import { useComics } from "@/composables/marvelApi";
import type { Comic } from "@/types/marvel";

import LoadingIndicator from "./LoadingIndicator.vue";

//… abbreviated
</script>
<template>
  <div>
    <LoadingIndicator v-if="isLoading" text="Loading comics..."/>
    // … abbreviated
  </div>
</template>
```

We do this to create more consistency when using recurring user interface patterns. Again, this is one of the strengths of a component-based architecture.

We can also create the visual representation of a comic. We're going to apply the abstractions right away. In practice, it happens more often that refactors occur while progressing on the code base. It's tricky to predict what code will be reused beforehand, so don't hesitate to refactor early and often, when the need arises. In our case, though, we have a different goal to teach, so we won't focus on the refactoring part.

Create a component called `CardView.vue` that contains the following code:

```
<template>
    <article class="p-4 bg-white rounded-lg shadow-xl place-content-
center text-slate-800">
    <header>
        <h1 class="pb-5 text-lg font-semibold "><slot name="header"></
slot></h1>
```

```
      </header>
      <slot></slot>
      </article>
  </template>
```

In this component, we are making use of slots. Slots are placeholders for specific dynamic content (components or text) that comes from the parent component. Slots are an excellent way to reuse templates and offer a lot of flexibility. Consider the `LoadingIndicator` component, which accepts only a text as a property. Restriction can be good, but sometimes you might prefer flexibility over constraint. Let's start to apply this generic component.

We'll create a `ComicCard.vue` component with the following contents: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/04.marvel/.notes/4.1-ComicCard.vue`

Let's break that component down, shall we? You should be familiar with most of the concepts, but I have managed to introduce a bit more. A particular addition is the following line:

```
const lf = new Intl.ListFormat('en');
```

The `Intl` is a standardized namespace that deals with language-sensitive functions. In our case, we are setting up a list specific formatter in the English (en) language and using it in the template to string together the list of creators. The list is provided as an array (that is, `["Evan You", "Sebastien Chopin", "Anthony Fu"]`). Using the `Intl` formatter, the result is readable text that is specific to that language: *Evan You, Sebastien Chopin, and Anthony Fu!*

We're using the computed values to create lists of characters for every comic (`charactersList`) and the creators of every comic (`creatorsList`).

In the template, we see how we are using the slots in the `CardView` component and filling it with our own templates:

```
<template>
  <CardView :data-testid="comic.id">
    <template v-slot:header>
      {{ comic.title }}
    </template>
    <template v-slot:default>
      <img
        class="aspect-[150/228] shadow-xl float-left mr-4"
        :src="`${comic.thumbnail.path}.${comic.thumbnail.extension}`"
        width="150"
      />
      //… abbreviated
    </template>
```

```
    </CardView>
</template>
```

We've defined the contents of the templates. We've designated `comic.title` for the header in the component. For the default slot, we're providing the markup for the contents of the card. In both cases, we let the `<CardView>` component handle the formatting and style, which also ensures a consistent user interface.

In the example code, you will see the shorthand notation for named slots:

```
<template #header>
  {{ comic.title }}
</template>
<template #default>
  //… abbreviated
</template>
```

For now, we'll remove the temporary change to `App.vue` because we're going to add this to a specific route view!

With the components we have prepared, we'll continue by moving those components to specific views and routes.

Marvelous routes in a single-page application

Now, let's take a look at the default setup of the application, because we've pre-installed the app to use `vue-router`. This configured the app with a couple of things:

- We have an `index.ts` file in the `router` folder

- In the `views` folder, we have two components called `HomeView.vue` and `AboutView.vue`

- In `App.vue`, we have some components called `RouterLink` and `RouterView`

That's how routes are tied together. Let's take a look at each of them.

The contents of the `router` folder define and configure the routes for the application. Routes define the different paths in your application and the components that should be rendered when those paths are accessed. Each route is represented as an object with properties such as `path`, `name`, and `component`.

The `path` property specifies the URL path, and the `component` property specifies the Vue component to render. The `name` is not required and more meant as a human-readable identifier for a route.

Using the default configuration, it defined the home view for us and set up the other view to support code splitting to limit bundle sizes per route. So, this is a best practice out of the box!

We'll change the references to about into search, in preparation for our final result:

```
{
  path: '/search,
  name: search,
  // route level code-splitting
  // this generates a separate chunk (Search.[hash].js) for this route
  // which is lazy-loaded when the route is visited.
  component: () => import('../views/SearchView.vue')
}
```

We'll rename AboutView.vue to SearchView.vue. For the contents of the file, you can just strip most of it, we'll build something new later on. Something like this is fine for now:

```
<template>
  <div class="search">
    <h1>This is a search page</h1>
  </div>
</template>
```

To string it all back together, we can update the App.vue file so the RouterLink component points to /search instead of /about.

To be clear: you have to use the path that we've configured in the router file. The RouterView component is used to render the matched component based on the current route. It acts as a placeholder where the corresponding component is rendered. Whenever the route changes, the RouterView component will be automatically updated to render the new component. Remember slots? Consider RouterView as sort of a dynamic slot that can hold an entire view.

RouterLink in turn is used to create navigation links in your application and triggers navigation to the given route when clicked. The to attribute of RouterLink specifies the path or name of the target route.

Let's clean up the parts of the default configuration we don't need anymore:

```
<script setup lang="ts">
import { RouterLink, RouterView } from 'vue-router'
</script>

<template>
  <header>
    <div class="my-4 text-center">
      <h1 class="mb-4 text-6xl font-extrabold uppercase">Marvel
Explorer 🎬</h1>
      <nav>
        <RouterLink to="/" class="px-4 py-2 border-2 rounded-s-md
```

```
hover:text-slate-600">📋 Comics</RouterLink>
        <RouterLink to="/search" class="px-4 py-2 border-2 border-s-0
hover:text-slate-600 rounded-e-md">🕵 Heroes</RouterLink>
      </nav>
    </div>
  </header>

  <RouterView />
</template>
```

Since our `ComicsOverview.vue` component is ready, we can add it to `HomeView.vue`, replacing the `TheWelcome.vue` part:

```
<script setup lang="ts">
import ComicsOverview from '@/components/ComicsOverview.vue';
</script>

<template>
  <main>
    <ComicsOverview />
  </main>
</template>
```

If you now run the app, you can navigate between the home page, which loads the comics overview, and the almost empty search page.

We'll continue to work on getting more information into our application, because our application is limited now to just showing the first page of the API results.

Optional parameters

If you analyzed the network request coming from the Marvel API, you may have noticed that the comics we are showing are but the tip of the iceberg. There are a multitude of comics, and because of that volume, they are not being sent in one response. The API offers paged results. We can modify our app to mirror the features of the API!

If we open the router file, we can add an optional parameter to a route. It parses and exposes the value to be used in your application. The notation for a parameter (or *param* for short) is to prefix the name with a colon. We'll add a param called `page` to the `home` route:

```
{
  path: '/:page',
  name: 'home',
  component: HomeView
},
```

We have introduced a slight bug in our application. The application now *always expects a parameter*. For the home page, this is not always the case! The param should be optional. To mark a param as optional, we add a question mark as a suffix:

```
{
  path: '/:page?',
  name: 'home',
  component: HomeView
},
```

Hooray! We've successfully added an optional parameter. We can now introduce pagination to our comics overview.

Paging Dr Strange

The `vue` router exposes itself as a composable. A composable is a Vue superpower, used to encapsulate stateful logic for easy reuse. This means that the router composable, in this case, holds the state of the router, which we can use in any component!

This means we can open our `ComicsOverview.vue` file directly to implement pagination. Let's take a look at the `script` tag of the component and add a few lines: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/04.marvel/.notes/4.2-ComicsOverview.vue`

First, we're importing the composable (*line 4*) and register it to the route constant (*line 12*). Then we're adding two reactive variables (*lines 16, 17*) to track the page information. We access the parameter via the `route.params` object. Since we named the variable page we can access the corresponding property on the method. We use the `+route.params.page` (*lines 19-21*) as shorthand to convert the value to a numeric type, and we store it in the reactive `currentPage`.

Then, in `getComics`, we use the data coming from the endpoint to actualize the values (*lines 27, 28*).

Now that we know how many pages there are and what page we are on, we can use these properties to provide a simple `Pagination` component.

A simple pagination component

So, let's create a new component called `Pagination.vue` and add the following contents: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/04.marvel/.notes/4.3-Pagination.vue`

It should be pretty self-explanatory by now: we're adding the relevant props (for more flexibility, we provide a `path` property as well) and, based on our current page, we can render the links to the first, previous, or next and last page links and show the pagination status.

I would like to point out the aria-hidden property, which we use to mark decorative elements that add no value for screen readers. Now, we'll add it to the `ComicsOverview.vue` component by importing it and pasting the template under the overview of comics cards:

```
<template>
  <div>
    <LoadingIndicator v-if="isLoading" text="Loading comics..." />
    <div v-if="data && !isLoading">
      <div
        class="grid grid-flow-row grid-cols-1 gap-4 md:grid-cols-2
lg:grid-cols-4"
      >
        <ComicCard
          :comic="comic"
          :key="comic.id"
          v-for="comic in data"
        ></ComicCard>
      </div>
      <Pagination
        :total-pages="totalPages"
        path="/"
        :current-page="+currentPage"
      ></Pagination>
    </div>
  </div>
</template>
```

That works! We can click to the next page, but nothing happens. That's because our API doesn't support pagination yet. Let's look into adding it as a feature, so we'll open the `marvelApi.ts` file. First, we'll add the option for pagination to `useComics`:

```
export const useComics = async (page: number = 0): Promise<Comics> =>
{    const apiKey = import.meta.env.VITE_APP_MARVEL_API_PUBLIC;
  const MARVEL_API = //gateway.marvel.com/v1/public/
  const API_SIGN = apikey=${apiKey}

  const ITEMS_PER_PAGE = 20;
  const pagination = page ? &offset=${page * ITEMS_PER_PAGE} :

  const requestURI = ${MARVEL_API}/comics?${API_SIGN}${pagination}

  const res = await fetch(requestURI);
  const jsonRes = await res.json();
```

```
    return jsonRes.data;
  }
```

We're simply accepting a page number, and we'll use the predefined ITEMS_PER_PAGE to determine the offset (which is the way the Marvel API deals with pagination). Then, we store the query parameter and add it to the requestURI.

Now, we can flip over to the ComicsOverview component again to implement the pagination and connect the route parameter to the API request. To accomplish this, we add the following to the script block:

```ts
<script lang="ts" setup>
import { ref, onMounted, watch } from "vue";

// abbreviated…

const getComics = async (page: number = 0) => {
  isLoading.value = true;
  const comics = await useComics(page);

  currentPage.value = comics?.offset / comics?.limit || 0;
  totalPages.value = Math.ceil(comics.total / comics.limit);

  data.value = comics.results;
  isLoading.value = false;
};

watch(
  () => route.params.page,
  async (newPage) => {
    await getComics(+newPage);
  }
);

onMounted(async () => {
  await getComics(+currentPage.value);
});
</script>
```

We can simply add the page now to the getComics request and pass it down to the useComics composable. We do this onMounted, for when you enter the application from a URL directly. We also add a watch function, that keeps track of the route.params.page and requests a new page once the value changes. We're using the quick conversion to numeric here as well, as you can see by the plus sign.

With all of this in place, we can now browse through all 2,746 pages! As an extra exercise, why not figure out if you can expand the pagination component to show multiple pages.

Once you're ready to move forward, we'll refactor our app to use composables. They are functions that encapsulate (stateful) logic.

Composables, assemble!

Let's take a look at how we can leverage our composables and refactor the app to expand the functionalities a bit. Composables are all about reusability: it's their superpower in the Vue space, so let's put our previously created composable into action.

First, we will work on refactoring the `useComics` composable, where we will lightly apply the clean code principles. In our context, this will translate to applying the single responsibility principle and writing small and cohesive functions with meaningful names.

Refactoring useComics

We'll refactor in a non-destructive way too, leaving the existing `useComic` composable functional until we're ready to update that too.

We'll first move the static constants out of the function to the upper scope. We'll also import additional types that we will reference in functions. This way, we can still access them, but they are available throughout the file. I make it a practice to group these types of values at the top of the file for easy future reference:

```
import { Path } from '@/types/marvel'
import type { Comics, Characters, Character } from '@/types/marvel'


const apiKey = import.meta.env.VITE_APP_MARVEL_API_PUBLIC;
const MARVEL_API = //gateway.marvel.com/v1/public/
const API_SIGN = apikey=${apiKey}
const ITEMS_PER_PAGE = 20;

export const useComics = async (page: number = 0): Promise<Comics> =>
{

  const pagination = page ? &offset=${page * ITEMS_PER_PAGE} :

  const requestURI = ${MARVEL_API}/comics?${API_SIGN}${pagination}

  const res = await fetch(requestURI);
  const jsonRes = await res.json();
```

```
    return jsonRes.data;
  }
```

If we think about what our new composables should do and have in common, we can identify the following activities: determining the pagination, determining a search query, constructing the Marvel Developer API URL, and fetching and returning data. We'll create short, separate functions for each activity. These are not composables, and we won't expose them outside of the file.

Let's add the getPagination function, which accepts page number and translates it to a string translating the page to an **offset** (in line with what the Marvel API would expect):

```
const getPagination = (page?: number): string => {
    return page ? &offset=${page * ITEMS_PER_PAGE} : ''
};
```

To construct an additional string containing a search query, we add the following:

```
const getQuery = (query?: string): string => {
    return query ? &${query} : ''
};
```

The next addition is a function to construct the request URI, combining the static constants with the output of the getPagination and getQuery function:

```
const getRequestURI = (path: Path,query: string, pagination: string):
string => {
    const apiPath = ${MARVEL_API}/${path};
    return ${apiPath}?${API_SIGN}${query}${pagination};
};
```

And we'll add a function to do a request and return the result. In this case, we could potentially reuse this, so we can write it as if it were a composable, using the use prefix:

```
export const useFetch = async (requestURI: string): Promise<Comics |
Characters> => {
  const res = await fetch(requestURI);
  const jsonRes = await res.json();
  return jsonRes.data as Comics | Characters;
};
```

Finally, we can string all of the functions together in a composable that will allow us to interact with the Marvel Developer API:

```
interface ApiOptions {
  query?: string;
  page?: number;
```

```
}

export const useMarvelAPI = async (path: Path, options: ApiOptions):
Promise<Comics | Characters> => {
   const pagination = getPagination(options.page);
   const query = getQuery(options.query);

   const requestURI = getRequestURI(path, id, query, pagination);
   return useFetch(requestURI);
}
```

As you can see, we've created a function that can either return comics or characters, this will depend on the path variable we provide. Since the Marvel Developer API has similar mechanics for every endpoint, we were able to make a useful abstraction of the options we need.

The code we've added to the `MavelAPI.ts` file looks like this: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/04.marvel/.notes/4.4-MarvelAPI.ts`

We've expanded the file with abstractions to retrieve data from the endpoint, where we can reuse functions that are generic while being able to request specific content.

Let's investigate incorporating these functionalities into our application.

Reassembling functionalities

Now, we can update our existing `useComics` composable to build on top of this foundation. As you can see, we're now able to reduce the contents of the composable to a single line of code, providing only the path and current page:

```
export const useComics = async (page: number = 0): Promise<Comics> =>
{
   return await useMarvelAPI(Path.COMICS, { page }) as Comics
}
```

By running our code, the existing overview of comics with pagination should remain fully functional. This is a testament to writing clean code rather than the power of composables. It enables us to implement new, useful code with relative ease. In our next composable, we'll request a different type of information, as we will describe in the function.

The way we will interact with the search API is to provide the correct path to the API and, in our case, we'll use the `nameStartsWith` way of searching. We'll provide it, together with a dynamic search value, as part of the query:

```
export const useCharacterSearch = async (query: string, page: number =
0): Promise<Characters> => {
```

```
   return await useMarvelAPI(Path.CHARACTERS, { query:
 nameStartsWith=${query}, page }) as Characters
 }
```

This time, as you can see, we've changed the expected response type to `Characters` instead of `Comics`. Our IDE will be able to make a distinction between the two types when we're interacting with these composables.

We now have our two composables ready for use in the app. The way we refactored our file is something of a natural process in coding. Over time, requirements will change, so it is only logical that the code changes with it. Having our code split into small, focused functions will make it easier to understand and modify in the future.

When using composables, we normally follow the same practice of providing simple-to-use functions and rather splitting them into separate ones than combining them into one.

If you're interested in applying composables, I recommend checking out `https://vueuse.org/`. It hosts a collection of ready-to-use composables for everyday problems in Vue applications.

At this point, we've seen how the combination of a clean code mindset, combined with specific composables have helped us refactor our application's code into individual parts that are more readable and maintainable. We've also experienced refactoring code. Sometimes we refactor due to shifting needs, sometimes we just want to make existing code more readable.

Let's now look at adding more types of data to our application!

Managing the roster

With our brand new composable, we have easy access to more data from the Marvel Developer API! We'll move onto creating the Vue components that will allow the user interface to deal with searching.

We'll first create a variant of the `ComicCard.vue` named `CharacterCard.vue`. The component will be a bit simpler, so you can either paste the following contents in the file or create a copy of the `ComicCard.vue` and update it to match the contents:

```
<script setup lang="ts">
import type { Character } from "@/types/marvel";
import CardView from "./CardView.vue";
interface Props {
  character: Character;
}

const props = defineProps<Props>();
</script>
<template>
  <CardView :id="character.id">
```

```
    <img :src="`${character.thumbnail.path}.${character.thumbnail.
extension}`" class="float-left w-12 h-12 mb-2 mr-4 rounded-full
shadow-md aspect-square" />
    <template #header>{{ character.name }}</template>
    <div class="text-base max-w-prose">{{ character.description }}</
div>
  </CardView>
</template>
```

There's nothing special going on here. We're expecting a single property character, and that should match the type. We can therefore easily map the underlying properties to a simple HTML template.

Next, we'll create the main component to host all the user interface elements. We'll create a file called `SearchCharacter.vue`, and we'll start with just the template. This too should look familiar after creating `ComicsOverview.vue`. I've highlighted the key differences: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/04.marvel/.notes/4.5-SearchCharacter.vue`

We're providing a meaningful message when the data is loading (*line 3*), and we're showing results in a different matter, namely a `CharacterCard` (*lines 8-12*). In `Pagination`, we have provided the current path (*line 16*), and we've added a more specific message when no data is returned (*lines 25-28*).

Now, we'll implement a way of inputting a search query to lead into the `SearchCharacter` presentation.

Searching for heroes

Searching is an isolated, specific action, so in line with the Single File Components philosophy, we are going to create a specific component for this!

Let's create a form component by creating a new file called `SearchForm.vue`. Start with the `script` tag, and I'll explain some new things along the way:

```
<script lang="ts" setup>
import { ref } from "vue";
import type { Ref } from "vue";

const emit = defineEmits(['searchSubmit'])

interface Props {
  isSearching: boolean;
}

const props = defineProps<Props>();
const query: Ref<string> = ref("");
```

```
const search = (): void => {
   emit("searchSubmit", query.value);
};
</script>
```

There are two interesting things going on. The first highlighted line defines an emit. Emitting happens if we want to pass something (an event) upwards in the scope. Props go down, emits go up. By using the defineEmits, we wrap it so that Vue can keep track of the event at runtime and we're assigning it the name searchSubmit.

Next, we have a function called search that does nothing but emit the event by referencing its name and passing query.value as a parameter. In our parent component, we will be able to catch the event and its value.

It's time to add the template. Let's start as simply as possible:

```
<template>
   <form class="flex justify-center my-8" v-on:submit.stop="search">
      <input
         class="px-3 py-2 border rounded-md rounded-r-none
disabled:opacity-40 border-slate-300 text-slate-800 focus:outline-none
focus:border-slate-500"
         type="text"
         v-model="query"
         placeholder="Search..."
         :disabled="isSearching"
      />
      <button
         class="px-4 py-2 text-sm font-bold text-white transition-
colors duration-300 rounded-md rounded-l-none disabled:opacity-40 bg-
slate-500 hover:bg-slate-600"
         :disabled="isSearching"
         type="submit"
      >
         Q  Search
      </button>
   </form>
</template>
```

Here too, two things are important to note. The v-on:submit.stop statement is a built-in method that prevents the actual form from being submitted as an HTML form (which would lead to refreshing the page). Instead, on submit, it calls the search function.

In order to have any value reference, we can use the v-model to bind the value of query to the input field. This gives you two-way databinding.

Adding search

Although the form works, it doesn't feel like an app: we still have to manually submit the form. Let's upgrade the SearchForm before moving on to showing the results. We'll use a watcher to monitor the value of the query and trigger the search function when it has changed.

We'll update the code in the script tag to match the following:

```ts
<script lang="ts" setup>
import { ref, watch } from "vue";
import type { Ref } from "vue";

const emit = defineEmits(["searchSubmit"]);

interface Props {
  isSearching: boolean;
}

const props = defineProps<Props>();

const query: Ref<string> = ref("");
let timeout: number;

const search = (): void => {
  emit("searchSubmit", query.value);
};

const debouncedSearch = (): void => {
  clearTimeout(timeout);
  timeout = setTimeout(async () => {
    search();
  }, 500);
};

watch(query, (): void => {
  debouncedSearch();
});
</script>
```

It's easy to understand the importing of the watch function and defining the watcher. And instead of directly calling the search function, the watcher calls debouncedSearch.

Firing requests to an API is expensive in terms of resources. By debouncing the function, we run a timer of, in this case 500 milliseconds (ms). When the timer runs out, we then call the search function, which in turn emits the searchSubmit event. If, however, the debouncedSearch function is called before the timer was cleared, we simply reset the timer and wait another 500 ms.

An overview with superpowers

We can finally assemble the SearchCharacter component. Let's start with the script tag, since we left it out when we first started: https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/04.marvel/.notes/4.6-SearchCharacter.vue

Apart from importing our utilities and components, the core of the component is to trigger the search action on the Marvel Developer API. The getCharacterSearch (*lines 20-34*) takes care of updating the reactive values to translate to the UI and calculate the pagination. Its core is using our composable to retrieve the results, which are passed to the reactive data property.

The searchReset function (*lines 36-41*) makes sure we can always return to the initial state, for instance, when you'd want to clear the UI or when somebody searches for an empty value.

In the template itself, we only need to add the SearchForm so that our users can find their favorite hero from the Marvel Universe:

```
<template>
  <div>
    <SearchForm
      :is-searching="isSearching"
      @search-submit="search"
    />
    <LoadingIndicator v-if="isSearching" :text="Searching by
'${searchQuery}'..." />
    // ...abbreviated
  </div>
</template>
```

We have extended the app now with the very useful search function. This means we can put our new composable structure to the test. It allows us to focus on the implementation of the search form with debouncing, rather than fetching the data.

We've seen how our abstractions have helped to expand features with minimal effort. All we've built so far has mostly been about a "happy flow," where the individual parts work as expected. Since we're depending on a third-party API, we have no control over its stability and have to prepare for cases where the data is not returned. We'll focus on error handling in the next section.

A different vision

At this point, our app is functioning just fine. We can improve our app experience by making sure we can handle situations when the API returns an error. Let's see how we can make our app a bit more robust in that sense.

We'll add a page that will be able to display errors to the user when they occur. Let's start with a new file in the `views` folder called `ErrorView.vue`. Just create a template with the following contents:

```
<template>
  <main>
    Oops!
  </main>
</template>
```

We'll circle back to this file later. We can now at least create a new route in the `router/index.ts` file, which just duplicates similar logic from the `search` route:

```
import { createRouter, createWebHistory } from 'vue-router'
import HomeView from '../views/HomeView.vue'

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    {
      path: '/:page?',
      name: 'home',
      component: HomeView
    },
    {
      path: '/search',
      name: 'search',
      component: () => import('../views/SearchView.vue')
    },
    {
      path: '/error',
      name: 'error',
      component: () => import('../views/ErrorView.vue')
    }
  ]
})

export default router
```

If we then navigate to the `/error` route in the app, we should see our `errorpage`. Since the data is coming from an external API, we can't control it. This makes it an obvious weakness in the app.

On top of that, it is common practice to code defensively. If we look at the `marvelApi` composable file, we can add some safeguards around the composables that we're using in the app:

```
export const useComics = async (page: number = 0): Promise<Comics> =>
{
  try {
    return await useMarvelAPI(Path.COMICS, { page }) as Comics
  } catch {
    throw new Error('An error occurred while trying to read comics');
  }
}

export const useCharacterSearch = async (query: string, page: number =
0): Promise<Characters> => {
  try {
    return await useMarvelAPI(Path.CHARACTERS, { query:
nameStartsWith=${query}, page }) as Characters
  } catch {
    throw new Error('An error occurred while trying to search
comics');
  }
}
```

> **Note**
>
> You could also consider adding these `try/catch` blocks on `useFetch` and `useMarvelAPI`. On error, the error will propagate upwards through the call stack, which means it will be caught at the highest level.

We're going to simulate erroneous behavior in order to develop for these unforeseen circumstances. An easy way of doing this is to go to your `.env` file and temporarily rename the value of the `VITE_APP_MARVEL_API_PUBLIC` variable to `VITE_APP_MARVEL_API_PUBLIC_ERROR`. We will change it back once we're done! If you run the app, it will not be able to request anything, and you will see the error message we've set in the console.

Handling the error

Now we'll focus on dealing with the error in a user-friendly way. Let's start with the `ComicsOverview.vue` file. We'll wrap the contents of the `getComics` function with another `try/catch` block.

In this case, the user cannot recover the state from this error, so it doesn't make sense to remain on this page, since it's completely broken. We'll redirect the user to our error page instead. This means we'll import the `useRouter` composable from `vue-router` and instantiate it on the component. We'll modify file and specifically the `getComics` function accordingly: `https://github.com/`

`PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/`
`blob/main/04.marvel/.notes/4.7-ComicsOverview.vue`

We're importing (*line 4*) and registering the router (*line 14*) and use the `router` to redirect to a new route when an error occurs (*lines 27-29*).

Next, we'll create a component to show in the `ErrorView.vue`. Let's create a new component called `ErrorMessage.vue` with some static content:

```
<template>
  <article
    class="p-4 mx-4 my-24 bg-white rounded-lg shadow-xl place-content-
center text-slate-800"
  >
    <header>
      <h1 class="pb-5 text-lg font-semibold">Oops! 🤯</h1>
    </header>
    Something went wrong!
  </article>
</template>
```

And we'll update `ErrorView.vue` to load the component:

```
<script setup lang="ts">
import ErrorMessage from '@/components/ErrorMessage.vue'
</script>

<template>
  <main>
    <ErrorMessage />
  </main>
</template>
```

If we run our code, we should get redirected to the `/error` path as long as the API treats us as if we're not authorized. We'll add `errorhandling` to `SearchCharacter.vue` as well in similar fashion:

```
<script lang="ts" setup>
import { ref } from "vue";
import type { Ref } from "vue";
import { useRouter } from "vue-router";

// … abbreviated

const router = useRouter();
```

```
// … abbreviated

const getCharacterSearch = async (query: string, page: number = 0) =>
{
  try {
    if (query !== "") {
      isSearching.value = true;
      searchQuery.value = query;
      const search = await useCharacterSearch(query, page);

      currentPage.value = search?.offset / search?.limit || 0;
      totalPages.value = Math.ceil(search.total / search.limit);

      data.value = search.results;
      isSearching.value = false;
    } else {
      searchReset();
    }
  } catch (e) {
    router.push("/error");
  }
};
// … abbreviated
</script>
```

In this case, after you've attempted a search, the app will redirect to the same page. It would be nice if we could provide a bit of context to our user, so they can better understand what went wrong. Fortunately, we have access to the error message we've thrown in the `catch` block.

Adding query parameters

We'll modify the `router.push` action so that it passes some additional information to the destination.

It's an easy change that we'll apply to both the `ComicsOverview.vue` and the `SearchCharacter.vue` line. Let's change this:

```
} catch (e) {
  router.push("/error");
}
```

And we'll change it so that it provides information on the `query` parameter:

```
} catch (e) {
  router.push({ path: 'error', query: { info: e as string }})
}
```

In this case, we'll pass the message from the error message as a `query` parameter to the route. Note that it is not meant to send a large amount of text, but it serves as a good example of using the `query` parameters.

Lastly, we can modify the `ErrorMessage.vue` file to read into the `query` parameter in order to show it on the component. We can achieve that by reading into the route by utilizing the `useRoute` composable once the component is mounted. The file would look like this:

```ts
<script setup lang="ts">
import { ref, onMounted } from "vue";
import type { Ref } from "vue";
import { useRoute } from "vue-router";

const route = useRoute();
const errorMessage: Ref<string> = ref("");

onMounted((): void => {
  errorMessage.value = route.query.info as string;
});
</script>
<template>
  <article
    class="p-4 mx-4 my-24 bg-white rounded-lg shadow-xl place-content-
center text-slate-800"
  >
    <header>
      <h1 class="pb-5 text-lg font-semibold">Oops! 😵</h1>
    </header>
    {{ errorMessage }}
  </article>
</template>
```

If you now end up in the error state, you should see a message stating more accurately what went wrong. Don't forget to rename your `VITE_APP_MARVEL_API_PUBLIC` variable once you're done!

At this point we've made good progress on pretty common techniques and principles. In this chapter, we have introduced composables to the app, which bring reusable functionalities. We've also added client side routing and were able to create links in our apps' user interface as well as apply dynamic routing and passing additional parameters.

As an extra bonus, we've introduced basic error handling and learned a bit more about favorite Marvel comic books.

Summary

In this chapter, we've learned how to add multiple pages and navigate by several means: by using the router-link component or manipulating the routes programmatically. We've created composables in order to use and reuse logic within our application. For a better user experience, we learned how we can handle errors in a user-friendly manner.

In our next chapter, we'll build an application using Vuetify, a third-party component library. Component libraries allow us to speed up development by making use of ready-made components. In addition, we'll introduce an application state using Pinia, where we can modularly store data (or a state) to be shared between components throughout our application.

Part 2:
Intermediate Projects

In this part, you will iterate on using external APIs to build data-rich applications. You will also learn how to handle more complex application state, apply the basics of data storage and retrieval, and identify when and how to use short-term storage or long-term, persistent storage solutions. You will be introduced to using web technologies to build applications beyond the web and deploy them anywhere.

This part has the following chapters:

- *Chapter 5*, *Building a Recipe App with Vuetify*

- *Chapter 6*, *Creating a Fitness Tracker with Data Visualization*

- *Chapter 7*, *Building a Multi-Platform Expense Tracker Using Quasar*

5

Building a Recipe App with Vuetify

In this chapter, we'll leverage the power of a third-party component library to quickly scaffold out a user interface and explore the powers and usage of a store in the context of an app. We will build a meal planner where a user can browse recipes to add them to a weekly calendar. The state of the week planner will be stored on the user's machine to make sure it's available on returning visits.

In this chapter, we'll cover the following topics:

- Applying and customizing Vuetify to scaffold out views

- Speeding up development using a component library

- Refactoring strategies

- Understanding state

- The usefulness of structuring stores using Pinia

Technical requirements

We'll be using **Vuetify** (`https://vuetifyjs.com/en/`) in this chapter, a popular component library for Vue.js 3 applications. We also need to register for an API key at `https://spoonacular.com/` to retrieve recipe data.

To manage our applications' state, we'll use **Pinia** (`https://pinia.vuejs.org/`). We'll use a composable from **VueUse** (`https://vueuse.org/`) to leverage `localStorage` in our app.

The code for this chapter can be found in this book's GitHub repository: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/tree/main/05.mealplanner`.

A new Vue project

We're ready to initialize a new project, but we'll use the Vuetify installer this time. Vuetify is a wrapper around the Vue installer, with presets for common Vuetify project configurations. In the CLI, type the following command to proceed to the next steps of the installer wizard:

```
npm create vuetify
```

Now, do the following:

1. Choose `vue-meal-planner` as the project's name.
2. Use the **Base** (Vuetify, VueRouter) installation.
3. Select **TypeScript** using the arrow keys.
4. Select the **npm** option to install the dependencies.

If you navigate to the new projects folder, you can run the local development server with `npm run dev`. The result should look very similar to what's shown in *Figure 5.1*:

Figure 5.1 – The initialized Vuetify application

Before we continue, we also need an API key to make the example a bit closer to reality. This will also allow us to search for actual recipes. To register at Spoonacular, you can follow these steps:

1. Visit `https://spoonacular.com/`.
2. Navigate to the **Food** API.
3. Sign up via **Email** and choose a password.

4.   Confirm your email address to complete the registration process.

5.   After logging in, go to **Profile** to reveal the API key.

Create a .env file in the root of your project and add the following line:

```
VITE_APP_SPOONACULAR_API=Replace this with the key
```

We are now ready to create a meal planner application.

Let's get cooking

First, we'll make sure we have a decent boilerplate project to begin with and start by replacing the contents of App.vue with the following:

```
<script setup lang="ts">
</script>


<template>
  <v-layout>
    <v-container class="main">
      <main>
        <router-view />
      </main>
    </v-container>
    <v-footer app><span class="text-light-green">My Meal Planner</
span> - &copy; {{ new Date().getFullYear() }}</v-footer>
  </v-layout>
</template>
```

We'll expand on this later. Note that in the generated Vue component, the order of the <template> and <script> tags is different. I prefer starting with the <script> tag because that holds the logic that ties to the <template> tag, but both are valid:
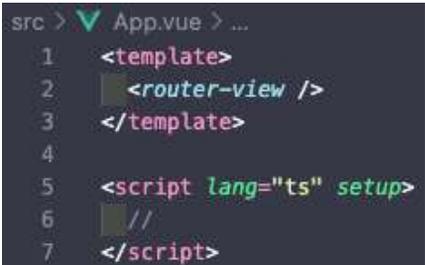
Figure 5.2 – The template and script tag have a different order than in our examples

On the home view, we will build our meal planner as a representation of the upcoming 7 days. First, we'll start with a component that can render several days based on a given date.

We'll create a component in the `components` folder called `CalendarDays.vue`: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/05.mealplanner/.notes/5.1-CalendarDays.vue`.

First, let's take a look at the `script` tag. It accepts props so that it can execute the `generateCards` function, which generates an array of *cards* with a `date` property for each card. We've added the following content just to have something to show in the template:

```
<template>
  <v-table>
    <thead>
      <tr>
        <th class="text-left">Upcoming days</th>
      </tr>
    </thead>
    <tbody>
      <tr v-for="card in cards" :key="card.date.toString()">
        <td class="py-4">
          {{ card.content }}
        </td>
      </tr>
    </tbody>
  </v-table>
</template>
```

In the template, we're using the Vuetify `table` component to render a table. The Vuetify components are prefixed with the `v-` identifier.

We render the table rows using the generated entries from the cards array. We'll expose the component to the view by creating a `MealPlanner.vue` component where we import our component:

```
<script setup  lang="ts">
import CalendarDays from './CalendarDays.vue';
</script>
<template>
    <calendar-days :date="new Date()" :days="7" />
</template>
```

We're not doing anything special here other than instantiating the `CalendarDays` component with our desired number of plannable days. While we're in this folder, we can delete `HelloWorld.vue`, and replace the reference in the `views\Home.vue` component so that our application shows `MealPlanner` on the home page.

Let's improve it by showing formatted dates. We'll create a small composable for this in a to-be-created `composables` folder. Let's call the file `formatters.ts`:

```
const getOrdinalSuffix = (day: number): string => {
    const suffixes = ["th", "st", "nd", "rd"];
    const remainder = day % 100;
    return suffixes[(remainder - 20) % 10] || suffixes[remainder] ||
suffixes[0];
};
export const useFormatDate = (date: Date): string => {
    const day = date.getDate();
    const month = date.toLocaleString("default", { month: "long" });
    const ordinal = getOrdinalSuffix(day);
    return ${day}${ordinal} of ${month};
};
```

Here, we're just adding some clever code that spells out the first, second, or third suffixes based on the given date. We're leveraging bits of the browser's built-in APIs, but with a bit of extra formatting. We can now print out the dates in a readable format for our generated cards, but we'll get to that in a moment.

Let's also create a composable to help us interact with the Spoonacular API. We'll create a file called `recipeApi.ts` in the `composables` folder. The contents should be familiar and resemble the functions we used in the previous chapter: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/05.mealplanner/.notes/5.2-recipeApi.ts`.

With the default installation of Vuetify, we ended up with a default `AppBar.vue` component in the `src/layouts/default` folder. Let's modify it so that it fits the purpose of the app:

```
<template>
  <v-app-bar flat>
    <v-app-bar-title>
      <v-icon icon="mdi-silverware-fork-knife" />
      Meal planner
    </v-app-bar-title>
  </v-app-bar>
</template>
```

The Vuetify components make it relatively easy to build an app with sensible defaults. In the next section, we will learn how to quickly expand our application using the available components.

Quick development with Vuetify

The app we've built so far isn't much use to us yet. Let's turn this into a working meal planner! Since we are going to want to abstract and compartmentalize, we'll start by splitting some of the code of the `CalendarDays.vue` component.

First, we'll create a new component, called `CalendarCard.vue`. We will use this to represent a calendar item and use the date formatter we've created:

```
<script setup lang="ts">
import { useFormatDate } from "@/composables/formatters";

interface Card {
  date: Date;
}

const props = defineProps<{
  card: Card;
}>();
</script>
<template>
  <v-sheet class="d-flex justify-space-between">
    <v-sheet class="ma-2 pa-2">
      <h2 class="text-h2">{{ useFormatDate(card.date) }}</h2>
    </v-sheet>
  </v-sheet>
</template>
```

In `CalendarDays.vue`, we can replace the inline representation by importing our newly created `CalendarCard` component:

```
<script setup lang="ts">
import { ref } from "vue";
import CalendarCard from "@/components/CalendarCard.vue";
// …abbreviated
```

Then, we'll add the component to the template:

```
<template>
  <v-table>
    // … abbreviated
    <tbody>
      <tr v-for="card in cards" :key="card.date.toString()">
```

```
        <td class="py-4">
          <calendar-card :card="card" />
        </td>
      </tr>
    </tbody>
  </v-table>
</template>
```

We also need to add a new route that will be able to display all planned recipes. We'll add the recipes later. First, we'll create a table to display recipes. We'll feed it recipes via a prop from another component.

Let's create `RecipeTable.vue`: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/05.mealplanner/.notes/5.3-RecipeTable.vue`.

We're using the Vuetify components to create a table representation of the list of recipes that will be provided in the property. The `openPreview` function (*lines 2, 14-16, and 34*) is one of the features that the table will support in the future. When we're implementing this, we will make sure that the emitted event will be picked up by the parent component. Let's quickly build the parent component.

Let's create a `RecipesList.vue` component. It will feature the table for displaying past and future recipes, using Vuetify components: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/05.mealplanner/.notes/5.4-RecipesList.vue`.

I've added a little bit of code to generate some mock data (*lines 12-24*). It can sometimes be helpful to focus on the template, and in those cases, it's helpful to be able to have granular control over the data and support multiple scenarios. When writing the unit tests to match this file, you could even reuse this code!

The other part is how we're listening to the event that gets emitted from the `RecipeTable` component (*lines 69 and 76*). We trigger the `openPreview` function (*lines 26-28*) on the click event. We'll also need a view and a route to be able to navigate to these components.

Let's create a `RecipeView.vue` component in the `src/views` folder that simply loads our component:

```
<script setup lang="ts">
import RecipesList from "@/components/RecipesList.vue";
</script>

<template>
  <RecipesList />
</template>
```

Next, we'll expand the route configuration, which Vuetify generated for us in `src/router/index.ts`:

```
import { createRouter, createWebHistory } from 'vue-router'

const routes = [
  {
    path: '/',
    component: () => import('@/layouts/default/Default.vue'),
    children: [
      {
        path: '',
        name: 'Home',
        component: () => import(/* webpackChunkName: "home" */ '@/
views/Home.vue'),
      },
      {
        path: 'recipes',
        name: 'Recipes',
        component: () => import('@/views/RecipesView.vue')
      },
    ],
  },
]

// …abbreviated
```

Finally, we can update `src/layouts/default/AppBar.vue`:

```
<template>
  <v-app-bar flat>
    <v-btn id="hamburger-activator" icon="mdi-menu"> </v-btn>
    <v-menu activator="#hamburger-activator">
      <v-list>
        <v-list-item>
          <v-btn flat block><router-link to="/">Home</router-link></v-
btn>
          <v-btn flat block
            ><router-link to="/recipes">Recipes</router-link></v-btn
          >
        </v-list-item>
      </v-list>
    </v-menu>

    <v-app-bar-title>
      <v-icon icon="mdi-silverware-fork-knife" />
```

```
      Meal planner
    </v-app-bar-title>
  </v-app-bar>
</template>
```

With these lines of code, we've added a hamburger menu with a toggle. It just works! This is a very powerful feature of using a component library: it not only provides styled components but also provides commonly used patterns for interaction.

I highly recommend checking out the Vuetify documentation as it provides an extensive list of the available components and examples of how to use them. In the app we're building, we're only touching lightly upon the usage of the component library, but you also have the option of optimizing the components so that they fit a more specific goal or purpose.

We can now navigate to two different views and build a limited version of the final app. In the next section, we'll connect the Spoonacular recipes to our app!

Connecting the recipes to our app

In this section, we'll connect the API to our app, which allows users to start planning meals for upcoming days. We'll explore patterns to interact with an app using the Vuetify components.

A bit of additional setup

Because we are going to deal with asynchronous data, we'll add some helper components. First, we'll create an `AppLoader.vue` component in the `src/components` folder, which acts as a loading indicator:

```
<template>
  <v-container class="fill-height" fluid>
    <v-row align="center" justify="center">
      <v-col cols="12" sm="8" md="4">
        <div class="text-center my-8">
          <v-progress-circular
            indeterminate
            color="light-blue"
            :size="80"
            :width="10"
          ></v-progress-circular>
        </div>
      </v-col>
    </v-row>
  </v-container>
</template>
```

And while we're at it, we can also add a dedicated component to handle our links. We'll name it AppLink.vue:

```ts
<script setup lang="ts">
const props = defineProps({
  to: {
    type: String,
    required: true,
  },
});
</script>
<template>
    <router-link :to="to" class="text-light-blue">
        <slot></slot>
    </router-link>
</template>
```

We can insert AppLink into AppBar.vue immediately by replacing router-link with our new component. This component adds style across our applications' links. Note that we're keeping the markup deliberately close to the original router-link:

```ts
<script setup lang="ts">
import AppLink from '@/components/AppLink.vue';
</script>
<template>
  <v-app-bar flat>
    <v-btn id="hamburger-activator" icon="mdi-menu"> </v-btn>
    <v-menu activator="#hamburger-activator">
      <v-list>
        <v-list-item>
          <v-btn flat block><app-link to="/">Home</app-link></v-btn>
          <v-btn flat block
            ><app-link to="/recipes">Recipes</app-link></v-btn
          >
        </v-list-item>
      </v-list>
    </v-menu>
<!-- ... abbreviated -->
</template>
```

Now that we can navigate, let's continue by exposing the data from the API in our application.

Our API connection

We'll now focus on the CalendarCard.vue and CalendarDays.vue components. We'll add functionalities to search for a recipe and add it to a day, view it, and remove it.

We'll start in CalendarCard.vue by adding an event to signal that a user selected a certain date:

```ts
<script setup lang="ts">
import { useFormatDate } from "@/composables/formatters";

const emits = defineEmits(["daySelected"]);

const addRecipeToDay = (card: Card): void => {
  emits("daySelected", card);
}

interface Card {
  date: Date;
}

const props = defineProps<{
  card: Card;
}>();
</script>
<template>
  <v-sheet class="d-flex justify-space-between">
    <v-sheet class="ma-2 pa-2">
      <h2 class="text-h2">{{ useFormatDate(card.date) }}</h2>
    </v-sheet>
    <v-sheet class="ma-2 pa-2">
      <v-btn text @click="addRecipeToDay(card)" icon="mdi-plus"></v-btn>
    </v-sheet>
  </v-sheet>
</template>
```

In the user interface, we've added a button that emits the current card when it's clicked. Now, let's modify CalendarDays.vue so that it picks up this event and shows a dialog to search for recipes:

```ts
<script setup lang="ts">
import { ref } from "vue";
import type { Ref } from "vue";
import CalendarCard from "@/components/CalendarCard.vue";

const props = defineProps({
```

```
  date: {
    type: Date,
    required: true,
  },
  days: {
    type: Number,
    required: false,
    default: 7,
  },
});

interface Card {
  date: Date;
  content: string;
}

const generateCards = (startDate: Date, numberOfDays: number): Card[]
=> {
  const cards: Card[] = [];
  const currentDate = new Date(startDate);

  for (let i = 0; i < numberOfDays; i++) {
    const date = new Date(currentDate.getTime());
    const content = Card ${i + 1};

    cards.push({ date, content });
    currentDate.setDate(currentDate.getDate() + 1);
  }

  return cards;
};
const cards = ref<Card[]>(generateCards(props.date, props.days));

const dialogVisible: Ref<boolean> = ref(false);
const dateSelected: Ref<Date | null> = ref(null);

const recipeDialogOpen = (card: Card): void => {
  dateSelected.value = card.date;
  dialogVisible.value = true;
};

const recipeDialogClose = (): void => {
  dateSelected.value = null;
```

```
    dialogVisible.value = false;
};

</script>

<template>
  <v-table>
    <thead>
      <tr>
        <th class="text-left">Upcoming days</th>
      </tr>
    </thead>
    <tbody>
      <tr v-for="card in cards" :key="card.date.toString()">
        <td class="py-4">
          <calendar-card :card="card" @daySelected="recipeDialogOpen"
/>
        </td>
      </tr>
    </tbody>
  </v-table>
  <v-dialog v-model="dialogVisible" scrollable>
    <v-card>
      <v-card-title> Search for a recipe to add to this day </v-card-
title>
      <v-card-actions>
        <v-btn color="primary" block @click="recipeDialogClose"
          >Close Dialog</v-btn
        >
      </v-card-actions>
    </v-card>
  </v-dialog>
</template>
```

In the updated code, we're storing the state of the dialog in a newly created variable and using a component provided by Vuetify to open a dialog, where we can also close and restore the variables to their initial values.

Selecting a recipe

Next, we'll build a small search component to display in the dialog. This will allow users to search for a recipe. When selected, we'll pass the details of the recipe to the CalendarDays.vue component.

Create a file called `RecipeSearch.vue`:

```ts
<script setup lang="ts">
import { ref, watch } from "vue";
import type { Ref } from "vue";
import { useRecipeSearch } from "@/composables/recipeApi";
import type { RecipeResults } from "@/types/spoonacular";

const emits = defineEmits(["recipeSelected"]);

const searchQuery: Ref<string> = ref("");
const searchResults: Ref<RecipeResults[] | []> = ref([]);

const getSearchResults = async () => {
  const result = await useRecipeSearch(searchQuery.value);
  searchResults.value = result.results;
};

let timeout: ReturnType<typeof setTimeout>;
const debouncedSearch = (): void => {
  clearTimeout(timeout);
  timeout = setTimeout(async () => {
    getSearchResults();
  }, 500);
};

watch(searchQuery, (): void => {
  debouncedSearch();
});

const recipeSelected = (result: RecipeResults): void => {
  emits("recipeSelected", result);
};
</script>

<template>
  <v-car flat>
    <v-card-text>
      <v-text-field v-model="searchQuery" label="Search"></v-text-field>
    </v-card-text>
    <v-divider></v-divider>
    <v-list v-if="searchResults">
```

```
        <v-list-item v-for="(result, index) in searchResults"
:key="index">
        <v-list-item-title @click="recipeSelected(result)"
class="list-item">{{
            result.title
        }}</v-list-item-title>
      </v-list-item>
    </v-list>
  </v-car>
</template>

<style scoped>
.list-item {
  cursor: pointer;
}

.list-item:hover,
.list-item:active {
  text-decoration: underline;
}
</style>
```

We're making use of a debounced watcher, the same as with the Marvel search component. We can use the composable to retrieve the results based on a simple text-based search query and display the results in a list.

When a user clicks on a list item, we'll emit the event and send the corresponding recipe as context to the parent component.

Let's implement the addition and removal of recipes on a day!

Adding and removing a meal

Implementing the search functionality is now very easy! We can import the component in the setup tag and then place the components' tags in the template. We do have to add a listener since we're emitting an event on the search component.

Let's take a look:

```
<script setup lang="ts">
import { ref } from "vue";
import type { Ref } from "vue";
import type { RecipeResults } from "@/types/spoonacular";
```

```typescript
import CalendarCard from "@/components/CalendarCard.vue";
import RecipeSearch from "@/components/RecipeSearch.vue";

interface Today {
  id: number;
  title: string;
  readyInMinutes: number;
}

interface Card {
  date: Date;
  content: string;
  today: Today[];
}

// …abbreviated

const insertRecipeOnDay = (recipe: RecipeResults): void => {
  if (dateSelected.value) {
    cards.value = cards.value.map((card) => {
      if (card.date.getTime() === dateSelected.value?.getTime()) {
        return { ...card, today: [...card.today, recipe] };
      }
      return card;
    });
    recipeDialogClose();
  }
};
</script>

<template>
  <v-table>
    <!-- abbreviated -->
  </v-table>
  <v-dialog v-model="dialogVisible" scrollable>
    <v-card>
      <v-card-title> Search for a recipe to add to this day </v-card-
title>
      <recipe-search @recipeSelected="insertRecipeOnDay" />
      <v-card-actions>
        <v-btn color="primary" block @click="recipeDialogClose"
          >Close Dialog</v-btn
        >
      </v-card-actions>
```

```
    </v-card>
  </v-dialog>
</template>
```

With that, we've added the `recipe-search` component to the page. With the `insertRecipeOnDay` function, we can modify our cards collection by adding the selected recipe (which was emitted through the search component) and adding it to a new property on a card: today.

Before we display recipes for a day, let's also add a method for removing a recipe on a day based on a recipe ID and the date (this way, we can support multiple recipes on one day and similar recipes across multiple days). We can add the following function to the script tag:

```
const removeRecipeFromDay = (recipe: { id: number }, date: Date): void
=> {
  cards.value = cards.value.map((card) => {
    if (card.date.getTime() === date.getTime()) {
      return {
        ...card,
        today: card.today.filter((today) => today.id !== recipe.id),
      };
    }
    return card;
  });
};
```

This function simply modifies the `today` collection by filtering out any recipe that matches the given ID and date. In the template, we'll add a listener for an event on the `CalenderCard.vue` component, like this:

```
<calendar-card :card="card" @daySelected="recipeDialogOpen"
@recipeRemoved="removeRecipeFromDay" />
```

Now, let's update the `CalendarCard.vue` component so that it shows the recipes for a day and then add the option to remove them:

```
<script setup lang="ts">
import { useFormatDate } from "@/composables/formatters";
import AppLink from "./AppLink.vue";

const emits = defineEmits(["daySelected", "recipeRemoved"]);

const addRecipeToDay = (card: Card): void => {
  emits("daySelected", card);
}
```

```
const recipeRemoved = (recipe: Today, date: Date): void => {
  emits("recipeRemoved", recipe, date);
}

interface Today {
  id: number;
  title: string;
  readyInMinutes: number;
}

interface Card {
  date: Date;
  today: Today[];
}

const props = defineProps<{
  card: Card;
}>();
</script>
<template>
  <v-sheet class="d-flex justify-space-between">
    <v-sheet class="ma-2 pa-2">
      <h2 class="text-h2">{{ useFormatDate(card.date) }}</h2>
    </v-sheet>
    <v-sheet class="ma-2 pa-2">
      <v-btn text @click="addRecipeToDay(card)" icon="mdi-plus"></v-
btn>
    </v-sheet>
  </v-sheet>

  <v-col>
    <v-card v-for="today in card.today" :key="today.id" class="my-4">
      <v-card-title>
        <app-link :to="/recipe/${today.id}">{{ today.title }}</app-
link>
      </v-card-title>
      <v-card-text>{{ today.readyInMinutes }} minutes</v-card-text>

      <v-card-actions>
        <v-spacer></v-spacer>
        <v-btn
          text
          icon="mdi-trash-can-outline"
          @click="recipeRemoved(today, card.date)"
```

```
      ></v-btn>
    </v-card-actions>
  </v-card>
  </v-col>
</template>
```

In the template, we've just added an emitter to emit the `recipeRemoved` event that provides the recipe and date as context. Here, we use Vuetify components to create a repeating card layout to show any recipes that are added for that day.

As you can see, we also provide a link to the details page of the recipe, so we need to build one! But before we do that, let's take a look at our app at this point. You should be able to use the app interface to show several upcoming days. When selecting a date, we can search for a recipe using the Spoonacular API and add one or more recipes to our meal planner. We can also remove recipes from our meal planner.

We are not saving anything yet, which means that refreshing the browser or navigating to a different page empties the meal planner! That's something we will work on in the next chapter, but before we do that, we must add the cooking instructions to a separate route.

Let's create a `CookingInstructions.vue` component in the `components` folder with the following contents:

```
<script setup lang="ts">
import { ref, onMounted } from "vue";
import type { Ref } from "vue";
import type { Recipe } from "@/types/spoonacular";

import { useRecipeInformation } from "@/composables/recipeApi";

import AppLoader from "./AppLoader.vue";

const props = defineProps({
  id: {
    type: Number,
    required: true,
  },
  activePanel: {
    type: Number,
    default: 0,
  },
});

const recipe: Ref<Recipe | null> = ref(null);
```

```
const getRecipeDetails = async (id: number): Promise<void> => {
  const data = (await useRecipeInformation(id.toString())) as Recipe;
  recipe.value = data;
};

const panel = ref<number | null>(1);

onMounted(() => {
  getRecipeDetails(props.activePanel);
});
</script>

<template>
  <AppLoader v-if="!recipe" />

  <v-container v-else fluid>
    <v-col>
      <v-img height="200" :src="recipe.image" cover v-if="recipe.
image" />
      <h1 class="text-h3 ma-4">{{ recipe.title }}</h1>
      <v-chip
        class="ma-2 my-4"
        color="primary"
        :key="cuisine"
        v-for="cuisine in recipe.cuisines"
      >
        {{ cuisine }}
      </v-chip>

      <v-expansion-panels variant="accordion" v-model="panel">
        <v-expansion-panel>
          <v-expansion-panel-title class="text-h5"
            >Summary</v-expansion-panel-title
          >
          <v-expansion-panel-text>
            <div v-html="recipe.summary" class="text-body-1"></div>
          </v-expansion-panel-text>
        </v-expansion-panel>
        <v-expansion-panel>
          <v-expansion-panel-title class="text-h5"
            >Instructions</v-expansion-panel-title
          >
          <v-expansion-panel-text>
```

```
                <div v-html="recipe.instructions" class="text-body-1"></
div>
            </v-expansion-panel-text>
          </v-expansion-panel>
        </v-expansion-panels>
      </v-col>
    </v-container>
</template>
```

Note that we are using our `AppLoader` component because the contents will come directly from the API. Other than that, most of the layout uses Vuetify components to display the details coming from the endpoint.

Next, we'll create the view to load this component. In the `views` folder, we'll create a `RecipeView.vue` file:

```
<script setup lang="ts">
import { useRouter } from "vue-router";
import CookingInstructions from "@/components/CookingInstructions.
vue";

const router = useRouter();
const recipeId = Number(router.currentRoute.value.params.id);
</script>

<template>
  <cooking-instructions :id="recipeId" :active-panel="1" />
</template>
```

Let's add it to the router (`src/router/index.ts`) with a new entry, using `id` as a parameter:

```
import { createRouter, createWebHistory } from 'vue-router'

const routes = [
  {
    path: '/',
    component: () => import('@/layouts/default/Default.vue'),
    children: [
      {
        path: '',
        name: 'Home',
        // route level code-splitting
        // this generates a separate chunk (about.[hash].js) for this
route
        // which is lazy-loaded when the route is visited.
```

```
        component: () => import(/* webpackChunkName: "home" */ '@/
views/Home.vue'),
      },
      {
        path: 'recipes',
        name: 'recipes',
        component: () => import('@/views/RecipesView.vue')
      },
      {
        path: '/recipe/:id',
        name: 'recipe',
        component: () => import('../views/RecipeView.vue')
      }
    ],
  },
]

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes,
})

export default router
```

That's it for this section. While we have some functionality, it isn't usable because our data is not persistent. We need to build a stateful application. And since managing state can be tedious to do by hand, we'll make use of Pinia to help us with that!

You may have noticed that, by using Vuetify components, we have less logic in our application to deal with the state of the user interface. Our `script` tags now contain mostly functions that are tied to the features of the application.

This is one of the benefits of using a component library in your application: you can focus on the specific features rather than on building interactive user interface elements.

Using Pinia for state management

In this section, we'll focus on making our application stateful using Pinia. This means we will have to refactor existing code, optimize certain flows, and add new features to our application.

Stateful applications

We use the term *stateful applications* to describe applications that can use, save, and persist data for a certain amount of time. A state can be temporary (while a session lasts) or of a more permanent nature when stored in a database.

The state is contextual to the current user and is typically not shared between users. In short, it is representative of the current user's state of interacting with an application.

Adding Pinia

Pinia is a framework for managing states of applications built using Vue.js 3. It aims to facilitate sharing and interacting with a state or store by leveraging composables and simple syntax.

Let's add Pinia to our project by installing it using the command line:

```
npm install pinia
```

Next, we need to create our Pinia instance and pass it to the app as a plugin. Open the `main.ts` file to make the following changes:

```
/**
 * main.ts
 *
 * Bootstraps Vuetify and other plugins then mounts the App
 */

// Components
import App from './App.vue'

// Pinia
import { createPinia } from 'pinia'

// Composables
import { createApp } from 'vue'

// Plugins
import { registerPlugins } from '@/plugins'

const pinia = createPinia()

const app = createApp(App)

registerPlugins(app)

app.use(pinia)

app.mount('#app')
```

That's it! We can now create our stores and make our app even better.

The first store

We'll start with something small. If you've opened the cooking instructions for a recipe, you may have noticed the loading indicator. That's because the data is coming from an external API. If you refresh the page, you will notice that you have to wait for data to load again. You can see it in the **Network** tab of your developer tools as well. We can use a store to at least cache requests once they've been resolved, to prevent additional requests to the same resource and improve the performance of the app.

For stores, we'll create a folder called `stores` in the `src` folder. Let's add a `cache` folder and create an `index.ts` file. In this file, we'll use the `defineStore` method to create a store called `cache` (the names need to be unique):

```
import { defineStore } from "pinia";

export const useCacheStore = defineStore('cache', () => {
    return { }
});
```

This is how you define any store – we use composable use notation. The composable consists of a (now empty) object that we will complement with the functions we want to expose. We need a function to cache data and one to return cached data, which will look like this:

```
import { defineStore } from "pinia";

export const useCacheStore = defineStore('cache', () => {
    const cachedData = (): void => {}

    const cacheData = (): void => {}

    return { cachedData, cacheData }
});
```

We can define functions and choose which functions we want to return. In this case, we'll return both of the functions.

In our store, we can use native functions from Vue, such as `reactivity`! We'll define a constant named `cache` and, using both of our functions, read from the cache or add data to the cache:

```
import { ref } from "vue";
import { defineStore } from "pinia";

export const useCacheStore = defineStore('cache', () => {
    const cache = ref<any>([]);
```

```
    const cachedData = (key: string): any => {
        try {
            return cache.value[key]
        } catch (e) {
            return undefined;
        }
    }

    const cacheData = (key: string, data: any): void => {
        cache.value[key] = data
    }
    return { cachedData, cacheData }
});
```

Now, this is a very simplistic design, but for our purposes, it works. Let's see how we can implement this in our CookingInstructions.vue component.

In our script tag, we need to import the cache store and initialize the store on a store constant:

```
import { useCacheStore } from "@/stores/cache";
const store = useCacheStore();
```

That's all we need to do to use the methods in our store. Now, we'll modify the getRecipeDetails function so that it only uses the external API if no data is found on the cache by the unique key per response:

```
<script setup lang="ts">
import { ref, onMounted } from "vue";
import type { Ref } from "vue";
import type { Recipe } from "@/types/spoonacular";

import { useRecipeInformation } from "@/composables/recipeApi";

import AppLoader from "./AppLoader.vue";

import { useCacheStore } from "@/stores/cache";
const store = useCacheStore();

const props = defineProps({
  id: {
    type: Number,
    required: true,
  },
});
```

```
const recipe: Ref<Recipe | null> = ref(null);

const getRecipeDetails = async (id: number): Promise<void> => {
  const cacheKey = recipe-details-${props.id};
  if (store.cachedData(cacheKey)) {
    recipe.value = store.cachedData(cacheKey) as Recipe;
  } else {
    const data = (await useRecipeInformation(id.toString())) as
Recipe;
    store.cacheData(cacheKey, data);
    recipe.value = data;
  }
};

const panel = ref<number | null>(1);

onMounted(() => {
  getRecipeDetails(props.id);
});
</script>
```

The cache is not stored on the user's machine but stored in the session of the Vue application. On refresh, the data will be lost, but when navigating backward or forward in the app, the cache will be available. When the data is stored on a unique identifier, and using the store, we can reach the contents of the store from anywhere within the Vue.js application.

Do you remember the incomplete features of the meal planner? Let's build a store to centralize the way we manage those recipes.

The meal planner store

In this section, we'll create a dedicated store for our meal planner capabilities. So, create a folder called planner in the stores folder and create an index.ts file where we'll define our store:

```
import { defineStore } from "pinia";

export const usePlannerStore = defineStore('planner', () => {
})
```

We're going to add some functions to interact with recipes. In this case, we would like persistence so that the information is stored for longer periods. We'll use the useLocalStorage composable from VueUse by installing the dependency in our project:

```
npm install @vueuse/core
```

We want our planner to be persistent (and reactive), so let's add that functionality first:

```
import { ref } from "vue";
import { defineStore } from "pinia";
import { useLocalStorage } from "@vueuse/core"

interface Recipe {
    id: number;
    date: Date;
}

export const usePlannerStore = defineStore('planner', () => {

    const recipes = ref<Recipe[] | any>(useLocalStorage('planner',
[]));

    return { recipes }
})
```

When using the store, we can access the recipes from anywhere by accessing the store and requesting the recipes. We'll add two more methods, for adding and removing recipes:

```
import { ref } from "vue";
import { defineStore } from "pinia";
import { useLocalStorage } from "@vueuse/core"

interface Recipe {
    id: number;
    date: Date;
}

export const usePlannerStore = defineStore('planner', () => {
    const recipes = ref<Recipe[] | any>(useLocalStorage('planner',
[]));

    const addRecipe = (recipe: Recipe) => {
        recipes.value.push(recipe)
    }

    const removeRecipeByIdDate = (options: { id: number, date: Date })
=> {
        const { id, date } = options;
        const recipeIndex: number = recipes.value.findIndex((recipe:
Recipe) => recipe.id === id && new Date(recipe.date).setHours(0, 0, 0,
0) === new Date(date).setHours(0, 0, 0, 0))
```

```
        recipes.value.splice(recipeIndex, 1)
    }

    return { recipes, addRecipe, removeRecipeByIdDate }
})
```

With this in place, we can retrieve the recipes in the `MealPlanner.vue` file:

```ts
<script setup lang="ts">
import { storeToRefs } from "pinia";
import CalendarDays from "./CalendarDays.vue";

import { usePlannerStore } from "@/stores/planner";
const store = usePlannerStore();
const { recipes } = storeToRefs(store);
</script>

<template>
  <calendar-days
    :date="new Date()"
    :days="7"
    :recipes="recipes"
    :key="recipes-${recipes.length}"
  />
</template>
```

Using `storeToRefs` from Pinia ensures that the values from our store are automatically converted into reactive properties! We're passing the recipes down to `CalendarDays.vue`, so let's continue with the implementation.

Now, in the `CalendarDays.vue` component, we're receiving the recipes as a property, but we'll also make sure we can add and remove recipes from the planner. First, we'll focus on processing the recipes by adding the property definition and updating the `generateCards` function, where we'll map the recipes to each generated day:

```
const props = defineProps({
  date: {
    type: Date,
    required: true,
  },
  days: {
    type: Number,
    required: false,
    default: 7,
  },
```

```
  recipes: {
    type: Array,
    required: false,
    value: [],
  },
});

const generateCards = (startDate: Date, numberOfDays: number): Card[]
=> {
  const cards: Card[] = [];
  const currentDate = new Date(startDate);

  for (let i = 0; i < numberOfDays; i++) {
    const date = new Date(currentDate.getTime());
    const content = Card ${i + 1};

    const recipesThisDay = props.recipes?.filter((recipe: any) => {
      const recipeDate = new Date(recipe.date).setHours(0, 0, 0, 0);
      return recipeDate === date.setHours(0, 0, 0, 0);
    }) as Today[];
    cards.push({ date, content, today: recipesThisDay });
    currentDate.setDate(currentDate.getDate() + 1);
  }

  return cards;
};
```

We can't see it in action yet because we have no means of adding recipes to our store. Let's add the store to the component and modify the `insertRecipeOnDay` function so that we can save the recipes in our planner store:

```
const insertRecipeOnDay = (recipe: RecipeResults): void => {
  if (dateSelected.value) {
    store.addRecipe({ ...recipe, date: dateSelected.value });
    recipeDialogClose();
  }
};
```

If you open the app and add a recipe to a date, it gets added just as before, with the difference that on refresh, the item is preserved!

Now, we can do something nifty using our cache store: we can preload the data to the cache to speed up the user experience when opening the cooking instructions after adding them. With our stores combined, it's just a couple of lines.

We'll import the cache store and, using the predetermined key, load the information and write it to the cache store. I tend to move the store references to the top of the file so that I can quickly glance over the capabilities that a component has at its disposal:

```ts
<script setup lang="ts">
import { ref } from "vue";
import type { Ref } from "vue";
import type { Recipe, RecipeResults } from "@/types/spoonacular";
import { useRecipeInformation } from "@/composables/recipeApi";

import { usePlannerStore } from "@/stores/planner";
const store = usePlannerStore();

import { useCacheStore } from "@/stores/cache";
const cacheStore = useCacheStore();

// …abbreviated

const preloadRecipe = async (id: number): Promise<void> => {
  const cacheKey = recipe-details-${id};
  if (!cacheStore.cachedData(cacheKey)) {
    const data = (await useRecipeInformation(id.toString())) as
Recipe;
    cacheStore.cacheData(cacheKey, data);
  }
};

const insertRecipeOnDay = (recipe: RecipeResults): void => {
  if (dateSelected.value) {
    preloadRecipe(recipe.id);
    store.addRecipe({ ...recipe, date: dateSelected.value });
    recipeDialogClose();
  }
};

// …abbreviated
</script>
```

Now, if you've selected a recipe, if you visit the cooking instructions, you'll notice that the content is there instantly! The added value of this approach depends on a couple of factors. If we expect that, on average, users want to navigate to a detailed view after adding a recipe, it makes perfect sense.

In other cases, it only adds an additional API call. So, use this pattern only when needed. In our case, it serves as a demonstration of using the cache store from multiple entry points in our application.

We have to make sure that we can remove the recipe as well. We're going to apply that logic to the card that holds the recipe. We don't have to centralize these functions since we have the store to take care of this for us! Therefore, we can remove the removeRecipeFromDay method (highlighted in the following code) from the CalendarDays component, as well as the event listener in the template:

```
<calendar-card
  :card="card"
  @daySelected="recipeDialogOpen"
  @recipeRemoved="removeRecipeFromDay"
/>
```

Now, we can zoom in on the CalendarCard.vue component to add the ability to remove recipes to/from this component. We'll start by removing the recipeRemoved event (highlighted in the following code) and function:

```
const emits = defineEmits(["daySelected", "recipeRemoved"]);

// …abbreviated

const recipeRemoved = (recipe: Today, date: Date): void => {
  emits("recipeRemoved", recipe, date);
};
```

We'll create a new function after importing the planner store in this component. In the function, we'll call removeRecipeByIdDate from the store and pass the current context:

```
<script setup lang="ts">
import { useFormatDate } from "@/composables/formatters";

const emits = defineEmits(["daySelected"]);

import { usePlannerStore } from "@/stores/planner";
const store = usePlannerStore();

import AppLink from "./AppLink.vue";

interface Today {
  id: number;
  title: string;
  readyInMinutes: number;
}

interface Card {
  date: Date;
```

```
  today: Today[];
}

const props = defineProps<{
  card: Card;
}>();

const addRecipeToDay = (card: Card): void => {
  emits("daySelected", card);
};

const removeFromDay = (recipes: { id: number; date: Date }): void => {
  const { id, date } = recipes;
  store.removeRecipeByIdDate({ id, date });
};
</script>
```

In the template, we'll modify the removal button by calling the new function with the correct parameters:

```
<v-card-actions>
  <v-spacer></v-spacer>
  <v-btn
    text
    icon="mdi-trash-can-outline"
    @click="removeFromDay({ id: today.id, date: card.date })"
  ></v-btn>
</v-card-actions>
```

With this, we can document and use actions in places that make sense within the context of the app. With the store, we've created a central state where we can access and manipulate different components without needing to pass properties from one component to the next. Having a library such as Pinia integrated with the Vue environment makes it a straightforward choice since it can fully leverage the reactive capabilities of Vue out of the box!

Computed store values

To stress the reusability aspect, we will finally take a look at RecipesList.vue, which we've filled with static content. Since the meal planner only shows the upcoming few days, we may want to show the full extent of past and future planned recipes.

We have two tabs in RecipesList – one for showing upcoming recipes and one for past recipes. While we could ingest all the recipes from the store and apply some sorting with a centralized store, it makes more sense to handle it close to the source.

We can use computed values in stores, just like Vue components! To display them, we'll internally sort the recipes and provide two values:

```
import { ref, computed } from "vue";
import { defineStore } from "pinia";
import { useLocalStorage } from "@vueuse/core"

import type { Recipe } from "@/types/spoonacular";
interface RecipeList extends Recipe {
    date: Date;
}

export const usePlannerStore = defineStore('planner', () => {
    const recipes = ref<Recipe[] | any>(useLocalStorage('planner',
[]));

    const recipesSortedByDate = () =>
        recipes.value.sort((a: { date : Date }, b: { date: Date }) =>
new Date(a.date).getTime() < new Date(b.date).getTime() ? -1 : 1)

    const pastRecipes = computed(() => {
        const sorted = recipesSortedByDate();
        return sorted.filter((recipe: RecipeList) => {
            const date = new Date(recipe.date);
            return date < new Date();
        }) as RecipeList[]
    })

    const futureRecipes = computed(() => {
        const sorted = recipesSortedByDate();
        return sorted.filter((recipe: RecipeList) => {
            const date = new Date(recipe.date);
            return date >= new Date();
        }) as RecipeList[];
    })

    const addRecipe = (recipe: Recipe) => {
        console.log('addRecipe', recipe)
        recipes.value.push(recipe)
    }

    const removeRecipeByIdDate = (options: { id: number, date: Date })
=> {
        const { id, date } = options;
```

```
        const recipeIndex: number = recipes.value.findIndex((recipe:
Recipe) => recipe.id === id && new Date(recipe.date).setHours(0, 0, 0,
0) === new Date(date).setHours(0, 0, 0, 0))
        recipes.value.splice(recipeIndex, 1)
    }

    return { recipes, pastRecipes, futureRecipes, addRecipe,
removeRecipeByIdDate }
});
```

As I mentioned earlier, it is perfectly fine to have non-exposed methods in our store. With the final `return` statement, we can decide what methods to expose on the module. The functions are nothing extraordinary. After sorting by date, they filter and return dates from the past or future.

Let's connect the dates to the `RecipesList.vue` component. We will focus on the `script` tag since we built the interface previously. We'll remove the highlighted parts from the following code:

```
<script setup lang="ts">
import { ref, computed, onMounted } from "vue";
import type { Ref } from "vue";
import { storeToRefs } from "pinia";

import type { Recipe } from "@/types/spoonacular";
interface RecipeList extends Recipe {
  date: Date;
}

import RecipeTable from "./RecipeTable.vue";

const { pastRecipes, futureRecipes } = storeToRefs(store);

// return a date in the future:
const addDays = (days: number): Date => {
  const date = new Date();
  date.setDate(date.getDate() + days);
  return date;
};

// generate some mock data for now:
const recipes = [
  { id: 1, title: "test", date: addDays(1) },
  { id: 2, title: "test2", date: addDays(1) },
  { id: 2, title: "test3", date: addDays(-1) },
];
```

```
const openPreview = (recipe: { title: string }): void => {
  console.log(`opening recipe ${recipe.title}`);
};

const pastRecipes = computed(() =>
  recipes.filter((recipe: RecipeList) => {
    const date = new Date(recipe.date);
    return date < new Date();
  })
);

const futureRecipes = computed(
  () =>
    recipes.filter((recipe: RecipeList) => {
      const date = new Date(recipe.date);
      return date >= new Date();
    }) as RecipeList[]
);
const tab: Ref<string> = ref("upcoming");

onMounted(() => {
  if (futureRecipes.value.length === 0) {
    tab.value = "past";
  }
});
</script>
```

Without the mockup code, it looks a lot more readable already! We'll also connect the `openPreview` event to another dialog. We'll just reuse the existing `CookingInstructions.vue` component, but you could also consider creating a specific preview of your own.

Let's change the `script` tag so that it matches the following:

```
<script setup lang="ts">
import { ref, onMounted } from "vue";
import type { Ref } from "vue";
import { storeToRefs } from "pinia";
import type { Recipe } from "@/types/spoonacular";

import { usePlannerStore } from "@/stores/planner";
const store = usePlannerStore();

import RecipeTable from "./RecipeTable.vue";
import CookingInstructions from "./CookingInstructions.vue";
```

```
import AppLink from "./AppLink.vue";

const { pastRecipes, futureRecipes } = storeToRefs(store);

const dialogVisible: Ref<boolean> = ref(false);
const selectedRecipe: Ref<Recipe | null> = ref(null);

const openPreview = (recipe: Recipe): void => {
  selectedRecipe.value = recipe;
  dialogVisible.value = true;
};

const tab: Ref<string> = ref("upcoming");

onMounted(() => {
  if (futureRecipes.length === 0) {
    tab.value = "past";
  }
});
</script>
```

Then, we'll add a dialog from Vuetify to the template code to show CookingInstructions:

```
<template>
  <div v-if="pastRecipes.length === 0 && futureRecipes.length === 0">
    No recipes yet. Add some to your planner!
  </div>
  <div v-else>
   <!-- abbreviated -->
    </v-window>

    <v-dialog v-model="dialogVisible" class="dialog" scrollable>
      <v-card v-if="selectedRecipe">
        <cooking-instructions :id="selectedRecipe.id" />
        <v-card-actions>
          <v-btn text>
            <app-link :to="/recipe/${selectedRecipe.id}"
              >Cooking instructions</app-link
            ></v-btn
          >
          <v-spacer />
          <v-btn @click="dialogVisible = false" icon="mdi-close"></v-
btn>
        </v-card-actions>
```

```
        </v-card>
      </v-dialog>
   </div>
</template>
```

With this in place, navigating from the preview to the Cooking instructions page is almost instant again since we're caching the contents and only have to load them on the first request. Using the store, we can interact with the date in a centralized and reusable way. Let's try and solidify this by adding a final component to our app.

Rating the recipes

As a final addition, we'll demonstrate the reusability of composables by adding a rating feature to every recipe. It will store the recipe ID and a rating of 1 through 5 stars. We should be able to read and update the rating from anywhere in the application.

First, we'll create the store. For that, we'll add a folder called `rating` with the following contents in the `index.ts` file:

```
import { ref } from "vue";
import { defineStore } from "pinia";
import { useLocalStorage } from "@vueuse/core"

interface Rating {
    id: number;
    rating: number;
}

export const useRatingStore = defineStore('rating', () => {
    const ratings = ref<Rating[] | any>(useLocalStorage('rating',
[]));

    const getRatingById = (id: number) => {
        const rating = ratings.value.find((rating: Rating) => rating.
id === id)
        return rating?.rating;
    }

    const saveRating = (rating: Rating) => {
        const ratingIndex = ratings.value.findIndex((r: Rating) =>
r.id === rating.id)
        if (ratingIndex === -1) {
            ratings.value.push(rating)
        } else {
            ratings.value[ratingIndex] = rating
```

```
        }
    }

    return { getRatingById, saveRating }
})
```

Now, we'll create a component called `RecipeRating.vue`. It will use the rating store and, based on a provided ID, retrieve any ratings for that ID. It will also allow you to add a new rating for that particular ID, overwriting the old value.

Vuetify has a premade component for this, so when we're combining stores and a component library, we can quickly build interactive components:

```ts
<script setup lang="ts">
import { ref } from "vue";
import type { Ref } from "vue";

import { useRatingStore } from "@/stores/modules/rating";
const store = useRatingStore();

const props = defineProps<{
  id: number;
  readonly?: boolean;
}>();

const rating: Ref<number> = ref(store.getRatingById(props.id));

const saveRating = () => {
  store.saveRating({id: props.id, rating: rating.value});
};
</script>
<template>
  <div class="flex items-center">
    <v-rating
      v-model="rating"
      color="light-green"
      :readonly="readonly"
      half-increments
      item-aria-label="This item is rated {0} of {1}"
      hover
      @click="saveRating"
    ></v-rating>
  </div>
</template>
```

The important parts are highlighted, although you should be familiar with them by now. We can add this component anywhere in our application where we have access to a recipe ID.

Let's start with `CookingInstructions.vue`. We need to import the component and then add it to our template:

```ts
<script setup lang="ts">
// …abbreviated

import AppLoader from "./AppLoader.vue";
import RecipeRating from "./RecipeRating.vue";

// …abbreviated
</script>

<template>
  <app-loader v-if="!recipe" />

  <v-container v-else fluid>
    <v-col>
      <v-img height="200" :src="recipe.image" cover v-if="recipe.
image" />
      <h1 class="text-h3 ma-4">{{ recipe.title }}</h1>

      <recipe-rating :id="recipe.id" />

      <!-- abbreviated -->

    </v-col>
  </v-container>
</template>
```

Feel free to add the `rating` component on other mentions as well. It's just two lines of code!

With that, we've iterated over our Marvel app by utilizing a component library to speed up development time. It allows us to build interfaces with relative ease. We've also refactored and updated our code since our needs have changed. This situation reflects how real-world code bases evolve.

Summary

In this chapter, we combined two concepts to build an application that scales well in development. As we've seen, by adopting more and more of the principles of both the component library as well as a centralized store, the more readable and simplified our code becomes.

Using a component library such as Vuetify provides us with a quick way of adding interactive elements to a user interface that are well tested and easy to use out of the box. Coming with means of customization and theming, we can make sure that our implementation follows any style guide.

This would be a good time to try your hand at customizing the user interface, either by setting up themes and styles or just by using the classes and properties on existing components.

By adding state to our app, we've made it usable and reusable for our users. In our example, we've stored our data in the browser, which isn't as portable. However, it does give us a practical look at dealing with data and caching resources. With the methods in a central place, it becomes easier to refactor the way we would store data.

We've deliberately not built a perfect app from the beginning and instead demonstrated how a refactoring process evolves with the needs and features of app development. Admittedly, it's still not perfect at this point. As an extra exercise, you could try and see whether you can apply some of the lessons we've learned so far.

In the next chapter, we'll solidify our Vuetify knowledge by building another application using the component library. We'll make things more interesting by building a data resource. We'll learn about not just reading from an endpoint but also writing and storing data by building a simple fitness tracker!

6
Creating a Fitness Tracker with Data Visualization

Up until this point, we've relied on stateless apps or storing the state on the users' browser. In this chapter, we'll cover using a database to store data in a centralized place, and we'll learn how to modify and read from the data source. We'll use the opportunity to incorporate some data visualizations as well, using a third-party library.

While we're using a database and have to set up tables, this is by no means a guide to production-ready database configuration and management. I suggest brushing up on those skills in different ways. It does serve as a valuable prototype to familiarize yourself with patterns concerning database handling.

Again, we'll build upon the knowledge we've acquired so far, and we'll incorporate composables, a store, and a component library to build our product.

In this chapter, we'll cover the following topics:

- Creating dashboards and reports
- Retrieving data with Supabase
- Storing data using Supabase
- Adding various visualizations using `vue-chartjs`

Technical requirements

There's some overlap in requirements from the previous chapter. We'll make use of **Vuetify** (`https://vuetifyjs.com/en/`) and **Pinia** (`https://pinia.vuejs.org/`). For storing data, we'll make use of **Supabase** (`https://supabase.com/`), which is an open source database provider with built-in authentication. For the database, I've prepared a script to create databases and another one to add example data.

Here's the GitHub link: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/tree/main/06.fitness`.

We'll cover those steps during the chapter. Lastly, for data visualization, we'll install and use `vue-chartjs` (`https://vue-chartjs.org/`), which is a Vue-compatible wrapper for the `chart.js` (`https://www.chartjs.org/`) library.

Creating a client

In order to start our project, we'll use the Vuetify installer, just as we did in the last chapter. Here's the command for that:

```
npm create vuetify
```

Choose `vue-fitness-tracker` as the project name and select these options, as shown in the following screenshot:
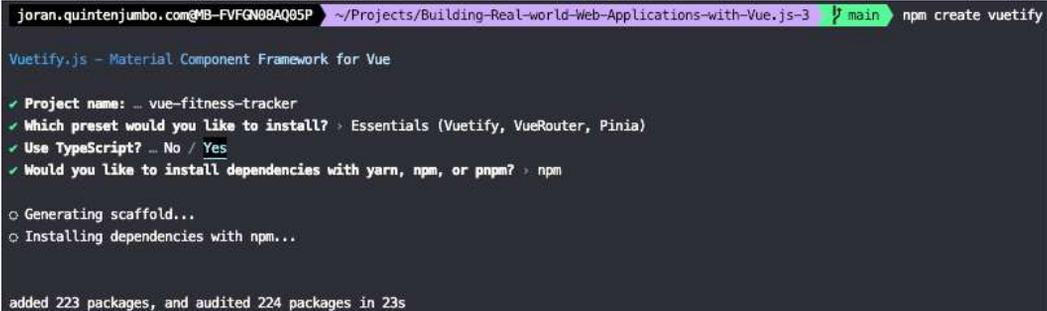
Figure 6.1 – Setting up the Vuetify project

With our project initialized, we'll create and configure a database to store our data.

Setting up the database

After registering for a free account on **Supabase** (`https://supabase.com/`), you'll end up in the dashboard to create a new project. Let's use `fitness-tracker` as the name and choose a strong database password. For the region, pick one that is geographically close to you for better latency. We'll stick with the free plan!

On the next page (*Figure 6.2*), you will see the project API keys:
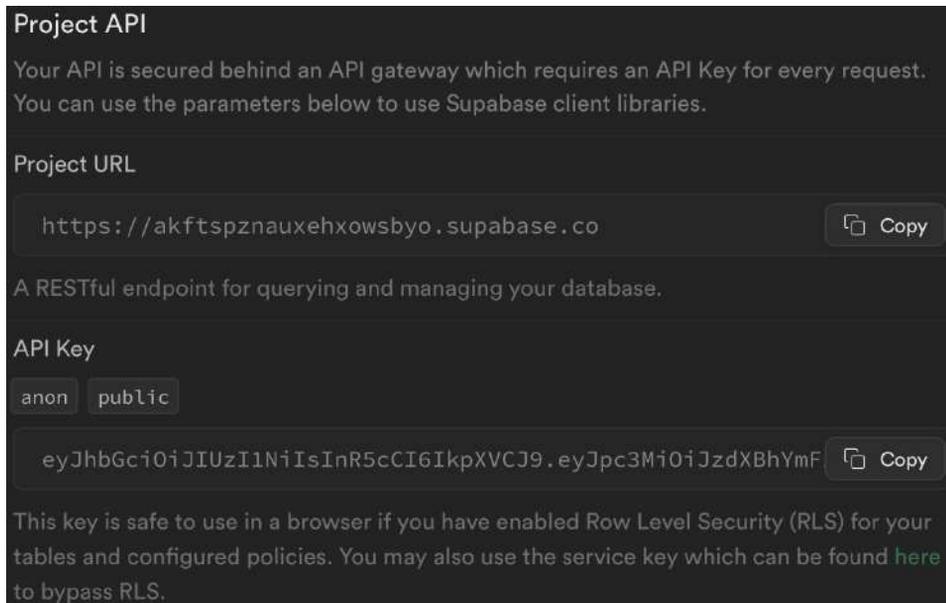
Figure 6.2 – Overview of the project API keys

We'll store them in our `.env` file in the root of our project:

```
VITE_SUPABASE_URL=YOUR_SUPABASE_URL
VITE_SUPABASE_ANON_KEY=YOUR_SUPABASE_ANON_KEY
```

Note that sharing keys like this via a client-side app always exposes them to the public. Luckily, Supabase has its own means of ensuring authentication while interacting with the database.

I've created a script that sets up the database with the table structure for our app. Via the dashboard and SQL editor, you can add and execute the query from the `example-structure.sql` file in the example repository, as shown in *Figure 6.3*:
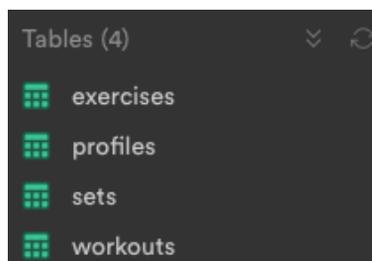
Figure 6.3 – Once the query is successfully executed, you should
see four tables in the table editor overview

Once that's executed, you can set up some data using the example-exercises.sql script, as seen in *Figure 6.4*:


```sql
1    INSERT INTO
2        exercises (name, color)
3    VALUES
4        ('Barbell squat','#ef476f'),
5        ('Overhead press', '#ffd166'),
6        ('Deadlift', '#06d6a0'),
7        ('Bench press', '#118ab2'),
8        ('Cable crunch', '#073b4c');
9
10   SELECT * FROM exercises;
```

Results ∨

id	created_at	name	color
"7fa3219e-9127-4	"2023-07-05 20::	"Barbell squat"	"#ef476f"
"8d797a0e-6dd9-4	"2023-07-05 20::	"Overhead press'	"#ffd166"
"c428f7ec-165f-4	"2023-07-05 20::	"Deadlift"	"#06d6a0"
"e752a532-7140-4	"2023-07-05 20::	"Bench press"	"#118ab2"
"0570e7f1-157a-4	"2023-07-05 20::	"Cable crunch"	"#073b4c"

Figure 6.4 – Inserting example exercises into the exercise table

To streamline interactions with the database, we'll use the Supabase JavaScript client by installing the dependency:

```
npm install @supabase/supabase-js
```

Let's turn that package into a composable that handles our database connection within the app. In src/composables, we'll create a supabase.ts file and add the following contents:

```typescript
import { createClient } from '@supabase/supabase-js'

const supabaseUrl = import.meta.env.VITE_SUPABASE_URL
const supabaseAnonKey = import.meta.env.VITE_SUPABASE_ANON_KEY

export const useSupabaseClient = createClient(supabaseUrl,
supabaseAnonKey);
```

One final change I made was cleaning up the `layouts` folder with the boilerplate components. I deleted all files except `Default.vue` and updated its contents to match this:

```
<template>
  <v-app>
    <v-main>
      <router-view />
    </v-main>
  </v-app>
</template>
```

That's our baseline to start building our app. Next, we're going to make sure that the app shows individual results, using the built-in tooling that Supabase provides.

Handling the user

The goal of the app is to allow for individual metrics to be tracked and viewed, so for that purpose, we need to make sure that we can identify our users. Supabase supports authentication out of the box, and we're going to use a very basic method: a **magic link**.

A magic link allows you to sign up and sign in with just a valid email address. On logging in, the service sends an email containing a unique identifier, and when clicked, the user is verified to that email address. In our case, the backend handles verifying whether it's a new user or an existing one, which is perfect for our use case.

Since we can identify users, we need to connect our app to retrieve information that Supabase provides. We can also introduce authentication to make sure that users have access to parts that they want to use or visit.

User store

We will want to have access and the ability to update the status of the user at all times, so we'll set up a user store in Pinia that will keep track of the current state and provide actions on updating the state and logging in and out.

We'll go over the contents of the store, after creating a `user.ts` file in the `src/store` folder with the following contents: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.1-user.ts`.

In the session (*line 11*), we'll store the authentication state of the user. It can either be `null` for not logged in or the state can hold an object (as defined in the `UserSession` interface), which gets populated by the Supabase authorization service.

With the `login` (*lines 13-20*) and `logout` (*lines 22-30*) methods, we are calling the Supabase authentication services and executing a provided callback function. We will see these in action shortly!

To store the user, we have the `insertProfile` function (*lines 32-44*), which upserts any authenticated user to our database for future reference.

Note

Storing personal data may be subject to local law and governance. Be very diligent and transparent in what you store, why you're storing it, and how you remove personal data.

`setUserSession` (*lines 47-49*) simply passes the data to the state for further reference. Lastly, `userIsLoggedIn` (*lines 51-58*) checks whether the current session data is still valid, and if not, it returns `false`. We can use this for quick assessments on displaying user interface elements.

Having the store in place, we can incorporate profiles in our app with some sensible safety measures.

Authenticating users

Let's create a form where the user can provide an email address that will result in a **one-time password** (**OTP**) generated by Supabase. We'll create a form called `FormLogin.vue` in the `components` folder: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.2-FormLogin.vue`.

As you can see, we're using our user store to dispatch a `login` action (*lines 11-14*, *25*) with the email address provided by the user. The template is built with Vuetify components. It holds a `form` and a `dialog` component that is triggered on submission, to inform the user to look at their email.

The OTP login sends a login link to the provided email address, which means the user will enter the app from an external link. We need to make sure to try to validate the user's session when the app gets loaded. For that, we'll update the `App.vue` file in the root:

```ts
<script setup lang="ts">
import { onMounted } from "vue";
import { useUserStore } from "@/store/user";
import { useSupabaseClient } from "@/composables/supabase";

const userStore = useUserStore();
onMounted(async () => {
  const { data } = await useSupabaseClient.auth.getSession();

  if (data && data.session && data.session.user) {
    await userStore.insertProfile(data.session);
```

```
    userStore.setUserSession(data.session);
  }

  useSupabaseClient.auth.onAuthStateChange((_, _session) => {
    userStore.setUserSession(_session);
  });
});
</script>


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

In this script, we'll validate the user session via Supabase. On receiving data, we store it on the user store and use the store to upsert a profile in our database. We also keep track of the state change, in order to handle updated tokens or invalidation.

Protected routes

Having access to the state of the user, we can use the `beforeEnter` lifecycle hook to validate if a user is allowed access to a route. The `beforeEnter` method acts as middleware and executes a function whereby you can decide how to handle the route change.

We'll first create a view for the login state, called `Login.vue`, in the `views` folder:

```
<script lang="ts" setup>
import FormLogin from '@/components/FormLogin.vue';
</script>


<template>
  <form-login />
</template>
```

In the router file, we'll add a function called `loginGuard` (*lines 7-14*) to check whether the user is logged in, and we'll call that function on the `beforeEnter` method for protected routes (*line 37*). If the user session is present, you will be allowed to follow the route. Otherwise, you will be redirected to the newly added `loginRoute` function (*lines 12*, *39-44*): `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.3-router-index.ts`.

If you run the development app, on the first visit, you will be presented with a login form since you are not yet authenticated, as shown in *Figure 6.5*:

Figure 6.5 – The user is not logged in and is redirected to the /login route

After providing your email address, a fullscreen popup appears to direct our user to the next steps. If you open your email client, you should receive an email containing a magic link shortly:

Figure 6.6 – Default email containing a magic link

Clicking on the magic link opens a new browser window and should now direct you to the home page, which is only accessible for logged-in users:

Figure 6.7 – The home page is only visible to logged-in users

This means if you're seeing the Vuetify default home page, you are a logged-in user! You have no way of logging out for now, so let's work on that to complete the authentication flow.

Logging out

To complete the user flow, we'll add a feature for the user to log out as well. For that, we'll add a menu with a button that is attached to the user logout method on the store.

Let's create an `AppMenu.vue` file in the `src` folder and add **Log out** and **Log in** buttons. We'll expand upon the menu later as well: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.4-AppMenu.vue`.

In the menu, depending on the user state (*lines 5-6*, *10*, *18*), we'll show a button to either log in or log out. With some small modifications to our `App.vue` file, we can quickly include the `AppMenu.vue` file for our app (*lines 6*, *21-26*): `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.5-AppMenu.vue`.

We have now completed the user flow in our app. We can have new users logging in and existing users logging in, and logged-in users can sign out of the app. As you can see, we'll let Supabase handle the logic, and we're just consuming the data from Supabase.

This is a common pattern in frontend development, to leave the authentication to the server and never the client. For our upcoming features, we don't have to worry about who has access to what data since we've configured Supabase (with **Row Level Security** (**RLS**) policies (`https://supabase.com/docs/guides/auth#row-level-security`)) and the authentication methods to take care of the data for us.

Now that we have a way for users to sign up and log in, we can start adding features for adding personal data to the app.

App state

To make it easier for us to control the state of the app, we'll add a new store to track the current state of the user interface.

Vuetify has created a placeholder app store for us in the `store/app.ts` file, so we'll add some features to handle page transitions (*lines 28*, *30-35*), toggling a menu (*lines 18*, *20-26*) and controlling a dialog (*lines 37-55*) for app-level notifications: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.6-app.ts`.

Having access to these kinds of user interface utilities in a centralized place eliminates the need to repeat certain patterns in our app, such as showing or hiding a dialog. It means that those utilities are part of the app and are therefore available throughout the whole app.

Centralized dialog

Let's update the `FormLogin.vue` file to make use of the store options on the app level. We can clean up the existing dialog options and replace them with calling store methods: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.7-FormLogin.vue`.

As you can see, by using the generic app store (*lines 5*, *8*), we can now also easily add additional dialogs; for instance, when an email address is missing (*lines 13-17*) or when the OTP has been sent (*lines 18-24*). The only thing we need to do is add a central place to show the dialog, and we'll open and modify `App.vue` for that: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.8-App.vue`.

This way, we have a dialog (*lines 13*, *31-39*) that's part of the app, and we can control it from anywhere using our store (*lines 4*, *11*)! The same goes for the app menu, so let's modify our app to have centralized control over the app menu.

Centralized app menu

We can apply a similar pattern to a menu. We'll convert it to a drawer link feature that slides in and out of view. Let's start with modifying `AppMenu.vue` by adding the necessary app store references:

```ts
<script setup lang="ts">
import { storeToRefs } from "pinia";
import { useUserStore } from "@/store/user";
import { useAppStore } from "@/store/app";

const userStore = useUserStore();
const appStore = useAppStore();

const { userIsLoggedIn } = storeToRefs(userStore);
const { drawer } = storeToRefs(appStore);

const goToPage = (page: string): void => {
  appStore.navigateToPage(page);
};
</script>

<template>
  <v-navigation-drawer v-model="drawer" app>
    <v-list dense v-if="userIsLoggedIn">
      <v-list-item @click="userStore.logout()">
        <template v-slot:prepend>
```

```
          <v-icon icon="mdi-account-arrow-right"></v-icon>
        </template>
        <v-list-item-title>Log out</v-list-item-title>
      </v-list-item>
    </v-list>
    <v-list dense v-else>
      <v-list-item @click="goToPage('/login')">
        <template v-slot:prepend>
          <v-icon icon="mdi-account"></v-icon>
        </template>
        <v-list-item-title>Login</v-list-item-title>
      </v-list-item>
    </v-list>
  </v-navigation-drawer>
</template>
```

In the template, we've wrapped the menu in a `navigation-drawer` component from Vuetify, which uses the `drawer` state variable to display as opened or closed. We've also replaced the `router-link` component with a method for navigating to new pages.

To complete the setup, we need to modify the `App.vue` file to adapt to the new interface and finalize the app layout:

```
<script setup lang="ts">
// ...abbreviated

const userStore = useUserStore();
const appStore = useAppStore();

const { pageTitle, dialog } = storeToRefs(appStore);
const currentYear = new Date().getFullYear();

onMounted(async () => {
  // ...abbreviated
});
</script>

<template>
  <v-app>
    <app-menu />
    <v-app-bar app style="position: relative">
      <v-app-bar-nav-icon @click="appStore.toggleDrawer()"></v-app-
bar-nav-icon>
      <v-toolbar-title>🏋 Fittest Pal - {{ pageTitle }}</v-toolbar-
title>
```

```
    </v-app-bar>

    <v-main>
      <!-- abbreviated -->
    </v-main>

    <v-footer app>
      <span>&copy; {{ currentYear }} 🏋 Fittest Pal Fitness Tracker</
span>
    </v-footer>
  </v-app>
</template>
```

So far, we have the foundation of an app that supports a personalized experience, depending on authentication by a third party, and the ability to have both public and secured routes. We have some centralized features to control the state of the app's user interface. The next step will be to add features where the user can insert their own data!

We've already prepared and set up our database with tables and some prefilled exercises. Feel free to take a look at the tables and the contents of the exercises table, because it will help you understand our next steps.

We have a table exercise where different types of gym exercises are stored. The user data (limited to just an email address) is stored in the table. If you've signed up via the login form, you should see your email address already there! We have workouts where the training is logged per user, and the sets table combines performed exercises with a workout. Lastly, we have added a view for a dashboard, which we'll build later on.

Now, let's make sure users can add data to the database by building an exercise tracker.

Exercise tracking

Let's add a new route to have our users add a routine to the database. Let's start by adding a new route entry on the routes:

```
    {
      path: 'track',
      name: 'Track',
      component: () => import(/* webpackChunkName: "track" */ '@/
views/Track.vue'),
      beforeEnter: loginGuard
    },
```

As you can see, this is a page that is only accessible by authenticated users. Our entry also means that we need to create a view, called `Track.vue`, so let's continue:

```ts
<script lang="ts" setup>
import TrackExercise from "@/components/TrackExercise.vue";
</script>

<template>
  <track-exercise />
</template>
```

We'll complete the initialization with the creation of an empty `TrackExercise.vue` file in the `components` folder, and we'll focus on creating an interface that matches the way we want to track activities.

To navigate to our route, we can modify our `AppMenu.vue` file. Since we can expect some more menu items, we can define a pattern for items in the `script` block and have the template repeat over those items. We'll start with a single item in the collection, like this:

```ts
<script setup lang="ts">
// ...abbreviated
const menuItems = [
  {
    icon: "mdi-dumbbell",
    title: "Track",
    page: "/track",
  }
];
</script>
```

In our template, we'll expand `v-list` for logged-in users to iterate over our `menuItems` collection:

```
<v-list dense v-if="userIsLoggedIn">
  <v-list-item
    v-for="item in menuItems"
    :key="item.title"
    @click="goToPage(item.page)"
  >
    <template v-slot:prepend>
      <v-icon :icon="item.icon"></v-icon>
    </template>
    <v-list-item-title>{{ item.title }}</v-list-item-title>
  </v-list-item>
  <v-list-item @click="userStore.logout()">
    <template v-slot:prepend>
```

```
        <v-icon icon="mdi-account-arrow-right"></v-icon>
      </template>
      <v-list-item-title>Log out</v-list-item-title>
    </v-list-item>
  </v-list>
```

We can now, from the menu, navigate to our new page in the app.

Next, we can define input fields. We'll start by creating a date picker for our users.

Selecting a date

We want to add a date to our workout. Vuetify at the moment supports datepicker as an experimental feature. We need to explicitly import it into our newly created TrackExercise. vue file, and in addition, we'll configure some variables to keep track of the user interface state as well as the selected date:

```
<script setup lang="ts">
import { ref } from "vue";
import type { Ref } from "vue";
import { VDatePicker } from "vuetify/labs/VDatePicker";

const showDialogDate: Ref<boolean> = ref(false);
const selectedDate: Ref<any[] | undefined> = ref(undefined);
</script>
```

In our template, we'll build controls to add a workout, beginning with a date selection:

```
<template>
  <v-container>
    <v-row class="align-center justify-space-between mb-6">
      <v-btn @click="showDialogDate = true">
        <span v-if="selectedDate">Change date</span>
        <span v-else>Select date</span>
      </v-btn>
      {{ selectedDate }}

      <v-dialog v-model="showDialogDate" center>
        <v-date-picker
          v-model="selectedDate"
          show-adjacent-months
          @click:cancel="showDialogDate = false"
          @click:save="showDialogDate = false"
          style="margin: 0 auto"
        ></v-date-picker>
```

```
        </v-dialog>
      </v-row>
    </v-container>
  </template>
```

This should look pretty familiar. We've added a button that controls a dialog and have some configuration for the dialog. The reason we're not using our app-wide dialog is that this contains more advanced content and has hooks attached. The app dialog is meant to display short messages to our users.

If you try this out, you'll notice that selecting a date results in an unformatted date shown in the interface. We'll fix that before we move on by using the `computed` method and the browser's built-in `Intl` API:

```ts
<script setup lang="ts">
import { ref, computed } from "vue";
// ... abbreviated

const formattedDate: Ref<string> = computed(() => {
  if (selectedDate?.value?.length) {
    return new Intl.DateTimeFormat("en-US", {
      weekday: "long",
      year: "numeric",
      month: "long",
      day: "numeric",
    }).format(selectedDate.value[0]);
  }
  return "";
});
</script>
```

In our template, we'll replace `{{ selectedDate }}` with a nicely formatted date representation:

```
<h4 class="text-h5">{{ formattedDate }}</h4>
```

Adding a routine

For adding a routine, we want the user to be able to select a routine from our database. We want to offer something of a routine picker, so let's build one! The component should read the exercises from the database and have a user select one to add properties of a set (weight and repetitions).

With a need for centralized data, we can create a store for all our fitness-related data and methods. Let's create a `fitness.ts` file in the store, and we'll start with retrieving exercises from the database: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.9-fitness.ts`.

The store exposes the exercises and a method of retrieving them. Since we want to immediately populate the `exercises` list, we can call the `getExercises` method (*lines 13-28*, *29*) when initializing the store!

We'll add more of this store in the future, but for now, we can start using the data in a new component: an **exercise picker**. In a new file in the `components` folder called `SelectExercise.vue`, we'll import the store and use it to populate a Vuetify `select` component:

```ts
<script setup lang="ts">
import { storeToRefs } from "pinia";
import { useFitnessStore } from '@/store/fitness'
const fitnessStore = useFitnessStore()

const { exercises } = storeToRefs(fitnessStore)
</script>

<template>
  <v-select
    v-if="exercises"
    label="Select exercise"
    :items="exercises"
    item-title="name"
    item-value="id"
  ></v-select>
</template>
```

Very straightforward! We access the store and map the values to the `v-select` component. In addition to `exercise`, we want the user to be able to add `weight` and `repetitions` as part of a routine. So, let's wrap our created component in a parent called `AddRoutine.vue`: https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.10-AddRoutine.vue.

Let's break this down before we move to the template. We have the `exercise` (*line 10*) and `routine` (*line 9*) variables, where a routine consists of an exercise in combination with a set of weights and repetitions.

An example of `routines` could be similar to this:

```
{ "exercise": "7fa3219e-9127-4189-ae30-d340aaf0f9e6",
  "routines": [
    { "weight": "10", "repetitions": "10" },
    { "weight": "10", "repetitions": "10" }
  ]
}
```

The flow starts with selecting an exercise, and the watcher (*lines 38-40*) then prepares the routine to be filled with new values (clearing the `routines` array). The interface in the template is modeled to the weight and repetitions, as you will see in a moment.

As usual, several rows can be added to the `routines` property using the `addRow` function (*lines 16-21*). If a user makes an error, a row can be removed from the property (*lines 23-25*).

The `add` function (*lines 33-36*) emits the `routine` object to the parent component and resets any values on the form.

In the template part, we start with the `select-exercise` component (*line 48*) to trigger the flow, and we use Vuetify expansion panels to show the forms for adding a set (*lines 52-86*) as well as give a summary of sets that you've added (*lines 87-131*).

The second expansion panel also uses a Vuetify badge to indicate the number of unsaved changes (*lines 92-100*). At the bottom, there's a button (*lines 135-144*) that calls the `emit` function to have the `routine` object sent to the parent.

We can now move back to the `TrackExercise.vue` file to pick up on the emitted event and combine the selected date with the modeled routine to eventually store it in the database.

In the `script` block, we'll add the following code to keep track of the routine and the child component:

```
<script setup lang="ts">
// ...abbreviated
import type { Routine } from "@/types/fitness";
import AddRoutine from "./AddRoutine.vue";
const routines: Ref<Routine[]> = ref([]);
const showDialogRoutine: Ref<boolean> = ref(false);

const addRoutineToExercise = (newRoutine: any) => {
  showDialogRoutine.value = false;
  routines.value.push(newRoutine);
};
</script>
```

In our template, below the representation of the date picker, we can add a dialog for creating a routine:

```
<template>
  <v-container>
    <v-row class="align-center justify-space-between mb-6">
      // ...abbreviated
    </v-row>

    {{ routines }}

    <v-row class="mb-6">
      <v-btn
        block
        size="x-large"
```

```
        @click="showDialogRoutine = true"
        v-if="selectedDate"
      >
        Add routine
      </v-btn>
      <v-dialog v-model="showDialogRoutine" fullscreen>
        <v-card>
          <v-card-text>
            <add-routine @add="addRoutineToExercise" />
          </v-card-text>
          <v-card-actions>
            <v-btn color="primary" @click="showDialogRoutine = false"
              >Close</v-btn
            >
          </v-card-actions>
        </v-card>
      </v-dialog>
    </v-row>
  </v-container>
</template>
```

After this code change, once a date has been selected, we show a button to start adding a routine. The routine selection component opens in a dedicated dialog, and on the add event, it calls the addRoutineToExercise function, which adds a newline to the object in this component. You can try it out yourself since we're showing {{ routines }} inline for the moment. It should look similar to *Figure 6.8*:

Figure 6.8 – Our interface for compiling a workout

Our next step will be to format the display of the workout that we want to add to the database and, of course, store the workout in the database itself.

Saving to the database

We've created a feature for our users to model a workout so that we can start saving the hard work. We're going to use our fitness store file for this, so let's add some new methods and export them for usage: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.11-fitness.ts`.

We need to have access to the user ID, so we'll import the `userStore` function (*line 2*), and we'll import some of the data types that we'll use.

We've added our main `saveWorkout` function (*lines 40-70*), which executes two helpers: first, it saves the workout to the workout table using the `insertWorkout` function (*lines 11-25*). On retrieving the `id` property from the workout, we can start saving the sets. For that, we iterate over the routines to compile a list of sets, and we can save them all at once with the `insertSets` method (*lines 27-38*).

With those additions, let's flip back to our `TrackExercise.vue` file to add a **Save** button to call the `saveWorkout` action. We'll import both the fitness and app store:

```
import { useFitnessStore } from "@/store/fitness";
const fitnessStore = useFitnessStore();

import { useAppStore } from "@/store/app";
const appStore = useAppStore();
```

We'll also add functions to validate if a workout can actually be saved – one to reset the form state and one that passes the information to our store action:

```
const canSaveWorkout = computed(() => {
  return routines.value.length > 0;
});

const reset = () => {
  routines.value = [];
  selectedDate.value = undefined;
};

const saveWorkout = () => {
  if (selectedDate.value && routines.value?.length > 0) {
    fitnessStore.saveWorkout({
      date: selectedDate.value,
      routines: routines.value,
    });
    appStore.showDialog({
      title: "Success",
      contents: "Workout saved successfully",
```

```
      });
      reset();
    } else {
      appStore.showDialog({
        title: "Error",
        contents: "Please select a date and add at least one routine",
      });
    }
  };
```

Again, we're using the app dialog to show our systems' messages; nice and reusable. We'll end our form with the conditional **Save** button at the bottom of our template:

```
<template>

// …abbreviated

    <v-row class="mb-6">
      <v-btn
        block
        size="x-large"
        :disabled="!canSaveWorkout"
        @click="saveWorkout"
        v-if="selectedDate"
        >Save workout</v-btn
      >
    </v-row>
  </v-container>
</template>
```

You can try it out. The data should show up in the tables of your Supabase instance. In *Chapter 7,* we'll start to retrieve this data in various ways. As you can see, sometimes during development it makes sense to go back and forth between various components that make up the chain.

I'm trying to demonstrate this process a bit since it's close to the development in practice. It is rare to come up with the ideal solution (or specifications!) in one go.

Our hard work visualized

It is one thing to store the data. For the user, the data only has value if we can present it within a certain context. We've done a small exercise when displaying the compilation of a routine before saving. In this part, we'll see a couple of different examples of displaying the data.

We'll make sure to accommodate an empty state (so feel free to delete any items or try a new login), and after we've added some workouts, we'll find ways to display the data.

Let's start by replacing the home page. In this case, we already have a route and we have the `Home.vue` view, but we'll remove the reference to the `HelloWorld.vue` component and create an empty `History.vue` `<template>` component instead. And then, in `Home.vue`, we'll reference the `History.vue` file instead of `HelloWorld.vue`.

A view-based dashboard

We can start with a quick component to show the latest statistics of the user. When executing the script on the database, it included a view, called `workout_dashboard_view`. This is like an aggregate of read-only queries that we can in turn query as if it were a table on its own.

We'll add methods for getting data to the fitness store, similar to what we did with the exercises, by adding and exposing the `dashboard` variable (*lines 15*, *42*), which in turn gets data from the `getDashboard` method (*lines 17-40*, *42*): `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.12-fitness.ts`.

With that in place, we can create a component to access the dashboard from the store and show the contents. Let's call it `WorkoutStats.vue`, and we'll add a reference to the fitness store and load the dashboard:

```
<script setup lang="ts">
import { computed, onMounted } from "vue";
import { storeToRefs } from "pinia";

import { useFitnessStore } from "@/store/fitness";
const fitnessStore = useFitnessStore();

const { dashboard } = storeToRefs(fitnessStore);

const daysSinceLastWorkout = computed(() => {
  if (!dashboard.value) return 0;
  const lastWorkout = new Date(dashboard.value.last_workout_date);
  const today = new Date();
  const diffTime = Math.abs(today.getTime() - lastWorkout.getTime());
  const diffDays = Math.floor(diffTime / (1000 * 60 * 60 * 24));
  return diffDays;
});

onMounted(() => {
  fitnessStore.getDashboard();
});
</script>
```

We're adding one function to calculate the difference between today and the last exercise date, but it's a concise setup. In the template, we can expect four values for the dashboard, each of which we'll show on its own card:

```
<template>
  <v-row>
    <v-col cols="12" sm="3" m="2" class="d-flex justify-space-
between">
      <v-card class="align-self-stretch flex-grow-1">
        <v-card-title>{{ daysSinceLastWorkout }}</v-card-title>
        <v-card-text>Days since last workout</v-card-text>
      </v-card>
    </v-col>
    <v-col cols="12" sm="3" m="2" class="d-flex justify-space-
between">
      <v-card class="align-self-stretch flex-grow-1">
        <v-card-title>{{ dashboard?.total_workouts || 0 }}</v-card-
title>
        <v-card-text>Total workouts</v-card-text>
      </v-card>
    </v-col>
    <v-col cols="12" sm="3" m="2" class="d-flex justify-space-
between">
      <v-card class="align-self-stretch flex-grow-1">
        <v-card-title>{{ dashboard?.cumulative_weight || 0 }}</v-card-
title>
        <v-card-text>Cumulative weight moved</v-card-text>
      </v-card>
    </v-col>
    <v-col cols="12" sm="3" m="2" class="d-flex justify-space-
between">
      <v-card class="align-self-stretch flex-grow-1">
        <v-card-title>{{
          dashboard?.most_weight_single_workout || 0
        }}</v-card-title>
        <v-card-text>Most weight in a single workout</v-card-text>
      </v-card>
    </v-col>
  </v-row>
</template>
```

We're using an expression to default to 0 values if we have no results from the dashboard. Once it gets filled with values, though, we populate the template with the values. There is some room for optimization since there's some repetition involved in the template. This would be a good exercise to improve on your own!

Now, we have some means of motivating our users to start filling in more exercises! Let's now see if we can display individual workouts too.

History and overview

To retrieve our workouts, we'll add a new method to our fitness store. We have another view lined up for this particular goal! Let's take a look at the updated fitness store file: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.13-fitness.ts`.

As you can see, with the workouts (*line 15*) and the method of retrieving the workouts (*lines 20-46*), we've exposed a new dataset to show in the app.

With these changes, we can add some additional content to our `History.vue` file. We'll begin by importing the data from the store and simply outputting it in the template: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.14-History.vue`.

This addition provides us with an overview of workout entries from the database view (*lines 6-7*, *9*, *28*), which are presented in expansion panels from Vuetify (*lines 36-45*). But as you can see, we need to massage the data a bit, since it's now showing an individual line per combination of workout and exercise. We want to group the data by workout, so we'll create a function to transform our data and add a bit more structure. Take a look at the next iteration of the `History.vue` file: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.15-History.vue`.

This will net us with unique IDs so that we can identify every workout, and we've remodeled the data so that every unique workout has a collection of corresponding exercises as a child (*lines 23-33*). We're doing something similar with exercises: grouping them based on the `exercise_name` property.

In this case, we're opting for this particular approach because we only have to query the database once. There are multiple ways of optimizing database queries, and we're now choosing to take the data from the database as is and model it in our app to suit our needs.

We've also updated the panels to iterate over `workoutIds` and display the restructured (part of the) workout based on the `id` property (*lines 53-63*).

As a final step, we can create a small component to show the exercise. We'll create a `GroupedExerciseView.vue` component and add the following contents: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.16-GroupedExerciseView.vue`.

We can then import and use the Vue component in our panel by passing the exercise set as a property:

```
<script setup lang="ts">
// ...abbreviated
```

```
import GroupedExerciseView from "./GroupedExerciseView.vue";
// ...abbreviated
</script>
```

We can then pass it to the template:

```
<template>
  <v-container>
    <workout-stats class="mb-4" />
    <h1>Past workouts</h1>
    <v-expansion-panels v-model="panel" multiple v-if="workouts">
      <v-expansion-panel v-for="id in workoutIds" :key="id">
        <v-expansion-panel-title
          >{{
            formattedDate(new Date(workoutsGroupedById[id].workout_
created_at))
          }}
        </v-expansion-panel-title>
        <v-expansion-panel-text>
          <grouped-exercise-view
            :exercise="set"
            v-for="(set, index) in
setsByExerciseName(workoutsGroupedById[id].sets)"
            :key="index"
          />
        </v-expansion-panel-text>
      </v-expansion-panel>
    </v-expansion-panels>
  </v-container>
</template>
```

We can use this last component, slightly modified in our overview of adding a workout as well, to display what the user will be saving in a nicely formatted manner.

The data will be slightly different, so we'll create a wrapper component to modify the data before sending it as an exercise to the GroupedExerciseView.vue component. We'll name the new file ExerciseGrouping.vue: https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/. notes/6.17-ExerciseGrouping.vue.

Here, we're also using the component, but we make sure that data in a different format is modified to fit with the component. To make this visible, we'll import the ExerciseGrouping.vue component in our TrackExercise.vue file and display its values:

```
<script setup lang="ts">
// ...abbreviated
```

```
import ExerciseGrouping from "./ExerciseGrouping.vue";
// ...abbreviated
</script>
<template>
  <v-container>

    // ...abbreviated

    <exercise-grouping
        :key="index"
        v-for="(row, index) in routines"
        :exercise-id="row.exercise || 'Unknown'"
        :routines="row?.routines"
        class="mb-6"
      />

    // ...abbreviated
  </v-container>
</template>
```

This helps our users in tracking their future exercises even better. These are relatively simple representations of data. Let's see if we can add more complex visualizations such as graphs.

Graphs

When working with larger sets of data, it is very common to need visualization at some point. We'll implement different graphs by adopting a third-party library (`chart.js`) and have it render our tracked data! At this point, it would be helpful to have multiple workouts tracked over a certain timespan as well. This creates a more useful experience in visualizing the data.

With specific and complex challenges, it is often more efficient to reach for a third-party library than build a solution on your own. In this case, we'll have a look at a library that connects `chart.js` to Vue.js and apply it to our app.

Depending on vue-chartjs

We use a library for displaying charts. In this case, the `vue-chartjs` package helps us integrate the framework-agnostic `chart.js` with our Vue.js app. It is quite common to use a third-party wrapper to better embed an underlying library into your framework.

This usually helps in abstracting concepts and behavior that we know from our framework and translating it to and from the library, which acts unaware of its ecosystem. This way, we don't have to deal with the integration layer and can focus on adding features that matter to our end users.

Let's build some graphs! We're going to install both `vue-chartjs` and the core `chart.js` libraries:

```
npm i vue-chartjs chart.js
```

We'll create a new route called `graph` with a view that loads an empty component and an addition to the menu.

In the router file, we add the following entry:

```
    {
      path: 'graph',
      name: 'Graph',
      component: () => import(/* webpackChunkName: "graph" */ '@/
views/Graph.vue'),
      beforeEnter: loginGuard
    },
```

This indicated to us that we need to create a `Graph.vue` file in the `views` folder, so let's do that:

```
<script lang="ts" setup>
import Graph from "@/components/Graph.vue";
</script>

<template>
  <graph/>
</template>
```

And we'll create a `Graph.vue` component to start building different kinds of graphs on the route. Let's start with a `panel` expansion template: `https://github.com/PacktPublishing/ Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06. fitness/.notes/6.18-Graph.vue`.

As soon as we add the entry to the `menuItems` collection in our `AppMenu.ts` file, we can navigate to our final page for this project:

```
const menuItems = [
  // ...abbreviated
  {
    icon: "mdi-chart-line",
    title: "Graph",
    page: "/graph",
  },
];
```

As you can see from the page, we'll implement three types of charts, with various content of our tracked workouts.

Piece of pie (chart)

As we will see, generating a chart using a library is very straightforward! We have to bear in mind that a `chart` component expects data in a fixed format, as in this example:

```
chartData: {
  labels: [ 'January', 'February', 'March' ],
  datasets: [ { data: [40, 20, 12] } ]
},
```

This inevitably means that we need to do some data remodeling, so let's do that first. We actually need a bit more of a detailed response from our data, so we'll add the following functions to our fitness store:

```
    const workoutsWithSets: Ref<WorkoutFromDatasource | []> = ref([]);

    const getWorkoutsWithSets = async (options: GetWorkoutsOptions = {
order: 'ascending' }): Promise<void> => {
        try {


            // ...abbreviated

            const { data, error, status }: any = await
useSupabaseClient
                .from('workouts')
                .select(
                  id, created_at,
                  sets (
                    workout_id, weight, repetitions,
                    exercises ( name, color )
                  )
                )

    // ...abbreviated

return { exercises, getExercises, saveWorkout, workouts, getWorkouts,
dashboard, getDashboard, workoutsWithSets, getWorkoutsWithSets }
```

The key difference is in the `select` query, where we also request sets with their properties. Of course, we also need to return these new methods from the store. The data from the database, however, is not ready to be used in the graph, since it expects a different format. Again, this happens regularly in real-life scenarios, so let's build a solution.

We create a new store called `graph.ts`, and we'll start with a function that returns data for our pie chart: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.19-graph.ts`.

Here, we're getting the workout data and returning it as an object that's compatible with a pie chart.

Now, we'll create a component called `GraphPie.vue` in our `components` folder where we use the store, with some configuration for the graph type, to render a pie chart based on workout data: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.20-GraphPie.vue`.

With our `PieChart` component completed, we'll import it into the `Graph.vue` component, and then we can replace the line in the collapsible:

```
<script setup lang="ts">
import { ref } from "vue";
import type { Ref } from "vue";

import GraphPie from '@/components/GraphPie.vue'

const panel: Ref<Number[]> = ref([0]);
</script>
<template>
  <v-container>
    <h1>Graphs</h1>
    <v-expansion-panels v-model="panel" accordion>
      <v-expansion-panel>
        <v-expansion-panel-title
          >Workout distribution all time (sets & reps)</v-expansion-
panel-title
        >
        <v-expansion-panel-text>
          <graph-pie />
        </v-expansion-panel-text>
      </v-expansion-panel>
      // ...abbreviated
    </v-expansion-panels>
  </v-container>
</template>
```

Depending on the availability and content of the data, you will end up with a user interface that resembles this:

Graphs
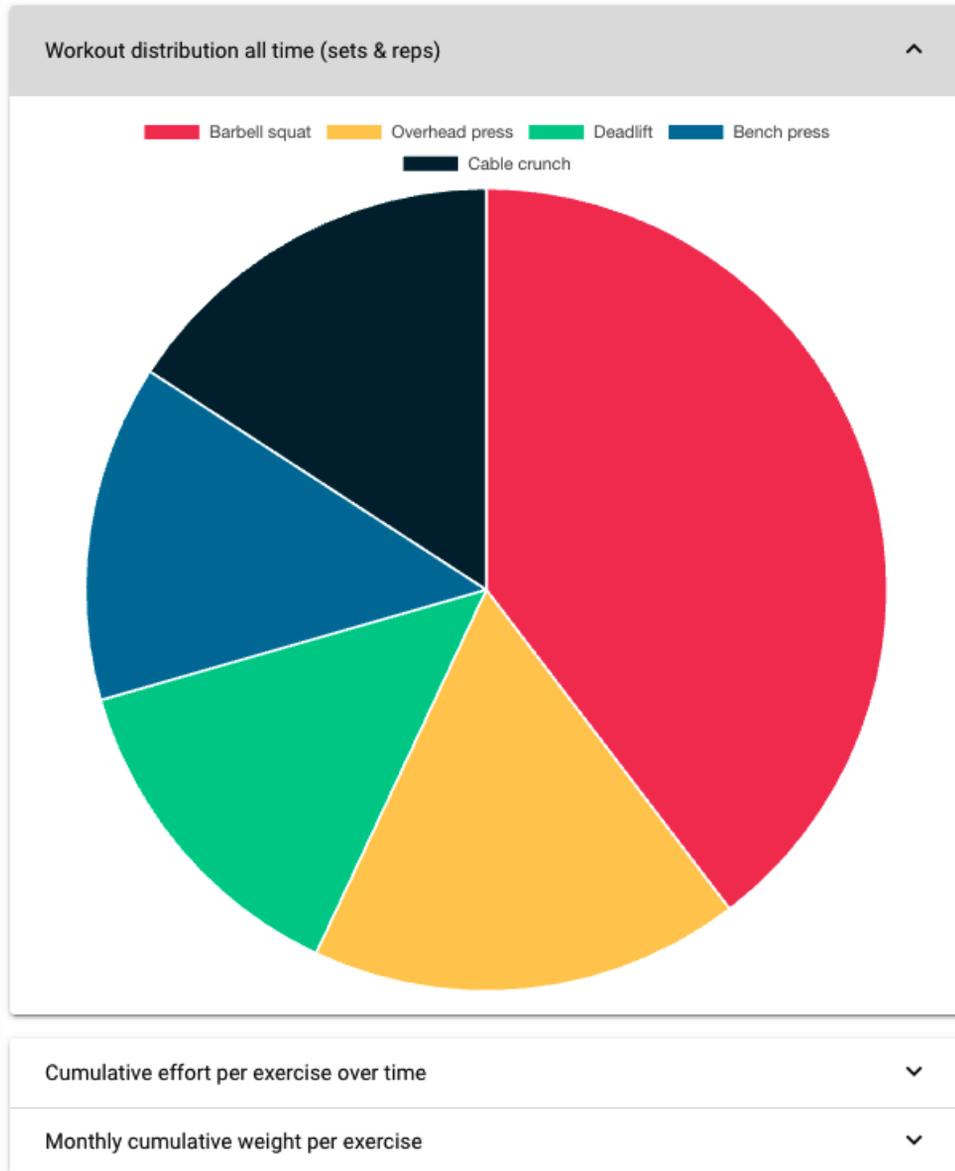
Figure 6.9 – Example workout data represented in a pie chart

Using `vue-chartjs`, it's been relatively easy for us to visualize datasets. We have worked on making sure that our data is formatted in the way the library expects. With that in mind, we can proceed in expanding to different types of visualizations.

More charts!

Looking at our collapsible examples, we're going to build two more charts, just to get a feel of multiple implementations. We'll update our graph store with more features. We'll create an internal method that we can reuse in both our graphs and then create a public method on the store to retrieve data: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.21-graph.ts`.

We're creating a helper function (`createGraphData`, *lines 11-46*) to gather and format the data based on a monthly average. For our two distinct types of graphs, we'll create one function called `getGraphMonthlyAverage` (*lines 48-58*). Since the library expects data in a predefined format, we can use the same data presented in different ways.

To showcase this, we'll create a bar graph as well as a line graph from the same data. First, the bar graph; the code is very similar to the pie graph, with our abstractions in place. We'll call this component `GraphBar.vue`: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.22-GraphBar.vue`.

This is how the bar graph would render to the browser:
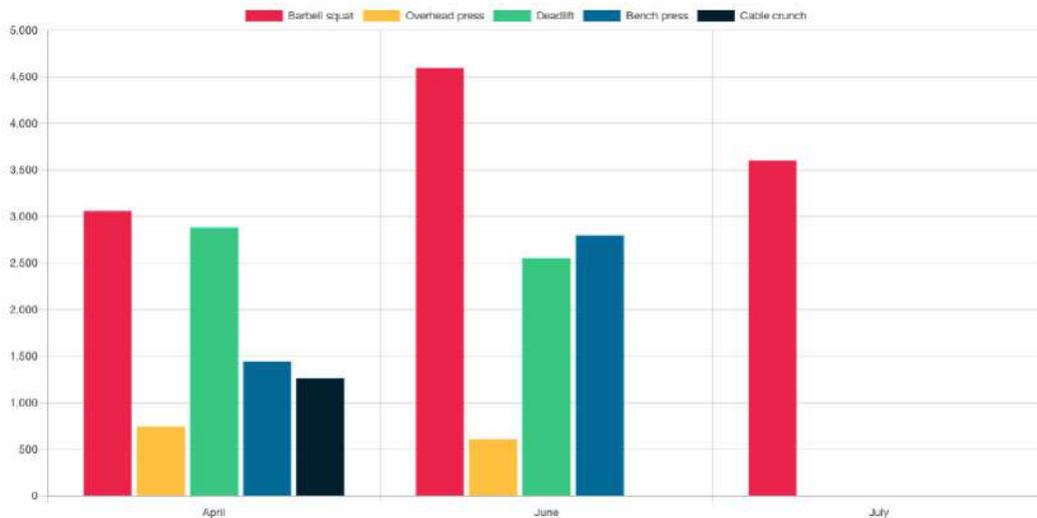
Figure 6.10 – Example workout data represented in a bar chart

Not so different from our previous graph component! We'll do the same for a line graph, called `GraphLine.vue`, as you might have guessed: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/06.fitness/.notes/6.23-GraphLine.vue`.

This is how the line graph would render to the browser:
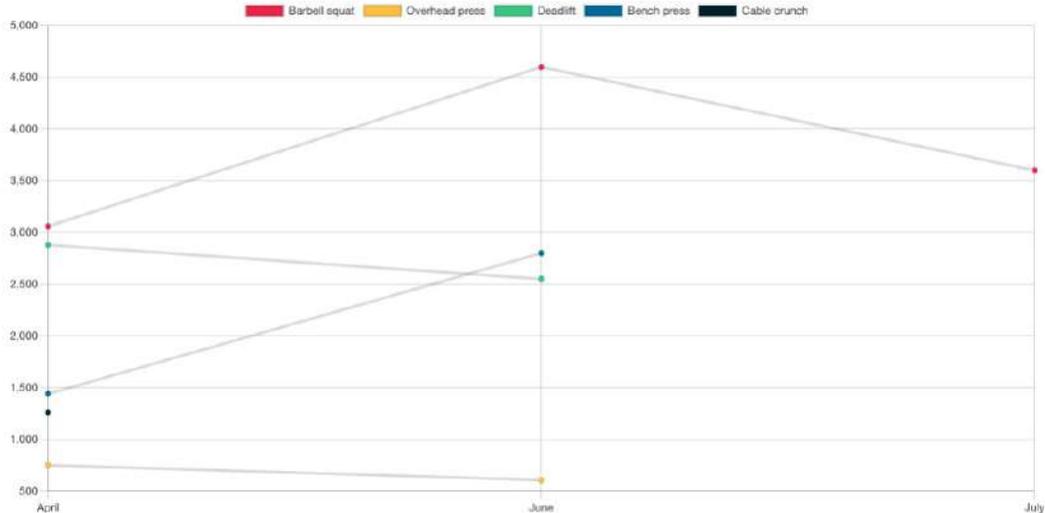
Figure 6.11 – Example workout data represented in a line chart

We could even abstract repetitive code in a Vue component that is capable of rendering the data as different types of graphs. That would be a nice exercise for you to pursue as an extra assignment. Try swapping out the getGraphPie data with the getGraphMonthlyAverage data, and you'll see that the graph just accepts these new values since they follow the right shape.

To complete this chapter, we'll add the components to our user interface, in the Graph.vue file:

```ts
<script setup lang="ts">
import { ref } from "vue";
import type { Ref } from "vue";

import GraphPie from '@/components/GraphPie.vue'
import GraphLine from '@/components/GraphLine.vue'
import GraphBar from '@/components/GraphBar.vue'

const panel: Ref<Number[]> = ref([0]);
</script>
<template>
  <v-container>
    <h1>Graphs</h1>
    <v-expansion-panels v-model="panel" accordion>
      <!-- abbreviated -->
      <v-expansion-panel>
        <v-expansion-panel-title
```

```
            >Monthly cumulative weight per exercise</v-expansion-panel-
title
          >
          <v-expansion-panel-text>
            <graph-bar />
          </v-expansion-panel-text>
        </v-expansion-panel>
        <v-expansion-panel>
          <v-expansion-panel-title
            >Cumulative effort per exercise over time</v-expansion-
panel-title
          >
          <v-expansion-panel-text>
            <graph-line />
          </v-expansion-panel-text>
        </v-expansion-panel>
      </v-expansion-panels>
    </v-container>
  </template>
```

With that, we've built an app where users can log in and track and retrieve their individual results.

Summary

We've seen how we can use the concepts of stores, composables, and nested components to build a relatively complex user flow. For security, we depend on the authorization model by Supabase, which helps us achieve our results in an efficient way.

Taking a look at the Supabase structure and data is helpful in understanding the way certain endpoints store and offer their data. Up until this point, we've just been consuming data. Under the hood, every Supabase instance is a dedicated PostgreSQL database. If you want to know more about PostgreSQL, I highly recommend checking out *Developing Modern Database Applications with PostgreSQL* by Dr. Quan Ha Le and Marcelo Diaz, at `https://www.packtpub.com/product/developing-modern-database-applications-with-postgresql/9781838648145`.

With more complex tasks, it makes sense to take a step-by-step approach to building the feature, which is something I've also demonstrated. This sometimes means revisiting certain files to make small additions while we learn more about features along the way. Even when requirements are very clear upfront, it can be really helpful to break features up into smaller parts and follow their path in the app file structure while building.

We've applied a certain level of abstraction but also took a pragmatic approach in parts. It can be difficult to find this balance, but I do tend to favor some repetition to prevent over-engineering: this often results in more readable code that is easier to make small changes to.

I do favor separating a component into parts with a specific role. The store composition is a good example of this, just as with our Supabase client file. With the **separation of concerns** (**SoC**), we can limit the complexity of each individual part to make it more scalable and manageable in the future.

We've built apps specifically for the web. In the next chapter, we'll see how we can create projects to target different environments.

7

Building a Multiplatform Expense Tracker Using Quasar

In this chapter, we'll iterate the topics and techniques we covered in *Chapter 6*. We'll build a similar app using Vue and rely on Supabase to store our data. In this chapter, however, we'll focus on building an app that can be deployed on multiple platforms other than the web.

We'll use Quasar (`https://quasar.dev/`) as our framework of choice since it allows us to choose multiple different types of platforms. For the sake of simplicity, we'll focus on creating a desktop application based on Electron (`https://www.electronjs.org/`). Both Quasar and Electron are well-maintained open source projects with excellent documentation and active communities.

In this chapter, we'll cover the following topics:

- Solidifying what we learned previously
- Familiarizing yourself with different frameworks
- Understanding the value of platform-agnostic development
- Building a native app using web technologies
- Learning the key differences between web and native

Technical requirements

In this chapter, we'll be reusing most of the requirements from *Chapter 6* since we'll be building an application with similar capabilities. This will help you identify how the framework impacts the architecture of an app.

We'll be depending heavily on Quasar (`https://quasar.dev/`) as our foundational framework. Since the framework also offers UI patterns (`https://quasar.dev/components`), we don't need Vuetify in this project. We will use Pinia (`https://pinia.vuejs.org/`) to handle our application state. To store the data, we'll create a new project in Supabase (`https://supabase.com/`), an open source database provider with built-in authentication. For the database, I've prepared a script to create the databases and another one to add example data. We'll cover those steps in the *Setting up the database* section.

The final product is located in this book's GitHub repository at `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/tree/main/07.expenses`.

Setting up the database

We'll start by fulfilling our database needs. We already have a free account (see *Chapter 6*). We'll create a new project called `expense-tracker`, set a strong database password, and assign a geographically close region.

You'll be redirected to a view that provides you with **Project URL** and **API Key** details, as shown in *Figure 7.1*:

Figure 7.1 – API settings for our expense-tracker project

Since our application isn't ready, we need to note the URL and API key down in a safe place or simply revisit this page once we get to the application.

For this project, I've prepared a script to create the required tables and settings for our database called `example-structure.sql`. Open the SQL Editor in Supabase, then paste and run the contents of the script:

```
1    --
2    -- Name: categories; Type: TABLE; Schema: public; Owner: postgres
3    --
4
5    CREATE TABLE public.categories (
6        id uuid DEFAULT gen_random_uuid() NOT NULL PRIMARY KEY,
7        name text,
8        color text,
9        profile_id uuid
10   );
11
12   ALTER TABLE public.categories OWNER TO postgres;
13
14   --
15   -- Name: profiles; Type: TABLE; Schema: public; Owner: postgres
16   --
17
18   CREATE TABLE public.profiles (
19       id uuid DEFAULT gen_random_uuid() NOT NULL PRIMARY KEY,
20       name text,
21       email_address text,
22       updated_at timestamp with time zone DEFAULT now()
23   );
```

```
Results  ∨                                              ♥    ∨    RUN ⌘↵

Success. No rows returned
```

Figure 7.2 – Success message after running the example-structure.sql script

Our expense tracker will be able to organize expenses into different categories, so I've also created a script to insert a set of example categories in the `example-categories.sql` file. You can paste the contents and run this file in the SQL Editor too:

```
1   INSERT INTO
2       categories (name, color)
3   VALUES
4       ('Food & Dining','#AEC6CF'),
5       ('Groceries', '#F49AC2'),
6       ('Transportation', '#B39EB5'),
7       ('Housing', '#FEC8D8'),
8       ('Entertainment', '#C8B9D4'),
9       ('Shopping', '#FDCB92'),
10      ('Health & Wellness', '#A5D0B0'),
11      ('Travel', '#FFD700'),
12      ('Education', '#D5E7D8'),
13      ('Personal care', '#FFB6C1'),
14      ('Bills & Utilities', '#D0E6F2'),
15      ('Debt Payments', '#F0E68C'),
16      ('Miscellaneous', '#F0D9B5')
17
```

Results ⌄   ♥   ⌄   RUN ⌘ ↵

Success. No rows returned

Figure 7.3 – Success message after running the example-categories.sql script

Now that our database has been set up, we can start creating a new project.

Using Quasar to build a project

We're going to follow the default setup and installation guide at `https://quasar.dev/start/quick-start`. In the CLI, we'll run `npm init quasar` and select the configuration, as shown in *Figure 7.4*:

```
Need to install the following packages:
  create-quasar@1.3.0
Ok to proceed? (y) y


 .d88888b.
d88P" "Y88b
888     888
888     888 888  888  8888b.  .d8888b  8888b.  888d888
888     888 888  888     "88b 88K         "88b 888P"
888 Y8b 888 888  888 .d888888 "Y8888b. .d888888 888
Y88b.Y8b88P Y88b 888 888  888     X88 888  888 888
 "Y888888"   "Y88888 "Y888888 88888P' "Y888888 888
       Y8b

✔ What would you like to build? › App with Quasar CLI, let's go!
✔ Project folder: … ex.expensetracker
✔ Pick Quasar version: › Quasar v2 (Vue 3 | latest and greatest)
✔ Pick script type: › Typescript
✔ Pick Quasar App CLI variant: › Quasar App CLI with Vite
✔ Package name: … packt-expense-tracker
✔ Project product name: (must start with letter if building mobile apps) … Packt Expense Tracker
✔ Project description: … An Expense Tracker
✔ Author: … Joran Quinten <joran@joranquinten.nl>
✔ Pick a Vue component style: › Composition API with <script setup>
✔ Pick your CSS preprocessor: › Sass with SCSS syntax
✔ Check the features needed for your project: › ESLint, State Management (Pinia)
✔ Pick an ESLint preset: › Prettier
```

Figure 7.4 – Creating a new project using the Quasar CLI

This will install the project and its dependencies. Once the initialization is completed, we can navigate to the project folder and install the Supabase JavaScript client via the CLI:

```
npm install @supabase/supabase-js
```

To conclude the initialization, we'll create a .env file with the Supabase API keys:

```
VITE_SUPABASE_URL=YOUR_SUPABASE_URL
VITE_SUPABASE_ANON_KEY=YOUR_SUPABASE_ANON_KEY
```

We can verify our installation by running the following command in the command line:

```
npx quasar dev
```

The example project will be installed, as shown in *Figure 7.5*:
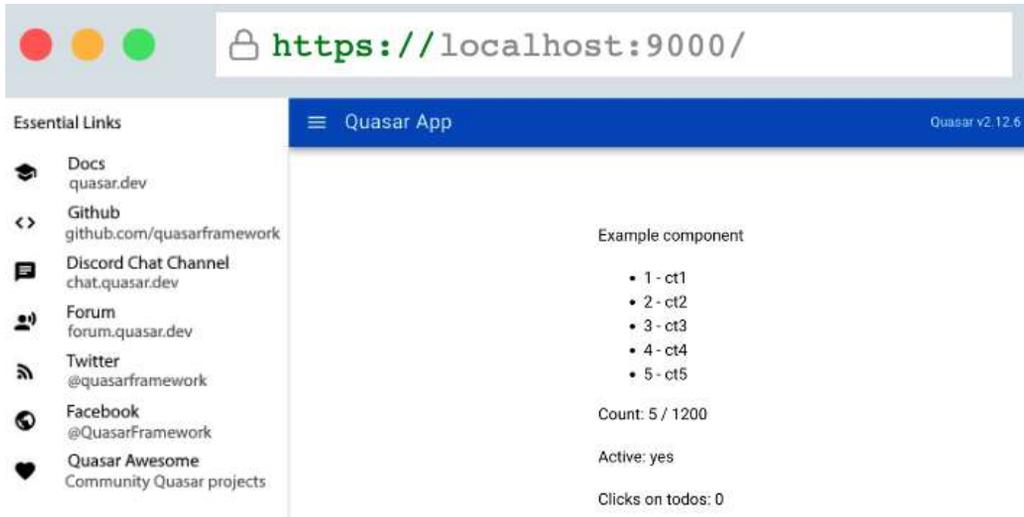
Figure 7.5 – The default project using Quasar

Since our goal is to work on a desktop app, we can easily run a development command for that environment as well:

```
npx quasar dev -m electron
```

The first time this command is run, it will need to install some dependencies to run the environment. This will result in an output similar to the following:
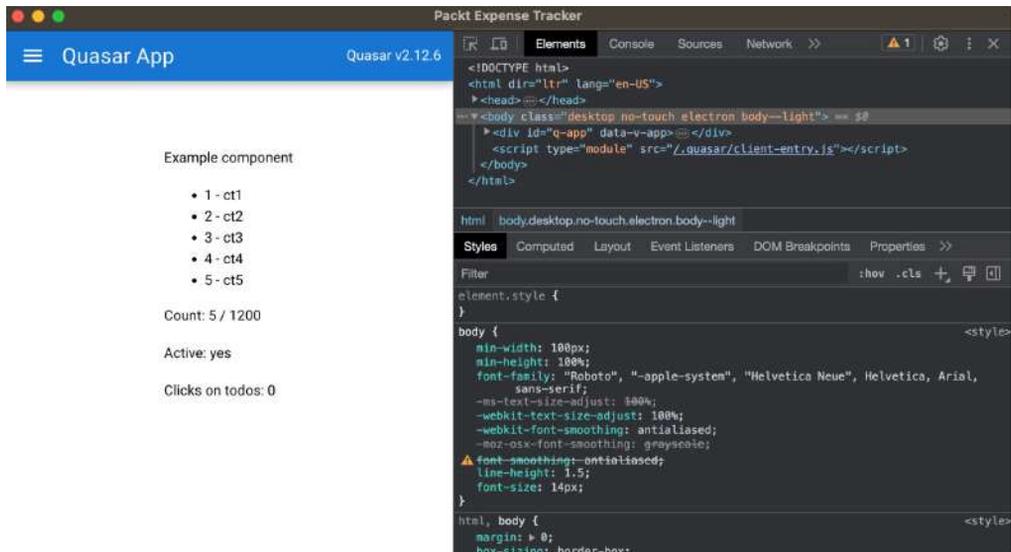
Figure 7.6 – Running Quasar in Electron development mode

See how easy it is to target a different environment? Of course, Electron is very close to our browser environment, so it will behave very similarly. We are going to develop and debug our application in the browser. In virtually all cases, we can rely on this framework to deliver and compile our code to specific platforms.

Targeting Android or iOS is a bit more complicated. It will make use of Capacitor to build a native-like shell that acts as a proxy between the operating system and the application. If you're interested in mobile deployments, I highly recommend referring to the Quasar guide: `https://quasar.dev/quasar-cli-vite/developing-capacitor-apps/introduction`.

With our app running on the web and Electron, we have our starting point to build the expense tracker!

> **Note**
>
> We're using Quasar as a framework due to its bundling and building capabilities, but Quasar also offers a packed library of ready-to-use Vue components (`https://quasar.dev/components`). In our example code, you will be able to recognize them by the *q-* prefixes in the component name. We won't dive into how the component works, so I'd like to refer you to the official (and great) docs, which you can find at `https://quasar.dev/docs` and `https://quasar.dev/components`.

Let's have a look at how we can connect our Supabase instance with a frontend application, shall we?

Authenticating with Supabase and Quasar

Having an application rather than a website means that external hyperlinks, such as the OTP method of signing in via Supabase, will not work out of the box. Handling these issues is a bit too advanced for this chapter, so we'll opt for signing in via email and password. To have Supabase and our Quasar application integrate nicely, I'm loosely basing our implementation on the following online resource: `https://dev.to/tvogel/getting-started-with-supabase-and-quasar-v2-kdo`.

The `src/boot` folder is meant for scripts that need to be executed before we initialize the Vue.js application (`https://quasar.dev/quasar-cli-vite/boot-files/`). In our case, we need to utilize the boot files because we want to execute logic before we change a route, to see whether a user has access. This means we need to handle our authentication and Supabase client in scripts that are executed before the main scripts of our app.

First, we'll create the `src/boot/supabase.ts` file with the contents of the following file: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/07.expenses/.notes/7.1-supabase.ts`.

We'll use the `router-auth.ts` file as well and place it in the same folder: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/07.expenses/.notes/7.2-router-auth.ts`.

We can add these scripts to our `quasar.config.js` file by simply referencing the file on the `boot` property, which will then look like this:

```
// ...abbreviated
boot: ['supabase', 'router-auth'],
// …abbreviated
```

The preceding configuration tells our app to run the given scripts before we initialize (or boot) the application.

With our basic boot scripts in place, we can look at our routes.

Routes and app structure

Now, we'll add some routes so that we can build out our app and apply the `router-auth` scripts to the correct routes. Let's remove all of the files from the `src/pages` folder, except the `ErrorNotFound.vue` page. We'll add the following pages with the same structure:

- `AccountPage.vue`
- `Expenses Page.vue`
- `CategoriesPage.vue`

Replace the contents of `<h1 class="text-h1">Home</h1>` with a relevant title for every page using the following template:

```
<script setup lang="ts"></script>

<template>
  <q-page class="column items-center justify-center">
    <h1 class="text-h1">Home</h1>
  </q-page>
</template>
```

We'll build the functionalities of each in the *Managing categories* and *Showing expenses and an overview* sections. But first, we need to integrate the authentication into our routes.

First, let's look at the `src/router/index.ts` file: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/07.expenses/.notes/7.3-router-index.ts`.

We'll make two changes: we'll import the `init` function of our Supabase boot script (*line 11*) and we'll execute the function in the route function (*line 23*). We can now define the routes that lead to the pages we've already created.

Let's look at the `routes.ts` file: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/07.expenses/.notes/7.4-routes.ts`.

The file contains familiar code but with a slight difference – for every route that requires the user to be logged in, we've added a `meta` field:

```
meta: {
  requiresAuth: true,
},
```

Also, the route for the `/auth` route (*lines 29–31*) is different: it doesn't rely on the layout and directly imports the component. This is because the route will be the entry point of the app for non-authenticated users.

Now, if we look in our `src/boot/router-auth.ts` file, we will see that before each route change, we check if that meta field exists and then validate that the user `session` exists. If not, we redirect the user to the `fullPath` property, which translates to the home page.

It's time to put these features into practice by building signup and login features for the app.

Signing up and logging in

Let's work on offering users a way of both registering and signing in to our app. In our components folder, we'll create a form for logging in called `FormLogin.vue`: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/07.expenses/.notes/7.5-FormLogin.vue`.

We'll also create a (very similar) file for signing up called `FormSignUp.vue`: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/07.expenses/.notes/7.6-FormSignUp.vue`.

The key difference between the two forms is the method we're calling on submit, which is either the `supabase.auth.signInWithPassword` or `supabase.auth.signUp` method. We're not interested in abstracting anything here. It's not always the best approach to try and over-optimize everything that might be repetitive. In this case, we prefer readability and simplicity over similar code between the two files.

Let's incorporate the two forms on a newly created `AuthPage.vue` page:

```
<script setup lang="ts">
import { ref } from 'vue';
import type { Ref } from 'vue';
```

```
import FormLogin from 'src/components/FormLogin.vue';
import FormSignUp from 'src/components/FormSignUp.vue';

const tab: Ref<'login' | 'sign-up'> = ref('login');
</script>

<template>
  <div class="column items-center justify-center">
    <h1 class="text-h2">Home</h1>
    <q-card class="column">
      <q-tabs
        v-model="tab"
        active-color="primary"
        indicator-color="primary"
        align="justify"
        narrow-indicator
      >
        <q-tab name="login" label="Log in" />
        <q-tab name="sign-up" label="Sign up" />
      </q-tabs>

      <q-separator />

      <q-tab-panels v-model="tab" animated>
        <q-tab-panel name="login">
          <FormLogin />
        </q-tab-panel>
        <q-tab-panel name="sign-up">
          <FormSignUp />
        </q-tab-panel>
      </q-tab-panels>
    </q-card>
  </div>
</template>
```

In the preceding code, we used the Quasar `tab` component to offer the two forms on the page. We should now be able to sign up for a new account. However, there's a small catch.

The default settings of Supabase require us to confirm email addresses on signup. For simplicity's sake, we need to disable this feature. Log into your Supabase dashboard, navigate to **Authentication | Providers**, and then expand the **Email** panel. There, we need to disable the **Confirm Email** option, as shown in *Figure 7.7*:
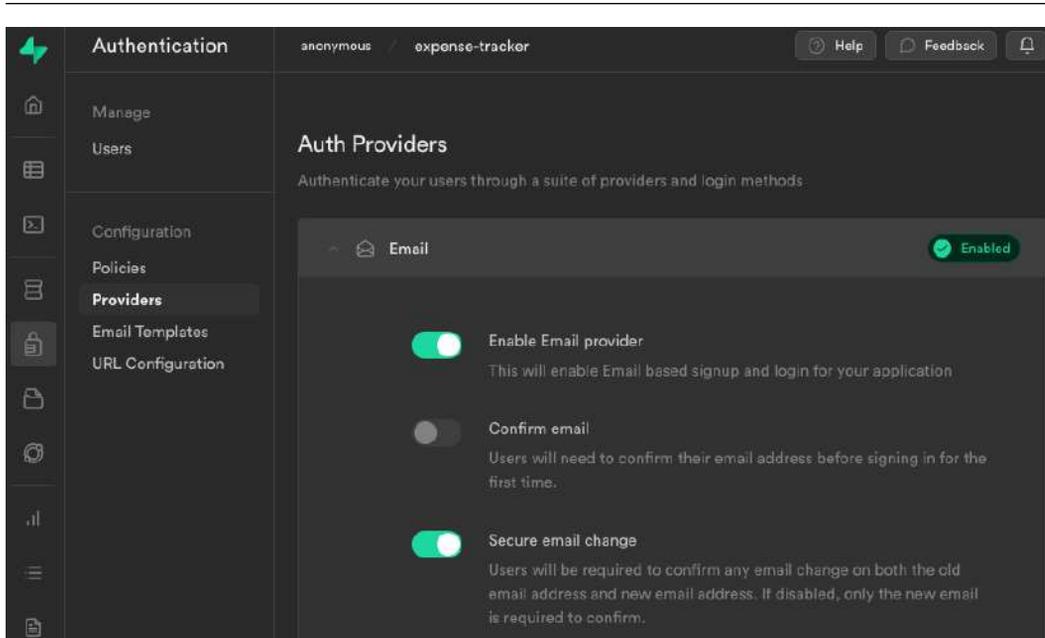
Figure 7.7 – Disabling the Confirm email option in Supabase

Once we've saved this setting, we can register as a new user. You can keep the Supabase dashboard open and navigate to **Profiles** to verify your newly created account!
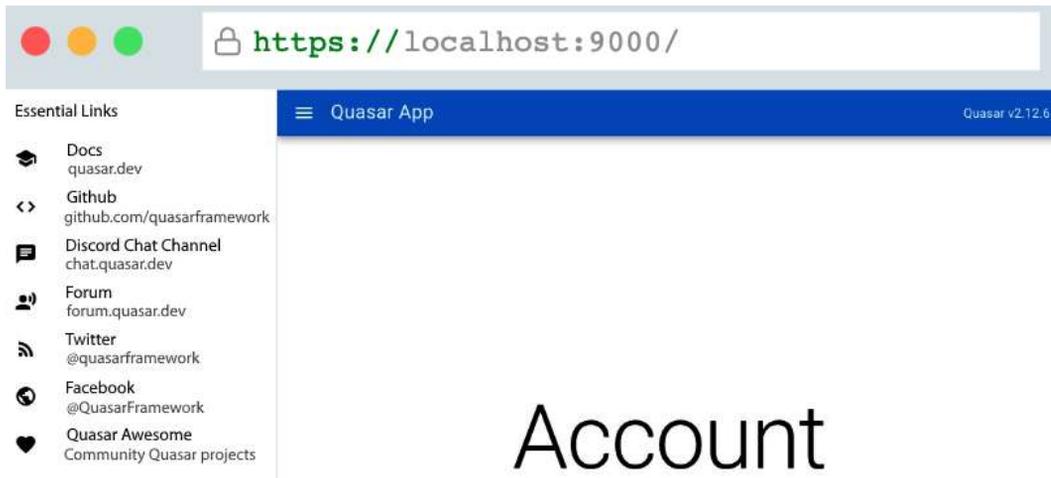
Figure 7.8 – Initial logged-in state in the web view

In the web app, we'll be redirected to the (mostly empty) **Account** page, as shown in *Figure 7.8*. Now we're getting somewhere.

First, we'll slightly modify the existing `src/components/EssentialLink.vue` component: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/07.expenses/.notes/7.7-EssentialLink.vue`.

We're replacing the link with a router link integration. The `<q-item />` component provides support so that we can use it as a router link entity out of the box! Let's open `layouts/MainLayout.vue` so that we can change the default layout for our use case. You can modify the header any way you see fit, but let's focus on **Menu**. Next, we want to update the `essentialLinks` constant to reflect the pages we've created and want to show in **Menu**. We'll replace it with the following contents:

```
const essentialLinks: EssentialLinkProps[] = [
  {
    title: 'Account',
    caption: 'Manage my account settings',
    icon: 'face',
    link: '/',
  },
  {
    title: 'Expenses',
    caption: 'Track my expenses',
    icon: 'toll',
    link: '/expenses',
  },
  {
    title: 'Categories',
    caption: 'Manage my expense categories',
    icon: 'settings',
    link: '/categories',
  },
];
```

You can, of course, see these changes in the app as soon as you save them!

We haven't allowed our users to log out of their session in the app yet. In the next section, we'll make sure our users can sign out as well.

Logging out

Let's add a button to sign out as well. We'll create a standalone component for this in the `components` folder called `ButtonSignOut.vue`:

```
<script setup lang="ts">
import { supabase } from 'src/boot/supabase';
import { useRouter } from 'vue-router';
const router = useRouter();
```

```
const signOut = async (): Promise<void> => {
  try {
    const { error } = await supabase.auth.signOut();
    if (error) {
      throw new Error('Logout failed');
    }
  } catch (error) {
    console.error(error.message);
  } finally {
    router.go(0);
  }
};
</script>


<template>
  <q-btn @click="signOut()">Sign out</q-btn>
</template>
```

We're calling the Supabase method of signing out again and instructing the router to go to the first entry of its history table.

Now, we can flip back to `MainLayout.vue`: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/07.expenses/.notes/7.8-MainLayout.vue`.

If we look at the contents of the `<q-drawer>` component (*lines 20–39*), we will see the addition of a logout button (*lines 29–32, 48*). Feel free to add the sign-out button to, say, the **Account** page as well.

> **Note**
>
> Since we're creating new files as well as modifying existing starter files, we have mixed coding styles in our app. Sometimes, the code block precedes the template, and sometimes, the other way around. For our current implementation, this is not problematic, although when collaborating or working on large projects, it is highly recommended to use a consistent coding style and strictly keep to it.

So far, we've created the basic features for a user-centered app: signing up, logging in, and logging out. We've also applied the authorization to different parts of our app using routes. Now, we'll focus on adding specific features to the app, such as expense tracking.

Expense tracking features

A big part of tracking our future expenses is being able to organize them into different categories. We'll start by adding a Pinia store to work with the category data in our app. This is very similar to the exercises we completed in *Chapter 6*. Let's create a `src/store/categories.ts` file: `https:// github.com/PacktPublishing/Building-Real-world-Web-Applications- with-Vue.js-3/blob/main/07.expenses/.notes/7.9-categories.ts`.

As you can see, we're using the `session` and `supabase` scripts from our `boot` scripts to interact with the logged-in user session and database connection.

To show the categories, we'll create a component in our components folder called `CategoryList. vue`: `https://github.com/PacktPublishing/Building-Real-world-Web- Applications-with-Vue.js-3/blob/main/07.expenses/.notes/7.10- CategoryList.vue`.

Apart from showing the categories, the `removeOwnCategory` function (*lines 29–39*) allows users to remove categories that they've added themselves. All of the actions are dispatched via the store we've created.

In the next section, we'll create features to make the app a bit more personal, by allowing our users to manage categories for themselves.

Managing categories

Having a function to remove categories means we have to build a component to add custom categories. So, we'll create a new component called `CategoryAdd.vue`: `https://github. com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue. js-3/blob/main/07.expenses/.notes/7.11-CategoryAdd.vue`.

This exposes a sticky **floating action button** (**FAB**) that's tied to toggling a small form in a dialog that accepts name and color entries and upserts the entry in the database with the user's profile ID. This is another good example of a component with a clear purpose that's contained in a single file. This is how we build complex applications!

Having the feature of adding personal categories means we can show these individual categories together with the default categories.

To complete the categories functionality, we will import both components we've just created and list them on `CategoriesPage.vue`:

```ts
<script setup lang="ts">
import CategoryList from 'src/components/CategoryList.vue';
import CategoryAdd from 'src/components/CategoryAdd.vue';
</script>
```

```
<template>
  <q-page class="column items-center justify-center">
    <h1 class="text-h1">Categories</h1>
    <category-list />
    <category-add />
  </q-page>
</template>
```

With our categories in place, we can finalize the app by exposing the feature to add expenses!

Adding expenses

To add expenses, we'll start by creating a Pinia store. The store will contain an overview of expenses for our users and have some methods for retrieving and adding expenses. Let's create a `src/store/expenses.ts` file: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/07.expenses/.notes/7.12-expenses.ts`.

The most important feature of our app is its ability to track expenses! Having our store in place, we can create a component specifically for that called `src/components/ExpenseAdd.vue`: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/07.expenses/.notes/7.13-ExpenseAdd.vue`.

As you can see, this component has a lot of similarities with adding a category, although this form depends on categories to be present on the Vuetify Select component. Both are FAB components that toggle a dialog to facilitate this feature.

We can add our new `ExpenseAdd` component to `ExpensesPage.vue` like this:

```
<script setup lang="ts">
import ExpenseAdd from 'src/components/ExpenseAdd.vue';
</script>

<template>
  <q-page class="column items-center justify-center">
    <h1 class="text-h1">Expenses</h1>
    <expense-add />
  </q-page>
</template>
```

Now, you should be able to add expenses. You can verify this by looking at the tables in the Supabase dashboard. For our users, we'll have to start creating views in the app, so that will be our next step!

Showing expenses and an overview

In the `src/components` folder, we'll create a `CategoryOverview.vue` component that will aggregate the categories and expenses from the database into a combined view: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/07.expenses/.notes/7.14-CategoryOverview.vue`.

Similar to the category overview, we have another overview of categories, but this one is displayed in a slightly different layout. This will serve as our new starting point. Let's finalize `ExpensesPage.vue` by adding the `CategoryOverview` component as well:

```ts
<script setup lang="ts">
import ExpenseAdd from 'src/components/ExpenseAdd.vue';
import CategoryOverview from 'src/components/CategoryOverview.vue';
</script>

<template>
  <q-page class="column items-center justify-center">
    <h1 class="text-h1">Expenses</h1>
    <expense-add />
    <category-overview />
  </q-page>
</template>
```

If you run your app and navigate to the **Expenses** page, you should see something similar to the following:
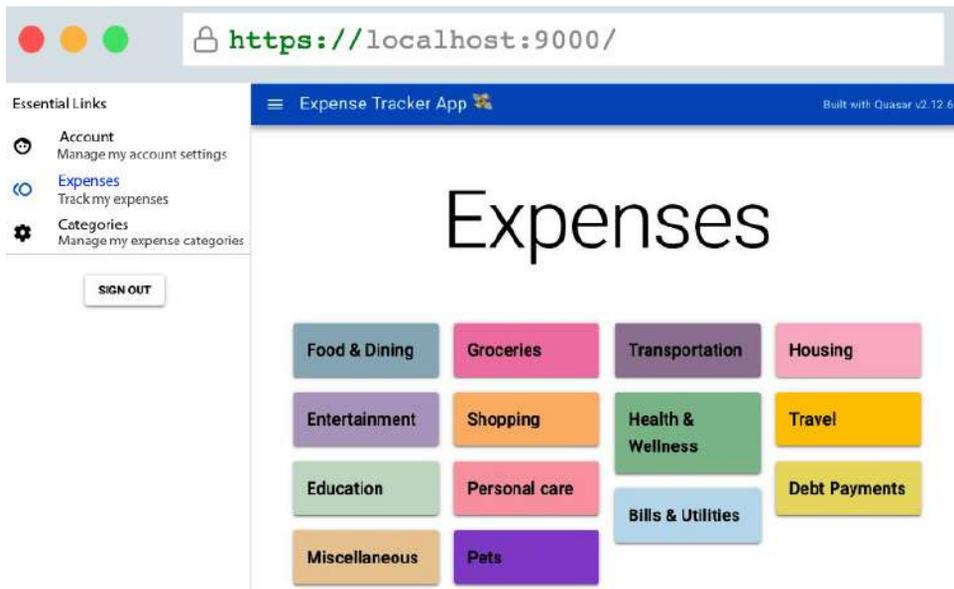
Figure 7.9 – First overview of the Expenses page

To add value to this page, we need to start showing our expenses here. We can create a specific component for this in the components folder, which we'll call `ExpensesCategoryTotal.vue`:

```ts
<script setup lang="ts">
import { computed } from 'vue';
import { useExpensesStore } from 'src/stores/expenses';
import { storeToRefs } from 'pinia';
import { Expense } from 'src/types/expenses';

const expensesStore = useExpensesStore();

const { expenses } = storeToRefs(expensesStore);

interface Props {
  categoryId: string;
}

const props = defineProps<Props>();

const totalPerCategory = computed(() => {
  return expenses.value.reduce((total: number, expense: Expense):
number => {
    if (expense.category_id === props.categoryId) {
      return total + expense.amount;
    }
    return total;
  }, 0);
});
</script>

<template>
  <q-card-section  align="right"> Total: {{ totalPerCategory }}
</q-card-section>
</template>
```

This is a component that we can provide with a category ID, and it will grab and tally all the values of `amount` as the total amount for that category. Once we've done this, we can add the `ExpensesCategoryTotal` component to our overview of categories with ease:

```ts
<script setup lang="ts">
// …abbreviated
import ExpensesCategoryTotal from './ExpensesCategoryTotal.vue';
// …abbreviated
```

```
      </script>

  <div class="masonry">
    <!-- abbreviated -->
        <q-card
          clickable
          class="q-ma-sm"
          :style="{ backgroundColor: category.color }"
        >
          <q-card-section>
            <div class="text-h6">{{ category.name }}</div>
          </q-card-section>
          <expenses-category-total :category-id="category.id" />
        </q-card>
    <!-- abbreviated -->
</template>
```

Depending on what expenses you've inserted, you will now see an overview of the total expenses per category in the tiled overview we created. We're almost there!

It would be nice if users could get some more insights into their expenses, right? So, let's add a more detailed overview of the expenses in a certain category. Luckily, in our store, the functionality to query the database on expenses per category is already exposed. It is an extra call to Supabase, so we want to be a bit diligent with additional requests, which means we will only load the data when requested.

Let's start by adding the script to handle the request and store the data in the `CategoryOverview.vue` file:

```
<script setup lang="ts">
// ...abbreviated
import type { ExpenseWithCategory } from 'src/types/expenses';

// ...abbreviated

const loading: Ref<boolean> = ref(true);
const showDialog: Ref<boolean> = ref(false);
const selectedCategoryName: Ref<string | undefined> = ref(undefined);
const expensesByCategoryId: Ref<ExpenseWithCategory[] | undefined> =
ref([]);

onMounted(async () => {
  // ...abbreviated
});
```

```
const getExpensesByCategoryId = async (
  categoryName: string,
  categoryId: string
) => {
  selectedCategoryName.value = categoryName;
  const expenses = await expensesStore.
getExpensesByCategory(categoryId);
  if (expenses && expenses.length > 0) {
    expensesByCategoryId.value = expenses;
    showDialog.value = true;
  }
};


const getCategories = async () => {
// ...abbreviated
};
</script>
```

Now, in the template, we'll call `getExpensesByCategoryId` when clicking on a tile. So, let's modify `<q-card>` and add an `onClick` event:

```
<q-card
  class="q-ma-sm"
  :style="{ backgroundColor: category.color }"
  @click="getExpensesByCategoryId(category.name, category.id)"
>
  <q-card-section>
    <div class="text-h6">{{ category.name }}</div>
  </q-card-section>
  <expenses-category-total :category-id="category.id" />
</q-card>
```

Next, as a sibling of the masonry style element, we'll add dialog markup to show the expenses of the selected category:

```
<template>
  <div class="masonry">
    <!-- ...abbreviated -->
  </div>

  <q-dialog v-model=»showDialog»>
```

```
    <q-card v-if="expensesByCategoryId && expensesByCategoryId.length
> 0">
      <q-card-section>
        <div class="text-h6">
          {{ `${selectedCategoryName} expenses overview` }}
        </div>
      </q-card-section>

      <q-separator />

      <q-card-section style="max-height: 50vh" class="scroll">
        {{ expensesByCategoryId }}
      </q-card-section>
    </q-card>
  </q-dialog>
</template>
```

We will need to feed the given collection of expenses into a new component. This will be our last component before we finalize our app!

In the `components` folders, we'll create an `ExpensesList.vue` component and have it receive `expenses` as a prop:

```
<script setup lang="ts">
interface Props {
  expenses: {
    id: string;
    description: string;
    amount: number;
    created_at: Date;
  }[];
}

const props = defineProps<Props>();

const formatDate = (date: Date) => {
  const dateObj = new Date(date);
  return dateObj.toLocaleDateString();
};
</script>
<template>
  <q-list dense class="expenses" v-if="expenses">
    <q-item v-for="expense in expenses" :key="expense.id">
```

```
    <q-item-section>
      <q-card-section class="flex row justify-between">
        <div>{{ expense.description }}</div>
        <div>{{ expense.amount }}</div>
        <div>{{ formatDate(expense.created_at) }}</div>
      </q-card-section>
    </q-item-section>
  </q-item>
  </q-list>
</template>

<style scoped>
.expenses {
  min-width: 400px;
  max-width: 80vw;
}
}
</style>
```

Now, we can wrap the details of an expense and the categories together with CategoryOverview.
vue by importing the ExpensesList component and using that list component instead of rendering
the raw data:

```
<script setup lang="ts">
// ...abbreviated

import ExpensesCategoryTotal from './ExpensesCategoryTotal.vue';
import ExpensesList from './ExpensesList.vue';

// ...abbreviated
</script>
<template>
  <div class="masonry">
    <!-- ...abbreviated -->
  </div>

  <q-dialog v-model="showDialog">
    <q-card v-if="expensesByCategoryId && expensesByCategoryId.length
> 0">
      <q-card-section>
        <div class="text-h6">
          {{ `${selectedCategoryName} expenses overview` }}
        </div>
```

```
      </q-card-section>

      <q-separator />

      <q-card-section style="max-height: 50vh" class="scroll">
        <expenses-list :expenses="expensesByCategoryId" />
      </q-card-section>
    </q-card>
  </q-dialog>
</template>

<style lang="scss" scoped>
// ...abbreviated
</style>
```

At this point, we have an app where a user can manage categories and add expenses, which are shown in an overview.

With this as our basis, it would be a good exercise to try and expand the app with different functionalities on your own if you like. How would you add a date filter to the expenses? At what level would you introduce them and what will they affect? Or how about removing expenses?

In the next section, we'll convert our web application into a desktop application. The conversion can be done for every improvement you make in the code, so feel free to continue and add some new features later!

Building the app with the Quasar CLI

The Quasar CLI offers some commands to quickly build and publish an Electron app. It is important to realize that at this point, *the output of the default build script defaults to supporting only your current operating system and architecture*! That's a good way to test the app, so let's see what happens. We can generate our app code by running the following command in the terminal:

```
quasar build -m electron
```

This will take a bit longer to process than spinning up the development server: first, Quasar builds the files for the web and then uses that production-ready code with Electron to compile a native app. Once the processing is done, you can locate the build output files in the `/dist/electron` folder of your project. You should also be able to execute your app now!

Next, we'll improve the visual identity of the application by creating and providing our custom app icon.

A custom icon

With any application, an icon is as important as its name. With our web apps, we have omitted that part since there are plenty of resources for adding a favorite icon to a website. For our desktop app, however, we'll recreate the steps for adding an icon.

I downloaded a suitable icon from **Flaticon** (`https://www.flaticon.com/free-stickers`) and placed it in the `src/assets` folder. Quasar offers a small tool called **Icon Genie** (`https://quasar.dev/icongenie/introduction`) that we can run in the terminal using `npx`. We can generate the icon set with the following command:

```
npx icongenie generate -m electron -i ./src/assets/icon.png
```

If the `npx` command doesn't work, you can try to install the node package globally using the following terminal command:

```
npm i -g @quasar/icongenie
```

Once the global installation is done, you can run the package from your projects' folder, like so:

```
icongenie generate -m electron -i ./src/assets/icon.png
```

Once the script is done, you will find the output in the `src-electron/icons` folder. That's all there is to it!

Our default output is only suited for the platform we're developing with. What we're interested in is building for different platforms as well. We'll discover those options in the next section.

Packaging for different targets

By default, Quasar uses **Electron Packager** (`https://electron.github.io/electron-packager/main/`) under the hood to create the package for you. You could change it to **Electron Builder** (`https://www.electron.build/index.html`) as well, but for this example, we'll use the default.

If you open the `quasar.config.js` file, you can scroll to the `electron` property:

```
// ...abbreviated
  electron: {
      // extendElectronMainConf (esbuildConf)
      // extendElectronPreloadConf (esbuildConf)

      inspectPort: 5858,

      bundler: 'packager', // 'packager' or 'builder'
```

```
      packager: {
        // https://github.com/electron-userland/electron-packager/
blob/master/docs/api.md#options
        // OS X / Mac App Store
        // appBundleId: '',
        // appCategoryType: '',
        // osxSign: '',
        // protocol: 'myapp://path',
        // Windows only
        // win32metadata: { ... }

      builder: {
        // https://www.electron.build/configuration/configuration

        appId: 'packt-expense-tracker',
      },
    },
  },
// ...abbreviated
```

This is the default setup. Since we're dealing with the Electron Packager, our configuration goes in the packager property. To build for additional platforms, we can add the target platforms we want to build for as properties (as documented in the guide at https://electron.github.io/electron-packager/main/modules/electronpackager.html#officialplatform), and for each property, we can add additional configuration. We'll use the configuration properties and default presets to target specific architectures per platform. The following configuration will attempt to build the app for macOS (Darwin), Linux, and Windows (Win32) and target specific architectures as an example:

```
electron: {
  inspectPort: 5858,
  bundler: 'packager', // 'packager' or 'builder'

  packager: {
    platform: ['darwin', 'linux', 'win32'],
    darwin: {
      arch: ['x64'],
    },
    linux: {
      arch: ['x64', 'arm64'],
    },
    win32: {
        arch: ['x64'],
    },
```

```
    },
    // ...abbreviated
  },
```

Now, there are some caveats. If I run the preceding code on a Mac or Linux machine, I will need to install a Windows emulator to build for that platform. The good news is that if you run **HomeBrew** (`https://brew.sh/`), you can install **Wine** fairly easily with the following command:

```
brew install --cask wine-stable
```

The Quasar CLI provides you with that instruction. For non-macOS users, the app that you build will be unsigned. This means that users will need to manually accept the gatekeeping security warning. It will also be impossible to publish to the app store.

With the flexibility that Quasar and Electron offer in terms of native app development, it is still a very viable way of delivering apps that need to run on multiple platforms. Also, bear in mind that these apps may not be as performant concerning applications that were developed specifically for and on a target platform. Development is often about tradeoffs. However, having the possibility to develop and deliver a web app to a native platform is very useful.

And remember, the basis of our app is a **single-page application** (**SPA**) that we can build as a production-ready app by simply replacing the `-m electron` flag with `-m spa`. It will bundle and build our application so that it's ready for the web.

Summary

In this chapter, we unlocked a powerful capability: we started with what we learned in *Chapter 6* and built a web application. Using Quasar's available features, we processed our code and deployed our web application as a standalone desktop application that's suitable for multiple platforms.

We also adopted another framework to build an app. Instead of Vuetify, we relied on the default components that Quasar offers. This way, we've seen and experienced slight differences in code styles in building applications, using frameworks and build tools. We also experienced similarities, for instance, in the usage of Pinia as a centralized store.

This way of app building is not always the most suitable. There are some limitations and tradeoffs. On the upside, you only have to build one application and can deploy it to multiple targets. The cost efficiency of this development method makes it a serious candidate for multi-platform strategies.

In the next chapter, we'll work on something fun. We'll connect multiple devices to a single server and build a real-time quiz!

Part 3: Advanced Applications

This part of the book covers complex use cases, and you will learn how to break complexity down into individual parts and distribute responsibility accordingly. Also, you will learn the difference between, and use cases for, using Vue.Js as a frontend framework and adopting a meta-framework such as Nuxt to build server-side applications.

In addition, you will experience the process of investigating by prototyping, building, and iterating with experimental frameworks and solutions.

This part has the following chapters:

- *Chapter 8*, *Building an Interactive Quiz App*
- *Chapter 9*, *Experimental Object Recognition with TensorFlow*

8
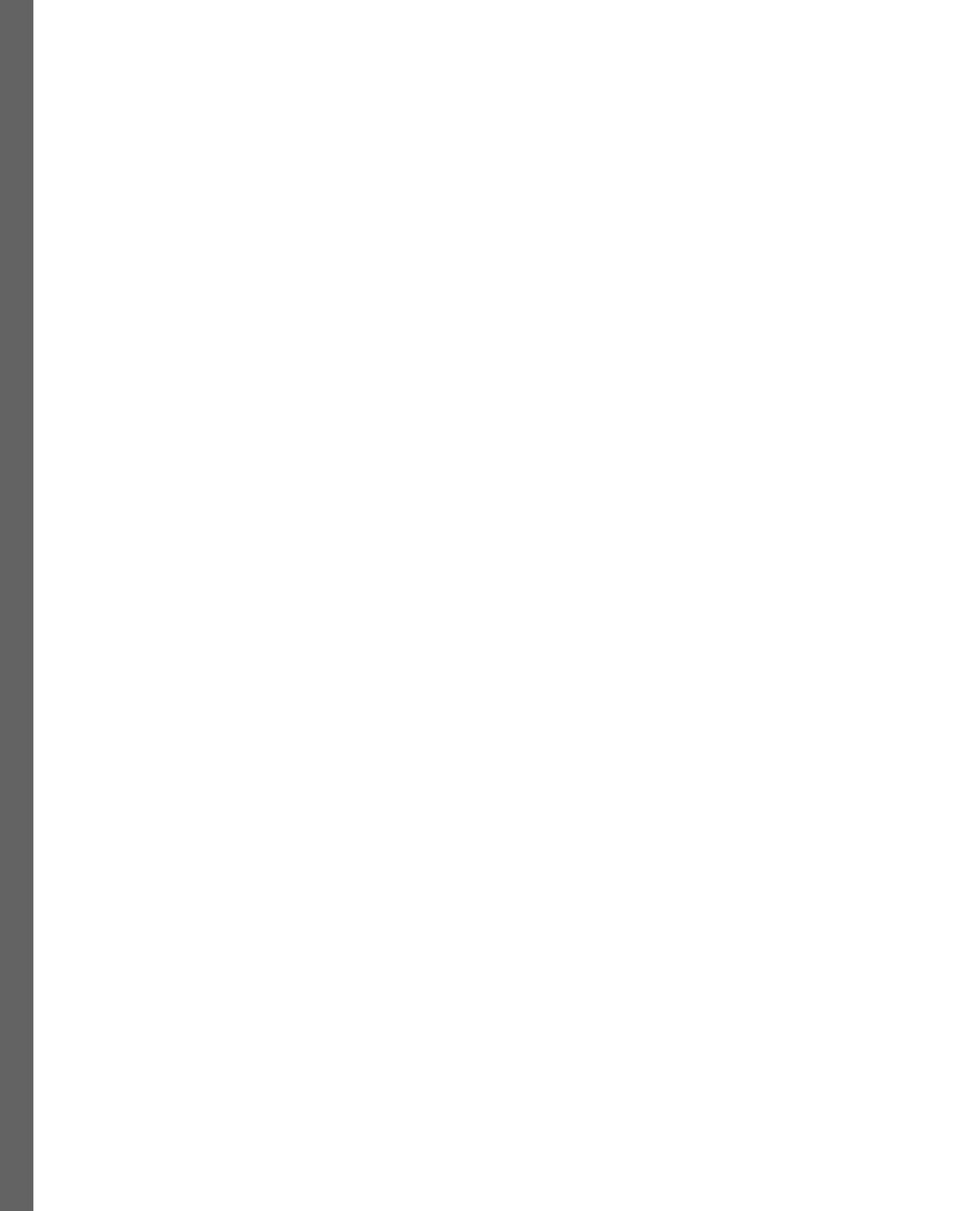
Building an Interactive Quiz App

We'll be stepping up the complexity in this chapter by creating a quiz app that has an admin panel and a real-time connection among multiple clients through the use of WebSockets. WebSockets differ from our usual endpoints by keeping the connection open, allowing for continuous updates to be sent from a central socket server to one or more clients. Using these features, we'll build a small-scale Kahoot clone.

For the admin panel, we'll use Nuxt (`https://nuxt.com/`). Nuxt is a framework that was built as an extension of the Vue philosophy but extended with server capabilities. Save for the Electron app in *Chapter 7*, all of our code can run in the browser of the client. Nuxt allows Vue code to be executed on a server. On top of that, it has a lot of extra capabilities that drastically improve the **developer experience** (**DX**). We'll encounter these capabilities as we progress through this chapter.

In this chapter, we'll cover the following topics:

- Familiarizing yourself with Nuxt and server-side rendering
- Understanding the difference between the REST API and WebSockets
- Client experience and server roles in complex application architecture
- Structuring application logic
- Using Node.js scripts in development environments

We're going to build three different applications that need to communicate with each other to form an interactive quiz.

Technical requirements

At the heart of our setup lies the **Server Quiz App** (**SQA**), which is scaffolded around Nuxt (`https://nuxt.com/`), Pinia for state management (`https://pinia.vuejs.org/`), Supabase for managing quiz data (`https://supabase.com/`), and Vuetify (`https://vuetifyjs.com/`) for rendering a management interface.

We'll build a standalone **Socket Quiz Server** (**SQS**) that uses Express (`https://expressjs.com/`) to process incoming requests and sets up a `socket.io` server (`https://socket.io/`) to maintain a real-time connection between clients.

Lastly, our **Client Quiz App** (**CQA**) will use Vuetify to render the quiz elements (`https://vuetifyjs.com/`).

You can find the code complete code for this chapter here: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/tree/main/08.quiz`.

Entities in the quiz app setup

To give you an idea of how our elements will work together, let's quickly take a look at the following figure:
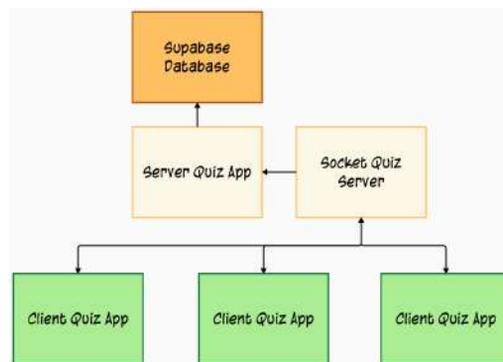
Figure 8.1 – Overview of the entities in the quiz app setup

The big change concerning our previous projects is that our client (CQA) is not directly communicating with the Supabase database anymore. Instead, it connects via the SQS, where it retrieves questions and scores and sends its answers back to the SQS. The SQS, in turn, communicates with the SQA to retrieve relevant quiz information and centralizes the session of a current active quiz between its clients (CQAs).

The SQA is used to manage the contents of quizzes and interact with the database.

> **A word beforehand**
>
> Since the setup is fairly complex, this chapter will not focus on, and instead leave out, security measures. It is good to realize the limitations of the project and don't treat this as production-ready code. Where possible, we'll mention implications or possible solutions briefly.

Let's dive in by setting up the database.

Setting up the database

As with previous chapters, we'll start this chapter by setting up our database. We'll create another project called `quiz`, set a strong database password, and select a geographically close region.

Remember to note the **Project URL** and **API Key** values!

> **Note**
> The intent of this chapter is not to focus on database management and the following settings should not be considered a best practice for a production app!

Follow these steps to set up the database:

1. Go to **Table Editor** and choose **Create a new table**. Here, do the following:

    I.   For the name, insert `quiz`.

    II.   Uncheck **Enable Row Level Security (RLS)** and confirm the dialog after reading its warning.

    III. In the **Columns** section, change the type of the **id** field to **uuid**.

    IV.   That's enough for this table, so click **Save**.

    We will only use this table as a grouping mechanism for questions, so we're keeping it as simple as possible.

2. Now, return to **Table Editor** and click **Create a new table**. Apply the following settings:

    I.   For the name, insert `questions`.

    II.   Again, uncheck **Enable Row Level Security (RLS)** and confirm the dialog after reading its warning.

    III. In the **Columns** section, change the type of the **id** field to **uuid**.

    IV.   Create a new column called **quiz_id** and click the **Edit foreign key relation** button:

         i.   In the side panel, select the `quiz` table and check that the **id** column is automatically entered.

         ii. Click **Save** to close the foreign key property.

    V.   Create a column called **question** and set its type to **text**.

    VI.   Create four columns called **answer_1** through **answer_4** with their type set to **text**.

    VII. Create a column called **correct** with its type set to **int2**.

3. Click **Save** to create the table.

We're not going to import any preset data since we'll use our SQA to handle insertions for us! Let's get started by building our first application in this project.

The SQA

To organize all our applications, we'll create subfolders for every project in our chapters' root folder. Since this app will run on Nuxt, we can use the `Nuxi` CLI to install our project for us. From the root of our project, we'll run the following command in the command line:

```
npx nuxi@latest init server
```

We'll simply pick `npm` as our package manager. Once the installation is done, navigate to the `server` folder and run `npm run dev` to start the application. By default, it will run on port `3000`. Upon opening the URL in your browser, you should see something like this:
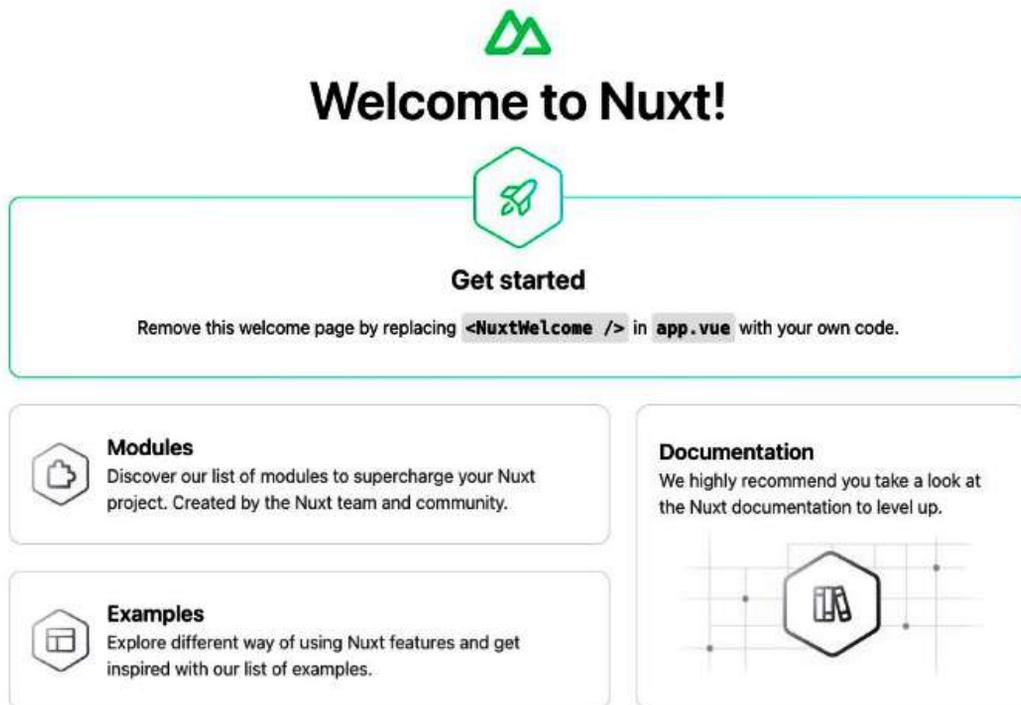
Figure 8.2 – The welcome screen of a fresh Nuxt installation

While this may not look like much, please inspect the source of this page. Instead of rendering a virtual DOM to a `<div id="app" />` element, Nuxt runs as a Node.js process, which (among other things) means that it supports server-side rendering of Vue components! This can be very beneficial since you don't have to rely on JavaScript being executed in the browser, which positively impacts search engine optimization, crawling support, rendering performance, and browser memory usage. For more information, please visit the Nuxt official docs (`https://vuejs.org/guide/scaling-up/ssr.html#server-side-rendering-ssr`).

With our foundation in place, let's add some extra capabilities using the Nuxt module system.

Modules and auto-imports

We'll start by adding Vuetify to our project. Nuxt has a solid community that contributes to certain modules for everyone to use. In our case, we'll use the *Nuxt Vuetify Module* (`https://nuxt.com/modules/nuxt-vuetify`). In our `server` folder, run the following command:

```
npm install --save-dev @invictus.codes/nuxt-vuetify
```

Nuxt modules can be registered and configured via the `nuxt.config.ts` file by changing its contents, as follows:

```
// https://nuxt.com/docs/api/configuration/nuxt-config
export default defineNuxtConfig({
  devtools: { enabled: true },
  modules: [
    '@invictus.codes/nuxt-vuetify'
  ],
  vuetify: {
    moduleOptions: {
      treeshaking: true,
      useIconCDN: true,
      styles: true,
      autoImport: true,
      useVuetifyLabs: true,
    }
  }
})
```

In the `modules` property, we register the module we want to use, and, optionally, on the (in this case) `vuetify` property, we configure how the module should behave.

That's all we need! We can now use the Vuetify templates in our application. Nuxt supports a concept called auto-importing, which means that for commonly used scripts, we don't need to explicitly write the import statement in our script block. Nuxt can figure out the required file at runtime! As you will see when we start to write our code, this will make our files very clean and readable.

File-based routing

Similar to the automated imports, Nuxt uses our familiar `vue-router` by default, and it's configured to create routes for you, based on a certain file structure (`https://nuxt.com/docs/getting-started/routing#routing`).

We'll start with a default layout by creating a `default.vue` file in the `layouts` folder: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.1-default.vue`.

As you can see, we're depending on Vuetify components to create a simple layout for us. You can already identify a couple of routes. We'll create them in the next section. For now, we can use the default layout in our app by updating the `app.vue` file in the SQA root folder with the following contents:

```
<template>
  <NuxtLayout>
    <NuxtWelcome />
  </NuxtLayout>
</template>
```

As you can see in the menu, we need a couple of routes: a home route, a route to manage quizzes, and a route where we can share a quiz.

We'll start with the home route by creating a `pages` folder in the root of the SQA folder and adding an `index.vue` file with the following contents:

```
<template>
  <v-card class="mx-auto" width="600">
    <template v-slot:title>Welcome to the Admin Panel</template>
    <v-card-text>
      <p>
        Via this interface, you can create and edit quizzes. This
panel is not perfect, but it works. As an extra challenge consider
implementing the following features:
      </p>
      <ul class="ma-4">
        <li>
          Adding meta data to a quiz, such as a title and make it more
identifiable in Admin Panel;
        </li>
        <li>Managing the order of Questions in a Quiz;</li>
        <li>Securing the Admin Panel via the Supabase OTP
authentication;</li>
        <li>Adding validation on the Question dialog inputs</li>
      </ul>
      <p>Good luck!</p>
    </v-card-text>
  </v-card>
</template>
```

Now, if we flip back to the app.vue file and replace the <NuxtWelcome /> component with the <NuxtPage /> component (notice that we don't need to import these components in our script block to use them?) while the development server is active, our home route now opens the contents of ./pages/index.vue in our app!

So, what's happening here? The NuxtPage component has some logic built in that can read the pages folder and dynamically create routes on the vue router instance that's initialized as a default part of Nuxt. Neat!

Now, if we want to route to a certain path, we can simply create a new folder in the ./pages folder with a name that matches the route. In our case, we'll create a quiz subfolder and add another index.vue file in that folder.

> **Note**
>
> Technically, you could also opt to create a file called quiz.vue in the ./pages folder. However, since we will add multiple routes that are part of the quiz domain, it's a better practice to group them in their dedicated folder.

We'll begin with a basic file as part of ./pages/quiz/index.vue:

```
<template>
  <div class="my-8">
    <h1 class="text-h3 mb-8">Choose quiz to edit</h1>
    <v-card class="mx-auto" max-width="600">
      <v-divider />
      <v-card-actions>
        <v-btn primary class="my-4 mx-4">❇ Create new Quiz</v-btn>
      </v-card-actions>
    </v-card>
  </div>
</template>
```

Now, when navigating to `http://localhost:3000/quiz` in your browser (or using the **Manage Quizzes** button in the navigation drawer), you should see the following page:
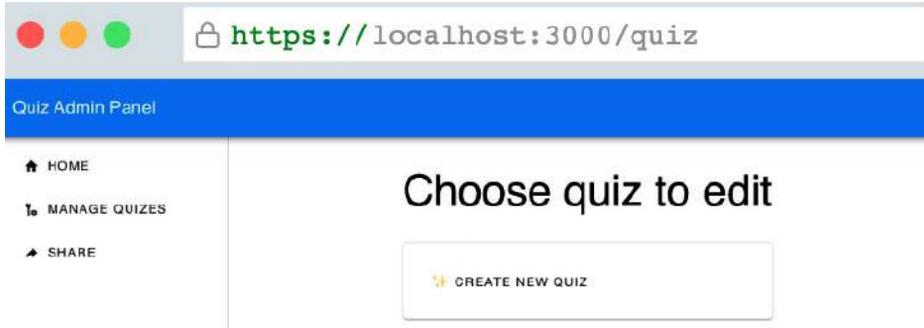
Figure 8.3 – File-based routing in action

Our static page doesn't help us that much, so we'll focus on establishing the connection and data from our database in the next section.

Reintroducing two familiar friends

As we did in *Chapters 6* and *7*, we'll depend on the Supabase JS client and Pinia. Let's see how that works.

First, we'll install the Supabase JS Client (`https://www.npmjs.com/package/@supabase/supabase-js`) with the npm command:

```
npm install @supabase/supabase-js
```

We'll also create a `./composables` folder with a `supabase.ts` file that has the following contents:

```
import { createClient } from '@supabase/supabase-js'

const supabaseUrl = import.meta.env.VITE_SUPABASE_URL
const supabaseAnonKey = import.meta.env.VITE_SUPABASE_ANON_KEY

export const useSupabaseClient = createClient(supabaseUrl as string,
supabaseAnonKey as string, { auth: { persistSession: false } });
```

Nuxt is set up to scan the `./composables` folder and extract the exports so that it supports auto imports for composables as well! There's one caveat: it only scans one level and excludes nested folders.

As you can see, we need to set up a `.env` file in the root of our SQA folder with the URL and API key that we received when we created our database. The `.env` file has the same setup as in *Chapters 6* and *7*:

```
VITE_SUPABASE_URL=YOUR_SUPABASE_URL
VITE_SUPABASE_ANON_KEY=YOUR_SUPABASE_ANON_KEY
```

With our client in place, we can add Pinia – this time, as a Nuxt module. Run the following command in the terminal to install the packages:

```
npm install pinia @pinia/nuxt
```

If you run into an ERESOLVE error during installation, please look at the tip provided in the installation guide (https://pinia.vuejs.org/ssr/nuxt.html#Installation).

We'll add the module to the modules array on the nuxt.config.ts property:

```
modules: [
  '@invictus.codes/nuxt-vuetify',
  '@pinia/nuxt'
],
```

For an even better coding experience, we can define the auto imports for Pinia functions as well by adding the following property to the Nuxt config:

```
pinia: {
  autoImports: [
    'defineStore',
    ['defineStore', 'definePiniaStore'],
  ],
},
```

The nuxt.config.ts file will look like this: https://github.com/PacktPublishing/ Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08. quiz/.notes/8.2-nuxt.config.ts. With Supabase and Pinia now in place, we can create our quiz store!

For our stores, we'll create the ./stores folder with a quiz.ts file that has the following contents: https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.3-quiz.ts.

If you compare this store to, for instance, the *User Store* from *Chapter 6*, you'll notice that a lot of imports are missing. That's because they're being handled by Nuxt! Let's quickly summarize the functionalities of the *Quiz Store* since we're not going to go in depth.

Our store exposes a list of all quizzes (*line 7*) and the method for retrieving that data from Supabase (*lines 11–30*). Our store also exposes the properties of a single quiz (*line 8*) and the accompanying method of retrieving quiz data from the database (*lines 62–82*). For both the quiz and answers to the quiz, we expose methods to upsert data and remove data. That's the basic management we need to continue with.

We'll revisit our ./pages/quiz/index.vue file to add management at the quiz level: https:// github.com/PacktPublishing/Building-Real-world-Web-Applications- with-Vue.js-3/blob/main/08.quiz/.notes/8.4-index.vue.

When you run the development server, you should be able to add a couple of new quizzes and have them show up in this overview, as shown in the following screenshot:
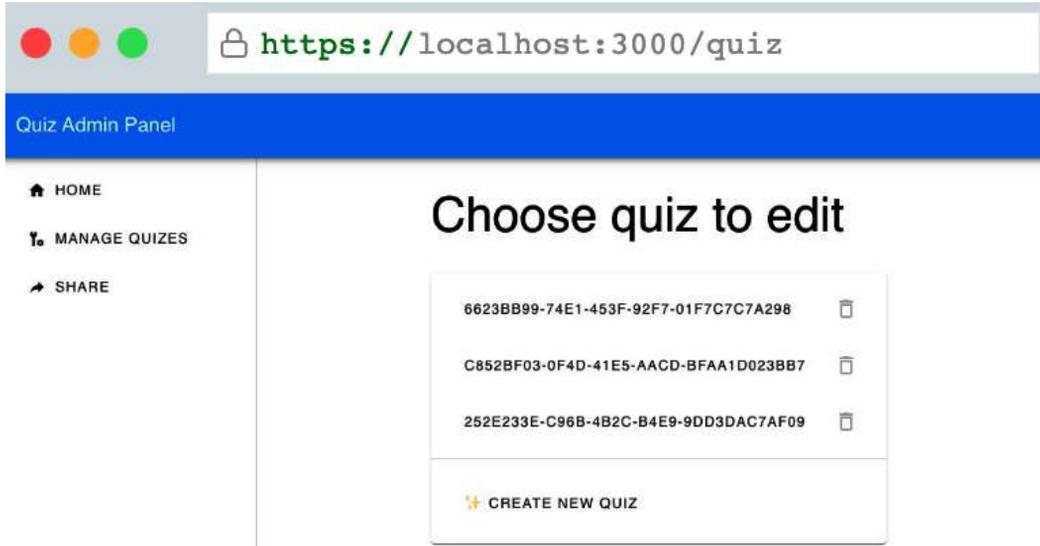
Figure 8.4 – Our application connected to the database via a Pinia store

We can now build a similar overview of quizzes for the /share route with ease by creating a ./share subfolder in the ./pages folder, creating an index.vue file, and pasting the following contents in that file: https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.5-index.vue.

We're getting there! If you noticed the <nuxt-link /> component in both pages we created, you may have noticed its similarities to <router-link />. The <nuxt-link /> component is a wrapper for it but also helps Nuxt in determining a map of all possible routes. Together with file-based routing, these occurrences and configuration of the <nuxt-link /> components help in determining the configuration of the vue-router implementation. You may have also noticed that both components link to a dynamic route as a child of /quiz and /share, respectively. Let's fix those!

Dynamic file-based routes

We'll start with the ./share route. As you can see by the markup of the link component, we're targeting a route with a parameter:

```
<nuxt-link :to="`/share/${quiz.id}`">
  <v-btn flat>{{ quiz.id }}</v-btn>
</nuxt-link>
```

Normally, we'd configure our Vue router configuration with something like the following example entry:

```
routes: [
  {
    path: '/share/:id',
    name: share,
    component: () => import('../pages/ShareDetail.vue')
  },
]
```

Using Nuxt, passing a parameter is as simple as marking it in the filename. Next to the `index.vue` file, we'll create a file called `[quiz_id].vue`. This is the equivalent result of defining the parameter in a route file. It tells Nuxt to instruct the router to create a route with a parameter of `quiz_id`. In the file, we can read the parameter by utilizing the `useRoute` composable and accessing the `params.quiz_id` property!

In the `[quiz_id].vue` file, we'll add the following contents: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.6-%5Bquiz_id%5D.vue`.

The page itself is nothing fancy: it just generates an anchor link element that points to a URL that we will need to build an app for in the final section of this chapter!

To complete the management interface, we'll create a page for editing questions in a certain quiz. First, as a requirement, we'll create a `./components` folder with a `FormQuestion.vue` file that has the following contents: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.7-FormQuestion.vue`.

In the file, we're directly mapping a form to the columns in our database. It's pretty straightforward to set up, although I'd like to point out the repeating setup for the answers (*lines 33–48*) where we use a `v-for` directive to generate the four answer fields and map each field's `v-model` using dynamic object keys (*line 36*).

Now, we'll create a dynamic route as a descendant of the `quiz` route by creating a `[quiz_id].vue` file in the `./pages/quiz` folder: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.8-%5Bquiz_id%5D.vue`.

With these pages finished, you should be able to create one or two small quizzes with multiple-choice questions. Before we continue, we need to have at least one, but preferably more quizzes in the database, with a couple of multiple-choice questions.

Why don't you try and create a quiz about the Vue ecosystem?

Setting up the SQS

Our next order of business is setting up the server that handles the requests from one or more clients. This will be a small standalone application and it will get its data from our Nuxt server since that already has a connection with the database instance. Creating an endpoint in Nuxt is something we haven't built yet because our Nuxt application is only capable of presenting a management app!

Nuxt API routes

As I mentioned previously, the Nuxt app runs as a Node process. When we request pages, it acts as a web server that interprets the Vue components and routes to return an HTML response. In addition to that, it can also act as a server at the same time! Nuxt uses the Nitro server engine (https://nuxt.com/docs/guide/directory-structure/server) to process requests on scripts in the ./server folder. It also supports file-based routing and parameters, similar to the ./pages folder.

To server our quizzes as part of a RESTful API, we'll create the ./server/api/quiz structure in our Nuxt project. In the quiz folder, we'll create an index.ts file. This will be available on /api/quiz requests to the Nuxt URL:

```
import type { QuizHead } from '../../types/quiz';
import { useSupabaseClient } from '../../composables/supabase';

export default defineEventHandler(async (): Promise<QuizHead[] | null>
=> {

  console.log("📄 Requesting quizzes from endpoint")

  const { data, error, status } = await useSupabaseClient
    .from(`quiz`)
    .select(`id, created_at`);

  if (error && status !== 406) console.error(error);

  return data
})
```

Nothing special is going on here. We've added console.log with a 📄 emoji, which will help us analyze the application flow once it's been created. If your development server is running (a restart might be required when adding new files), you should be able to request the quiz information via this URL: http://localhost:3000/api/quiz.

If we've set up our endpoint correctly, we should see something like this in the browser. This is the contents of our quiz table in JSON format:

Figure 8.5 – Example of the quiz API server from Nuxt

We'll add another entry point using the parameterized file notation. We'll create a [id].ts file next to the index.ts file: https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.9-%5Bid%5D.ts.

We're using this endpoint to quickly retrieve all questions and answers and the correct answer to be used in our setup. Now that we're able to retrieve quizzes and their details, we can finally build the next part of our setup: the SQS.

Setting up a basic Node project

In this section, we won't be using any Vue-related software. This part will depend mainly on Express (https://expressjs.com/) and Socket.io (https://socket.io/). This means we can't rely on helpful CLI tools to create our project for us. Luckily, it's not hard. Navigating back to the root of our project, we'll create a new folder called ./sockets. Via the CLI, we'll run the npm init command to define our project, where we'll simply accept all the defaults. Once done, we'll install Express and Socket.io and some TypeScript tooling packages via the CLI:

```
npm install express socket.io ts-node @types/node
```

We'll also create a tsconfig.json file in the ./sockets folder that contains the following configuration:

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "commonjs",
```

```
    "lib": ["dom"],
    "allowJs": true,
    "outDir": "build",
    "rootDir": "./",
    "strict": true,
    "noImplicitAny": true,
    "esModuleInterop": true,
    "resolveJsonModule": true
  }
}
```

For the game mechanics, I've provided a `Quiz` class that we can implement without going into any details. Create a `quiz.ts` file with the following contents: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.10-quiz.ts`.

Next, we can work on the socket connection. We'll create an `index.ts` file in the `./sockets` folder with the following contents:

```
const express = require("express");
const { createServer } = require("node:http");
const { Server } = require("socket.io");

import QuizGame from "./quiz";

const app = express();
const server = createServer(app);

// This needs to match the client url of the running app instance
const clientAppUrl = "http://localhost:5173";

let serverOptions = {};
if (process.env.NODE_ENV !== "production") {
  serverOptions = {
    ...serverOptions,
      cors: {
        origin: clientAppUrl, // cors is enabled for socketed
connections on localhost:5173
      },
    }
}

const io = new Server(server, serverOptions);
```

```
const game = new QuizGame();

// *****************************
// Listen on the port for events
// *****************************
server.listen(4000, () => {
  console.log("🎲 Server is running on port 4000");
});
```

This code scaffolds a blank server. As you can see, we've imported the Quiz class and used it to instantiate a game on the server. clientAppUrl is important: when using sockets, the **Cross-Origin Resource Sharing** (**CORS**; https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS) policy will block incoming traffic coming from localhost, unless its origins are provided to the server.

We do have another problem now, though: the Quiz class expects data that comes from our Nuxt server, remember? We need to have the Nuxt server running in tandem with the SQS. We can do this manually, in separate terminal windows, but we can also script the boot process for more convenience. After all, once our work on the SQS is done, we need to be able to run another app to act as a client!

Executing scripts in parallel

In this section, we'll focus on the root folder of our project, from where we'll trigger the scripts in sub-folders. We'll navigate to the root with the terminal and set up another Node project using npm init, accepting the defaults. Once we're done, we'll install some helper packages:

```
npm install nodemon npm-run-all ts-node @types/node open chalk
```

We'll also create a file called open.mjs with the following contents:

```
import chalk from 'chalk';
import open from 'open';

const urls = [{
    name: '📋 Quiz Admin Panel',
    url: 'http://localhost:3000',
}]

urls.forEach(url => {
    setTimeout(async (): void => {
        console.log(`🎲 Opening ${chalk.black.bgCyan(url.name)} at
${chalk.magenta(url.url)}`);
        await open(url.url);
    }, url.wait || 0);
});
```

In this file, we're using the open package to automatically open a browser window. To make the status more recognizable in the console, we use chalk to add color to certain parts of the logs. Now, when we open package.json, we can change the scripts property so that it matches the following sample:

```
"scripts": {
  "dev": "npm-run-all --parallel dev:*",
  "dev:sockets": "nodemon ./sockets/index.ts",
  "dev:server": "cd server && npm run dev",
  "dev:open": "node ./open.mjs"
},
```

So, what's happening here? The npm-run-all command triggers multiple commands in parallel mode – in our case, all scripts that are prefixed with dev:

- The dev:sockets script uses nodemon to run and watch for file changes in our socket project folder

- The dev:server script opens the Nuxt server from its folder

- The dev:open script executes the open.mjs script, which, in turn, opens a browser window on the predefined URL for the SQA

If you don't want the SQA to open every time you start the script, you could remove the dev:open script or remove the entry from the urls constant in the open.mjs file.

At least we can now control the execution of multiple scripts with a single command. If we run npm run dev from the root of the project, it will automatically start the Nuxt server as well as the SQS!

Let's take a step back and explain why a regular endpoint is not sufficient, as well as why we're building a complete socket server as a layer between the server and the client(s).

Why sockets?

We're using a socket server to act as the host of our quiz app. Sockets have the advantage over RESTful connections in that data transmission is bidirectional by default and a socket connection is stateful and persistent. This makes sockets ideal for applications that depend on instant updates, such as chat applications or, as in our case, online games (especially those that support collaboration or competition).

We're starting a socket server where clients can connect to. A handshake takes place between the server and client on connecting, which establishes the agreements and protocol between the two. This handshake allows the server to identify the client, which is useful when we're dealing with multiple clients but want to target individuals.

Once the connection has been established, both the client and server can send events with context. The server can broadcast to all clients or an individual. The clients only send events to the server. Depending on the event and the context, we can process the event.

Using Sockets.io, we can broadcast events using `io.emit('event', context)` and listen to events using `io.on('event')`.

In our application, we're going to see both methods in action on both the server and the client: the server handles the information being sent to clients and receives answers given by a client – we'll even use it to control the navigational state of the client app! From the client's side, we'll listen to the sent events and use the sockets to send answers to our quiz's answers.

Completing the SQS

We can finalize the SQS by replacing the contents of the `./index.ts` file in the `./sockets` folder with the following: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.11-index.ts`.

We've added the mechanics for communication now. The socket server gets initialized on *lines 27–76*. In the socket connection event, we have defined all the events the server needs to listen to:

- When a player has joined (*line 17*), the server signals other players with the current status (*lines 32, 130 –133*)

- When a player is ready to start playing (*lines 37–41*), a player is added to the current game (*line 38*), and the other players are notified

- If a player answers a question during a game (*lines 43–50*), the answer is processed and, if needed, the player's score is updated

- When a player selects a quiz (*lines 55–60*), the server queries the Nuxt server for the questions and updates it to the current game (*line 59*)

- When the quiz is started (*lines 62–66*), the current question is set to the first of the collection and the question is sent to the player (*lines 65, 82–88*)

We have some options to send players to certain views:

- There's a method to send a question to the players (*lines 82–88*)

- In between every third question, we can show the current score (*lines 114–118*)

- Once the quiz has ended, we direct players to the final score (*lines 120–125*)

As part of the game mechanics, answering the questions is time-based. The timer runs internally as part of the game, but we emit events once a timer has started (*lines 138–145*) or when it has ended (*lines 147–150*).

With all of this, we have the basic needs for an interactive quiz. It's not very robust, mind you, but it functions well enough to get our point across.

In the final section, we're going to create an app to complete this whole quiz application platform!

Creating the CQA

To wrap it all up, we're going to create the client app as a Vuetify project in our projects' root folder. Let's navigate to the root folder by using the terminal and typing `npm create vuetify` to begin the installation. We'll use the following settings:

- `app` as the project's name
- For the preset, we'll choose the default of `Base (Vuetify, VueRouter)`
- We'll select TypeScript
- To install dependencies, we'll select `npm`

Once the installation completes, we can open the folder in our IDE to start editing.

First, we'll make some changes to the `vite.config.ts` file, which helps our app to work in our multi-app environment. We'll add a new property called `clearScreen` with a value of `false`. This will prevent the process from clearing the logs (which will also contain our servers' logs). We can simply add it at the bottom of the file.

Next, we'll locate the `server` and `port` properties and change `port` to the new value – that is, `5173` (this corresponds to the CORS setting of our socket server). This prevents the Vuetify application from trying to occupy the same port as our Nuxt application! The bottom of the `vite.config.ts` file should resemble the following code:

```
// ...abbreviated
export default defineConfig({
  plugins: [
    // ...abbreviated
  define: { 'process.env': {} },
  resolve: {
    // ...abbreviated
  },
  server: {
    port: 5173,
  },
  clearScreen: false,
})
```

Before we continue working on the app, we will add our client app to our main development script. Let's open the `package.json` file from our project's root and add a script to run our Vuetify development script:

```
"scripts": {
  "dev": "npm-run-all --parallel dev:*",
  "dev:client": "cd app && npm run dev",
  "dev:sockets": "nodemon ./sockets/index.ts",
  "dev:server": "cd server && npm run dev",
  "dev:open": "node ./open.mjs"
},
```

Regarding the `open.mjs` file, we will add a URL that will automatically open the browser as well. We'll give it a timeout of `5000` milliseconds because it needs to wait for Nuxt and the socket server to have been initialized:

```
const urls = [
  {
    name: "📘 Quiz Admin Panel",
    url: "http://localhost:3000",
  },
  {
    name: "📱 Quiz App",
    url: "http://localhost:5173",
    wait: 5000,
  },
];
```

Now, we can boot the app from the projects' root by using the `npm run dev:client` command. We can also start up all of our scripts at once using the `npm run dev` command from the root of the project. This is very handy when working on features of the app since the app by itself is very reliant on the other entities running.

Setting up the app

We'll begin by setting up the routes and views for our app. We'll move back to our app folder to make some changes to the `./src/router/index.ts` file: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.12-index.ts`.

Now, for all those routes, we'll add a separate view Vue component in the `./src/views` folder. We'll use the following template as a basis (replace the title in the template with a corresponding one for that view):

```
<template>
  <div class="my-8">
    <h1 class="text-h3 mb-8">Home</h1>
  </div>
</template>
```

In the end, you'll end up with the following views:

Figure 8.6 – Newly created views to match our routes

We'll also simplify the layout by replacing the contents of `./src/layouts/default.vue` with the following:

```
<template>
  <v-layout>
    <v-app-bar class="bg-primary pa-4"
      ><h1 class="text-h5">Quiz time!</h1>
    </v-app-bar>
    <v-main>
      <RouterView />
    </v-main>
  </v-layout>
</template>
```

We can delete the `AppBar.vue` and `View.vue` files from the folder as well.

In the next step, we'll work on adding a connection to our server.

Adding the socket client

At this point, it's time to add the socket client to our app project. In the `./app` root folder, install the package by typing the following command:

```
npm install socket.io-client
```

To use Socket.io in our app, we'll create a file called `sockets.ts` in the `./src` folder with the following contents: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.13-sockets.ts`.

This file will handle the connection from the socket server and export the connection as a constant (*line 6*). In this file, we'll also store and expose the data we're receiving in a single object (*lines 18–29*). Since the object has multiple levels, we'll use `reactive` over `ref` because of its deep reactive model (`https://vuejs.org/api/reactivity-core.html#reactive`). This `state` will propagate any changes it receives wherever it's being used in our app.

Listening to socket events

Now, let's create a feature where we can start to open a quiz. We expect users to enter the app via a link that contains the ID of a quiz. For that, we'll update the `./src/views/Start.vue` file and completely replace its contents with the following:

```
<script setup lang="ts">
import { onMounted } from 'vue';
import { useRoute, useRouter } from 'vue-router';
import { socket } from "@/socket";

const route = useRoute();
const router = useRouter();

const init = (id: string): void => {
  socket.emit('quiz:select', id);
}

onMounted(() => {
  const id : string | string[] = route.params.id;
  if (id) {
    init(id.toString());
    router.push('/lobby');
  }
});
</script>
```

As you can see, we don't even need any template here! The component grabs the `id` value from the router parameters. The code uses the socket to emit the `quiz:select` event and then redirects the user to the `/lobby` route.

Automated route changes

Before we continue to the lobby, we'll modify the `./src/App.vue` file too so that it listens to specific state changes in the quiz and redirects to certain routes:

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

<script setup lang="ts">
import { computed, watch } from 'vue'
import { RouterView, useRouter } from 'vue-router'

import { state } from '@/socket'

const router = useRouter()
const quizStatus = computed((): string | null => state.quizStatus)

// when quizStatus changes, check if it's "ready" and if so, redirect
to /quiz
watch(quizStatus, (newStatus) => {
  if (newStatus === 'question') {
    router.push('/question')
  }
  if (newStatus === 'answer') {
    router.push('/answer')
  }
  if (newStatus === 'end') {
    router.push('/final')
  }
  if (newStatus === 'scoreboard') {
    router.push('/scoreboard')
  }
})
</script>
```

Here, we're importing `quizStatus` and watching for changes. Once an updated value matches one of our routes, we'll programmatically update the router with the corresponding view. Now, let's enter the lobby!

Player management in the lobby

In the lobby, we want to be able to add ourselves as a player, but we also want to see an overview of all currently connected players. To add a player, we'll create a new component in the `./src/components` folder called `PlayerAdd.vue`: https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.14-PlayerAdd.vue.

In this case, we will use the socket connection to emit events (*lines 9–11, 25 –27*) but we also read from the state to, for instance, only allow a player to connect once.

Our next component in the folder, called `PlayersOverview.vue`, will complement the lobby: https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.15-PlayersOverview.vue.

This component only reads from the state and presents the overview in a nicely formatted way, with means for a user to identify themselves in the lobby view.

We can now add these two components to the `./src/views/Lobby.vue` file to complete this feature:

```ts
<script setup lang="ts">
import PlayerAdd from '@/components/PlayerAdd.vue'
import PlayersOverview from '@/components/PlayersOverview.vue'
</script>

<template>
  <div class="my-8">
    <h1 class="text-h3 mb-8">Welcome to the lobby</h1>
    <player-add />
    <players-overview class="my-8" />
  </div>
</template>
```

This is our first opportunity to see our sockets in action! So, let's have a look. Once we've run the development script, all three of our applications will run in parallel. To enter the lobby, we need to access the CQA via the local development URL and provide an ID as a parameter to the `/start` route. Remember our admin panel? It has a section called **Share**, where we've shown a list of active quizzes. If you use the link from the admin panel, you should be redirected, via the `/start/{QUIZ_ID}` route, to the lobby:
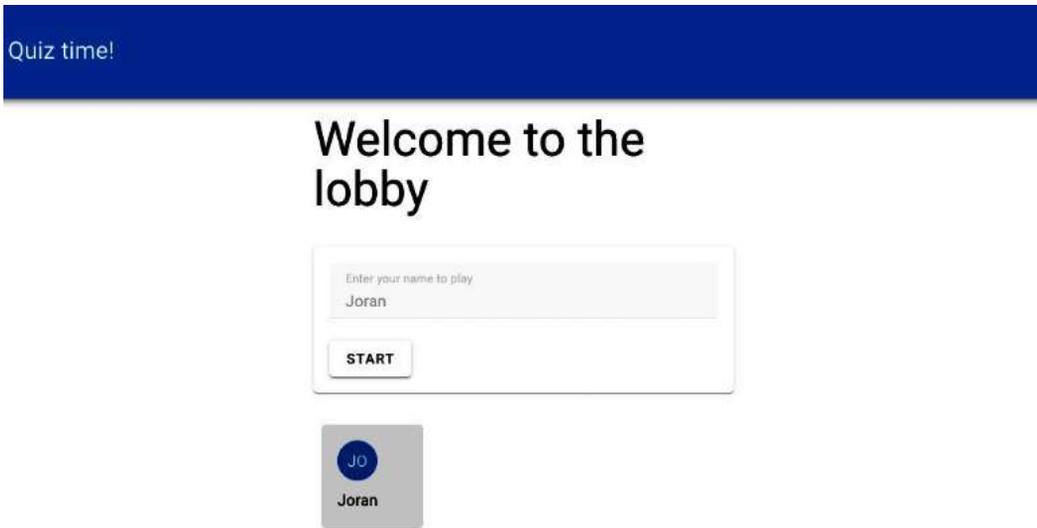
Figure 8.7 – A player has been added to the lobby

Now, without closing the existing browser window on the lobby, flip back to the admin panel and open the same link in a new browser window. You should, again, be redirected to the lobby. But in this case, you will enter as a new player. You should see the player we added previously in the overview, ready to play! If you add a new player name, the new player will be added to both browser windows in real time via the socket server!

Have a look at the console of our terminal. You will see messages indicating that the state of the players has changed:

```
⚞ Player ready tQ01ME_GswTMZhuKAAAD
⚞ Connection established osrzB4ZJsqx9RRYaAAAF
⚞ players:update [ { id: 'tQ01ME_GswTMZhuKAAAD', name: 'Joran', score: 0 } ]
⚞ quiz:select
🔲 Requesting quiz id 6623bb99-74e1-453f-92f7-01f7c7c7a298
⚞ players:update [
    { id: 'tQ01ME_GswTMZhuKAAAD', name: 'Joran', score: 0 },
    { id: 'osrzB4ZJsqx9RRYaAAAF', name: 'Player 2', score: 0 }
  ]
⚞ Player ready osrzB4ZJsqx9RRYaAAAF
```

Figure 8.8 – The console indicates the state of the players from the socket server

Since we've already connected our views and routes, we can trigger the start of a quiz. If one of the players hits **Start**, the quiz will automatically iterate over all of the questions and eventually end up at the **Final Score** view. Feel free to try it out. You will notice that the quiz gets processed by both browser windows at the same time!

We need some answers

Alright – it's time to build on the core of our app: the answering mechanism. For this and upcoming sections, I recommend preparing a quiz that has four questions. That way, we can test all of the mechanics for a reasonable duration.

We're going to create a new component in the `./components` folder called `QuestionForm.vue`: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.16-QuestionForm.vue`.

Similar to how we are adding and showing players, we are using the socket connection to emit our users' input (*line 12*), while also reading the actual state and presenting it in the UI (*lines 15–17*).

In the **Question** view (`./src/views/Question.vue`), we'll import and display this component next to showing the current question as the title of the view:

```
<script setup lang="ts">
import { computed } from 'vue'
import { state } from '@/socket'
import type { QuizQuestion } from '@/types/quiz'

import QuestionForm from '@/components/QuestionForm.vue';

const question = computed((): QuizQuestion => {
  return state.quizCurrentQuestion
})
</script>

<template>
  <div class="my-8">
    <h1 class="text-h3 mb-8">{{ question.question }}</h1>
    <QuestionForm />
  </div>
</template>
```

By adding this component to the view, our users can each try and answer the question. And how about we show the user the results as well? Let's create a new component called `AnswerResult.vue`: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.17-AnswerResult.vue`.

In the preceding file, we're using the state of the quiz coming from the socket server to display whether the individual user has answered correctly. As you can see, we can perfectly track which user is which and show them a customized interface. In this case, we're doing the filtering on the client app.

Again, we can add this `AnswerResult` component to the correct view – in this case, the `Answer.vue` file:

```
<script setup lang="ts">
import AnswerResult from '@/components/AnswerResult.vue'
</script>
<template>
  <div class="my-8">
    <h1 class="text-h3 mb-8">Answers</h1>
    <answer-result />
  </div>
</template>
```

At this point, the client application already starts to closely resemble the result we're going for:
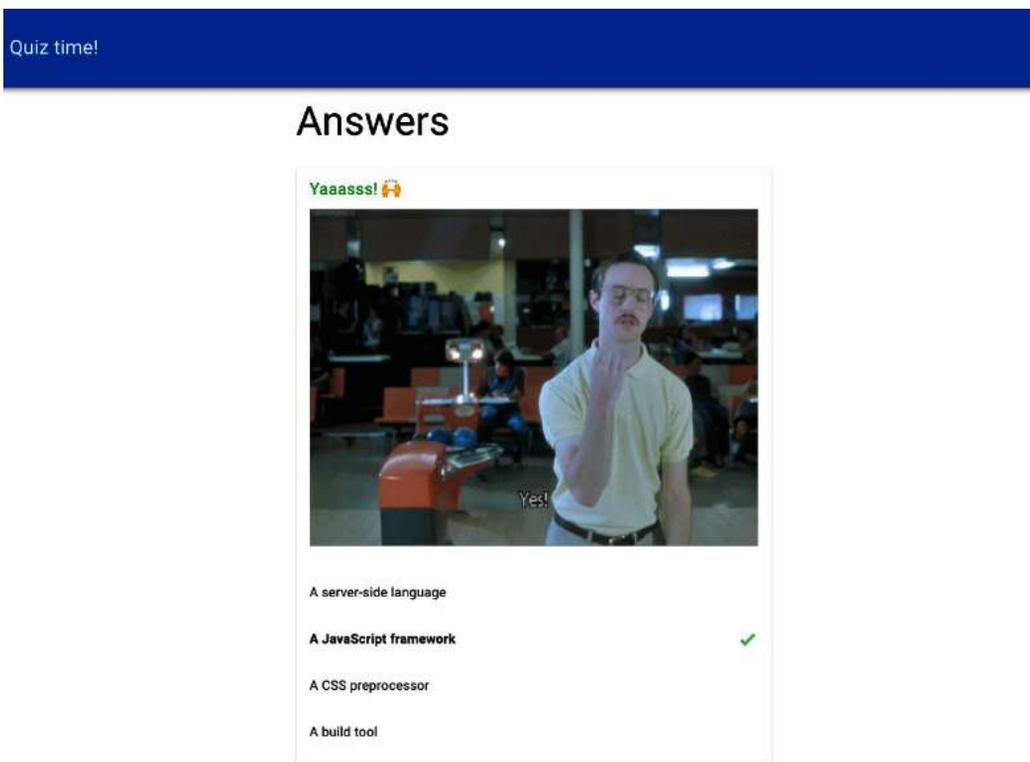
Figure 8.9 – Example of the quiz's state when answered correctly!

At this point, our work is getting a bit repetitive, so let's finalize our app by showing intermediate results and the final score!

Keeping and showing the score

For our scoreboard, we'll create another component called `ScoreBoard.vue` in the components folder: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.18-ScoreBoard.vue`.

We'll add the component to the **ScoreBoard** view:

```ts
<script setup lang="ts">
import ScoreBoard from '@/components/ScoreBoard.vue';
</script>

<template>
  <div class="my-8">
    <h1 class="text-h3 mb-8">Scoreboard</h1>
    <score-board />
  </div>
</template>
```

For the final score, we want to do something extra fun for the winner. We'll install the *Vue Confetti Explosion* component (`https://github.com/valgeirb/vue-confetti-explosion`) by running the following command in the `./app` folder's terminal:

```
npm install vue-confetti-explosion
```

And for our grand finale, we'll make sure to include confetti for the quiz winner by creating the following contents in a component called `FinalScoreBoard.vue`: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/08.quiz/.notes/8.19-FinalScoreBoard.vue`.

The important bit here is that we identify which user is the winner and shower that user with confetti (*lines 6, 22–24, 53–55*)!

The `FinalScoreBoard` component also needs to be added to its corresponding view file (`./src/views/FinalScore.vue`):

```ts
<script setup lang="ts">
import FinalScoreBoard from '@/components/FinalScoreBoard.vue';
</script>
<template>
  <div class="my-8">
    <h1 class="text-h3 mb-8">Scoreboard</h1>
    <final-score-board />
  </div>
</template>
```

And with that, our app is complete! You should now be able to complete the quiz with one or more players at the same time:
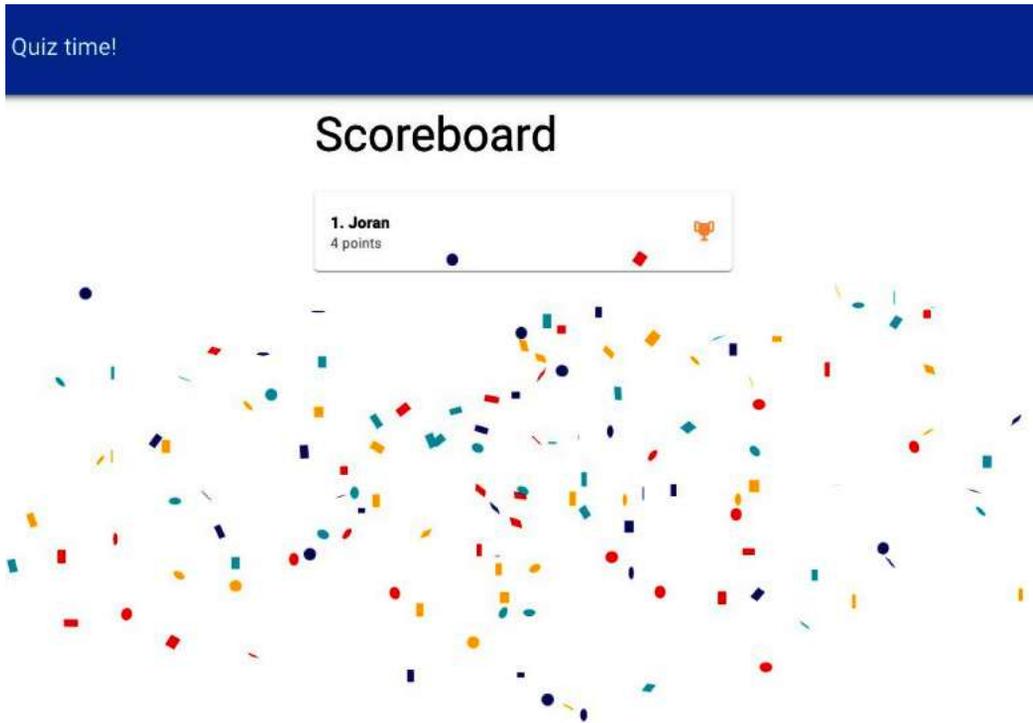
Figure 8.10 – There can be only one winner in this game

I think it's brilliant that our client apps have almost no state worth mentioning. All of it is handled by the central socket server!

Summary

This was quite a lengthy chapter that covered three different apps. I hope the distinct roles of the individual apps are clear. I also hope you appreciate the differences between a socket server setup and a RESTful server. We've used both in this chapter, each with its strengths.

Also, with the amount of code we've written in this chapter, it should be clear that this code is not at the level of sturdiness nor as secure as you would realistically want it to be when it's production-ready. I want to stress that this wasn't the focus of this chapter!

One of the new concepts that was introduced in this chapter was Nuxt. As you may have noticed, it has very powerful capabilities that enhance both the final product and the developer experience. You could consider Nuxt a default extension of any Vue application. I can get behind the philosophy of making it easy to do things right while making it hard to do things wrong. The opinionated setup that Nuxt encourages makes it easy to get started with.

The socket server is sort of an odd one between all of our projects. But as we can see from the implementation on the client side, its real-time updates fit very well with the reactivity model of Vue applications.

As a closing remark, we've also made some small quality-of-life improvements in our development workflow by creating scripts that automate repetitive tasks for us. You can consider it as a sort of stepping stone toward workspaces or monorepo setups, which expand even further upon managing projects that depend on each other.

In the next chapter, we'll scope down on the number of applications and build something fun too by using artificial intelligence and object recognition in a Vue app!

9

Experimental Object Recognition with TensorFlow

It's time for something a bit more experimental. As we've seen, **artificial intelligence** (**AI**) offers lots of new opportunities to explore when writing code assisted by AI as well as building solutions that are powered by AI. In this chapter, we'll take a look at **TensorFlow**. Google developed and published TensorFlow under an open source license. It enables developers to use and train machine learning models for different sorts of applications. You can find a curated list of demos on the TensorFlow website: `https://www.tensorflow.org/js/demos`.

We're going to apply a small part of the libraries that Google has published by leveraging the default published model for object recognition.

First, we'll build a small example prototype to discover some of the capabilities. Then, we'll apply our newly acquired knowledge to build something experimental and fun. It's another game, where you have to track down real-life objects using the camera in your browser!

In this chapter, we'll cover the following topics:

- Prototyping a concept to identify capabilities and limitations
- Leveraging multiple external APIs to build a multimedia app
- Using the browser's native **Camera**, **Text to Speech**, and **Media Stream** APIs

The example we'll be building has touch points with previous chapters and offers some potential opportunities for you to customize the application for your personal use case. I challenge you to create something unique here, based on the final code solution – maybe even a native app using what you learned about Quasar in *Chapter 7*!

Technical requirements

We'll build the main app on the **Vuetify** framework (`https://vuetifyjs.com/en/`) and **Pinia** (`https://pinia.vuejs.org/`) to manage the state. As stated previously, we'll leverage various **TensorFlow** libraries (`https://www.tensorflow.org/js/`) to incorporate some intelligence into our app.

You can find the complete code for this chapter here: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/tree/main/09.tensorflow`.

Let's get started with a prototype app!

Introduction to TensorFlow

When I need to research a new framework or technique, I find it very helpful to create a small application for it so that I can test it in complete isolation. We're going to apply the same approach with TensorFlow. The original idea is that we create an app using the object recognition library (`https://github.com/tensorflow/tfjs-models/tree/master/coco-ssd`) and apply the model to images from the camera on our device.

Setting up the project

Let's use a familiar framework to quickly build some boilerplate for our new project. We'll use the Vuetify CLI to create a new project for us:

1. Run `npm create vuetify@3.0.0` in the command-line interface.
2. Choose `vue-tensorflow` as the project's name.
3. Use the `Essentials (Vuetify, VueRouter, Pinia)` installation.
4. Select `TypeScript` using the arrow keys.
5. Select `npm` to install the dependencies.

If you navigate to the new project folder, you can run the local development server with `npm run dev`. The result should look very familiar to us since we've done this a few times now (see *Chapter 5*, *Figure 5.1*).

Next, we'll install the dependencies for using TensorFlow. The first two dependencies we'll install will help us in sourcing the CPU and WebGL to help with calculations in the algorithm. From the terminal, run the following command:

```
npm install @tensorflow/tfjs-backend-cpu @tensorflow/tfjs-backend-webgl
```

We'll use a pretrained model to help us with object recognition. **Coco SSD** (`https://github.com/tensorflow/tfjs-models/tree/master/coco-ssd`) can be used to identify multiple objects in a single image. We can install the model as a dependency of our project by running the following command:

```
npm install @tensorflow-models/coco-ssd
```

That's all we need for now!

> **Note**
>
> One of the limitations we'll be running into is that a pretrained model is trained to recognize a limited set of classes (classes refer to a classification of an object in a category). We only have access to some 80 different classes. We're going to have to work with this limitation.

To prepare the object recognition for future developments, we'll create a store to wrap the features. Since we selected Pinia during installation, an empty store has been initialized on the project. We'll create a new file called `objects.ts` in the `./store` folder: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/09.tensorflow/.notes/9.1-object.ts`.

We set some properties to track the status of the model. Bear in mind that it can take some time for a model to load, so we have to make sure that we inform the user so that they have a decent user experience. On store initialization, we must immediately call the `loadModel()` function, which loads the model on the store (*lines 14, 32–39*), for easy access throughout the app.

We've also added and exposed a `detect` function (*lines 22–30*). The function takes in an image and runs the image through the model. The result is an array of detected items with a certainty per item.

For now, this is enough for us to start working on an implementation. Now, let's build an interface for our prototype.

Performing and displaying a status check

It would be very valuable to see what the app is doing, especially since the first load of the model can take some time. We'll build a nice visual component to list the status of loading the model. Let's create a component called `StatusCheck.vue` in the `./components` folder:

```
<template>
  <v-list>
    <v-list-subheader>Status</v-list-subheader>
    <v-list-item>
      <v-list-item-title
        >AI Model
        <span v-if="isModelLoading">Loading...
```

```
            <v-progress-circular indeterminate :size="16"
color="primary" />
          </span>
      </v-list-item-title>
      <v-list-item-subtitle v-if="isModelLoaded">Loaded!</v-list-item-
subtitle>
      <template v-slot:append v-if="isModelLoaded">
        <v-icon icon="mdi-check" color="success"></v-icon>
      </template>
    </v-list-item>
  </v-list>
</template>

<script setup lang="ts">
import { watch } from "vue";
import { useObjectStore } from "@/store/object";
import { storeToRefs } from "pinia";

const objectStore = useObjectStore();
const { isModelLoading, isModelLoaded } = storeToRefs(objectStore);

const emit = defineEmits(["model-loaded"]);
watch(isModelLoaded, () => {
  if (isModelLoaded.value) emit("model-loaded");
});
</script>
```

This component is simply listing the status from the store in a nicely formatted way. It also emits the model-loaded event when the model is loaded so that we can pick up on the event. Let's have the model loading status show up in our app. We can delete the HelloWorld.vue file from the ./components folder and replace the contents of ./view/Home.vue with the following:

```
<template>
  <v-container>
    <StatusCheck />
  </v-container>
</template>

<script lang="ts" setup>
import StatusCheck from "@/components/StatusCheck.vue";
</script>
```

Now, we can run our app for the first time. You will notice that it takes a while to load at first, but after some time, you should see something similar to the following:
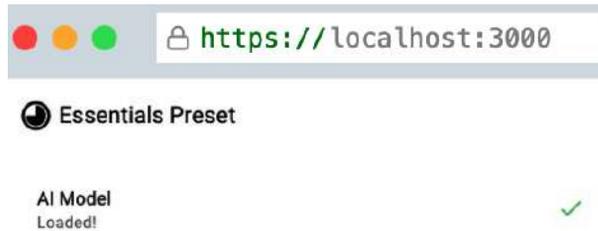
Figure 9.1 – Visualizing the status of the model

Now that our model has been loaded, we can use it! We'll build an image upload field and have the model analyze the contents of the image.

Selecting an image

We'll begin by creating a new component in the components folder. We'll call it ImageDetect. vue and start with the following contents:

```
<template>
  <v-container>
    <StatusCheckSimple @model-loaded="modelLoaded = true" />
    <v-file-input @change="inputFromFile" v-model="image"
accept="image/png, image/jpeg" :disabled="!modelLoaded" />
    <v-img :src="url" height="100"></v-img>
  </v-container>
</template>

<script setup lang="ts">
import { ref } from "vue";
import type { Ref } from "vue";

import StatusCheckSimple from "./StatusCheck.vue";

const image: Ref<File | any | undefined> = ref(undefined);
const imageToDetect: Ref<HTMLImageElement | undefined> =
ref(undefined);
const url: Ref<string | undefined> = ref(undefined);

import { useObjectStore } from "@/store/object";
import { storeToRefs } from "pinia";
const objectStore = useObjectStore();
```

```
const { detected } = storeToRefs(objectStore);

const modelLoaded: Ref<boolean> = ref(false);

const inputFromFile = (event: any): void => {
  const file = event.target.files[0];
  image.value = [file];
  imageToDetect.value = dataToImageData(file);
};

const dataToImageData = (dataBlob: Blob | MediaSource):
HTMLImageElement => {
  const objUrl = URL.createObjectURL(dataBlob);
  const img = new Image();
  img.onload = () => {
    URL.revokeObjectURL(img.src);
  };
  img.src = objUrl;
  url.value = objUrl;
  return img;
};
</script>
```

As shown in the template, we're moving some template logic to this file. We're using the `<StatusCheck` `/>` component with the `@model-loaded` event to determine whether the image detection controls should be visible or active.

In the scripts, we first set some of the variables we need to track the images that are being selected in the browser. Once the user changes the contents of the file, we can load the image in the browser's memory so that we can display it in the placeholder.

We'll go to `./views/Home.vue` and replace its contents to load this new component:

```
<template>
  <v-container>
    <ImageDetect />
  </v-container>
</template>

<script lang="ts" setup>
import ImageDetect from "@/components/ImageDetect.vue";
</script>
```

Now, we have a feature that provides images and we have a store that should be able to detect objects on images. Let's start to connect those by adding the store references to the `script` tag and adding a button to trigger the detection: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/09.tensorflow/.notes/9.2-ImageDetect.vue`.

As shown on *line 7*, we're ready to display detected objects.

> **Note**
>
> In terms of limitations, I mentioned that the model is capable of identifying several objects. The list can be found here: `https://github.com/tensorflow/tfjs-models/blob/master/coco-ssd/src/classes.ts`.
>
> I recommend trying this feature out with images of a person, or one of the listed classes.

After applying the detection, you should end up with something like this:
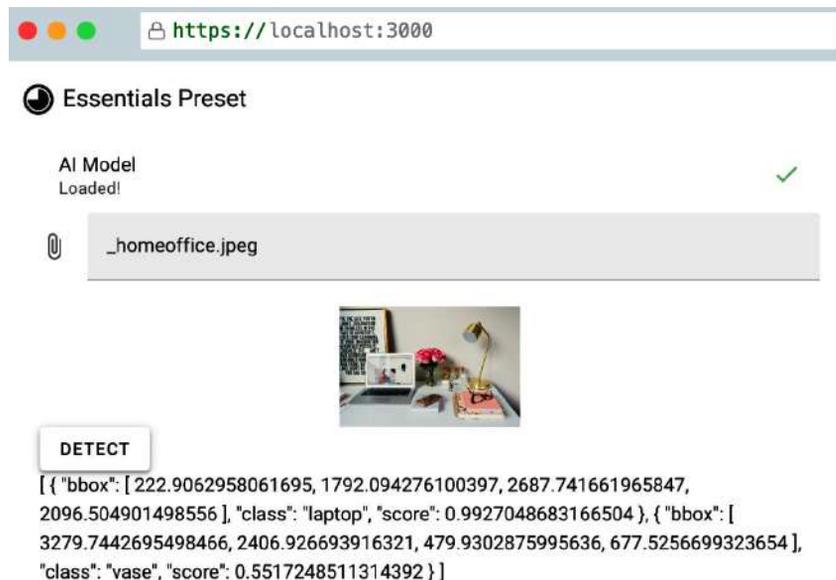
Figure 9.2 – Object recognition based on an uploaded image

This is already pretty interesting, but let's see if we can apply some more features. First, we'll look into formatting the results nicely: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/09.tensorflow/.notes/9.3-ImageDetect.vue`.

As shown on *lines 12–22*, we've added a nicely formatted list of detected items. We use the `roundNumber` function (*lines 18, 65–67*) to round the percentages.

Let's explore adding an additional feature and see if we can give our application a voice by exploring the Speech Synthesis API.

Adding a voice to the app

Since we're looking at non-traditional input for our app (using images rather than a mouse and keyboard), it's interesting to explore different ways of presenting information as well. Modern browsers have a built-in feature for converting **Text To Speech** (**TTS**) called **SpeechSynthesisUtterance** (`https://developer.mozilla.org/en-US/docs/Web/API/SpeechSynthesisUtterance`). Let's work on an addition for our prototype where we can explore this.

This API is pretty straightforward to set up. We will start by creating a new component called `TextToSpeech.vue` in the `./components` folder that will accept the text as a prop:

```
<template>
  <v-btn @click="tts" prepend-icon="mdi-microphone"
:disabled="isSpeaking">Speak</v-btn>
</template>
<script setup lang="ts">
import { ref } from "vue";
import type { Ref } from "vue";

const props = defineProps<{
  message: string;
}>();

const isSpeaking: Ref<boolean> = ref(false);

const tts = async () => {
  const { message } = props;
  const msg = new SpeechSynthesisUtterance();
  msg.text = message;
  msg.rate = 0.8;
  msg.pitch = 0.2;
  await window.speechSynthesis.speak(msg);
  msg.onstart = () => isSpeaking.value = true;
  msg.onend = () => isSpeaking.value = false;
};
</script>
```

In the `tts` function, we can see how we can access the API and send a message to speak. Since we want to disable the button while speech is active, we're keeping track of the `onstart` and `onend` callback functions and updating the `isSpeaking` variable accordingly. We're playing around a bit with the `rate` and `pitch` settings as well.

There are some more options we have when configuring `SpeechSynthesisUtterance` as we can read in the documentation. However, unfortunately, I've found that there are some limits. There are some mismatches between browsers and the support of certain languages is not very stable or usable. The `TextToSpeech.vue` component, however, should work in our application, so let's add speech to our app!

With the component stored, we'll add it to the template of `ImageDetect.vue` (don't forget to import the component!):

```
<template>
  <v-container>
    <!-- abbreviated -->
    <div v-if="detected">
      <v-list>
        <v-list-item v-for="(item, index) in detected" :key="index">
          <!-- abbreviated -->
        </v-list-item>
      </v-list>
      <TextToSpeech :message="speech" v-if="speech"></TextToSpeech>
    </div>
  </v-container>
</template>
<script setup lang="ts">
import { ref } from "vue";
import type { Ref } from "vue";

import StatusCheckSimple from "./StatusCheckSimple.vue";
import TextToSpeech from "./TextToSpeech.vue";

// ...abbreviated
</script>
```

As you can see from the template, we need to provide the component with `speech`. Let's have a look at the code: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/09.tensorflow/.notes/9.4-ImageDetect.vue`.

We've added a couple of helpers here. We want the speech to only name unique classes, so we're adding a computed variable called `uniqueObjects` that filters all duplicate entries (*lines 71–75*). The computed `speech` value (*lines 77–85*) takes in that list and joins them using the Intl API, which we also used in *Chapter 4*! The output is what we can send safely to the `<TextToSpeech />` component.

Try it out if you want! Our prototype is functional, which is all we need to be able to learn from it.

Learning from the prototype

So, with this micro app in place, we can experiment a bit. I was running into two major problems:

- Object recognition works, but it is very limited to the classes from the pretrained model. Providing a self-trained model should be possible, but it is a bit too complex to handle in the scope of this topic.

- The TTS capabilities between browsers are not very stable or reliable, especially between languages.

My initial idea was to create an app that would use the camera feed to point out objects that we could then learn to translate. With those two limitations, it's not going to be feasible to build. Luckily, we can still have some fun with the reliable features, without needing to modify the model.

Let's build a little game where we need to collect objects. We can have the existing classes list and prune it a bit so that it fits our use case. Let's go on a scavenger hunt!

Scavenge Hunter

In this section, we'll build a small app that can run on a web browser, preferably on a mobile phone. With *Scavenge Hunter*, the goal is to collect certain items from a list. We can use parts of the classes list to control the items our user needs to collect and in that case, we're sure to be able to detect those objects!

Once an object has been detected, we're going to add a score based on the find and certainty of the model. Since we can't guarantee that objects are being recognized properly, we should also be able to skip an assignment. Instead of uploading an image, we're going to use the camera stream!

Setting up the project

We can continue using the prototype we built or create a new project if we'd like. In the case of the latter, the dependencies and store are required, so we'd need to repeat the relevant steps provided in the *Setting up the project* and *Performing and displaying a status check* sections.

Let's see how we can turn the foundation of our prototype into a little game, shall we?

Generic changes

We're going to start with a configuration file. We need to create this file in the root of the project as `config.ts`:

```
export default Object.freeze({
    MOTIVATIONAL_QUOTES: [
        "Believe in yourself and keep coding!",
        "Every Vue project you complete gets you closer to victory!",
        "You're on the right track, keep it up!",
        "Stay focused and never give up!"
    ],
    DETECTION_ACCURACY_THRESHOLD: 0.70,
    SCORE_ACCURACY_MULTIPLIER: 1.10, // input scores are between
DETECTION_ACCURACY_THRESHOLD and 1
    MAX_ROUNDS: 10,
    SCORE_FOUND: 100,
    SCORE_SKIP: -150,
})
```

It can be very helpful to have this sort of configuration files in a central place so that we don't have to spend time hunting settings down in individual files. Feel free to modify the game configuration values in the `config.ts` file!

Let's also open the `./index.html` template so that we can update the title tag to the new project's name – that is, *Scavenge Hunter*.

We'll also create two new view files in the `./views` folder. It's okay to just paste some placeholder content here, like so:

```
<template>
  <div>NAME OF THE VIEW</div>
</template>
```

We need a view for the finding state, called `Find.vue`, and one for the end of a game, called `End.vue`. We'll add the contents later, in the *Building the finish screen* and *Skipping to the end* sections. With the views in place, we can update the `./router/index.ts` file with the following contents: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/09.tensorflow/.notes/9.5-index.ts`.

We're also going to simplify the interface a bit more. In the `./layouts/default` folder, delete the `AppBar.vue` and `View.vue` files. In the `Default.vue` file, replace its contents with the following:

```
<template>
  <v-app>
    <v-main>
      <router-view />
    </v-main>
  </v-app>
</template>
```

Now, we should be able to run the app, but there's not much new to do at the moment. Let's add some core features via Pinia stores.

Additional stores

I usually start by designing and setting up the stores since they usually act as a central source of information and methods. First, we're going to replace the contents of the `./store/app.ts` file with contents that are very similar to those from *Chapter 6*: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/09.tensorflow/.notes/9.6-app.ts`.

It's a trimmed-down version of the app store we used to build our fitness tracker, but we've removed all the unnecessary features.

Since we're dealing with a predefined list of classes, we're going to add those to the `object.ts` store as an additional value:

```
// ...abbreviated

export const useObjectStore = defineStore('object', () => {
    // ...abbreviated

    const loadModel = async () => {
        // ...abbreviated
    }
    loadModel();


    // Full list of available classes listed as displayName on the
following link:
    // https://raw.githubusercontent.com/tensorflow/tfjs-models/
master/coco-ssd/src/classes.ts
    const objects: string[] = ["person", "backpack", "umbrella",
"handbag", "tie", "suitcase", "sports ball", "bottle", "wine glass",
"cup", "fork", "knife", "spoon", "bowl", "banana", "apple", "orange",
"broccoli", "carrot", "chair", "couch", "potted plant", "bed", "dining
```

```
table", "toilet", "tv", "laptop", "remote", "cell phone", "microwave",
"oven", "sink", "refrigerator", "book", "clock", "vase", "scissors",
"teddy bear", "hair drier", "toothbrush"];


    return { loadModel, isModelLoading, isModelLoaded, detected,
detect, objects }
})
```

I've not added all of the categories and instead selected the classes that we could find in someone's home. You can change this to what you think is reasonable to have on hand (especially for testing purposes).

Let's also introduce some game mechanics by adding a `./store/game.ts` store file: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/09.tensorflow/.notes/9.7-game.ts`.

This store contains references to the rounds that are being played and which are being skipped (*lines 19–23*), keeps track of the score (*line 23*), and helps in selecting a category from the list of objects we've defined in the `object` store. In particular, `getNewCategory` (*lines 28–45*) is interesting since it pulls a randomized category from the `objects` collection while making sure it's always a unique new category.

As a final step in this section, we'll replace the contents of the `./App.vue` file: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/09.tensorflow/.notes/9.8-App.vue`.

This connects the app store's capabilities to the interface. Now, we can continue building up our little game!

Starting a new game

We'll start by creating a button that triggers the conditions for a new game. In the `components` folder, we'll create a `StartGame.vue` component, which is nothing more than a button with some actions on it:

```
<template>
  <v-btn
    :disabled="!canStart"
    @click="newGame"
    prepend-icon="mdi-trophy"
    append-icon="mdi-trophy"
    size="x-large"
    color="primary"
    ><slot>Start game!</slot></v-btn
  >
</template>
```

```
<script lang="ts" setup>
import { useAppStore } from "@/store/app";
import { useGameStore } from "@/store/game";
import { storeToRefs } from "pinia";
const gameStore = useGameStore();
const appStore = useAppStore();
const { canStart } = storeToRefs(gameStore);
const { reset } = gameStore;

const newGame = () => {
  reset();
  appStore.navigateToPage("/find");
};
</script>
```

As you can see, we're relying on the store to tell the button whether the button should be disabled. We trigger a new game by calling the `reset()` function of `gameStore` and calling a `navigateToPage` function on `appStore`. Now, we should be able to place this button component on the `Home.vue` view. Let's update that view completely with the following contents:

```
<template>
  <v-card class="pa-4">
    <v-card-title>
      <h1 class="text-h3 text-md-h2 text-wrap">z Scavenge Hunter</h1>
    </v-card-title>
    <v-card-text>
      <p>Welcome to "Scavenge Hunter"! The game where you find
things!</p>
    </v-card-text>
    <StatusCheck />
    <v-card-actions class="justify-center">
      <StartGame />
    </v-card-actions>
  </v-card>
</template>

<script lang="ts" setup>
import StartGame from "@/components/StartGame.vue";
import StatusCheck from "@/components/StatusCheck.vue";
</script>
```

If you're running the app now, you'll notice that it's impossible to start the game. Since we want to use the user's camera feed, we need to request access. We're going to expand the `StatusCheck. vue` file to also make sure we have access to a camera. We can use a composable from the `VueUse` library for this. So, from the terminal, let's install the `VueUse` package with the following command:

```
npm i @vueuse/core
```

With this dependency, we can update the `StatusCheck.vue` file. The changes to that component are quite extensive, so use the source from `https://github.com/PacktPublishing/ Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/09. tensorflow/.notes/9.9-StatusCheck.vue`.

Apart from some additional formatting on our model loading status and some template changes that show the actual status, most changes take place in the script. The `usePermission` composable returns a reactive property that lets us know if the user has granted access to use the camera. If both the model is loaded and the user has granted camera access, the game can start (*lines 61–65*). As you can see, we're using the `watch` function on multiple values by providing them as arrays (line 61) to the `watch` function.

In the `onMounted` hook (*lines 67–81*), we manually attempt to request a video stream. Once the stream starts, we immediately close it down since we don't need the stream, just the permission. The permission is persistent throughout our visit.

Building the finish screen

Before we dive into the image streams and object-hunting aspects, we'll build the final screen. We'll create a component in the `./components` folder to display the result of a game called `ScoreCard. vue`: `https://github.com/PacktPublishing/Building-Real-world-Web- Applications-with-Vue.js-3/blob/main/09.tensorflow/.notes/9.10- ScoreCard.vue`.

In the component, we're just displaying some of the metrics that were being collected on playthrough. They are all properties that are part of `gameStore`, so we have easy access to them.

In `End.vue`, we'll import the `ScoreCard.vue` file and make some additions to the template:

```
<template>
  <v-card class="pa-4">
    <v-card-title>
      <h1 class="text-h3 text-md-h2 text-wrap">It's over!</h1>
    </v-card-title>
    <v-card-text>
      <p>Let's see how you did!</p>
    </v-card-text>
    <ScoreCard />
```

```
    <v-card-actions class="justify-center">
      <StartGame>Play Again?</StartGame>
    </v-card-actions>
  </v-card>
</template>

<script lang="ts" setup>
import ScoreCard from "@/components/ScoreCard.vue";
import StartGame from "@/components/StartGame.vue";
</script>
```

There's not much going on here apart from the `<StartGame />` component, which we have reused to simply trigger a new game. That's how you use slots! Now, we can work on the middle section!

Skipping to the end

First, let's make sure we can complete a (very limited) flow by skipping all assignments. We're going to implement the basic game flow in the `./views/Find.vue` file. Let's take a look at the `script` tag since we have a lot going on in this file: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/09.tensorflow/.notes/9.11-Find(script).vue`.

At the top of the `script` tag, we're loading the properties and methods from the stores (*lines 3–15*). We use `appStore` to navigate to different pages and `gameStore` because that contains information about the progress of the current game.

We have some computed values that help in presenting and formatting data nicely. `currentRound` (*lines 17–19*) displays the progress of the game. We use `isPlaying` (*lines 21–23*) to determine the boundaries of the rounds versus the maximum set of rounds. Lastly, we have some fun randomized motivational quotes (*lines 25–29*) that we've loaded from our configuration file.

There are two methods in this component. One is to `skip` (*lines 31–39*) a round. The `skip` function tracks the number of rounds skipped (*line 32*) and modifies the player's `score` (*lines 33–37*). We must make sure the score doesn't fall below `0`. After skipping, we call the `newRound` method.

The `newRound` function (*lines 41–47*) tracks what should happen: either the number of rounds has reached the maximum and we should navigate to the `End` state, or we should load a new category using the `getCategory` function from the store. To ensure we get started when we enter this `Find` state, we will call that `newRound` function in the `onMounted` hook.

Next, let's look at the template of the `Find.vue` file, where we connect the computed values and methods to a basic interface: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/09.tensorflow/.notes/9.12-Find(template).vue`.

Again, there's not much special going on here. We're using the `<SkipRound />` component with the `@skipped` event to make sure we can move forward in rounds, regardless of whether we've been able to use object recognition.

Running the app at this stage should give us a result similar to the following:
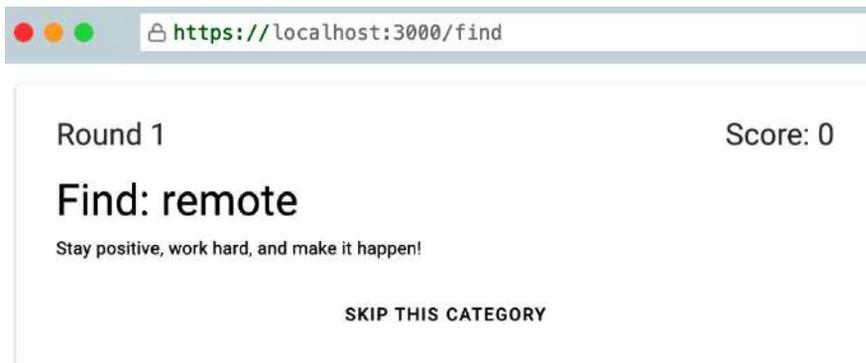
Figure 9.3 – The basic game flow in place

You should be able to complete the entire flow now by skipping all of the rounds. A game like this makes more sense on a mobile device than a laptop or personal computer, so this would be a good time to make sure we can test the app properly.

Testing on a mobile device

If you're building an app for a specific use case, it makes a lot of sense to test those cases as early as possible! While we can open the app in mobile views in our browser, it would make a lot of sense to run it on a mobile device as well. The first thing we can do is automatically expose the development server host by updating the dev script in the `package.json` file:

```json
{
    "scripts": {
    "dev": "vite --host",
    "build": "vue-tsc --noEmit && vite build",
    "preview": "vite preview",
    "lint": "eslint . --fix --ignore-path .gitignore"
  },
  "dependencies": {
    // ...abbreviated
  },
  "devDependencies": {
    // ...abbreviated
```

```
    }
  }
```

This change automatically serves the content through your local network, so as long as your mobile device and development server are on the same network, you can access the app via the network address:

Figure 9.4 – Exposing the development server to the network

We're not there yet, though. The media feed is only accessible over a secure connection. Going with **Vite's** recommendation in the official documentation (https://vitejs.dev/config/server-options.html#server-https), we'll install a plugin for this using the terminal:

```
npm install --save-dev @vitejs/plugin-basic-ssl
```

Once the installation is completed, we'll update the vite.confis.ts file so that it can use the plugin:

```
// Plugins
import vue from '@vitejs/plugin-vue'
import vuetify, { transformAssetUrls } from 'vite-plugin-vuetify'
import basicSsl from '@vitejs/plugin-basic-ssl'

// Utilities
import { defineConfig } from 'vite'
import { fileURLToPath, URL } from 'node:url'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [
    basicSsl(),
    vue({
      template: { transformAssetUrls }
    }),
    // ...abbreviated
  ],
  // ...abbreviated
})
```

After saving, we can restart the development server. The contents are now served over an HTTPS protocol. It is not using a signed certificate, so you will probably receive a warning from the browser upon first entry. You can now validate each step using your mobile device as well!

With that, we've built a basic flow from start to finish and we can test it on a mobile device. The game itself is not very interesting yet though, right? It's time to add some object recognition to the game!

Object recognition from the camera

This will be a change that involves a couple of steps. First, we'll introduce a component that can capture video from the browser. We'll create a `CameraDetect.vue` component in the `./components` folder: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/09.tensorflow/.notes/9.13-CameraDetect.vue`.

The code in the `CameraDetect.vue` component uses composables from the `@vueuse` package to interact with the browsers' `Devices` and `userMedia` APIs. We're using `useDevicesList` to list the available cameras (*lines 33–40*) and populate a `<v-select />` component (*lines 4–14*). This allows the user to switch between available cameras.

The user needs to manually activate a camera (also when switching between cameras) for security reasons. The button in the component toggles the camera stream (*lines 44–46*). To display the stream, we use `watchEffect` to pipe the stream into the `video` reference (*lines 48–50*). We can display the camera feed to the user by referencing the stream in the `<video />` HTML component (*line 20*).

Our stream is the replacement for the file upload of our prototype. We already have our store prepared to detect objects, so now, we'll connect the stream to the `detect` function.

Detecting and recognizing objects on a stream

One of the changes from our prototype is the way we provide images to the object recognition method. Using a stream means that we need to continuously process input, just as fast as the browser can.

Recognizing objects

Our `detect` method from `objectStore` needs to be able to determine if the recognized objects are the objects we are looking for. We'll add some capabilities to the function in the `object.ts` file:

```
    // ...abbreviated
    const detect = async (img: any, className?: string) => {
        try {
            detected.value = []
            const result = await cocoSsdModel.detect(img)
            const filter = className ? (item: DetectedObject) =>
 (item.score >= config.DETECTION_ACCURACY_THRESHOLD && item.class ===
```

```
className) : () => true
            detected.value = result.map((item: DetectedObject) =>
item).filter(filter).sort((a: DetectedObject, b: DetectedObject) =>
b.score - a.score)
        } catch (e) {
            // handle error if model is not loaded
        }
    };
    // ...abbreviated
```

Here, we're adding an optional parameter called `className`. If it's provided, we define a `filter` function. The filter is applied to the collection of recognized objects. If no `className` is provided, that filter function just defaults to returning `true`, which means it doesn't filter out any objects. We only do this to provide backward compatibility for the `<ImageDetect />` component.

> **Note**
>
> When working on existing code bases, you have to keep these sorts of compatibility issues in mind while developing. In our case, backward compatibility was needed for a prototype function, so it's not vital for our app. I'm highlighting this because, in large-scale applications with low test coverage, you may run into these solutions.

With our changes to the `object.ts` file, we can pass the stream to `objectStore`.

Detecting objects from the stream

We'll begin by passing the video stream's contents to our updated `detect` function from `objectStore`. We'll also include `gameStore` so that we can pass the current category as the `className` property. Let's add these lines to the `CameraDetect.vue` file to get ourselves set up:

```
import { ref, watchEffect, watch } from "vue";
// ...abbreviated
import { storeToRefs } from "pinia";

import { useObjectStore } from "@/store/object";
const objectStore = useObjectStore();
const { detected } = storeToRefs(objectStore);
const { detect } = objectStore;

import { useGameStore } from "@/store/game";
const gameStore = useGameStore();
const { currentCategory } = storeToRefs(gameStore);
// ...abbreviated
```

Don't forget about the `watch` hook that we import from Vue; we'll need it to monitor camera activity! Next, we'll add a function called `detectObject` to our scripts:

```
const detectObject = async (): Promise<void> => {
  if (!props.disabled) {
    await detect(video.value, currentCategory.value);
  }
  window.requestAnimationFrame(detectObject);
};
```

What's happening here? We've created a recursive function that continuously calls the `detect` method by passing the `video` and `currentCategory` values. To throttle the calls, we're using `window.requestAnimationFrame` (https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame). Normally, this API is meant to query the browser when animating: the browser will accept the callback function once it's ready to process it. This is perfect for our use case as well!

We can trigger the initial call as soon as the video is enabled. The `watch` hook we've imported can monitor the `enabled` variable and call the `detectObject` function once the video has been enabled:

```
watch(enabled, () => {
  if (enabled.value && video.value) {
    video.value.addEventListener("loadeddata", detectObject);
  }
});
```

Finally, once we've found a match, we need to signal this to our application. We'll add an `emit` event called `found` to trigger once the `detected` property has been populated with items:

```
const emit = defineEmits(["found"]);
watch(detected, () => {
  if (detected.value?.length > 0) {
    emit("found", detected.value[0]);
  }
});
```

We're returning the top match from the collection of `detected` items to the parent component.

> **Note**
>
> You can make testing easier by temporarily modifying the `objects` property in `objectsStore` so that it holds a couple of values of objects you have on hand, such as `person`. Later, you can restore the list to its previous state.

Using Vue's DevTools, you can test the app again. If you open the DevTools and navigate to the **Timeline** and **Component events** panels, once the camera has made a positive match, you will see continuous events being emitted (well, once for every animation frame):
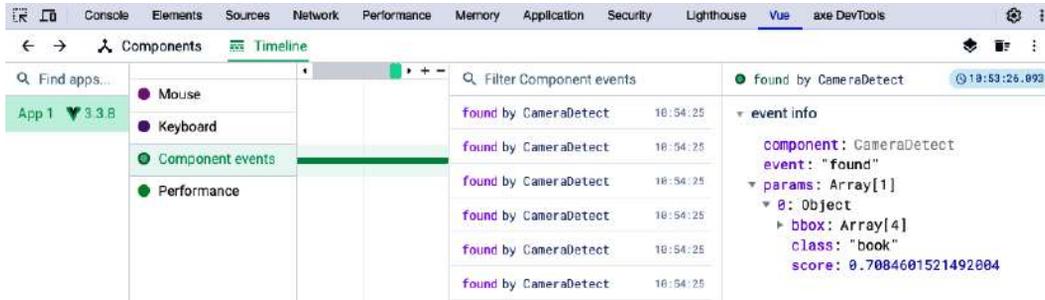
Figure 9.5 – Positive matches being emitted by the <CameraDetect /> component

We can now connect the emitted event to the Find state. So, let's move over to the ./views/Find.vue file so that we can pick up on the found event and pull it into our little game!

Connecting detection

If we open the Find.vue file, we can now add the event handler on the component to the template. We'll also provide a disable property to control the camera by changing the component line to the following:

```
<CameraDetect @found="found" :disabled="detectionDisabled" />
```

In the script block, we have to make some changes to both pick up on the found event and provide the value for the detectionDisabled property. Let's look at the new component code: https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/09.tensorflow/.notes/9.14-Find.vue.

We've added the detectionDisabled reactive variable (*line 51*) and are passing it down to the <CameraDetect /> component. In the existing skip function, we're setting the value of detectionDisabled to false (*line 68*). We're also adding the found function (*lines 78–86*), where we update the detectionDisabled value as well and process a new score by calculating the certainty of the recognized object (*lines 81–83*) and updating gameStore (*line 84*). Similar to the skip function, we call the newRound function to progress the game.

Once the newRound function has been called, we update the detectionDisabled variable and set it to true to continue detection.

This would be another good time to test the app. In this case, upon detection, you will rapidly progress through the rounds toward the end. If recognition seems unreliable, you can lower DETECTION_ ACCURACY_THRESHOLD in the ./config.ts file.

Wrapping up the game flow

Although the game is now functional, it's not playable since we're not giving enough feedback to the user. With appStore at our disposal, one of the easiest solutions is to use the dialog! Once we've incorporated that, our mini-game will be complete!

First, we'll update the CameraDetect.vue file by adding the reference to the dialogVisible reactive value. To do this, add the following to the script tag:

```
// ...abbreviated
import { useAppStore } from "@/store/app";
const appStore = useAppStore();
const { dialogVisible } = storeToRefs(appStore);
// ...abbreviated
```

Next, we'll use dialogVisible in our detectObject function to assess whether it should call the detect function from objectStore:

```
// ...abbreviated
const detectObject = async (): Promise<void> => {
  if (!props.disabled && !dialogVisible.value) {
    await detect(video.value, currentCategory.value);
  }
  window.requestAnimationFrame(detectObject);
};
// ...abbreviated
```

This doesn't affect our code yet since dialog has never been visible up until now. We'll remedy that by making some changes to the Find.vue file as well. To define the contents of the dialog, we'll add the following computed value to the script tag:

```
// ...abbreviated
const dialogEndLine = computed(() =>
  objectsFound.value + skips.value >= objectsLimit.value
    ? "You're done!"
    : "Get ready for the next round!"
);
// ...abbreviated
```

This returns a motivating line to display to the user. Feel free to modify this! The two functions that we'll change are called `found` and `skipped`. Let's have a look at the updated `found` function first:

```
const found = (e: { class: string; score: number }) => {
  detectionDisabled.value = true;
  objectsFound.value++;
  const newScore = Math.round(
    config.SCORE_FOUND * (e.score + 1) * config.SCORE_ACCURACY_
MULTIPLIER
  );
  score.value += newScore;
  newRound();
  appStore.showDialog(
    "Congratulations! 🎉",
    `<p>You've scored ${newScore} points by finding <strong>${e.
class}</strong>!</p><p>${dialogEndLine.value}</p>`
  );
};
```

As you can see, we're simply using the `appStore` method of `showDialog` to present a dialog to the user. The `<CameraDetect />` component is now able to detect when a dialog is visible and will stop detecting in the background. For the `skipped` function, we'll add the following:

```
const skipped = () => {
  detectionDisabled.value = true;
  skips.value++;
  if ((score.value + config.SCORE_SKIP) <= 0) {
    score.value = 0;
  } else {
    score.value += config.SCORE_SKIP;
  }

  newRound();
  appStore.showDialog(
    "Oh no! 😿",
    `<p>Skipping cost you ${-config.SCORE_SKIP} points!</p><p>${
      dialogEndLine.value
    }</p>`
  );
};
```

As you can see, these are very similar changes! Again, feel free to modify these contents to your liking as well.

Our game is now done! Hooray! We've now almost concluded our collection of applications. I think this game lends itself to being enriched with even more capabilities and more customization so that you can make it your own mini-game. From previous chapters, we've touched upon a lot of additional techniques and concepts you could apply or just get creative.

Summary

We started this chapter with a small prototype to experiment with a new sort of technology. Building something in an isolated environment helps you quickly understand how a certain technology can be adopted in an existing environment. As you've experienced, we were running into limitations that could not be resolved. In this case, it didn't matter that much, since we have few business requirements to deal with.

We also learned how to leverage existing and available APIs from the browser itself to build something unconventional. When putting together a portfolio, standing out with unique projects can make you stand out as a developer. Building little projects while combining multiple technologies can help you understand how you can compose applications with them. This is a more intensive approach but results in a much better understanding of technologies.

Feel free to spend some time customizing projects from previous chapters. In the final chapter, we are going to create a portfolio to host online. This will be the perfect showcase for everything you've achieved so far!

Part 4: Wrapping Up

The final part brings all the previous topics together. You will learn how to optimize Nuxt for a static site purpose and how to deploy to a web host. Then, we will look into automating workflows such as the deployment process. This section gives you the freedom to customize the output for your personal use and connects all previous chapters into a presentable portfolio.

This part has the following chapter:

- *Chapter 10*, *Building a Portfolio with Nuxt.js and Storyblok*

10
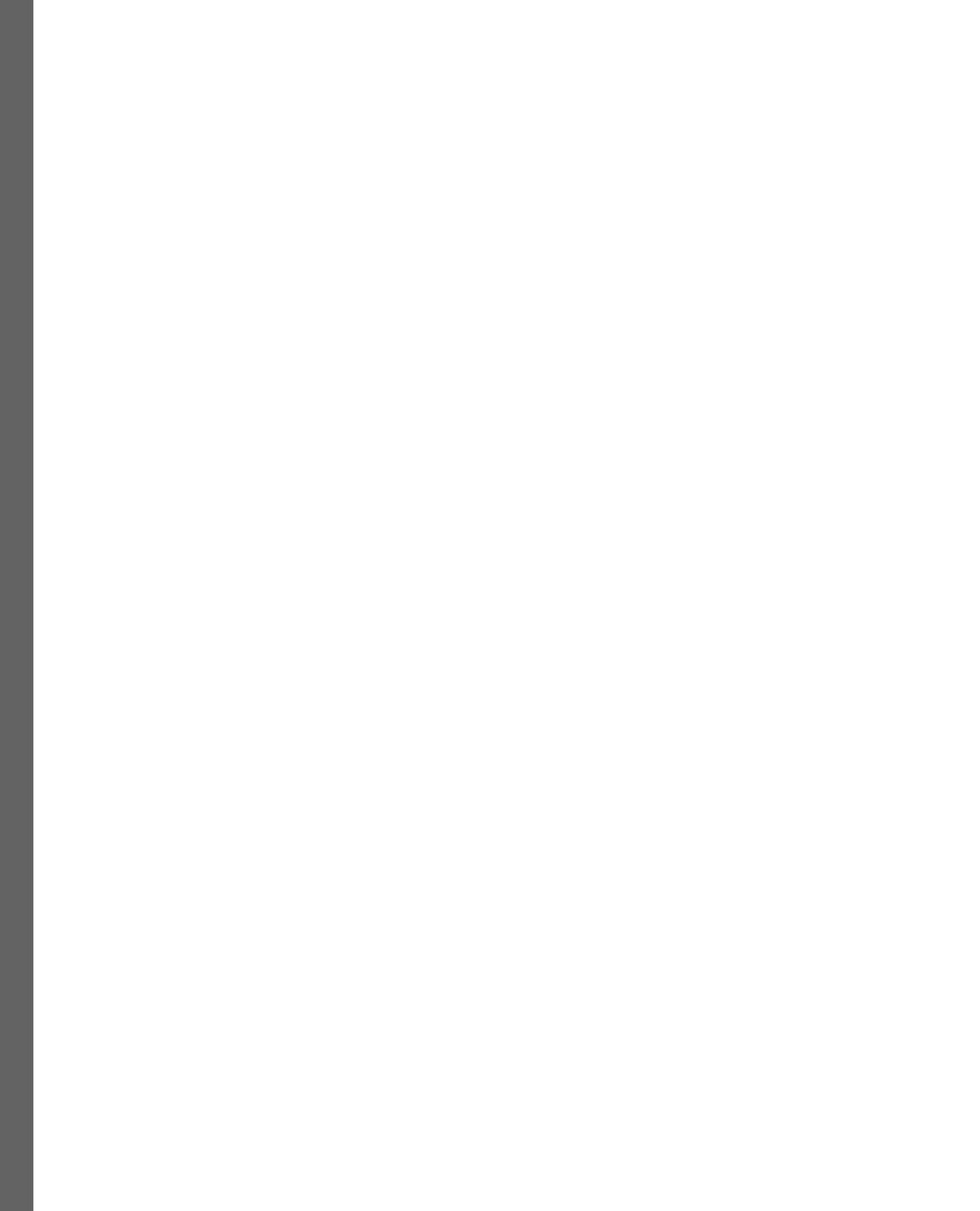
Creating a Portfolio with Nuxt.js and Storyblok

If you've been checking off the topics from the to-do list of applications from *Chapter 1*, you may have noticed that we've reached the final chapter. To celebrate our achievements, we'll create a portfolio where we can showcase our completed projects from the past, with the flexibility to add future projects as well. We'll also look into deploying the portfolio to an online space via an automated process.

We'll build the portfolio using Nuxt (`https://nuxt.com/`), which will greatly speed up our development process. The content will be stored in a Storyblok (`https://www.storyblok.com/`) space. For publication, we'll make use of Netlify (`https://www.netlify.com/`), which is a very developer-friendly platform for hosting modern web apps.

In this chapter, we'll cover the following topics:

- Refresher on using Nuxt and using Nuxt as a static site renderer
- Learning to apply a headless CMS to organize and manage content
- Connecting Nuxt and Storyblok using existing integrations
- Applying proven patterns to optimize a website
- Automating deployment to a public host

While we're going to focus on building the essentials of a portfolio website, you should be able to make modifications to further personalize the final product so that it can continue serving as your personal portfolio website to showcase your talents.

> **Note**
> Parts of the project in this chapter are based on a guide published by Storyblok: `https://www.storyblok.com/tp/add-a-headless-CMS-to-nuxt-3-in-5-minutes`.

Technical requirements

Like in *Chapter 8*, we'll use Nuxt (`https://nuxt.com/`) as the framework to build our portfolio website. For our styling and interactions, we'll use a UI library that is part of the Nuxt ecosystem: Nuxt UI (`https://ui.nuxt.com/`). We need to run our local development in SSL mode, for which we will use `mkcert` (`https://github.com/FiloSottile/mkcert`) to generate a locally trusted development certificate.

Our content will be managed and stored using Storyblok (`https://www.storyblok.com/`), which offers an excellent headless **Content Management System** (**CMS**) solution with a free tier we can use. A headless CMS is a system that focuses primarily on the content and aims to separate the content from the presentation. In our case, our presentation is handled by Nuxt, but it could be any application that we grant access to the content. This separation of concerns is a concept that helps in building scalable applications. Once we have built our portfolio, we will use Netlify (`https://www.netlify.com/`) to publish our portfolio on a public URL.

When developing this project, it will help if you have some data prepared beforehand to prevent switching contexts during development. Let's go through previous completed projects. For each project we want to display, we prefer a couple of screenshots (*Screenshot.rocks* is an excellent browser plugin for creating styled screenshots from your window: `https://screenshot.rocks/`) and have descriptions prepared.

You can find the complete code for this chapter here: `https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/tree/main/10.portfolio`.

With several projects prepared, we can get started with the initial content setup!

Setting up Storyblok

Registering on the Storyblok website is pretty self-explanatory. After the initial registration, you will end up in a demo space. We will skip the demo (you can always revisit this as part of **Spaces**) and choose to **Create a new space**, where we will choose the appropriate name and server location:

Create your new space

A space is an area to manage and store content for a specific project

New space
Start from scratch

Demo space
Explore Storyblok

Name your new space *

Nuxt Portfolio

Server Location * ❓

EU ⌄

Create space
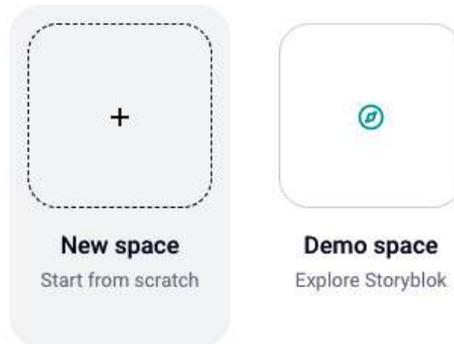
Figure 10.1 – Creating a new Storyblok space for our portfolio

After creation, you may encounter a modal referring to a trial period. You can select the free community plan directly or after the 14-day trial period.

The welcome screen will look like this:
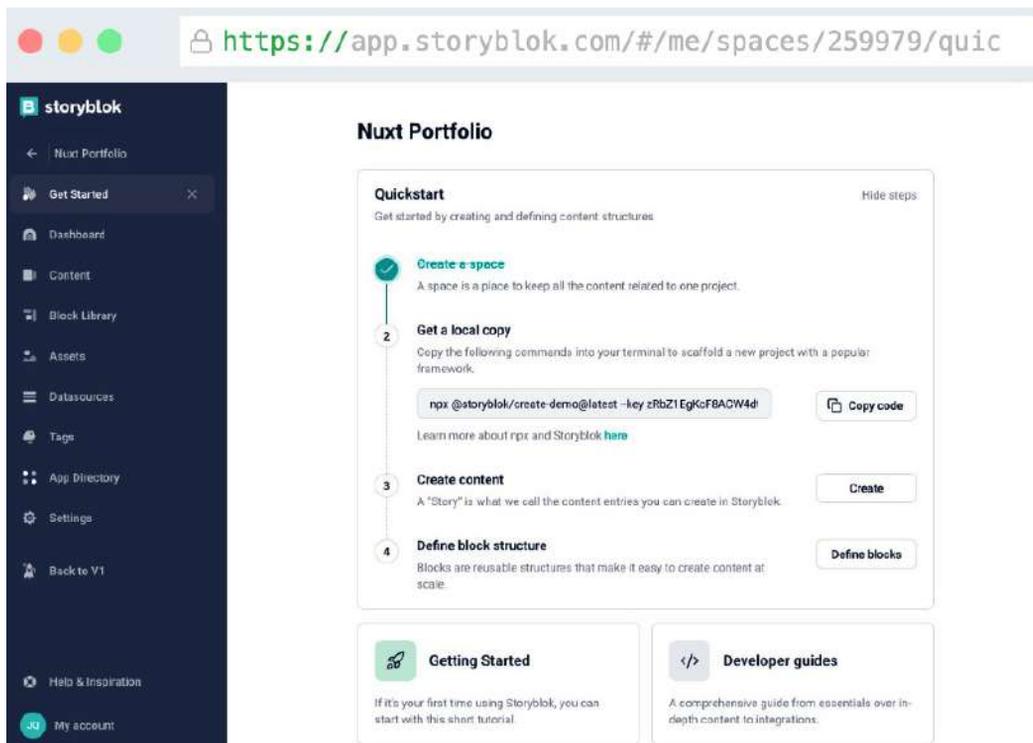
Figure 10.2 – The Storyblok dashboard

For the sake of our portfolio, we can close and ignore the **Get Started** section. We'll mainly focus on the **Settings**, **Content**, **Block Library**, and **Assets** sections.

If we open the **Content** section, we'll see an entry titled **Home**. Let's open it and take a look:
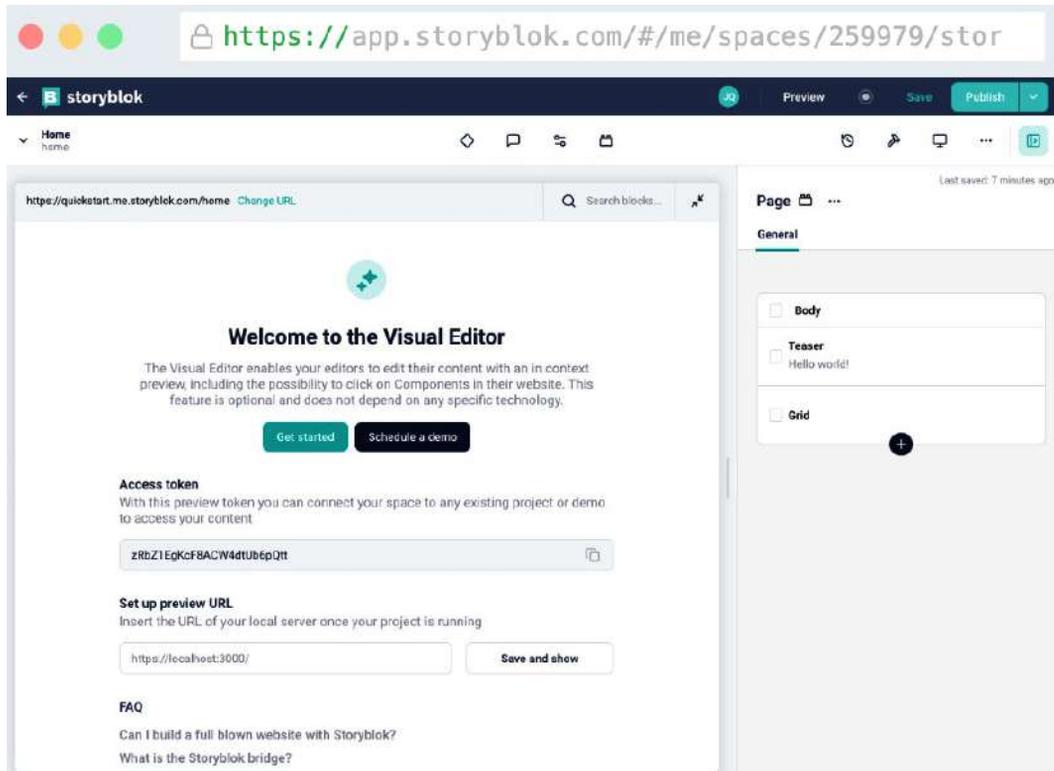
Figure 10.3 – The initial wizard on the Home entry

This screen lets us know that we can preview our content in what Storyblok refers to as an **in-context preview**. This means we can load our portfolio in the Storyblok environment to offer a very realistic preview of our portfolio!

Let's set this up to see what this means in practice. We will need (you may have guessed it by this point) the **Access token** value to connect the CMS to our application.

Initializing the Nuxt portfolio

We'll use the nuxi Nuxt CLI tool to create a new project, similar to what we did in *Chapter 8*:

```
npx nuxi@3.8.0 init portfolio
```

Again, we'll choose npm as our package manager. If you haven't installed the mkcert tool (https://github.com/FiloSottile/mkcert) to generate SSL certificates, you need to follow the appropriate installation instructions to proceed: https://github.com/FiloSottile/mkcert#installation.

Once the installation is done, we can generate a localhost certificate using the following command in the command-line interface:

```
mkcert localhost
```

The command should result in an output similar to the following:
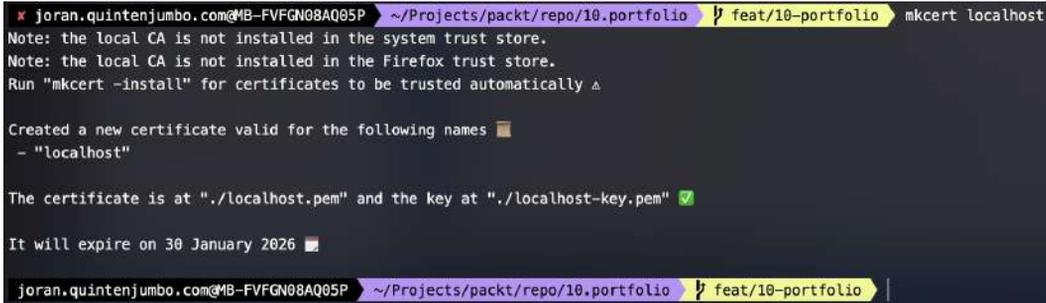
Figure 10.4 – Successfully generated SSL certificate

The next thing we need to do is update the development script in the package.json file so that we can run the Nuxt development servers in SSL mode. Find the following command:

```
"dev": "nuxt dev",
```

Replace it with the following line, which adds the certificate and points the process to a .env file:

```
"dev": "NODE_TLS_REJECT_UNAUTHORIZED=0 nuxt dev --dotenv .env --https
--ssl-cert localhost.pem --ssl-key localhost-key.pem",
```

We'll also create a .env file containing the following information (the **Access token** value can be found on the **Welcome** page in Storyblok):

```
NUXT_STORYBLOK_ACCESS_TOKEN=replace_with_your_access_token
```

If you have skipped the Storyblok **Welcome** page, you can review the token via the Storyblok menu. Navigate to **Settings**, then **Access tokens** – it should be listed there.

Now, we must run the npm run dev command from the terminal to start the development server. Note that you can access the server now via the HTTPS protocol!

On the Storyblok **Welcome** page, we can now fill in the **Set up preview URL** field so that it contains the local URL (this should be similar to the prefilled placeholder). Don't forget the trailing slash!

If you get an error in Storyblok, you may need to open the URL in your browser first as the browser might flag the local certificate as not private. If you accept the certificate in your browser, you can reload the Storyblok interface; it should show the Nuxt welcome page:
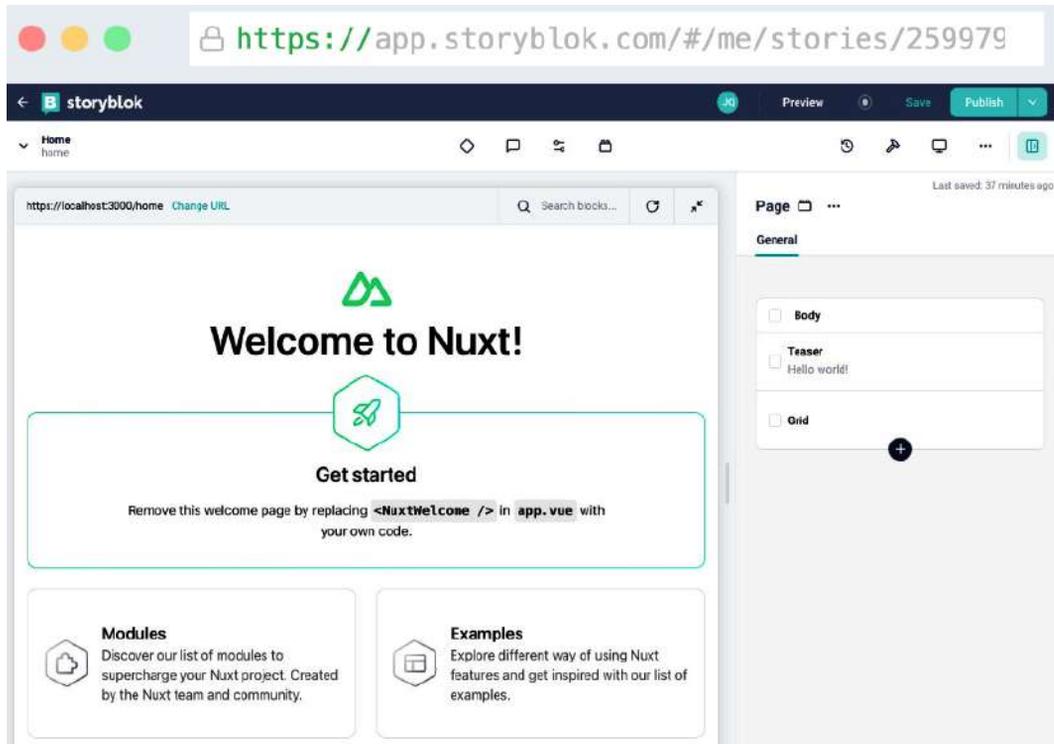
Figure 10.5 – The Nuxt welcome screen loaded in the Storyblok interface!

This is pretty neat! You're able to see a locally run preview from an online CMS interface! Nothing is editable yet, but we'll work on this next.

Installing Nuxt modules

For now, we can stop the Nuxt server to install some Nuxt modules we'll be using. In the terminal, use the following command to install the Storyblok module (https://github.com/storyblok/storyblok-nuxt):

```
npm i @storyblok/nuxt@5.7.4
```

The *Storyblok Nuxt* module is an SDK that has opinionated, and therefore low-code, integration with Nuxt. It supports auto-importing and can map Vue components directly to Storyblok entities with very little effort.

We need to add a little configuration before the *Storyblok Nuxt* module can function properly, so let's open the `nuxt.config.ts` file and modify it so that it has the following contents:

```
// https://nuxt.com/docs/api/configuration/nuxt-config
export default defineNuxtConfig({
  devtools: { enabled: true },
  modules: [
    ['@storyblok/nuxt', {
      accessToken: process.env.NUXT_STORYBLOK_ACCESS_TOKEN,
      apiOptions: {
          region: "eu"
        }
      }
    ]
  ],
})
```

If you're operating from a different region, be sure to pick the corresponding region.

While we're setting up the modules, we can add the UI library (`https://ui.nuxt.com/`) at this point too. We'll install the library from the terminal with the following command:

```
npm i @nuxt/ui@2.9.0
```

Once the installation is done, we need to register the module by adding the module's name to the `modules` property in `nuxt.config.ts`:

```
// https://nuxt.com/docs/api/configuration/nuxt-config
export default defineNuxtConfig({
  devtools: { enabled: true },
  modules: [
    ['@storyblok/nuxt', {
      accessToken: process.env.NUXT_STORYBLOK_ACCESS_TOKEN,
      apiOptions: {
        region: "eu"
      }
    }
    ],
    '@nuxt/ui'],
})
```

Before we start adding our own content to the CMS, let's test whether we can show the contents from Storyblok in our application.

First, we'll open the **Home** entry in the **Content** section. Using **Entry configuration**, we'll tell Storyblok that this page is published on the root of the domain, using the **Real path** property:

Figure 10.6 – Updating the Real path value for our home page

This change makes the contents of the **Home** page available on the main URL of our application. As mentioned previously, Storyblok offers a low-code SDK to integrate the CMS with our application. "Low-code" often means that we need to adopt certain patterns.

One of the things that the SDK does for us, for instance, is map the contents from the Block library to Vue components.

For this to work, we need to place those components in a `./storyblok` folder at the root of our project. For our test, we need to create two files. First, we'll start with `Page.vue` in the `./storyblok` folder:

```
<template>
    <div v-editable="blok">
       <StoryblokComponent v-for="blok in blok.body" :key="blok._uid"
:blok="blok" />
    </div>
  </template>

  <script setup lang="ts">
  defineProps({ blok: Object })
  </script>
```

Next, we'll create a `Teaser.vue` file in the same folder:

```
<template>
  <div v-editable="blok" v-if="blok">
    {{ blok.headline }}
  </div>
</template>

<script setup lang="ts">
defineProps({ blok: Object });
</script>
```

The `v-editable` directives are important to note in these files since they signal back to Storyblok that the contents of these blocks are indeed editable. We can now start creating a `./pages` folder in our application and temporarily create an `index.vue` file with the following contents:

```
<script setup>
const story = await useAsyncStoryblok('home')
</script>
```

```
<template>
  <StoryblokComponent v-if="story" :blok="story.content" />
</template>
```

Let's create a simple layout for our website. First, we'll place a `Header.vue` file in a `./components` folder: https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/10.portfolio/.notes/10.1-Header.vue.

We must also create another file called `Footer.vue` in the same folder: https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/10.portfolio/.notes/10.2-Footer.vue.

In both of these templates, we're using some Nuxt UI components, but other than that, nothing special is going on. We'll complete the base layout by creating a `default.vue` file in the `./layouts` folder:

```
<template>
  <div class="flex flex-col min-h-screen">
    <Header />
    <div class="flex-grow">
      <main class="container mx-auto">
        <slot />
      </main>
    </div>
    <Footer />
  </div>
</template>
```

With the page structure (albeit limited) in place, we can remove the `app.vue` file from our project's root. In the Storyblok interface, you will see a **Hello world!** message, replacing the Nuxt welcome view. Opening the development server in a browser window will yield the same result. Now, if you change the text in Storyblok and save the change, you will see it reflected in both the Storyblok interface as well as directly in the browser!
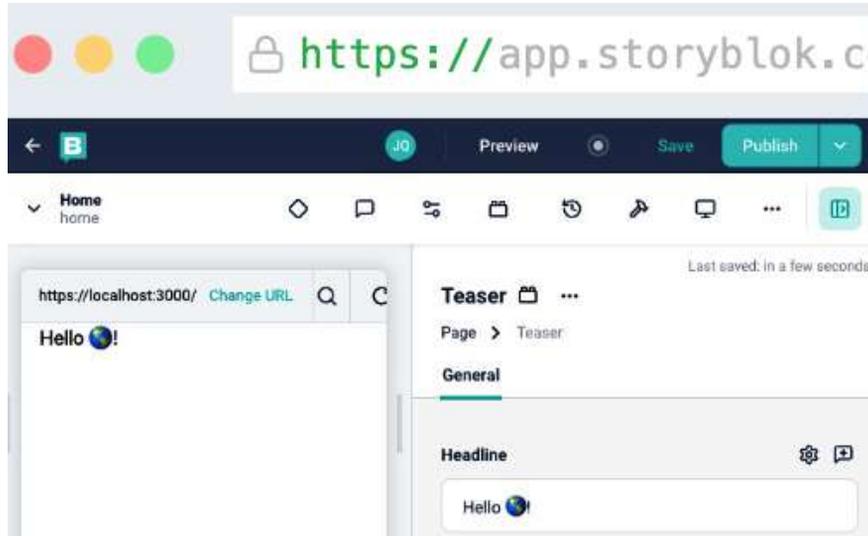
Figure 10.7 – Updating and saving the contents of the home page

Here, you can see how very little we needed to manually configure and set up to establish a connection! If we use the correct naming convention for our Storyblok components, the SDK can automatically populate the components with the content from Storyblok using the slug. On top of that, those components are editable in the Storyblok editor view.

Apart from automatically mapping contents to components, the SDK also helps in identifying the URL and providing the corresponding contents. In the next section, we'll add more content to the CMS so that we can start to explore this connection a bit better!

What we've done so far is set up pages. With Storyblok, you can easily create and modify pages using predetermined components. These are the **bloks**. Having a collection of reusable blocks allows editors to create a page and compose the contents by adding bloks and configuring them. Having the preview option available allows editors to see the changes before they're deployed, which is a valuable feature of Storyblok.

Working with multiple content types

To define the types and content to use on our pages, we can navigate to **Block library** in Storyblok. By default, you will see four existing blocks: **feature**, **grid**, **page**, and **teaser**. Between the blocks, there are two different types. **page** is of the **Content** type, while the rest are of the **Nestable** type. The key difference is that **Content type** is a top-level type of content. You can consider it a sort of page with a unique URL. You use this to hold different compositions of the lower level of content type – that is, **Nestable**. These are the building blocks of the pages: you can add them to a page to build the content you like.

There's also **Universal type**, which can act as either of the aforementioned types. We won't use it in our example, though.

Let's create a new block for our portfolio purposes by clicking on the **+ New Block** button. For the technical name, we'll have to fill in how the content will be identified as part of a JSON structure. We'll go with `portfolio` since lowercase is preferred in this case. This will be a **Content type** block, and we'll configure the type by clicking on the **Add Block** button. Next, we'll add the minimum fields we'll need to present a project in our portfolio. Let's go with the following settings:

1. Add a `title` field name of the **Text** type.

2. Add an `image` field name of the **Asset** type:

    I.   After creation, configure the field to only allow **Images**.

3. Add a `description` field name of the **Textarea** type:

    I.   After creation, configure the field to allow a maximum length of `150` characters.

4. Add a `body` field name of the **Richtext** type.

Now, we'll start to add some entries to the portfolio by navigating to the **Content** section of the Storyblok interface.

Configuring the portfolio

We want our portfolio to be published on a `/portfolio` path, where the slug is added to the URL. On the `/portfolio` path, eventually, we will want to show an overview of projects. In the **Content** section, we'll choose **+ Create new** to create a **Folder** parameter called `Portfolio`. The slug will be filled in automatically. Note the **Content type** field, where we can restrict what types of content can be shown as part of the folder. We'll keep the default settings for now, but we'll circle back to that option in a few steps. Click **Create** to add the folder to our content.

Because we want to present an overview of projects on the root section of the portfolio, we'll navigate to the folder and create a new entry, but this time, we'll choose **Story**. For the name, we'll choose `Home`, and we must check the **Define as root for the folder** checkbox when creating it. For the content type, we'll use the default, which is **Page**. Storyblok will attempt to open the page, which will result in either the server being unavailable if you haven't started it or a **404 not found** error since we have not defined a target for this route. That's okay – we'll solve that in the *Mapping content to code* section. We need to configure this page using **Entry configuration** and check the **Define as root for the folder** checkbox. This option removes the slug value, which is precisely what we will need.

Although we added a type page here, we want to restrict the rest of the content so that it exclusively has `Portfolio` entries. For that, we'll move to the root of the **Content** section in Storyblok and check the `Portfolio` folder entry so that we can edit the folder properties once more:

Figure 10.8 – Updating the settings for the Portfolio folder

This opens the same modal as when we created the folder. We'll use this to restrict further additions to the contents of the `Portfolio` folder by checking the **Restrict content types** option and selecting the **Portfolio** type:

Figure 10.9 – Restricting the content type for the Portfolio folder

This doesn't affect our previously created home page, but it will affect future entries. So, let's create a couple of portfolio entries. You can choose what you want to showcase – I recommend adding *at least two items* for the sake of this section. We will add more later to showcase the preview's capabilities!

Mapping content to code

First, we'll focus on regular pages since we have more than one to deal with now. We can remove the `./pages/index.vue` file completely and replace it with a file called `./pages/[...slug].vue`:

```ts
<script setup lang="ts">
const { slug } = useRoute().params as { slug: string[] };

const story = await useAsyncStoryblok(
  slug && slug.length > 0 ? slug.join("/") : "home", { version:
"draft" }
);
</script>

<template>
  <div>
    <StoryblokComponent v-if="story" :blok="story.content" />
  </div>
</template>
```

With this code in place, we should be able to render a simplistic page. Let's take a look at the home page for the **portfolio** section. If we open it in Storyblok, we'll see a blank page. In the **Content** panel on the right-hand side, we can add a new block – for instance, the **Teaser** block.

Let's drag the **Teaser** block in and give it a fitting title for a portfolio overview page:
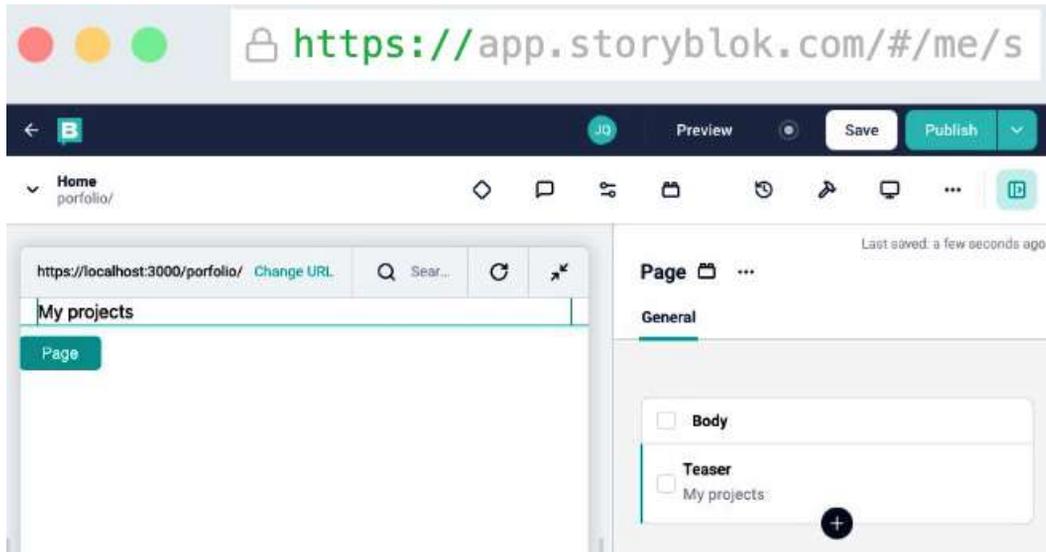
Figure 10.10 – Rendering a Teaser block on the page in preview mode

Again, the preview we're seeing is immediately visible when we access the development URL in the browser as well!

So, what's happening here? We've used the Nuxt capability of file-based routing to land our application on the `[...slug].vue` page. On this page, we're reading the `slug` route parameter to query the Storyblok content using the `useAsyncStoryblok` composable. In the template, we're relying on the `<StoryBlokComponent />` component that the Storyblok SDK exposes to dynamically load the corresponding components from the `./storyblok` folder to reflect items in the Block library from the CMS. Because of the auto-imports by the Nuxt framework, we have a very clean setup for our page!

You may have noticed that the `./storyblok` folder's contents do not match the number of blocks in the library. Let's quickly fix that: you will need to make sure these are always aligned to support the different scenarios.

We'll fix this with a minimal setup for the components. First, we'll create a `Feature.vue` component in the `./storyblok` folder with the following contents:

```
<script setup lang="ts">
defineProps({ blok: Object });
</script>

<template>
```

```
    <div v-editable="blok" v-if="blok">
      <h3>
        {{ blok.name }}
      </h3>
    </div>
</template>
```

Then, in the same folder, we'll create a `Grid.vue` file:

```
<script setup lang="ts">
defineProps({
  blok: {
    type: Object as () => { columns: any },
    required: true,
  },
});
</script>

<template>
  <div v-editable="blok" v-if="blok" class="flex mx-auto">
    <StoryblokComponent
      v-for="blok in blok.columns"
      :key="blok._uid"
      :blok="blok"
    />
  </div>
</template>
```

As you can see, this file is slightly different from a `Feature.vue` file since it can contain nested Storyblok components. We can simply feed those to `<StoryblokComponent />` to render the correct component for every possible level of recursiveness.

Presenting the portfolio section

Our **portfolio** section consists of roughly two cases: we want to present an overview of all portfolio items on the home page for the section and we want to present an individual case on its own page.

First, let's create the overview by creating a new block in **Block Library**. We'll call it `portfolio-all` and make it a nestable block. In the field editor, add the following fields:

1. Add a `title` field name of the **Text** type.

2. Add a `description` field name of the **Textarea** type:

    I. After creation, configure the field to allow a maximum length of `150` characters.

3.　Add a `body` field name of the **Richtext** type.

After saving, we can update the **portfolio** home page. We can remove the **Headline** block since we don't need it anymore. Drag in the new **Portfolio All** component and add some sensible text:
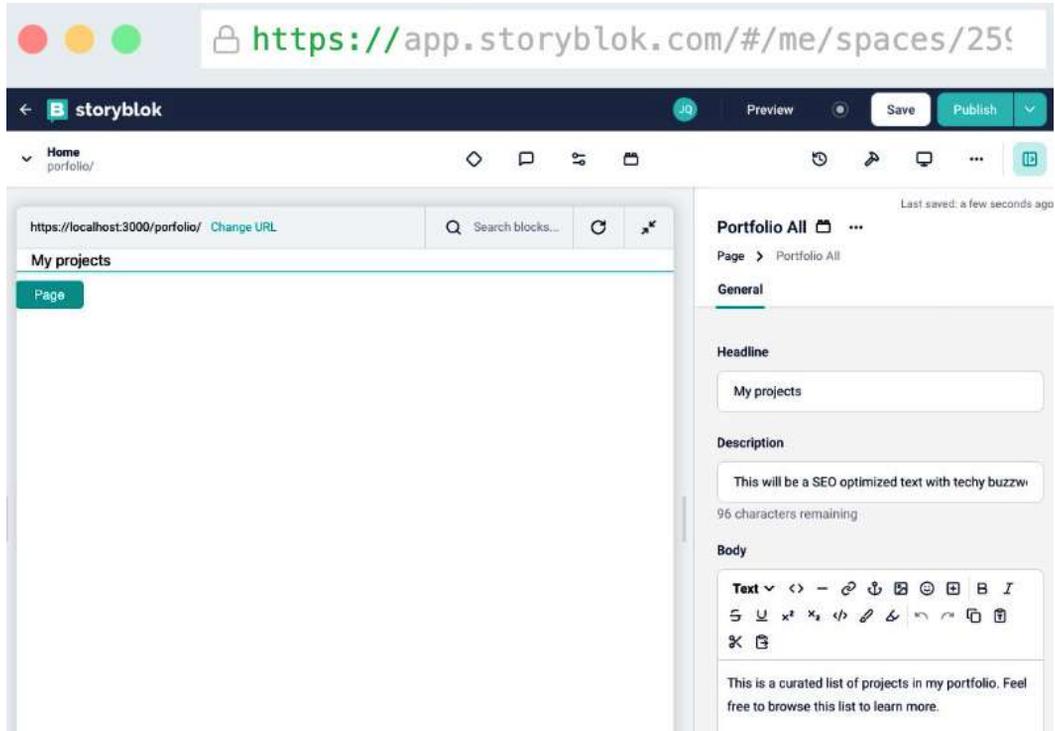
Figure 10.11 – Configuring the Portfolio All block

As we've learned, we need to make sure we have a counterpart component in our `./storyblok` folder. The naming convention follows camel case, so we need to create a `PortfolioAll.vue` component in the folder (https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/10.portfolio/.notes/10.3-PortfolioAll.vue):

```
<script setup lang="ts">
import type { Ref } from "vue";
import type { StoryblokProject, StoryBlok } from "@/types/storyblok";

const props = defineProps({
  blok: { type: Object as () => StoryBlok, required: true, },
});
```

```
const projects: Ref<StoryblokProject[] | null> = ref(null);

const storyblokApi = useStoryblokApi();
const { data } = await storyblokApi.get("cdn/stories", {
  version: "draft",
  starts_with: "portfolio",
  is_startpage: false,
});

const richTextBody = computed(() => renderRichText(props.blok. body));

projects.value = data.stories;
</script>
<template>
  <div>
    <h1 class="text-2xl mb-4 text-primary">
      {{ blok.headline }}
    </h1>
    <div v-html="richTextBody" class="py-4" />
    <div class="grid grid-cols-2 gap-4">
      <UCard
        v-for="project in projects"
```

The `<PortfolioAll />` component is a bit more complicated. While it has a lot in common with the `Page` component in the sense that it is getting props, we also manually request additional data using the `useStoryblokApi` composable (lines 11-16). We're getting all the stories on the portfolio path. `is_startpage` signals that we want to exclude the home page of the portfolio section. On receiving the data, we store it in the `projects` reference (lines 9 and 20) so that we can iterate over it in the template and we use Nuxt UI's `<ULink />` component to render a link to the project's page.

There's another addition: since we've defined a rich-text field for the *body* of the home page, we need to make sure it gets rendered properly. We can use the `renderRichText` function from Storyblok (*line 18*) as a computed value.

Let's build the portfolio detail page as well, to close this section for now. Again, we need to create the corresponding component in the `./storyblok` folder to have it automatically mapped to the URL. Since we're displaying the **Portfolio** block type, we'll create a `Portfolio.vue` component: https://github.com/PacktPublishing/Building-Real-world-Web-Applications-with-Vue.js-3/blob/main/10.portfolio/.notes/10.4-Portfolio.vue.

`Portfolio.vue /` is a pretty straightforward component. Because we've pre-defined the fields that are part of the **Portfolio** block, we can simply use the contents to render a common Vue template. In this case, the data is coming from the `./pages/[...slug.vue]` component, or more specifically, the `useAsyncStoryblok` composable, which feeds `<StoryblokComponent />`.

For a more visual representation, the following model helps explain how the components are rendered:
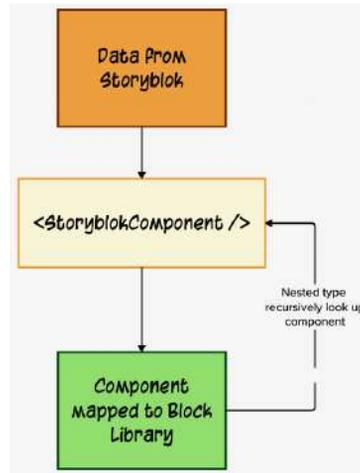
Figure 10.12 – From data in Storyblok to a component in the browser

When the data is coming from Storyblok, it bears a description of the component type. Using the slug page, we always land on a Storyblok component, which then attempts to match the type to the corresponding Vue component in the `./storyblok` folder. That component, in turn, could be a nested type, which means it contains another instance of the Storyblok component, and so on. Once you get the hang of this mental model, it's pretty straightforward. The amount of automation and assumption that Nuxt and the Storyblok module take care of can obfuscate the mechanics a bit.

With our portfolio in place, let's have a closer look at the content model and how we can better tune it to our needs.

Modifying the content model

Our headless CMS is now able to present basic layouts and contents to our portfolio. It's still very limited this way, so let's find out how we can work with the content model so that it supports new features.

Updating existing types

Let's assume that our **Teaser** block is a **Heading** block. I would recommend trying to keep blocks as static as possible, but let's see what we need to do to change a name. If we navigate to the **Block library** in CMS and hover over the **Teaser** block, we can enter edit mode with the context menu. In the **Config** tab, we have the option to update the name:
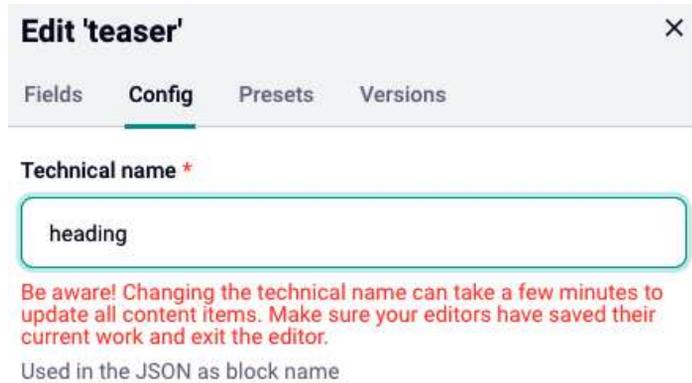
Figure 10.13 – Updating an existing block's name

The next step is simply renaming the `Teaser.vue` component to `Heading.vue` in the `./storyblok` folder. To keep stability in mind, I would strongly recommend planning ahead with content modeling instead of relying on rewrites of existing content types.

> **A rule of thumb**
>
> With a growing application, it becomes increasingly risky to introduce model updates that require existing components to be renamed or deleted. These are considered breaking changes and can cause your application to (momentarily) break. Storyblok will warn you that changes can take time to propagate throughout the content and that this can affect or prevent the app from generating all the pages. These sorts of warnings or errors should be treated with some caution.

Changing or removing functionalities always introduces some risks in code. It's much safer to add new features, as we will see next.

Expanding the block properties

Changes that add functionality are easier to implement, so let's see how we can expand the **Feature** block. Again, we'll open the edit view. We're going to add a couple of fields:

1. Add an `image` field name of the **Asset** type:

    I.   After creation, configure the field to only allow **Images**.

2. Add a `link` field name of the **Link** type.

Once we save these changes, we'll update the component so that we can experience the changes in the preview editor. Let's open the `./storyblok/Feature.vue` file and replace its contents:

```
<script setup lang="ts">
defineProps({ blok: {
  type: Object as () => { name: string, image: any, link: any },
} });
</script>

<template>
  <div v-editable="blok" v-if="blok" class="text-center">
    <ULink :to="blok.link.cached_url">
      <img
        v-if="blok.image"
        :src="blok.image.filename + '/m/200x0'"
        :alt="blok.image.alt"
      />
    </ULink>
    <header class="text-xl">
      {{ blok.name }}
    </header>
  </div>
</template>
```

We can update the default feature block on the home page of the website to, for instance, link to some portfolio items. You can also remove existing features if you don't want to use them. Having updated the component, Storyblok gives you a good visualization of what the result will look like.

The pages are still a bit empty now, and that's because we have no means of adding text. Let's create the component first this time, and then implement it in the CMS. We'll create a `RichText.vue` component in the `./storyblok` folder with the following contents:

```
<script setup lang="ts">
const props = defineProps({
  blok: { type: Object as () => { body: any }, required: true },
});
const richTextBody = computed(() => renderRichText(props.blok.body));
</script>

<template>
  <div v-editable="blok" class="prose">
    <div v-html="richTextBody" />
  </div>
```

```
</template>

<style scoped>
.prose {
  line-height: 1.8em;
}
</style>
```

Then, in Storyblok, we'll add a new block called `rich-text` as a nestable block. As part of the fields, we'll only add a single field called `body` of the **Richtext** type. Now, you can immediately add the content block and start adding additional content to the website!

We'll do a bit of refactoring with our blocks because the individual elements don't allow us to create a good-looking visual representation, especially when dealing with text pages. First, we'll create a Vue component in the `./storyblok` folder called `Article.vue`:

```
<script setup lang="ts">
defineProps({
  blok: { type: Object as () => { title: string, content: any},
required: true },
});
</script>

<template>
  <div v-editable="blok" class="article">
    <h2 v-if="blok.title" class="text-2xl mb-4">{{ blok.title }}</h2>
    <StoryblokComponent
      v-for="blok in blok.content"
      :key="blok._uid"
      :blok="blok"
    />
  </div>
</template>

<style scoped>
.article {
  max-width: 720px;
  margin: 2em auto;
}
</style>
```

Next, we'll flip over to the CMS to create a new block called `article` as a nestable block with the following fields:

1. Add a `title` field name of the **Text** type.
2. Add a `content` field name of the **Block** type.

The second field allows us to compose the article type with many different blocks that are available. It would be good practice to try and set up something such as a contact card block for yourself, for instance!

Taking a hint from the layout we've set up, we're still missing a page. With Storyblok, you should be able to create a **Me** story at the root of the project. You can use this space to introduce yourself and share your journey.

Mapping meta fields

Do you remember the **Description** fields we added to the **Portfolio All** and **Portfolio** blocks in the *Presenting the portfolio* section? We haven't mapped those! These fields are meant to provide some metadata to the page for **Search Engine Optimisation** (**SEO**) purposes. Using semantic HTML tags, we can describe the page contents specifically for indexing by search engines. Nuxt offers an out-of-the-box solution for controlling these tags and provides a composable (`https://nuxt.com/docs/getting-started/seo-meta#usehead`) for this.

In both cases, since the field name is the same, we can add the `meta` field by inserting the following code into the script tags:

```
useHead({
  meta: [
    { name: 'description', content: props.blok.description }
  ]
});
```

Now, if you look at the source of the website, you will see the tag and contents rendered:

```
</style>
<meta name="description" content="This will be a SEO optimized text with techy buzzwords">
<link rel="stylesheet" href="/_nuxt/node_modules/tailwindcss/tailwind.css">
<link rel="stylesheet" href="/_nuxt/node_modules/@nuxt/ui/dist/runtime/ui.css">
```

Figure 10.14 – The meta field has been rendered in the source code of the HTML

It's good to remember that we can use the content from the CMS not just for the visible parts of the website – we can process any content as a variable in our application. We could provide toggles or themes – anything really – where we can change the website without deploying new code. That's what makes a headless CMS a powerful tool when developing applications.

Adding new features

With these steps completed, you could try to add some more features to your portfolio. How about a nicely formatted contact sheet? Or a manageable resume section? With what we've built, you should be able to figure this out by yourself!

Generating a standalone website

We now have a functional portfolio. This type of website has very static content: it doesn't change depending on the visitor and will probably not receive daily updates in terms of code! This means that we can optimize the output of what we will deploy to the public.

The portfolio we've created is a perfect use case for static site generation. Normally, Nuxt would run as an active process on a server, where it can respond directly to requests and fetch data in real time. A good example is the *Quiz server* we built in *Chapter 8*. But since we don't need real-time data when publishing, we can leverage another capability of Nuxt. When generating a static site, Nuxt runs as an application initially and indexes all of the internal links. For every internal link, it will retrieve the data from the server once and then write the output to a static collection of files.

Let's see this in action! The command is already part of our `package.json` scripts, but we need to slightly modify it so that we can pass the API key we need to access the data:

```
…
  "scripts": {
    "build": "nuxt build",
    "dev": "NODE_TLS_REJECT_UNAUTHORIZED=0 nuxt dev --dotenv .env
--https --ssl-cert localhost.pem --ssl-key localhost-key.pem",
    "generate": "nuxt generate  --dotenv .env",
    "preview": "nuxt preview",
    "postinstall": "nuxt prepare"
  },
…
```

We can just run the following command in the terminal:

```
npm run generate
```

You will see that the required files for a static site are being generated. On completion, the *Nuxt* generation process will return a command you can use to test the website as a static site:

```
npx serve .output/public
```

Running this command will start a simple HTTP web server. There is no Nuxt process running! This is just the browser running HTML, CSS, and JavaScript files. All of the contents from the CMS are now bundled as part of the website. Well, except for the images – they are served from the Storyblok CDN, which is optimized for doing just that.

We'll see how we can make our static site public in the next section!

Publishing the static site

Any web host would be capable of serving these static files now. That's one of the benefits of a static-generated website: the amount of freedom when it comes to hosting them. We do have some requirements down the line, however, so we're choosing Netlify (https://app.netlify.com/) for our use case. If you register, you should be able to enter the free tier without any cost. While signing up, pick the categories that suit you best and pick a name for your team.

On the next screen, we'll go with the Netlify Drop way of deploying:
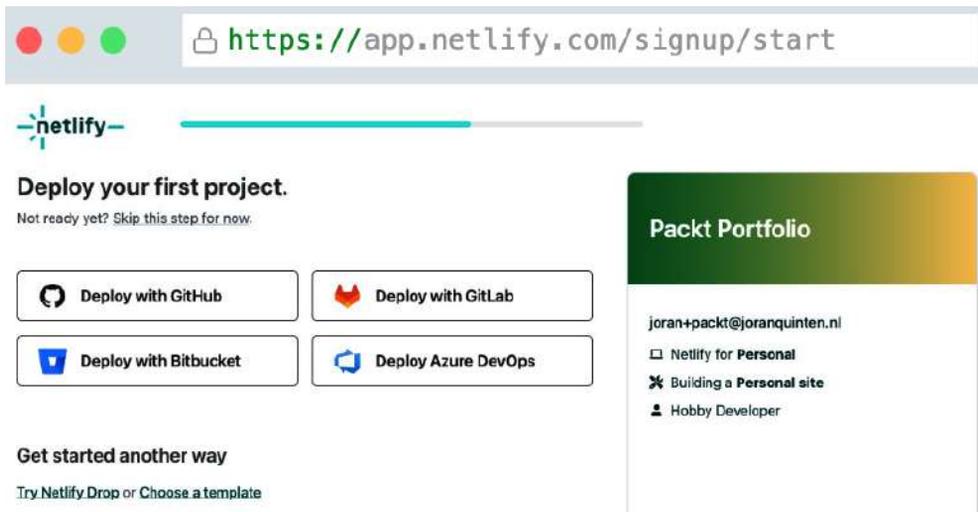
Figure 10.15 – Choosing your first method of deployment

Choosing **Try Netlify Drop** redirects you to a drag-and-drop space where you can simply drag the output files from our project's ./output/public folder. Make sure you drop the public folder and not the separate contents!

Once you've finished uploading everything, you will enter the dashboard for the first time. Clicking the **Open production deploy** button will send you to your published portfolio! The subdomain is generated by Netlify and contains random wording. You can opt to change it via the **Domain management** section:
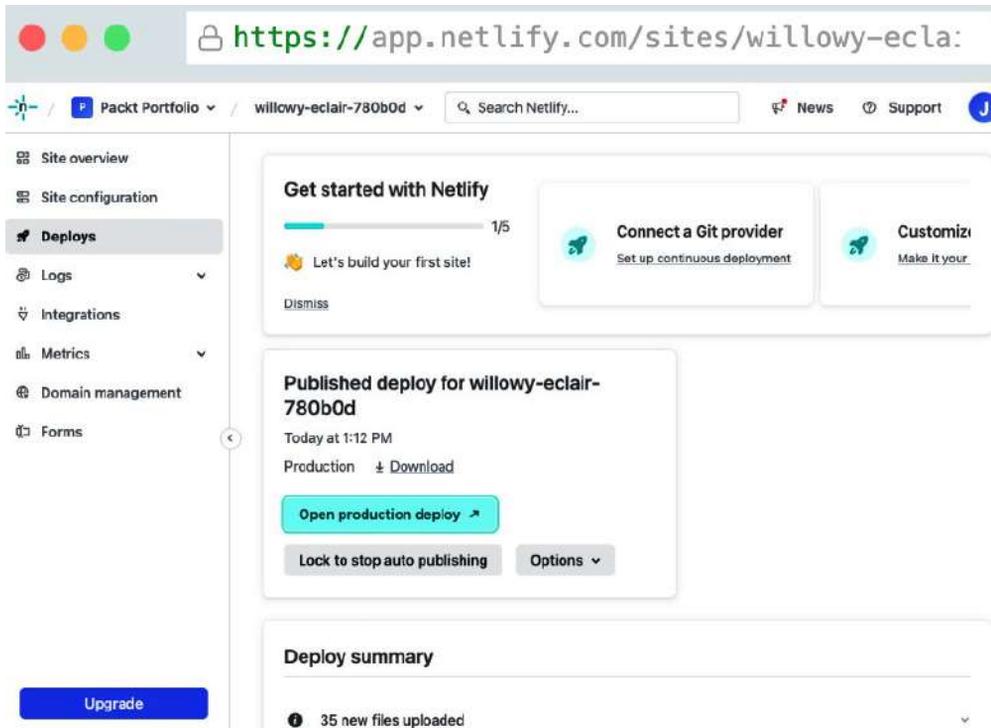
Figure 10.16 – The Netlify dashboard after the first deployment

The free tier of Netlify offers capabilities for processing domain names, pointing your own domain to this space.

The drawback of a statically generated site is that the website we deployed is completely detached from the content and our Nuxt server. This means that if we update the content or change a feature in the code, we need to regenerate the website and manually upload the output to Netlify. This works but is not ideal!

Luckily, with some extra effort and configuration, we can automate deployments when either the content changes or the code changes!

Automating the build and deployment on code change

Our Netlify environment does not know about our application. Netlify is a tool for deploying web apps. We can't connect Netlify to our local development environment. We need to host the source online as well.

GitHub (https://github.com/) has excellent integrations with Netlify and is very suitable for hosting and versioning code. Without diving into the details, GitHub can detect when code has been updated on the repository and can trigger certain actions when that happens.

First, we'll set up a repository where we can publish the code. Let's create a new empty repository:

Figure 10.17 – Creating a new repository in GitHub

Once the repository has been created, you can follow the next steps to attach our code to the repository using the terminal:

```
git init
git branch -m master main
git remote add origin https://github.com/{YOUR_USERNAME}/{REPOSITORY_
NAME}.git
git add .
git commit -am "published source to git"
git push -u origin main
```

This pushes our code to GitHub, which means we can connect it to Netlify! On the dashboard, navigate to **Site Configuration | Build & Deploy** and click the **Link repository** button. Then, you need to authorize Netlify on your GitHub account. Once you've done this, you'll see an overview of your repositories. Select the portfolio repository you just created to continue.

It might automatically detect that it's a Nuxt repository and Netlify will try and make some suggestions. We need to change three settings:

1. Change **Build command** to npm run generate.

2. Change **Publish directory** to ./output/public.

3. We need to add environment variables to add the contents of our .env file, with **Key** being NUXT_STORYBLOK_ACCESS_TOKEN and **Value** being what matches your access token. Make sure that it's saved!

The changes you've made should match these settings:

Figure 10.18 – Our new Netlify deployment settings

Deploying this will allow Netlify to read from the repository and pull in the latest version on a change. It will do this automatically. Then, it will run the `npm run generate` script to do the static site rendering on a virtual environment. The final step will be to deploy the contents of the output folder to the web domain.

It's that easy! If you work on your local code and commit and push the code to the repository, the change will automatically be deployed! It is good to note that we are not verifying whether our code is functional since we have no tests running on our application. It would be best practice to make sure that critical features of your code work.

Automating the build on content change

It would be perfect if we could do something similar on a content change rather than a code change, right? Luckily, there's a very convenient way of doing this: we can use webhooks to trigger actions. Storyblok will keep track of when content is ready to be published and it can trigger an action on the Netlify platform that executes a build just like with a code change. This time, however, Nuxt will fetch the latest version of the content during the generation process.

Let's see how we can set this up. First, we need to create a webhook in Netlify. This webhook is nothing more than a unique URL that triggers an internal action when it's being called.

Navigate to **Site configuration** and then locate the **Build hooks** section under **Build & Deploy**. If we click the **Add build hook** button, we can give it a meaningful name, such as `Deploy on Content Change`. The settings default to building the *main* branch, which is perfect. After a moment of saving, you will receive a unique URL with a pattern such as `https://api.netlify.com/build_hooks/UNIQUE_IDENTIFIER`.

Now, we can move to Storyblok to configure calling this webhook. If we navigate to the **Settings** and **Webhook** sections, we can manage webhooks! We'll create a new one, again with a meaningful name. In the **Endpoint URL** area, we can paste the endpoint from Netlify.

The *triggers* determine which events we want to call the webhook on. You can experiment a bit with the settings, but I recommend starting with this set:

Figure 10.19 – Selecting event triggers to call the webhook

Ideally, you want to select as few triggers as possible, simply because every build process takes time. Calling the webhook consecutive times just puts requests in a queue, which can lead to longer waiting times for your change to be visible.

Let's save these settings and change some content in our portfolio.

Tip

If you're not running the Nuxt development server, you can still edit the content using the panel on the right-hand side. You can even expand it to fill the screen using the **Show form view** button:
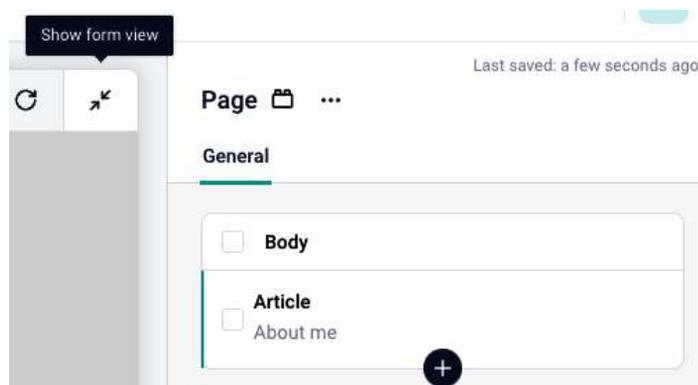
Figure 10.20 – Expanding the editing capabilities using the form view

Not having the development server just means that you don't have a live preview. It's not mandatory to access the content.

After saving, you can go over to the **Deploys** section on the Netlify dashboard to see that it triggered a new build.

You made it!

In this chapter, we've been working with different platforms and systems to connect them into a single application that renders a static website. One of the key takeaways is that we can combine the strengths of many specialized solutions to build a robust product we can deploy. I highly recommend checking out more resources on both Storyblok and Netlify since they work well with this type of web stack.

The combination of the live preview of Storyblok and the developer experience of using Nuxt as a framework makes these sorts of projects very easy to get started with and build something we can use! We've seen that Nuxt doesn't always have to run as a server – it can be used to generate the data from the server once and store the output. This approach is more sustainable and offers better performance than real-time data fetching.

With the way we've set up Storyblok and integrated it with Nuxt, it's a very accessible way for anybody to build a website. We've built a relatively simple example, but with an increasing number of blocks, the possibilities you can create scale up massively while still being very easy to understand for non-technical users.

Every configuration we've set up in this chapter was aimed at managing the website as easily as possible. Even the publication mechanisms add to that goal by both automating code deployments as well as automatically generating a new version when content is changed.

On top of all of this, we've created a platform where you can log your journey while working through the chapters of this book. I recommend completing the portfolio with projects or achievements you're proud of.

One of the goals of this book was to present real-world code. Real-world code is often pragmatic and may not always be optimized to perfection. Similarly, the examples in this book are not flawless. However, they effectively achieve the intended goals. Embracing pragmatism enables you to validate features with users and shorten feedback loops, which is a very valuable aspect. I've illustrated that refactoring and optimization are integral parts of a continuous process. I want to stress that, as you gain deeper insights into projects, you'll naturally prioritize which aspects require optimization.

Looking back, I hope you are proud of the work you've put in with all the different types of topics, technologies, and apps we've built throughout this book. I have tried to paint a broad picture of possible real-world scenarios while continuously increasing your level of Vue.js knowledge and experience. I would like to thank you for your interest in these topics and for following along during our journey. The investigating, building, and documenting process has also been very valuable for me, and I am grateful for the opportunity I've had to teach and share my knowledge and approach when writing code.

Index

‹packt›

Packt.com

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

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals

- Improve your learning with Skill Plans built especially for you

- Get a free eBook or video every month

- Fully searchable for easy access to vital information

- Copy and paste, print, and bookmark content

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

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

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:


Vue.js 3 Design Patterns and Best Practices

Pablo David Garaguso

ISBN: 978-1-80323-807-4

- What is the Vue 3 progressive framework.

- What are software principles and design patterns, how and when to implement them, and the trade-offs to consider.

- Setup your development environment using the new Vite bundler.

- Integrate in your applications state management, routing, multithreading, offline storage, and other resources provided to you by the browser, seldom taken advantage.

- Apply and identify design patterns to solve common problems in the architecture of your web application.
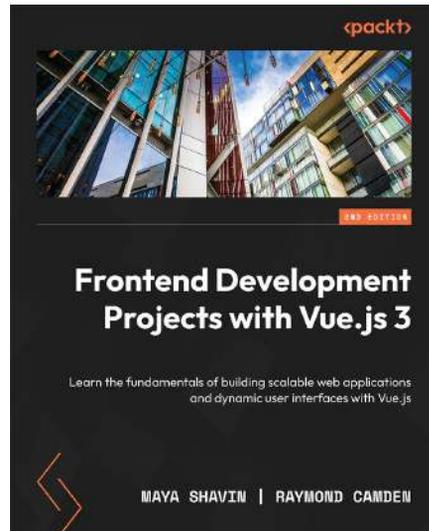
Frontend Development Projects with Vue.js 3

Maya Shavin | Raymond Camden

ISBN: 978-1-80323-499-1

- Set up a development environment and start your first Vue.js 3 project.
- Modularize a Vue application using component hierarchies.
- Use external JavaScript libraries to create animations.
- Share state between components and use Pinia for state management.
- Work with APIs using Pinia and Axios to fetch remote data.
- Validate functionality with unit testing and end-to-end testing.

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.

Hi!

I'm Joran Quinten, the author of Building Real-World Web Applications with Vue.js 3. I really hope you enjoyed reading this book and found it useful for increasing your productivity and efficiency in Web Application Development with Vue.js.

It would really help me (and other potential readers!) if you could leave a review on Amazon sharing your thoughts on Building Real-World Web Applications with Vue.js 3 here.

Go to the link below or scan the QR code to leave your review:

`https://packt.link/r/1837630399`

Your review will help me to understand what's worked well in this book, and what could be improved upon for future editions, so it really is appreciated.

Best Wishes,

Joran Quinten

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

https://packt.link/free-ebook/9781837630394

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly