Vue.js 3 Cookbook

Discover actionable solutions for building modern web apps with the latest Vue features and TypeScript

Heitor Ramon Ribeiro

Vue.js 3 Cookbook

Discover actionable solutions for building modern web apps with the latest Vue features and TypeScript

Heitor Ramon Ribeiro

Packt

BIRMINGHAM - MUMBAI

Vue.js 3 Cookbook

Commissioning Editor: Pavan Ramchandani
Acquisition Editor: Ashitosh Gupta
Content Development Editor: Akhil Nair
Senior Editor: Hayden Edwards
Technical Editor: Deepesh Patel
Copy Editor: Safis Editing
Project Coordinator: Kinjal Bari
Proofreader: Safis Editing
Indexer: Manju Arasan
Production Designer: Nilesh Mohite

First published: September 2020

Production reference: 1180920

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

ISBN 978-1-83882-622-2

`www.packt.com`

Contributors

About the author

Heitor Ramon Ribeiro has been developing web applications for over 15 years, constantly navigating between frontend and backend development. By following his passion for UX/UI and programming, he chose to stay in frontend development.

Heitor has built enterprise applications for businesses using Vue.js and the principles of clean architecture, shifting his course from legacy applications to the new world of **single-page applications** (**SPAs**) and **progressive web applications** (**PWAs**). He thinks that almost anything is possible today with a browser, and that JavaScript is the future of programming.

When he's not programming or leading a frontend team, he's with his family having fun, streaming their gaming sessions, or playing some first-person shooter games.

I would like to thanks my lovely wife, Raquel, for being with me every day helping and supporting me throughout this process of publishing my first book. To my son, Marco, your father loves you so much and does everything for you. To my family and friends who helped me during the development of this book, especially Patrick Monteiro, who helped me a lot.

To all the coworkers and companies that I've worked with over the last few years, who helped me and understood the importance of this project for me.

About the reviewers

Swanand Kadam is the creator of the first-ever online programming language in Hindi, built to improve coding literacy in rural India. He is an experienced web application and PWA architect who has designed and developed e-commerce, employee management, and custom software solutions to a range of businesses. Swanand is a consistent user of technologies including Vue.js, Firebase, Node.js, Google Cloud, and NoSQL databases. Swanand's articles have been published in top tech publications including Better Programming and Hackernoon. He is also an editor for InfoQ, where he talks about the latest trends in the world of software development.

Tyler VanBlargan is a frontend developer, primarily working with Vue.js and helping others learn about web development. When not working on new applications, Tyler can be found photographing his pets and playing board games.

Table of Contents

Preface

Vue is a minimal frontend framework that empowers developers to create web applications, prototypes, big enterprise applications, desktop applications, and mobile applications.

Vue 3 is a complete rewrite of Vue and brings changes to all the core APIs of the frameworks. This rewrite changes code that was written to flow in TypeScript. In Vue 3 we have all of the core APIs exposed, giving the possibility of using Vue to everyone.

The book starts with recipes for implementing Vue 3's new features in your web development projects and migrating your existing Vue apps to the latest version. You will get up and running with TypeScript with Vue, and find succinct solutions to common challenges and pitfalls faced in implementing components, derivatives, and animation, through to building plugins, adding state management, routing, and developing complete single-page applications (SPAs).

Some of the libraries, plugins, and frameworks used in this book might receive updates between the writing of this book and the time that you're reading it. So, please pay attention to any API changes or version changes that may have any breaking changes.

Who this book is for

This book is for web developers who wants to learn more about Vue and wants to improve their Vue skills. We'll start by presenting the Vue 3 and TypeScript technologies. In the subsequent chapters, the reader will be presented with the new concepts in Vue and their ecosystem plugins, UI frameworks, and advanced recipes.

By following the book from cover to cover, you will be able to create a Vue application, use all the essential Vue plugins, and employ the top Vue UI frameworks. If you are already familiar with Vue, you will discover relevant new patterns.

What this book covers

`Chapter 1`, *Understanding Vue 3 and Creating Components*, provides the reader with recipes on how to use the new Vue 3 APIs to create custom Vue components using Vue's exposed core API and the Composition API. This chapter also helps the reader along an initial upgrade path of a Vue 2 application to Vue 3.

`Chapter 2`, *Introducing TypeScript and the Vue Ecosystem* introduces the reader to the TypeScript superset and how to use it, starting with basic types, interfaces, and type annotations. The reader will become ready for the development of a Vue application with Vue CLI, TypeScript, and `vue-class-component`.

`Chapter 3`, *Data Binding, Form Validations, Events, and Computed Properties*, discusses the basic Vue developments and component concepts, including `v-model`, event listeners, computed properties, and `for` loops. The reader will be introduced to the Vuelidate plugin for form validation and how to use it on a Vue component, along with how to debug a Vue component with `vue-devtools`.

`Chapter 4`, *Components, Mixins, and Functional Components*, walks the reader through building components with different approaches, including custom slots for contents, validated props, functional components, and creating mixins for code reusability. It then introduces the reader to a set of different approaches for accessing child components' data, creating a dependency injection component and dynamic injected component, and how to lazy load a component.

`Chapter 5`, *Fetching Data from the Web via HTTP Requests*, shows the reader how to create a custom wrapper around the Fetch API for HTTP calls on JavaScript, how to use the wrapper in Vue, and how to implement custom asynchronous functions on Vue. The reader will also learn how to replace the Fetch API in the wrapper for axios, and how custom handlers can be implemented on axios.

`Chapter 6`, *Managing Routes with vue-router*, takes a look at Vue's routing plugin and how to use it on Vue to create routes for the pages of a Vue application. It introduces the process of managing router paths, dynamic paths with parameters on the router path, lazy loading the page component, creating middleware for authentication on the router, and using an alias and redirect.

`Chapter 7`, *Managing the Application State with Vuex*, explores the Vue state management plugin to help the reader understand how Vuex works and how it can be applied to their application. This chapter also provides the reader with recipes for creating Vuex modules, actions, mutations, and getters, and explores how to define the base state for the store.

`Chapter 8`, *Animating Your Application with Transitions and CSS*, explores the fundamentals of CSS animation and transitions by providing recipes for custom animations based only on CSS. These will be used with a Vue custom component to achieve a nice looking application and provide the best experience for the application's users.

`Chapter 9`, *Creating Beautiful Applications Using UI Frameworks*, take a look at popular UI frameworks. The reader will build a user registration form with Buefy, Vuetify, and Ant-Design with their design concept. The aim of the recipes in this chapter is to teach the reader how to create a good-looking application with a UI framework.

`Chapter 10`, *Deploying an Application to Cloud Platforms*, shows how to deploy a Vue application on custom third-party hosters such as Vercel, Netlify, and Google Firebase. Using the recipes in this chapter, the reader will learn how to automatically deploy their application with integrated repository hooks and auto-deploy functions.

`Chapter 11`, *Pro League – Directives, Plugins, SSR, and More*, explores advanced topics on Vue, including patterns, best practices, how to create plugins and directives, and how to use high-level frameworks such as Quasar and Nuxt.js to create applications.

To get the most out of this book

Vue 3 beta was the version available at the time of writing this book. All the code will be updated with the final release on the GitHub repository here: `https://github.com/PacktPublishing/Vue.js-3.0-Cookbook`

You will need Node.js 12+ installed, Vue CLI updated to the latest version, and a good code editor of some sort. Other requirements will be introduced in each recipe. All the software requirements are available for Windows, macOS, and Linux.

To develop mobile applications for iOS, you need a macOS machine to get access to Xcode and the iOS simulator. Here's a table summarizing all the requirements:

Software/hardware covered in the book	**OS requirements**
Vue CLI 4.X	Windows / Linux / macOS
TypeScript 3.9.X	Windows / Linux / macOS
Quasar-CLI 1.X	Windows / Linux / macOS
Nuxt-CLI 3.X.X	Windows / Linux / macOS

Visual Studio Code 1.4.X and IntelliJ WebStorm 2020.2	Windows / Linux / macOS
Netlify-CLI	Windows / Linux / macOS
Vercel-CLI	Windows / Linux / macOS
Firebase-CLI	Windows / Linux / macOS
Node.js 12+-	Windows / Linux / macOS
Python 3	Windows / Linux / macOS
Xcode 11.4 and iOS Simulator	macOS

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

Download the example code files

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

You can download the code files by following these steps:

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

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

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

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Vue.js-3.0-Cookbook`. In case there's an update to the code, it will be updated on the existing GitHub repository.

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

Conventions used

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

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

A block of code is set as follows:

```
<template>
 <header>
 <div id="blue-portal" />
 </header>
</header>
```

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

```
$ npm run serve
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Click on the **Email** button to be redirected to the **Email Sign up** form"

Warnings or important notes appear like this.

Tips and tricks appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There's more…

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

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

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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

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

Reviews

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

For more information about Packt, please visit `packt.com`.

1
Understanding Vue 3 and Creating Components

Vue 3 brings a lot of new features and changes for developers, all of them designed to aid development and improve the framework's overall stability, speed, and maintainability. Using other frameworks and libraries as inspiration, the Vue core team managed to achieve a great level of abstraction on the API where anyone can use Vue now, irrespective of whether they're a frontend developer or a backend developer.

In this chapter, we will learn how to upgrade our Vue project to the new version, and more about some of the new Vue features, such as the multiple root elements, the new attribute inheritance engine, how we can use the exposed reactivity API outside of Vue in another application, and how to create a component using the new composition API.

In this chapter, you will learn the following:

- What is new in Vue 3
- Upgrading your Vue 2 application to Vue 3
- Creating components with multiple root elements
- Creating components with attribute inheritance
- Using the reactivity and observable API outside the scope of Vue
- Creating a component using the composition API

What is new in Vue 3

You may be wondering how a new version of a framework could result in such hype on the internet? Imagine taking a car on the highway, doing a complete 360 roll, and then continuing to go full speed ahead in the same direction. This would cause a theatrical scene, and it's the perfect way to describe how Vue will go from version 2 to 3.

In this first part of the chapter, I will introduce you to the improvements on Vue, what was added to the framework, what has changed, and how it will impact the way you code a Vue application.

Improvements to the framework

There are numerous improvements to the Vue framework in this new release; all of them focused on making the framework better in every way possible. Here are some of the improvements that can impact the everyday development and usage of the framework by users and developers.

Under the hood

The outer shell looks the same as the old one, but the engine is a piece of art. In the new version, there is no leftover code from Vue 2. The core team built the framework from the ground up using TypeScript and rewrote everything geared to the maximum performance of the framework.

TypeScript was chosen to create a more maintainable code base for the Vue core team and the open-source community, and to improve the autocomplete features, such as **IntelliSense** or **typeahead** that the IDEs and code editors provide, without the need for special plugins and extensions.

Render engine

For Vue 3, a new render engine was developed using a new algorithm for the shadow DOM. This new render is totally exposed by the core of the framework by default, without the need to be executed by the framework. This makes it possible for new implementations of a completely new render function that can be injected into the framework and replace the original render engine.

In this new version of Vue, a new template compiler was written from scratch. This new compiler uses a new technique for cache manipulation and to manage the rendered elements, and a new hoisted method is applied to the creation of VNodes.

For cache manipulation, a new method is applied to control the position of the element, where the element can be a dynamic element with computed data or a response to a function that can be mutated.

The Vue core team has made an explorer where it's possible to see how the new template compiler renders the final `render` function. This can be viewed at `https://vue-next-template-explorer.netlify.app/`.

Exposed APIs

With all these modifications, it was possible to render all the Vue APIs exposed to usage within files outside the scope of application of Vue. It's possible to use the Vue reactivity or the shadow DOM in a React application, without the need to render a Vue application inside the React application. This explosibility is a way of transforming Vue into a more versatile framework, where it can be used anywhere, not just in frontend development.

New custom components

Vue 3 introduces three new custom components that can be used by the developer to resolve old problems. These components were present on Vue 2 but as third-party plugins and extensions. Now they are made by the Vue core team and added to the Vue core framework.

Fragments

In Vue 2, we always needed to have a parent node wrapping the components inside the single-file components. This was caused by the way in which the render engine of Vue 2 was constructed, requiring a root element on each node.

In Vue 2, we needed to have a wrapper element, encapsulating the elements that will be rendered. In the example, we have a `div` HTML element, wrapping two `p` HTML child elements, so we can achieve multiple elements on the page:

```
<template>
  <div>
    <p>This is two</p>
    <p>children elements</p>
  </div>
</template>
```

Now, in Vue 3, it's possible to declare any number of root elements on the single-file components without the need for special plugins using the new Fragments API, which will handle the multiple root elements. This helps to maintain a cleaner final code for the user, without the need for empty shells just for wrapping elements:

```
<template>
  <p>This is two</p>
  <p>root elements</p>
</template>
```

As we saw in the Vue 3 code, we were able to have two root `p` HTML elements, without the need for a wrapper element.

Teleport

A `Teleport` component, also known as a Portal component, as the name implies, is a component that can make an element go from one component to another. This may seem strange in the first instance, but it has a lot of applications, including dialogs, custom menus, alerts, badges, and many other customs UIs that need to appear in special places.

Imagine a header component, where you want a custom slot on the component so you can place components:

```
<template>
  <header>
    <div id="blue-portal" />
  </header>
</header>
```

Then, you want to display a custom button on this header, but you want to call this button from a page. You just need to execute the following code:

```
<template>
  <page>
   <Teleport to="blue-portal">
     <button class="orange-portal">Cake</button>
   </Teleport>
  </page>
</template>
```

Now, your button will be displayed on the header, but the code will be executed on the page, giving access to the page scope.

Suspense

When the wait for the data is taking longer than you would like, how about showing a custom loader for the user? This is now possible without the need for custom code; Vue will handle this for you. The `Suspense` component will manage this process, with a default view once the data is loaded, and a fallback view when the data is being loaded.

You can write a special wrapper like this:

```
<template>
  <Suspense>
    <template #default>
      <data-table />
    </template>
    <template #fallback>
      <loading-gears />
    </template>
  </Suspense>
</template>
```

The new Vue composition API will understand the current state of your component, so it will be able to differentiate if the component is loading or if it's ready to be displayed.

API changes

Some API changes were made in Vue 3 that were necessary in order to clean the Vue API and simplify development. Some of them are break changes, and others are additions. But don't worry; the Vue 2 object development was not removed, it's still there, and will continue to be used. This declaration method was one of the reasons why many developers choose Vue over other frameworks.

There are some break changes that will happen in Vue 3 that are important to learn more about. We will discuss the most important break changes that will be introduced in Vue 3, and how to deal with then.

In Vue 3, a new way of creating the components is being introduced – the composition API. This method will make the maintainability of your code better, and give you a more reliable code, where you will have the full power of TypeScript available.

Some minor break changes

There are some minor break changes that are present in Vue 3 that need to be mentioned. These changes relate to one method we used previously to write code, and that has now been replaced when using Vue 3. It's not a Herculean job, but you need to know about them.

Goodbye filters, hello filters! The Vue filters API

The way we used `filters` on Vue 2 is no longer available. The Vue filter has been removed from the API. This change was made to simplify the render process and make it faster. All filters, in the end, are functions that receive a string and return a string.

In Vue 2, we used to use `filters` like this:

```
{{ textString | filter }}
```

Now, in Vue 3, we just need to pass a `function` to manipulate the `string`:

```
{{ filter(textString) }}
```

The bus just left the station! The event bus API

In Vue 2, we were able to use the power of the global Vue object to create a new Vue instance, and use this instance as an event bus that could transport messages between components and functions without any hassle. We just needed to publish and subscribe to the event bus, and everything was perfect.

This was a good way to transfer data between components, but was an anti-pattern approach for the Vue framework and components. The correct way to transfer data between components in Vue is via a parent-child communication, or state management, also known as state-driven architecture.

In Vue 3, the `$on`, `$off`, and `$once` instance methods were removed. To use an event bus strategy now, it is recommended to use a third-party plugin or framework such as mitt (`https://github.com/developit/mitt`).

No more global Vue – the mounting API

In Vue 2, we were accustomed to importing Vue, and prior to mounting the application, use the global Vue instance to add the `plugins`, `filters`, `components`, `router`, and `store`. This was a good technique where we could add anything to the Vue instance without needing to attach anything to the mounted application directly. It worked like this:

```
import Vue from 'vue';
import Vuex from 'vuex';
import App from './App.vue';

Vue.use(Vuex);
const store = new Vuex.store({});

new Vue({
  store,
  render: (h) => h(App),
}).$mount('#app');
```

Now, in Vue 3, this is no longer possible. We need to attach every `component`, `plugin`, `store`, and `router` to the mounted instance directly:

```
import { createApp } from 'vue';
import { createStore } from 'vuex';
import App from './App.vue';

const store = createStore({});

createApp(App)
  .use(store)
  .mount('#app');
```

Using this method, we can create different Vue applications in the same global application, without the `plugins`, `store`, or `router` of the applications messing with one another.

v-model, v-model, v-model – multiple v-model

When developing a single-file component, we were stuck with a single `v-model` directive and a `.sync` option for a second update change. This meant us using a lot of custom event emitters and huge object payloads to handle data inside the component.

In this breaking change, a collateral break change was introduced that resulted in the `model` property (`https://vuejs.org/v2/api/#model`) being removed from the Vue API. This property is used in custom components that used to do the same thing that the new v-model directive now does.

The new way to use the `v-model` directive will change how the sugar syntax works. In Vue 2, to use a `v-model` directive, we had to create a component expecting to receive the `props` as `"value"`, and when there was a change, we needed to emit an `'input'` event, like the following code:

```
<template>
  <input
    :value="value"
    @input="$emit('input', $event)"
  />
</template>
<script>
export default {
  props: {
    value: String,
  },
}
</script>
```

In Vue 3, to make the syntactic sugar work, the `props` property that the component will receive and the event emitter will change. Now, the component expects a `props` named `modelValue` and it emits an event, `'update:modelValue'`, like the following code:

```
<template>
  <input
    :modelValue="modelValue"
    v-on:['update:modelValue']="$emit('update:modelValue', $event)"
  />
</template>
<script>
export default {
  props: {
    modelValue: String,
  },
}
</script>
```

But how about the multiple `v-model` directives? Understanding the `v-model` break change is the first step in getting to know how the new method of multiple `v-model` will work.

To create multiple `v-model` components, we need to create various `props` with the name of the model directive we want and emit `'update:value'` events where the value is the name of the model directive:

```
<script>
export default {
  props: {
    name: String,
    email: String,
  },
  methods: {
   updateUser(name, email) {
    this.$emit('update:name', name);
    this.$emit('update:email', email);
   }
  }
}
</script>
```

In the component where we want to use the multiple `v-model` directives, use the following code:

```
<template>
  <custom-component
    v-model:name="name"
    v-model:email="email"
  />
</template>
```

The component will have each `v-model` directive, bounded to the event the child is emitting. In this case, the child component emits `'update:email'` (the parent component) in order to be able to use the `v-model` directive with the email modifier. For example, you can use `v-model:email` to create the two-way data binding, between the component and the data.

Composition API

This is one of the most anticipated features of Vue 3. The composition API is a new way of creating Vue components, with an optimized way of writing code, and providing full TypeScript type checking support in your component. This method organizes the code in a simpler and more efficient way.

In this new way of declaring a Vue component, you just have a `setup` property that will be executed and will return everything your component needs in order to be executed, like this example:

```
<template>
  <p @click="increaseCounter">{{ state.count }}</p>
</template>
<script>
import { reactive, ref } from 'vue';

export default {
  setup(){
    const state = reactive({
      count: ref(0)
    });

    const increaseCounter = () => {
      state.count += 1;
    }

    return { state, increaseCounter }
  }
}
</script>
```

You will import the `reactivity` API from the Vue core to enable it in the object type data property, in this case, `state`. The `ref` API enables reactivity in the basic type value, like `count`, which is a number.

Finally, the functions can be declared inside the `setup` functions and passed down on the returned object. Then, everything is accessible in the `<template>` section.

Now, let's move on to some recipes.

Technical requirements

In this chapter, we will be using **Node.js** and **Vue-CLI.**

Attention Windows users! You need to install an NPM package called `windows-build-tools` to be able to install the following requisite packages. To do this, open Power Shell as an administrator and execute the following command:

```
> npm install -g windows-build-tools
```

To install Vue-CLI, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install -g @vue/cli @vue/cli-service-global
```

Creating the base file

In all recipes in this chapter, we will use this base template which we will create now. Make sure you follow these steps to create the file before starting the example in the recipe:

1. Create a new `.html` file in any folder and open it.
2. Create an `html` tag and add a `head` HTML element as a child. Inside the `head` HTML element, add a `script` HTML element with the `src` attribute defined as `http://unpkg.com/vue@next`:

   ```
   <html>
     <head>
       <script src="https://unpkg.com/vue@next"></script>
     </head>
   </html>
   ```

3. As a sibling of the `head` HTML element, create a `body` HTML element. Inside the `body` HTML element, add a `div` HTML element with the attribute `id` defined as `"app"`:

   ```
   <body>
     <div id="app">
     </div>
   </body>
   ```

4. Finally, as a sibling of the `div` HTML element, create a `script` HTML element, with empty content. This will be where we will place the code for the recipes:

   ```
   <script></script>
   ```

Upgrading your Vue 2 application to Vue 3

Upgrading your project from Vue 2 to Vue 3 can sometimes be done automatically, but in other cases, this needs to be done manually. This depends on how deep into the use of the Vue API you go with your application.

With projects made and managed by Vue-CLI, this process will be made seamlessly and will have a more straightforward approach compared to projects using a custom framework wrapper CLI.

In this recipe, you will learn how to upgrade your application using Vue-CLI and how to upgrade the project and the dependencies manually.

Getting ready

The prerequisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

In order to upgrade your Vue 2 project to Vue 3, you will have to split the upgrade into different parts. We have the upgrade of the framework itself, and then we have the ecosystem components, such as `vue-router` and `vuex`, and finally, the bundler that joins everything in the end.

The framework upgrade comes with break changes. There are some break changes that are presented in this book in the *What is new in Vue 3* section of this chapter, and others that may occur in a more advanced API schema. You have to manually update and check whether your components are valid for the upgrade on the framework.

Using Vue-CLI to upgrade the project

Using the latest version of Vue-CLI, you will be able to use Vue 3 in your project, out of the box, and you will be able to update your current project to Vue 3.

To update Vue-CLI to the latest version, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install @vue/cli-service@latest
```

Upgrading the project manually

To upgrade the project manually, you will have to first upgrade the project dependencies to their latest versions. You cannot use an old version of a Vue ecosystem plugin with Vue 3. To do this, perform the following steps:

1. We need to upgrade the Vue framework, the ESLint plugin (which Vue depends on), and the `vue-loader` for the bundler. To upgrade it, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > npm install vue@next eslint-plugin-vue@next vue-loader@next
   ```

2. We need to add the new Vue single-file component compiler as a dependency to the project. To install it, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > npm install @vue/compiler-sfc@latest
   ```

3. If you are using unit tests and the `@vue/test-utils` package on your project, you will also need to upgrade this dependency. To upgrade it, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > npm install @vue/test-utils@next @vue/server-test-utils@latest
   ```

4. For the Vue ecosystem plugins, if you are using `vue-router`, you will need to upgrade this too. To upgrade it, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > npm install vue-router@next
   ```

5. If your application is using `vuex` as the default state management, you will need to upgrade this too. To upgrade it, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > npm install vuex@next
   ```

Changing the starting files

With the new version of the packages, we will need to change our starting files. In a Vue project that was created with the Vue-CLI starter kit, you will find a file named `main.js` or `main.ts`. If you are using TypeScript, this file is located in the `src` folder. Now follow these instructions:

1. Open the `main.js` file in the `src` folder of your project. At the top of the file, where the packages are imported, you will see the following code:

   ```
   import Vue from 'vue';
   ```

   We need to change this to the new Vue exposed API method. To do this, we need to import `createApp` from the Vue package as follows:

   ```
   import { createApp } from 'vue';
   ```

2. Remove the global Vue static attribute definition of `Vue.config.productionTip` from your code.
3. The mounting function of your application needs to be changed. The old API will look like this:

   ```
   new Vue({
     router,
     store,
     render: (h) => h(App),
   }).$mount('#app');
   ```

   The old API should be changed to the new `createApp` API, as follows:

   ```
   createApp(App)
     .use(router)
     .use(store)
     .mount('#app')
   ```

4. Open your `vuex` store instantiation file (normally, this file is located in `src/store` and is named `store.js` or `index.js`).
5. Change the creation of the store from the instantiation of a new `vuex` class to the new `createStore` API. The vuex v3 class instantiation may look like this:

   ```
   import Vue from 'vue';
   import Vuex from 'vuex';

   Vue.use(Vuex);

   export default new Vuex.Store({
   ```

```
  state: { /* ... */ },
  mutations: { /* ... */ },
  actions: { /* ... */ },
  getters: { /* ... */ },
  modules: { /* ... */ },
});
```

You need to replace its content with the `createStore` API, which could look like this, for example:

```
import { createStore } from 'vuex';

export default createStore({
  state: { /* ... */ },
  mutations: { /* ... */ },
  actions: { /* ... */ },
  getters: { /* ... */ },
  modules: { /* ... */ },
});
```

6. In the `vue-router` ecosystem, we will need to replace the old API from the router creation with the new one. To do this, open the router creation file (in the `src/router` folder, normally named `router.js` or `index.js`).
7. Finally, in the creation file, replace the old `vue-router` class instantiation with the new `createRouter` API. The `vue-router` v3 class instantiation may look like this:

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

Vue.use(VueRouter);

export default new VueRouter({
  routes: [{
    path: '/',
    name: 'HomePage',
    component: () => import('pages/home'),
  }]
});
```

You will also need to replace the `new VueRouter` instantiation with the new `createRouter` and `createWebHistory` API, as in this example:

```
import {
  createRouter,
  createWebHistory,
} from 'vue-router';
```

```
Vue.use(VueRouter);

export default createRouter({
  history: createWebHistory(),
  routes: [{
    path: '/',
    name: 'HomePage',
    component: () => import('pages/home'),
  }]
});
```

How it works...

In the upgrading process, Vue has provided us with two ways to update our project. The first way is to use the Vue-CLI plugin, which tries to automate almost all the processes and changes needed for the upgrade.

The second way is to upgrade the project manually. This method requires the developer to upgrade all the dependencies to the latest version, install the new single-file component compiler, `@vue/compiler-sfc`, and change the entry files for the Vue application, router, and store to the new API.

Following the changes to the starter structure of the project, the developer needs to check the components to see whether there are any Vue 3 breaking changes present, refactor the component to the new Vue 3 APIs, and remove the deprecated APIs from Vue 2.

Creating components with multiple root elements

In Vue 3, it is possible to create components with multiple root elements, without the need for a wrapping element. This option is also known as a fragment.

In React, this has been possible for a long time, but in Vue, you need to use custom third-party plugins such as `vue-fragment` (`https://github.com/Thunberg087/vue-fragment`) to use this feature.

In this recipe, you will learn how to create a component with multiple root elements, and how it could be used with a `<template>` section and a `render` function.

How to do it...

In this recipe, we will create two examples of a multiple root element component, one with a `<template>` structure, and another with a `render` function. To do this, this recipe will be divided into two parts.

Creating the component with the <template> structure

In order to use the `<template>` structure in our example, we will be using the `template` property of the Vue object where we can pass a string or a template string as the value, which will be interpolated by the Vue script and rendered on the screen:

1. Using the base example from the 'Creating the base file' section, create a new file named `template.html` and open it.
2. In the empty `<script>` HTML element, create the constants `defineComponent` and `createApp` by object-destructuring the `Vue` global constant:

```
const {
  defineComponent,
  createApp,
} = Vue;
```

3. Create a constant named `component`, defined as the `defineComponent` method, passing a JavaScript object as an argument with three properties: `data`, `methods`, and `template`:

```
const component = defineComponent({
 data: () => ({}),
 methods: {},
 template: ``
});
```

4. In the `data` property, define it as a singleton function, returning a JavaScript object, with a property named `count` and with the default value as `0`:

```
data: () => ({
  count: 0
}),
```

5. In the `methods` property, create a property called `addOne`, which is a function that will increase the value of `count` by `1`:

```
methods: {
  addOne() {
    this.count += 1;
  },
},
```

6. In the `template` property, in the template string, create an `h1` HTML element with a title. Then, as a sibling, create a `button` HTML element with an event listener bound to the `click` event, triggering the `addOne` function when executed:

```
template: `
  <h1>
    This is a Vue 3 Root Element!
  </h1>
  <button @click="addOne">
    Pressed {{ count }} times.
  </button>
`
```

7. Finally, call the `createApp` function, passing the `component` constant as an argument. Then, prototype chain the `mount` function and, as an argument of the function, pass the `div` HTML element `id` attribute, `("#app")`:

```
createApp(component)
  .mount('#app');
```

Creating the component with the render function

In order to use the `<template>` structure in our example, we will be using the `template` property of the Vue object, where we can pass a string or a template string as the value, which will be interpolated by the Vue script and rendered on the screen:

1. Using the base example from the 'Creating the base file' section, create a new file named `render.html` and open it.

2. In the empty `<script>` HTML element, create the constants of the functions that will be used using the object destructuring method, calling the `defineComponent`, `h`, and `createApp` methods from the `Vue` global constant:

```
const {
  defineComponent,
  h,
  createApp,
} = Vue;
```

3. Create a constant named `component`, defined as the `defineComponent` method, passing a JavaScript object as an argument with three properties: `data`, `methods`, and `render`:

```
const component = defineComponent({
  data: () => ({}),
  methods: {},
  render() {},
});
```

4. In the `data` property, define it as a singleton function, returning a JavaScript object with a property named `count` and with the default value as `0`:

```
data: () => ({
  count: 0
}),
```

5. In the `methods` property, create a property called `addOne`, which is a function that will increase the value of `count` by `1`:

```
methods: {
  addOne() {
    this.count += 1;
  },
},
```

6. In the `render` property, perform the following steps:
    - Create a constant named `h1` and define it as the `h` function, passing `'h1'` as the first argument, and the title that will be used as the second argument.
    - Create a constant named `button`, which will be the `h` function, passing `"button"` as the first argument, a JavaScript object with the property `onClick` with a value of `this.addOne` as the second argument, and the content of `button` as the third argument.

- Return an array, with the first value as the `h1` constant, and the second value as the `button` constant:

```
render() {
  const h1 = h('h1', 'This is a Vue 3 Root Element!');
  const button = h('button', {
      onClick: this.addOne,
    }, `Pressed ${this.count} times.`);

  return [
    h1,
    button,
  ];
},
```

7. Finally, call the `createApp` function, passing the `component` constant as an argument, prototype chaining the `mount` function, and passing the `div` HTML element `id` attribute, `("#app")`, as an argument of the function:

```
createApp(component)
  .mount('#app');
```

How it works...

The new Vue component creation API needs to be executed by a function, `defineComponent`, and the JavaScript object that is passed as an argument maintains almost the same structure as the old structure in Vue 2. In the examples, we used the same properties, `data`, `render`, `methods`, and `template`, all present in Vue 2.

In the example with the `<template>` structure, we didn't have to create a wrapper element to encapsulate the content of our application component and were able to have two root elements on the component directly.

In the `render` function example, the same behavior occurs, but the final example used the new exposed `h` API, where it is no longer a parameter of the `render` function. A breaking change was present in the example; in the button creation, we had to use the `onClick` property inside the data JavaScript object, not the `on` property, with the `click` method. This happens because of the new data structure of the VNode of Vue 3.

Creating components with attribute inheritance

Since Vue 2, it has been possible to use attribute inheritance on components, but in Vue 3, attribute inheritance was made better and with a more reliable API to use in the components.

Attribute inheritance in components is a pattern that provides faster development of custom components based on HTML elements (such as custom inputs, buttons, text wrappers, or links).

In this recipe, we will create a custom input component with attribute inheritance applied directly to the `input` HTML element.

How to do it...

Here, we will create a component that will have a full attribute inheritance on a selected element on the DOM tree:

1. Using the base example from the *Creating the base file* section, create a new file named `component.html` and open it.
2. In the empty `<script>` HTML element, create the constants of the functions that will be used using the object destructuring method, calling the `defineComponent` and `createApp` methods from the `Vue` global constant:

```
const {
  defineComponent,
  createApp,
} = Vue;
```

3. Create a constant named `nameInput`, defined as the `defineComponent` method, passing a JavaScript object as an argument with four properties: `name`, `props`, `template`, and `inheritAttrs`. Then, we define the value of `inheritAttrs` as `false`:

```
const nameInput = defineComponent({
  name: 'NameInput',
  props: {},
  inheritAttrs: false,
  template: ``
});
```

4. In the `props` property, add a property called `modelValue` and define it as `String`:

   ```
   props: {
     modelValue: String,
   },
   ```

5. In the template property, within the template string, we need to do the following:
   - Create a `label` HTML element and add an `input` HTML element as a child.
   - In the `input` HTML element, define the `v-bind` directive as a JavaScript object with the destructed value of `this.$attrs`.
   - Define the variable attribute `value` as the received prop's `modelValue`.
   - Set the `input` attribute `type` as `"text"`.
   - To the `change` event listener, add an anonymous function, which receives an `event` as the argument, and then `emit` an event called `"update:modeValue"` with the payload `event.target.value`:

     ```
     template: `
     <label>
       <input
         v-bind="{
           ...$attrs,
         }"
         :value="modelValue"
         type="text"
         @change="(event) => $emit('update:modelValue',
                                    event.target.value)"
       />
     </label>`
     ```

6. Create a constant named `appComponent`, defined as the `defineComponent` method, passing a JavaScript object as an argument with two properties, `data` and `template`:

   ```
   const component = defineComponent({
     data: () => ({}),
     template: ``,
   });
   ```

7. In the `data` property, define it as a singleton function, returning a JavaScript object with a property named `name`, with the default value as `''`:

```
data: () => ({
  name: ''
}),
```

8. In the template property, within the template string, we need to do the following:
    - Create a `NameInput` component with a `v-model` directive bounded to the `name` data property.
    - Create a `style` attribute with the value `"border:0; border-bottom: 2px solid red;"`.
    - Create a `data-test` attribute with the value `"name-input"`:

    ```
    template: `
    <name-input
      v-model="name"
      style="border:0; border-bottom: 2px solid red;"
      data-test="name-input"
    />`
    ```

9. Create a constant named `app`, and define it as the `createApp` function, passing the `component` constant as the argument. Then, call the `app.component` function, passing as the first argument the name of the component you want to register, and as the second argument the component. Finally, call the `app.mount` function, passing `"#app"` as the argument:

```
const app = createApp(component);
app.component('NameInput', nameInput);
app.mount('#app');
```

How it works...

In Vue 3, in order to create a component, we need to execute the `defineComponent` function, passing a JavaScript object as an argument. This object maintains almost the same component declaration structure as Vue 2. In the examples, we used the same properties, `data`, `methods`, `props`, and `template`, all present in the V2.

We used the `inheritAttrs` property to block the auto application of the attributes to all elements on the components, applying them just to the element with the `v-bind` directive and with the `this.$attrs` object deconstructed.

To register the component in the Vue application, we first created the application with the `createApp` API and then executed the `app.component` function to register the component globally on the application, prior to rendering our application.

Using the reactivity and observable API outside the scope of Vue

In Vue 3, with the exposed APIs, we can use the Vue reactivity and reactive variables without the need to create a Vue application. This enables backend and frontend developers to take full advantage of the Vue `reactivity` API within their application.

In this recipe, we will create a simple JavaScript animation using the `reactivity` and `watch` APIs.

How to do it...

Here, we will create an application using the Vue exposed `reactivity` API to render an animation on the screen:

1. Using the base example from the 'Creating the base file' section, create a new file named `reactivity.html` and open it.
2. In the `<head>` tag, add a new `<meta>` tag with the attribute `chartset` defined as `"utf-8"`:

```
<meta charset="utf-8"/>
```

3. In the `<body>` tag, remove the `div#app` HTML element, and create a `div` HTML element with the `id` defined as `marathon` and the `style` attribute defined as `"font-size: 50px;"`:

```
<div
  id="marathon"
  style="font-size: 50px;"
>
</div>
```

4. In the empty `<script>` HTML element, create the constants of the functions that will be used using the object destructuring method, calling the `reactivity` and `watch` methods from the `Vue` global constant:

   ```
   const {
     reactive,
     watch,
   } = Vue;
   ```

5. Create a constant named `mod`, defined as a function, which receives two arguments, `a` and `b`. This then returns an arithmetic operation, `a` modulus `b`:

   ```
   const mod = (a, b) => (a % b);
   ```

6. Create a constant named `maxRoadLength` with the value `50`. Then, create a constant named `competitor` with the value as the `reactivity` function, passing a JavaScript object as the argument, with the `position` property defined as `0` and `speed` defined as `1`:

   ```
   const maxRoadLength = 50;
   const competitor = reactive({
     position: 0,
     speed: 1,
   });
   ```

7. Create a `watch` function, passing an anonymous function as the argument. Inside the function, do the following:
   - Create a constant named `street`, and define it as an `Array` with a size of `maxRoadLength`, and fill it with `'_'`.
   - Create a constant named `marathonEl`, and define it as the HTML DOM node, `#marathon`.
   - Select the element on the `street` in the array index of `competitor.position` and define it as `"🏃"` if the `competitor.position` number is even, or `"🚶"` if the number is odd.
   - Define `marathonEl.innertHTML` as `""` and `street.reverse().join('')`:

The emojis used in this recipe are **Person Running** and **Person Walking**. The emoji image may vary depending on your OS. The images presented in this recipe are the emojis for the Apple OS.

```
watch(() => {
  const street = Array(maxRoadLength).fill('_');
  const marathonEl = document.getElementById('marathon');
  street[competitor.position] = (competitor.position % 2 === 1)
    ? '🏃'
    : '🚶';

  marathonEl.innerHTML = '';
  marathonEl.innerHTML = street.reverse().join('');
});
```

8. Create a `setInterval` function, passing an anonymous function as the argument. Inside the function, define `competitor.position` as the `mod` function, passing `competitor.position` plus `competitor.speed` as the first argument, and `maxRoadLength` as the second argument:

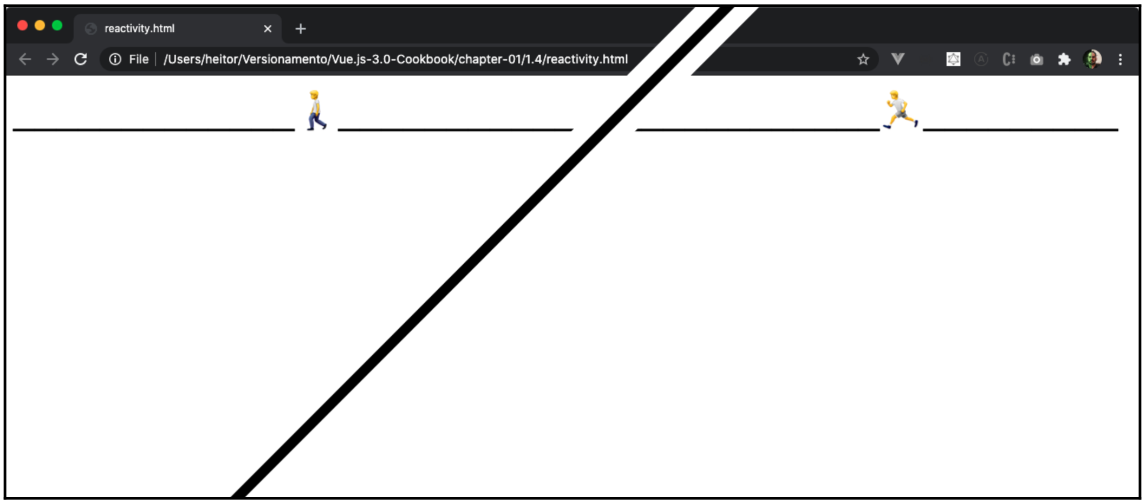
```
setInterval(() => {
  competitor.position = mod(competitor.position +competitor.speed,
    maxRoadLength)
}, 100);
```

How it works...

Using the exposed `reactive` and `watch` APIs from Vue, we were able to create an application with the reactivity present in the Vue framework, but without the use of a Vue application.

First, we created a reactive object, `competitor`, that works in the same way as the Vue `data` property. Then, we created a `watch` function, which works in the same way as the `watch` property, but is used as an anonymous function. In the `watch` function, we made the road for the competitor to run on, and created a simple animation, using two different emojis, changing it based on the position on the road, so that it mimics an animation on the screen.

Finally, we printed the current runner on the screen and created a `setInterval` function of every `100ms` to change the position of the competitor on the road:


reactivity.html
File | /Users/heitor/Versionamento/Vue.js-3.0-Cookbook/chapter-01/1.4/reactivity.html

Creating a component using the composition API

The composition API is a new way to write Vue components, based on the use of functions to compose the component, and it makes the organization and reusability of the code better.

This method is inspired by React Hooks and introduces the technique of creating a special function to compose the applications that can be shared without the need to be inside the Vue application because of the use of the exposed Vue APIs.

In this recipe, we will learn how to create an external function that fetches the user's geolocation and displays that data on the screen using the composition API.

How to do it...

Here, we will create a component using the composition API, which will fetch the user GPS position and show that information on the screen:

1. Using the base example from the 'Creating the base file' section, create a new file named `component.html` and open it.
2. In the empty `<script>` HTML element, create the constants of the functions that will be used using the object destructuring method, calling the `createApp`, `defineComponent`, `setup`, `ref`, `onMounted`, and `onUnmounted` methods from the `Vue` global constant:

```
const {
  createApp,
  defineComponent,
  setup,
  ref,
  onMounted,
  onUnmounted,
} = Vue;
```

3. Create a `fetchLocation` function and, inside this, create a `let` variable named `watcher`. Then, create a constant named `geoLocation` and define it as `navigator.geolocation`. Next, create a constant named `gpsTime` and define it as the `ref` function, passing the `Date.now()` function as the argument. Finally, create a constant named `coordinates` and define it as the `ref` function, passing a JavaScript object as the argument, with the properties `accuracy`, `latitude`, `longitude`, `altitude`, `altitudeAccuracy`, `heading`, and `speed` defined as `0`:

```
function fetchLocation() {
  let watcher;
  const geoLocation = navigator.geolocation;
  const gpsTime = ref(Date.now());
  const coordinates = ref({
    accuracy: 0,
    latitude: 0,
    longitude: 0,
    altitude: 0,
    altitudeAccuracy: 0,
    heading: 0,
    speed: 0,
  });
}
```

4. Then, inside the `fetchLocation` function, following the creation of the constants, create a function named `setPosition` with a parameter named `payload`. Inside the function, define `gpsTime.value` as the `payload.timestamp` argument and `coordinates.value` as the `payload.coords` argument:

```
function setPosition(payload) {
  gpsTime.value = payload.timestamp
  coordinates.value = payload.coords
}
```

5. Following creation of the `setPosition` function, call the `onMounted` function, passing an anonymous function as the argument. Inside the function, check whether the browser has the `geoLocation` API available, and define `watcher` as the `geoLocation.watchPostion` function, passing the `setPosition` function as the argument:

```
onMounted(() => {
  if (geoLocation) watcher =
geoLocation.watchPosition(setPosition);
});
```

6. After calling the `onMounted` function, create an `onUnmounted` function passing an anonymous function as the argument. Inside the function, check whether `watcher` is defined and then execute the `geoLocation.clearWatch` function, passing `watcher` as the argument:

```
onUnmounted(() => {
  if (watcher) geoLocation.clearWatch(watcher);
});
```

7. Finally, in the `fetchLocation` function, return a JavaScript object, and as the properties/values define, pass the `coordinates` and `gpsTime` constants:

```
return {
  coordinates,
  gpsTime,
};
```

8. Create a constant named `appComponent` and define it as the `defineComponent` function, passing a JavaScript object with the properties `setup` and `template` as the argument:

```
const appComponent = defineComponent({
  setup() {},
  template: ``
});
```

9. In the `setup` function, create a constant, which is an object destructuring with the properties `coordinates` and `gpsTime` of the `fetchLocation` function:

```
setup() {
  const {
    coordinates,
    gpsTime,
  } = fetchLocation();
}
```

10. Inside the `setup` function, create another constant named `formatOptions`, and define it as a JavaScript object with the properties `year`, `month`, `day`, `hour`, and `minute` as `'numeric'`. Then, define the property `hour12` as `true`:

```
const formatOptions = {
    year: 'numeric',
    month: 'numeric',
    day: 'numeric',
    hour: 'numeric',
    minute: 'numeric',
    hour12: true,
  };
```

11. Following the creation of the `formatOptions` constant, create a constant named `formatDate` and define it as a function, which receives a parameter named `date`. Then, return a new `Intl.DateTimeFormat` function, passing `navigator.language` as the first argument, and the `formatOption` constant as the second argument. Then, prototype chain the `format` function, passing the `date` parameter:

```
const formatDate = (date) => (new
  Intl.DateTimeFormat(navigator.language,
     formatOptions).format(date));
```

12. Finally, at the end of the `setup` function, return a JavaScript object with the properties defined as `coordinates`, `gpsTime`, and `formatDate` constants:

    ```
    return {
      coordinates,
      gpsTime,
      formatDate
    };
    ```

13. In the `template` property, do the following:
    - Create an `h1` HTML element with the text `"My Geo Position at {{ formatDate(new Date(gpsTime) }}"`.
    - Create a `ul` HTML element and add three `li` HTML elements as children.
    - In the first child element, add the text `"Latitude: {{ coordinates.latitude }}"`.
    - In the second child element, add the text `"Longitude: {{ coordinates.longitude }}"`.
    - In the third child element, add the text `"Altitude: {{ coordinates.altitude }}"`:

    ```
    template: `
      <h1>My Geo Position at {{formatDate(new
                                Date(gpsTime))}}</h1>
      <ul>
        <li>Latitude: {{ coordinates.latitude }}</li>
        <li>Longitude: {{ coordinates.longitude }}</li>
        <li>Altitude: {{ coordinates.altitude  }}</li>
      </ul>
    `
    ```

14. Finally, call the `createApp` function, passing the `appComponent` constant as an argument. Then, prototype chain the `mount` function, and, as an argument of the function, pass the `div` HTML element `id` attribute, `("#app")`:

    ```
    createApp(appComponent)
      .mount('#app');
    ```

How it works...

In this recipe, first, we imported the exposed APIs - `createApp`, `defineComponent`, `setup`, `ref`, `onMounted`, and `onUnmounted`, – as constants, which we will use to create the component. Then, we created the `fetchLocation` function, which has the responsibility of getting the user's geolocation data and returning it as reactive data that can be automatically updated when the user changes their location.

The ability to fetch the user GPS positions was possible because of the `navigator.geolocation` API present on modern browsers, which are able to fetch the user's current GPS position. Using this data provided by the browser, we were able to use it to define the variables created with the Vue `ref` APIs.

We created the component using the `setup` function of the Vue object declaration, so the rendering knows that we are using the new composition API as the component creation method. Inside the `setup` function, we imported the dynamic variables of the `fetchLocation` function and created a method that formats the date to use as a filter on the template.

Then we returned the imported variables and the filter, so they can be used on the template section. In the template section, we created a title adding the time of the last GPS position, used the filter to format it, and created a list of the user's latitude, longitude, and altitude.

Finally, we created the application using the `createApp` exposed API and mounted the Vue application.

See also

You can find more information about `Navigator.geolocation` at `https://developer.mozilla.org/en-US/docs/Web/API/Navigator/geolocation`.

You can find more information about `Intl.DateTimeFormat` at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl/DateTimeFormat`.

2
Introducing TypeScript and the Vue Ecosystem

TypeScript is a new Vue-based language, fully supported on **Vue 3**. It is now possible to use typed JSX (also know as TSX), type annotation, static verification of the code, and much more.

The Vue ecosystem is getting bigger each day, so to help us, the Vue team has developed some tools to improve project handling and management. Those tools are Vue CLI and Vue UI, which today are the main tools for local Vue development.

The Vue CLI tool is the beginning of every project; with it, you will be able to select the basic features or just a preset you had made, to create a new Vue project. After a project is created, you can use Vue UI to manage the project, add new features, check the status of the project, and do almost everything you previously needed to do in the command-line interface (CLI), with the addition of more features.

In these chapters, you learn more about TypeScript as a superset on JavaScript and how to use the power of the Vue CLI tool and Vue UI together to get a whole application up and running.

In this chapter, we'll cover the following recipes:

- Creating a TypeScript project
- Understanding TypeScript
- Creating your first TypeScript class
- Creating your first project with Vue CLI
- Adding plugins to a Vue CLI project with Vue UI
- Adding TypeScript to a Vue CLI project

- Creating your first TypeScript Vue component with `vue-class-component`
- Creating a custom mixin with `vue-class-component`
- Creating a custom function decorator with `vue-class-component`
- Adding custom hooks to `vue-class-component`
- Adding `vue-property-decorator` to `vue-class-component`

Technical requirements

In this chapter, we will be using **Node.js**, **Vue CLI**, and **TypeScript**.

Attention, Windows users—you need to install an `npm` package called `windows-build-tools` to be able to install the following required packages. To do it, open PowerShell as administrator and execute the following command:
`> npm install -g windows-build-tools`.

To install the **Vue CLI** tool, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install -g @vue/cli @vue/cli-service-global
```

To install **TypeScript**, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install -g typescript
```

Creating a TypeScript project

TypeScript is a typed superset of JavaScript that, when compiled, gives us plain JavaScript code. It seems like a new language, but in the end, it's still JavaScript.

What is the advantage of using TypeScript? The main advantage is the typed syntax, which helps with static checking and code refactoring. You can still use all the JavaScript libraries and program with the latest ECMAScript features out of the box.

When compiled, TypeScript will deliver a pure JavaScript file that can run on any browser, Node.js, or any JavaScript engine that is capable of executing ECMAScript 3 or newer versions.

Getting ready

To start our project, we will need to create an `npm` project. Open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm init -y
```

You also need to install TypeScript, so open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install typescript --only=dev
```

How to do it...

With our environment ready, we will need to start our TypeScript project. Let's create a `.ts` file and compile it:

1. To start our TypeScript project, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > tsc --init
   ```

   This will create a `tsconfig.json` file inside our folder. This is a compiler settings file. Here, you can define the target, which JavaScript libraries will be available on the development, the target ECMAScript version, the module generation, and much more.

   TIP When developing for the web, don't forget to add the **Document Object Model** (**DOM**) to the libraries on the `compilerOption` property inside the `tsconfig.json` file so that you can have access to the window and document object when developing.

2. Now, we need to create our `index.ts` file. Let's create some simple code inside the `index.ts` file that will log a math calculation in your terminal:

   ```
   function sum(a: number, b: number): number {
       return a + b;
   }

   const firstNumber: number = 10;

   const secondNumber: number = 20;

   console.log(sum(firstNumber, secondNumber));
   ```

This function receives two parameters, `a` and `b`, which both have their type set to `number`, and the function is expected to return a `number`. We made two variables, `firstNumber` and `secondNumber`, which in this case are both set to a `number` type—`10` and `20` respectively—so, it's valid to pass to the function. If we had set it to any other type such as a string, Boolean, float, or an array, the compiler would have thrown an error about the static type checking on the variable and the function execution.

3. Now, we need to compile this code to a JavaScript file. Open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> tsc ./index.ts
```

After the compilation, we can see the final file in `index.js`. If we look inside the file, the final code will be similar to this:

```
function sum(a, b) {
    return a + b;
}
var firstNumber = 10;
var secondNumber = 20;
console.log(sum(firstNumber, secondNumber));
```

You may be wondering: *where are my types?* As ECMAScript is a dynamic language, the types of TypeScript exist only at the superset level, and won't be passed down to the JavaScript file.

Your final JavaScript will be in the form of a transpiled file, with the configurations defined in the `tsconfig.json` file.

How it works...

When we create our TypeScript project, a file named `tsconfig.json` is created inside our folder. This file coordinates all the rules on the compiler and the static type checking during the development process. All developments are based on the rules defined in this file. Each environment depends on specific rules and libraries that need to be imported.

When developing, we can assign types directly to constants, variables, function parameters, returns, and much more. These types of definitions can prevent basic type errors and code refactoring.

After the development is done and we compile the project, the final product will be a pure JavaScript file. This file won't have any type of checking, due to the dynamic type of JavaScript.

This JavaScript file gets transpiled to the target model and defined on the configuration file, so we can execute it without any problems.

See also

You can find more information about TypeScript at `https://www.typescriptlang.org/docs/home.html`.

There is a guide to migrating from JavaScript at `https://www.typescriptlang.org/docs/handbook/migrating-from-javascript.html`.

A 5-minute lesson for TypeScript can be found at `https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html`.

Understanding TypeScript

TypeScript is a type-based language. Much of its power comes with the ability to use static code analysis with JavaScript. This is possible thanks to the tools that exist inside the TypeScript environment.

These tools include the compiler, which can provide static analysis during development and after compilation, and the ECMAScript transpiler, which can make your code available to run on almost any JavaScript engine.

Let's get to know more about the language, and how it works.

Getting ready

To start, we will need to create an `npm` project. Open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm init -y
```

You also need to install TypeScript, so open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install typescript --only=dev
```

Types

The main feature we get from using TypeScript is the **types**. In this section, we will learn about types, how to declare them, and how to use them.

These are some of the basic types in a statically typed language:

- String
- Number
- Boolean
- Arrays
- Tuple
- Enum
- Any
- Void
- Objects

Let's talk about some of these types and show how they can be used in TypeScript.

String

All the textual data on JavaScript will be treated as a **string**. To declare a string, we always need to surround it with double `(")` or single `(')` quotes, or the ``(`)`` grave accent, commonly known as a template string.

Declaring template strings inside text is not a problem with TypeScript. Template strings are a feature in ECMAScript that made it possible to add a variable inside a string without the need for concatenation:

```
const myText: string = 'My Simple Text';
const myTextAgain: string = "My Simple Text";
const greeting: string = `Welcome back ${myName}!`;
```

Number

In JavaScript, all numbers are floating-point values. In TypeScript, it's the same. Those numbers get a **number** type. In addition to the hexadecimal and decimal numbers, the binary and octal literals that were introduced in ECMAScript 2015 are treated like numbers too:

```
const myAge: number = 31;
const hexNumber: number = 0xf010d;
const binaryNumber: number = 0b1011;
const octalNumber: number = 0o744;
```

Boolean

The most basic type in the programming languages is the **boolean** values—a simple 1 or 0, and true or false. This is called a **boolean**:

```
const isTaskDone: boolean = false;
const isGreaterThen: boolean = 10 > 5;
```

Arrays

A group of elements in most of the languages is commonly called an **array**. In TypeScript, we can declare it in two different ways.

The most simple way is just to declare the type of the element followed by `[]` *(*square brackets*)* to denote that it is an **array** of the declared type:

```
const primeNumbers: number[] = [1, 3, 5, 7, 11];
```

Or, you can declare generically, using the `Array<type>` declaration. This is not the most common way used, but, depending on the code you are developing, you may need to use it:

```
const switchInstructions: Array<boolean> = [true, false, false, true];
```

Tuple

Tuples are a type of variable that has a specific structure. Structurally, a tuple is an array of two elements; both are a known type by the compiler and the user, but those elements don't need to have the same type:

```
let person: [string, number];
person = ['Heitor', 31];

console.log(`My name is ${person[0]} and I am ${person[1]} years old`);
```

If you try to access an element outside of the known indices, you will get an error.

Enum

Enums are similar to JavaScript objects, but they have some special attributes that help in the development of your application. You can have a friendly name for numeric values or a more controlled environment for the constants on the variables a function can accept.

A numeric enum can be created without any declaration. By doing this, it will start with the initial values of `0` and finish with the value of the final index number; or, you can get the name of the enum, passing the index of the enum value:

```
enum ErrorLevel {
    Info,
    Debug,
    Warning,
    Error,
    Critical,
}

console.log(ErrorLevel.Error); // 3
console.log(ErrorLevel[3]); // Error
```

Or, an enum can be declared with values. It can be an initial declaration that the TypeScript compiler will interpret the rest of the elements as an increment of the first one, or an individual declaration:

```
enum Color {
    Red = '#FF0000',
    Blue = '#0000FF',
    Green = '#00FF00',
}

enum Languages {
    JavaScript = 1,
    PHP,
    Python,
    Java = 10,
    Ruby,
    Rust,
    TypeScript,
}

console.log(Color.Red) // '#FF0000'
console.log(Languages.TypeScript) // 13
```

Any

As JavaScript is a dynamic language, TypeScript needed to implement a type that has no defined value, so it implemented the **any** type. The most used case for the any type any is when using values that came from a third-party library. In that case, we know that we are dropping the type checking:

```
let maybeIs: any = 4;
maybeIs = 'a string?';
maybeIs = true;
```

The main use of the any type is when you are upgrading a legacy JavaScript project to TypeScript, and you can gradually add the types and validations to the variables and functions.

Void

As the opposite of any, **void** is the absence of the type at all. The most used case is with functions that won't return any values:

```
function logThis(str: string): void{
    console.log(str);
}
```

Using void to type a variable is useless because it only can be assigned to undefined and null.

Objects

An **object** in TypeScripts has a special form of declaring because it can be declared as an interface, as a direct **object,** or as a type of its own.

Declaring an object as an interface, you have to declare the interface before using it, all the attributes must be passed, and the types need to be set:

```
interface IPerson {
    name: string;
    age: number;
}

const person: IPerson = {
    name: 'Heitor',
    age: 31,
};
```

Using objects as direct inputs is sometimes common when passing to a function:

```
function greetingUser(user: {name: string, lastName: string}) {
    console.log(`Hello, ${user.name} ${user.lastName}`);
}
```

And finally, they are used for declaring a type of object and reusing it:

```
type Person = {
    name: string,
    age: number,
};

const person: Person = {
    name: 'Heitor',
    age: 31,
};

console.log(`My name is ${person.name}, I am ${person.age} years old`);
```

Functions

In TypeScript, one of the most difficult types to declare is a **function**. It can get very complex in a just simple concatenation of the functional chain.

Declaring a function in TypeScript is a composition of the parameters that the function will receive and the final type that the function will return.

You can declare a simple function inside a constant, like this:

```
const sumOfValues: (a:number, b:number): number = (a: number, b: number):
number => a + b;
```

A more complex function declared inside a constant can be declared like this:

```
const complexFunction: (a: number) => (b:number) => number = (a: number):
(b: number) => number => (b: number): number => a + b;
```

When declaring a function as a normal function, the way to type it is almost the same as in a constant way, but you don't need to declare that the functions are a function. Here is an example:

```
function foo(a: number, b:number): number{
    return a + b;
}
```

Interfaces

TypeScript checks that the values of variables are the correct type and the same principle is applied to classes, objects, or contracts between your code. This is commonly known as "duck typing" or "structural sub-typing". Interfaces exist to fill this space and define these contracts or types.

Let's try to understand an **interface** with this example:

```
function greetingStudent(student: {name: string}){
    console.log(`Hello ${student.name}`);
}

const newStudent = {name: 'Heitor'};

greetingStudent(newStudent);
```

The function will know that the object has the property name on it and that it's valid to call it.

We can rewrite it with the interface type for better code management:

```
interface IStudent {
    name: string;
    course?: string;
    readonly university: string;
}

function greetingStudent(student: IStudent){
    console.log(`Hello ${student.name}`);
    if(student.course){
        console.log(`Welcome to the ${student.course}` semester`);
    }
}

const newStudent: IStudent = { name: 'Heitor', university: 'UDF' };

greetingStudent(newStudent);
```

As you can see, we have a new property called `course` that has a `?` declared on it. This symbolizes that this property can be nulled or undefined. It's called an optional property.

There is a property with a read-only attribute declared. If we try to change after it's declared on the variable creation, we will receive a compile error because it makes the property read-only.

Decorators

A new feature was introduced in ECMAScript 6—classes. With the introduction of these, the usage of decorators was made possible on the JavaScript engine.

Decorators provide a way to add both annotations and meta-programming syntax to class declarations and its members. As it's in a final state of approval on the TC-39 committee (where **TC** stands for **Technical Committee**), the TypeScript compiler already has this available to be used.

To enable it, you can set the flags on the `tsconfig.json` file:

```
{
    "compilerOptions": {
        "target": "ES5",
        "experimentalDecorators": true
    }
}
```

Decorators are a special kind of declaration that can be attached to a class, method, accessor property, or parameter. They are used in the form of `@expression`, where the expression is a function that will be called at runtime.

An example of a decorator that can be applied to a class can be seen in the following code snippet:

```
function classSeal(constructor: Function) {
    Object.seal(constructor);
    Object.seal(constructor.prototype);
}
```

When you create this function, you are saying that the object of the constructor and the prototype of it will be sealed.

To use it inside a class is very simple:

```
@classSeal
class Animal {
    sound: string;
    constructor(sound: string) {
        this.sound = sound;
    }
    emitSound() {
        return "The animal says, " + this.sound;
    }
}
```

These are just some examples of decorators and their powers to help you with the development of **object-oriented programming** (**OOP**) with TypeScript.

In conclusion

In summary, types are just a way to make our life easier in the process of development with TypeScript and JavaScript.

Because JavaScript is a dynamic language and doesn't have a static type, all the types and interfaces declared in TypeScript are strictly used just by TypeScript. This helps the compiler catch errors, warnings, and the language server to help the **integrated development environment** (**IDE**) on the development process to analyze your code as it is being written.

This is a basic introduction to TypeScript, covering the basics of the typed language, and how to understand and use it. There is much more to learn about its use, such as generics, modules, namespaces, and so on.

With this introduction, you can understand how the new **Vue 3** core works and how to use the basics of TypeScript in your project, and take advantage of the typed language on your project.

There is always more knowledge to find on TypeScript, as it is a growing "language" on top of JavaScript and has a growing community.

Don't forget to look at the TypeScript documentation to find out more about it and how it can improve your code from now on.

See also

You can find more information about TypeScript basic types at `https://www.typescriptlang.org/docs/handbook/basic-types.html`.

You can find more information about TypeScript functions at `https://www.typescriptlang.org/docs/handbook/functions.html`.

You can find more information about TypeScript enums at `https://www.typescriptlang.org/docs/handbook/enums.html`.

You can find more information about TypeScript advanced types at `https://www.typescriptlang.org/docs/handbook/advanced-types.html`.

You can find more information about TypeScript decorators at `https://www.typescriptlang.org/docs/handbook/decorators.html`.

View a cheatsheet on TypeScript types at `https://rmolinamir.github.io/typescript-cheatsheet/#types`.

Creating your first TypeScript class

In TypeScript, there is no main paradigm in which you write your program. You can choose between object-oriented, structural, or event functional.

In most cases, you will see an OOP paradigm being used. In this recipe, we will learn about creating a class inside TypeScript, its inheritance, the interface, and other properties that can be used inside the code.

Getting ready

To start our project, we will need to create an `npm` project. To do this, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm init -y
```

You also need to install TypeScript. To do this, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install typescript --only=dev
```

How to do it...

When writing a class inside a TypeScript file, we first need to have in mind what this class will do, what this class can be for, how it can be extended by another class through inheritance, and how it can be affected in the process.

Imagine that we have a basic `Animal` class. This class can have some basic properties such as its `name`, whether it produces a `sound`, its `family`, and the basic `food chain` this animal eats.

1. Let's start with the basics of the process, the `food chain`. We need to make sure that it's an innumerable list, and that each file that is using it will have the same value at the end. We just need to call a constant variable:

```
export enum FoodChainType {
    Carnivorous = 'carnivorous',
    Herbivorous = 'herbivorous',
    Omnivorous = 'omnivorous',
}
```

2. Now, we want to make the basic `interface` for our animal. We know that our animal has a `name`, can produce a `sound`, can be part of a `family`, and be in a `food chain` category. Using an interface in a class, we make a contract between the class and what will be exposed, helping in the development process:

```
interface IAnimal {
    name: string;
    sound?: string;
    family: string;
    foodChainType: FoodChainType;
}
```

3. With all that settled, we can make our `Animal` class. Each class can have its constructor. The class constructor can be simple, containing just some variables as arguments, or can be more complex and have an object as an argument. If your constructor will have any parameters, an interface or declaring the type of each parameter is needed. In this case, our constructor will be an object and will have only one parameter that is the same as the `Animal`, so it will extend the `IAnimal` interface:

```
interface IAnimalConstructor extends IAnimal {
}
```

4. Now, to make our class, we have declared the interfaces and enums that will be used. We will start by declaring that the class will implement the `IBasicAnimal` interface. To do this, we need to add some public elements that our class will have and declare those too. We will need to implement the functions to show what animal it is and what sound it makes. Now, we have a basic class that includes all the attributes for our animal. It has separate interfaces for the class and the constructors. The enum for the food chain is declared in a human-readable way, so the JavaScript imports of this library can execute without any problems:

```
interface IBasicAnimal extends IAnimal {
  whoAmI: () => void;
  makeSound: () => void;
}

export class Animal implements IBasicAnimal {
  public name: string;
  public sound: string;
  public family: string;
  public foodChainType: FoodChainType;

  constructor(params: IAnimalConstructor) {
    this.name = params.name;
    this.sound = params.sound || '';
    this.family = params.family;
    this.foodChainType = params.foodChainType;
  }

  public whoAmI(): void {
    console.log(`I am a ${this.name}, my family is ${this.family}.
    My diet is ${this.foodChainType}.`);
    if (this.sound) {
      console.log([...Array(2).fill(this.sound)].join(', '));
    }
  }

  public makeSound(): void {
    console.log(this.sound);
  }
}
```

5. Let's extend this class with a few lines of code and transform this `Animal` into a `Dog`:

```
import {Animal, FoodChainType} from './Animal';

class Dog extends Animal {
  constructor() {
    super({
      name: 'Dog',
      sound: 'Wof!',
      family: 'Canidae',
      foodChainType: FoodChainType.Carnivorous,
    });
  }n
}
```

This is a simple way of extending a parent class and using the parent's definition of the child to compose a new class with almost the same interface as the parent.

How it works...

Classes in TypeScript work the same as other classes in languages such as Java or C#. The compiler evaluates these common principles during development and compilation.

In this case, we made a simple class that had some public properties that were inherent by the children's classes. These variables were all readable and can be mutated.

There's more...

In TypeScript, we have a wide range of possible uses for classes, such as abstract classes, special modifiers, and using classes as interfaces. We've just covered the basics of the classes here to give us a good starting knowledge base. If you want to go deeper, the TypeScript documentation is very helpful and has a lot of examples that can help in the process of learning.

See also

You can find more information about TypeScript classes at `https://www.typescriptlang.org/docs/handbook/classes.html`.

View a cheatsheet on TypeScript classes at `https://rmolinamir.github.io/typescript-cheatsheet/#classes`.

Creating your first project with Vue CLI

When the Vue team realized that developers were having problems creating and managing their applications, they saw an opportunity to create a tool that could help developers around the world. The Vue CLI project was born.

The Vue CLI tool is a CLI tool that is used in terminal commands, such as Windows PowerShell, Linux Bash, or macOS Terminal. It was created as a starting point for the development of Vue, where developers can start a project and manage and build it smoothly. The focus of the Vue CLI team when developing was to give developers the opportunity to have more time to think about the code and spend less time on the tooling needed to put their code into production, adding new plugins or a simple `hot-module-reload`.

The Vue CLI tool is tweaked in such a way that there is no need to eject your tooling code outside the CLI before putting it into production.

When version 3 was released, the Vue UI project was added to the CLI as the main function, transforming the CLI commands into a more complete visual solution with lots of new additions and improvements.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To create a Vue CLI project, follow these steps:

1. We need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > vue create my-first-project
   ```

2. The CLI will ask some questions that will help with the creation of the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *Spacebar* to select an option:

   ```
   ? Please pick a preset: (Use arrow keys)
     default (babel, eslint)
   ❯ Manually select features
   ```

3. There are two methods for starting a new project. The default method is a basic `babel` and `eslint` project without any plugin or configuration, and the `Manually` mode, where you can select more modes, plugins, linters, and options. We will go for `Manually`.
4. Now, we are asked about the features that we will want on the project. Those features are some Vue plugins such as Vuex or Router (Vue-Router), testers, linters, and more:

   ```
   ? Check the features needed for your project: (Use arrow keys)
    ❯ Babel
      TypeScript
      Progressive Web App (PWA) Support
      Router
      Vuex
      CSS Pre-processors
    ❯ Linter / Formatter
      Unit Testing
    ❯ E2E Testing
   ```

5. For this project, we will choose `CSS Pre-processors` and press *Enter* to continue:

   ```
   ? Check the features needed for your project: (Use arrow keys)
   ❯ Babel
     TypeScript
     Progressive Web App (PWA) Support
     Router
     Vuex
   ❯ CSS Pre-processors
   ```

```
❯ Linter / Formatter
  Unit Testing
  E2E Testing
```

6. It's possible to choose the main **Cascading Style Sheets** (**CSS**) preprocessors to be used with Vue—`Sass`, `Less`, and `Stylus`. It's up to you to choose which fits the most and is best for you:

```
? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules
  are supported by default): (Use arrow keys)
  Sass/SCSS (with dart-sass)
  Sass/SCSS (with node-sass)
  Less
❯ Stylus
```

7. It's time to format your code. You can choose between `AirBnB`, `Standard`, and `Prettier` with a basic config. Those rules that are imported inside the `ESLint` can be always customized without any problem and there is a perfect one for your needs. You know what is best for you:

```
? Pick a linter / formatter config: (Use arrow keys)
  ESLint with error prevention only
❯ ESLint + Airbnb config
  ESLint + Standard config
  ESLint + Prettier
```

8. After the linting rules are set, we need to define when they are applied to your code. They can be either applied on save or fixed on commit:

```
? Pick additional lint features: (Use arrow keys)
  Lint on save
❯ Lint and fix on commit
```

9. After all those plugins, linters, and processors are defined, we need to choose where the settings and configs are stored. The best place to store them is on a dedicated file, but it is also possible to store then on the `package.json` file:

```
? Where do you prefer placing config for Babel, ESLint, etc.? (Use
arrow keys)
❯ In dedicated config files
  In package.json
```

10. Now, you can choose if you want to make this selection a preset for future projects so that you don't need to reselect everything again:

    ```
    ? Save this as a preset for future projects? (y/N) n
    ```

11. The CLI will automatically create the folder with the name you set in the first step, install everything, and configure the project.

You are now able to navigate and run the project. The basic commands on Vue CLI projects are as follows:

- `npm run serve`—For running a development server locally
- `npm run build`—For building and minifying the application for deployment
- `npm run lint`—To execute the lint on the code

You can execute those commands via the Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows).

There's more...

The CLI has a tool inside it called Vue UI that helps in the process of managing your Vue projects. This tool will take care of the project dependencies, plugins, and configurations.

Each `npm` script in the Vue UI tool is named as Tasks, and on those tasks, you can get real-time statistics such as—for example—the size of the assets, modules, and dependencies; numbers of errors or warnings; and more deep networking data for fine-tuning your application.

To enter the Vue UI interface, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue ui
```

See also

Find more information about the Vue CLI project at `https://cli.vuejs.org/guide/`.

Find more information about the development of Vue CLI plugins at `https://cli.vuejs.org/dev-guide/plugin-dev.html`.

Adding plugins to a Vue CLI project with Vue UI

The Vue UI tool is one of the most powerful additional tools for Vue development. It makes a developer's life easier, and at the same time can help manage the Vue projects.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

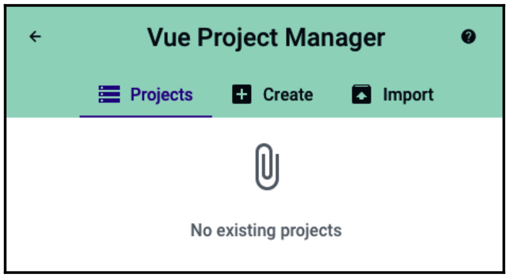
How to do it...

First, we need to create our Vue CLI project. To find how to create a Vue CLI project, please check the 'Creating your first project with Vue CLI' recipe. We can use the one we created in the last recipe or start a new one. Now, follow the instructions to add a plugin:

1. Open the Vue UI interface. Open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > vue ui
   ```

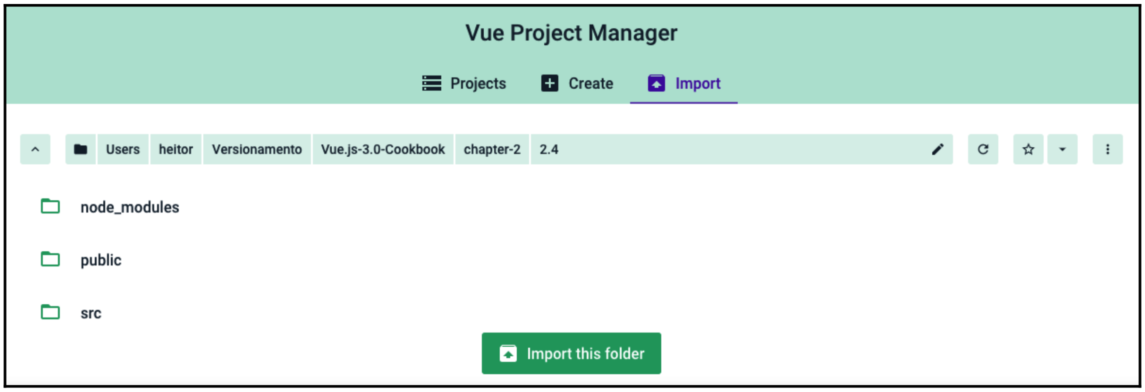
2. A new browser window will appear, with the **Vue UI** interface:



No existing projects

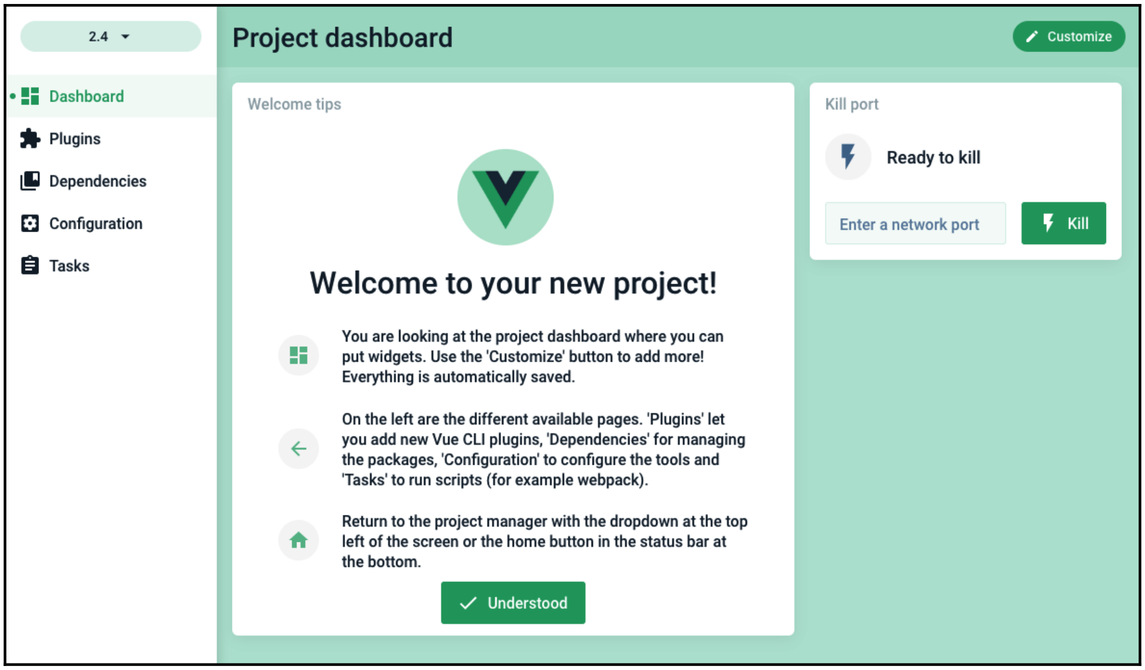
Here, you can list your projects, create a new project, or import an existing one.

3. Now, we will import the one we created:



You need to find the folder that you created and click on **Import this folder**.

4. After the folder is imported, the default **Dashboard** of the project will appear:



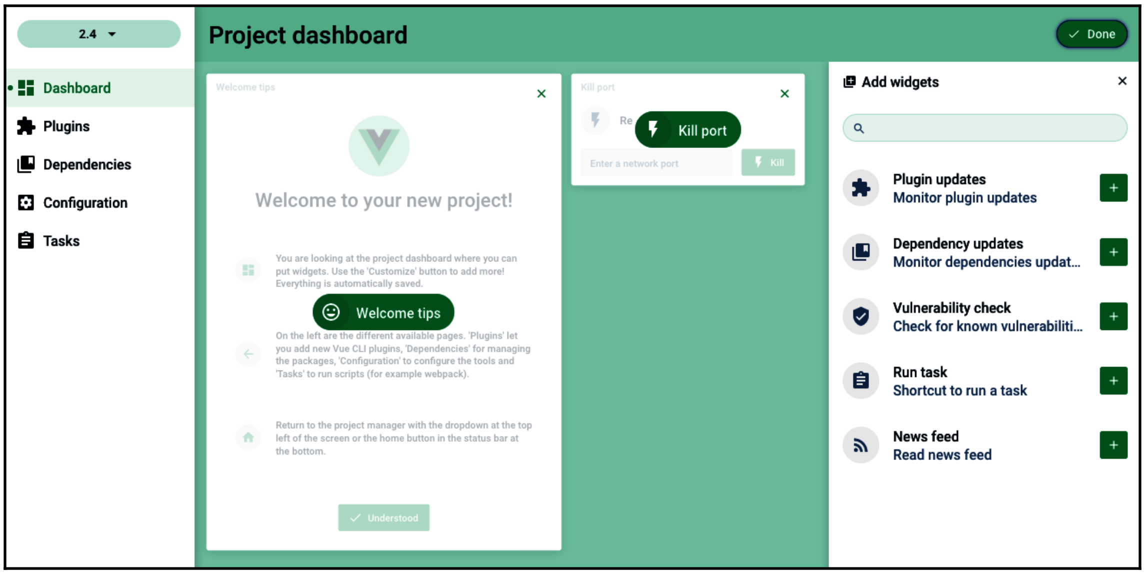
Welcome to your new project!

You are looking at the project dashboard where you can put widgets. Use the 'Customize' button to add more! Everything is automatically saved.

On the left are the different available pages. 'Plugins' let you add new Vue CLI plugins, 'Dependencies' for managing the packages, 'Configuration' to configure the tools and 'Tasks' to run scripts (for example webpack).

Return to the project manager with the dropdown at the top left of the screen or the home button in the status bar at the bottom.

Here, it's possible to customize your **Dashboard**, adding new widgets, by clicking on the **Customize** button on the top:

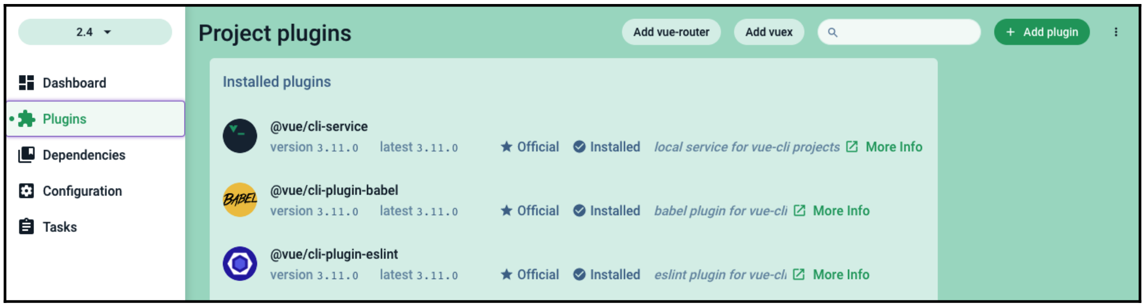


You are looking at the project dashboard where you can put widgets. Use the 'Customize' button to add more! Everything is automatically saved.

On the left are the different available pages. 'Plugins' let you add new Vue CLI plugins, 'Dependencies' for managing the packages, 'Configuration' to configure the tools and 'Tasks' to run scripts (for example webpack).

Return to the project manager with the dropdown at the top left of the screen or the home button in the status bar at the bottom.

5. To add a new plugin, you must click on the **Plugins** menu in the left-hand sidebar:



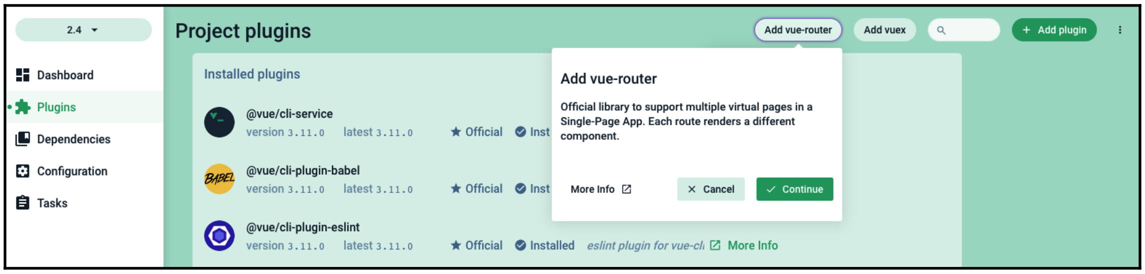
@vue/cli-service
version 3.11.0 latest 3.11.0 ★ Official Installed *local service for vue-cli projects* More Info

@vue/cli-plugin-babel
version 3.11.0 latest 3.11.0 ★ Official Installed *babel plugin for vue-cli* More Info

@vue/cli-plugin-eslint
version 3.11.0 latest 3.11.0 ★ Official Installed *eslint plugin for vue-cli* More Info

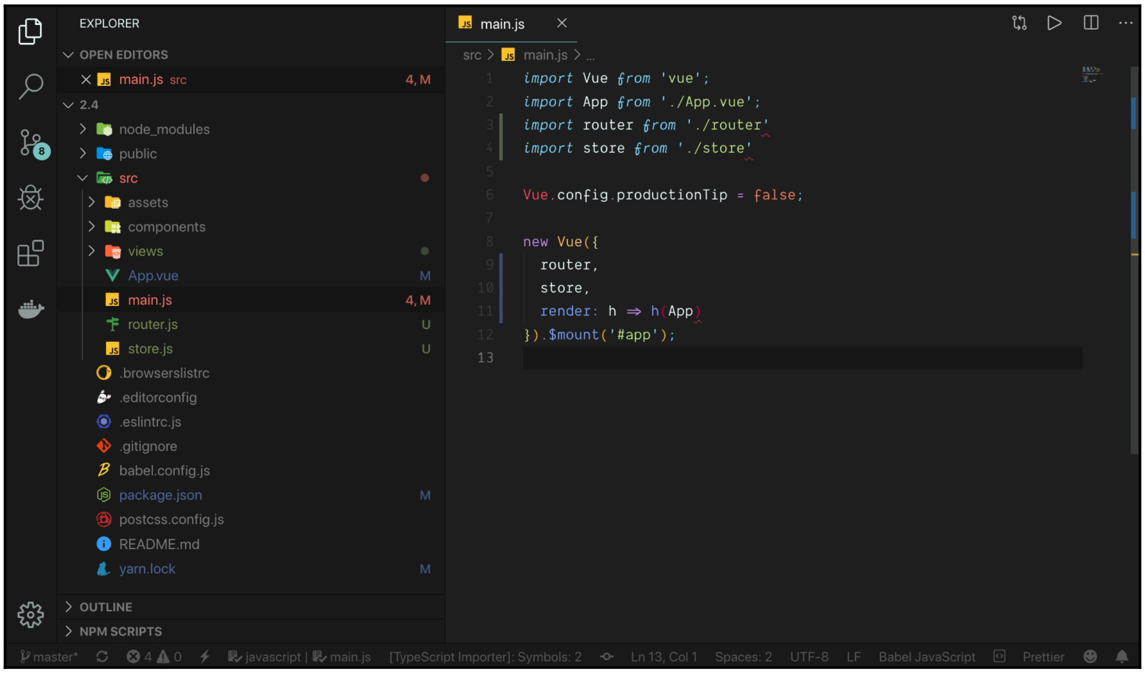
The base plugins that you added on the Vue CLI tool will be already listed here.

6. Now, we will add the base Vue ecosystem plugins—**vuex** and **vue-router**:



Official library to support multiple virtual pages in a Single-Page App. Each route renders a different component.

7. If you check your code, you will see that the `main.js` file was changed, and the `vuex (store)` and `vue-router (router)` plugins are now imported and injected to the Vue instance:



```
import Vue from 'vue';
import App from './App.vue';
import router from './router'
import store from './store'

Vue.config.productionTip = false;

new Vue({
  router,
  store,
  render: h => h(App)
}).$mount('#app');

```

How it works...

The Vue UI plugins work in conjunction with `npm` or `yarn` to automatically install the packages on your project, and then inject—when possible—the necessary conditions on the Vue instance.

If a plugin is a visual, directive, or a non-direct instantiated plugin, the Vue UI will install it and manage it, but you need to import it for use on your application.

Adding TypeScript to a Vue CLI project

Using TypeScript in a JavaScript project, even for static type checking, is good practice. It helps minimize the chance of errors and Object problems inside your project.

Adding TypeScript to a Vue project with the help of the Vue UI is very simple, and you will be able to use JavaScript code with TypeScript.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

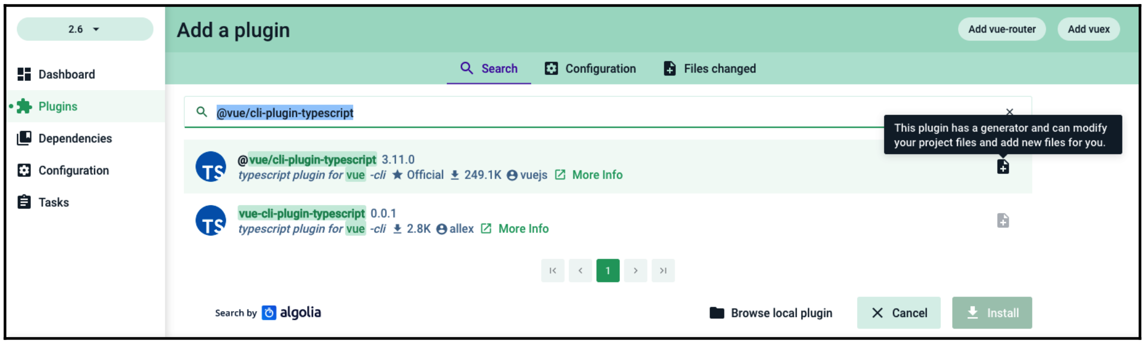
First, we need to create our Vue CLI project. To find how to create a Vue CLI project, please check the 'Creating your first project with Vue CLI' recipe. We can use the one we created in the last recipe or start a new one.

To add TypeScript to a Vue CLI project, follow these steps:

1. Open the Vue UI interface. Open the Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

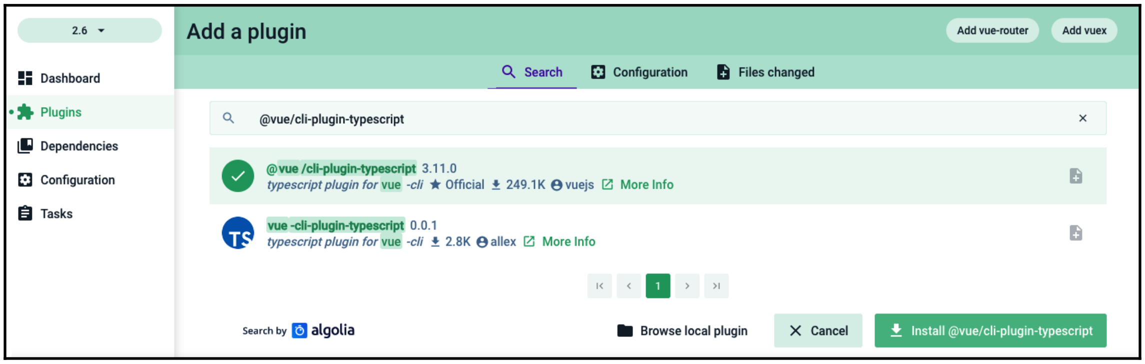
   ```
   > vue ui
   ```

2. On your project, go to the **Plugins** manager, click on **+ Add plugin**, and search for `@vue/cli-plugin-typescript`:


2.6
Dashboard
Plugins
Dependencies
Configuration
Tasks
Add a plugin
Add vue-router
Add vuex
Search
Configuration
Files changed
@vue/cli-plugin-typescript
@vue/cli-plugin-typescript 3.11.0
typescript plugin for vue-cli ★ Official 249.1K vuejs More Info
vue-cli-plugin-typescript 0.0.1
typescript plugin for vue-cli 2.8K allex More Info
1
Search by algolia
Browse local plugin
Cancel
Install

This plugin has a generator and can modify your project files and add new files for you.

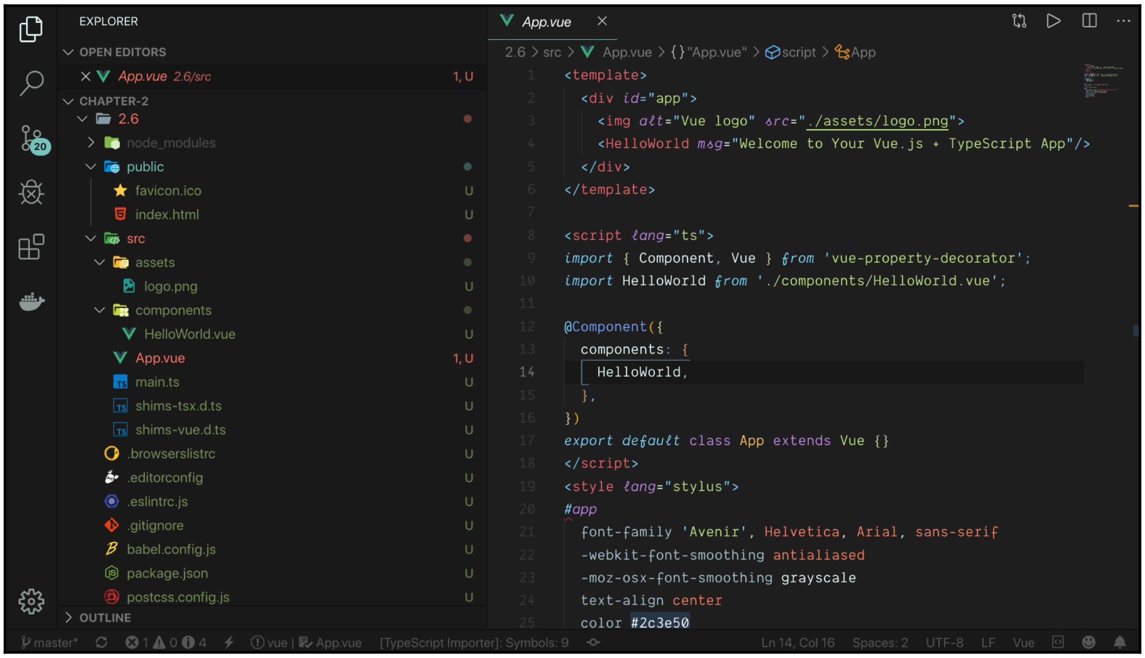
3. Now, click on the **Install @vue/cli-plugin-typescript** button at the bottom of the page:


2.6
Dashboard
Plugins
Dependencies
Configuration
Tasks
Add a plugin
Add vue-router
Add vuex
Search
Configuration
Files changed
@vue/cli-plugin-typescript
@vue /cli-plugin-typescript 3.11.0
typescript plugin for vue -cli ★ Official 249.1K vuejs More Info
vue -cli-plugin-typescript 0.0.1
typescript plugin for vue -cli 2.8K allex More Info
1
Search by algolia
Browse local plugin
Cancel
Install @vue/cli-plugin-typescript

4. You will be asked for some configuration settings after the plugin is downloaded, before the final installation:
   - **Use class-style component syntax?** Use the `vue-class-component` plugin with TypeScript.

- **Use Babel alongside TypeScript (required for modern mode, auto-detected polyfills, transpiling JSX)?** Activate Babel to transpile TypeScript in addition to the TypeScript compiler.
- **Use ESLint?** Use ESLint as a linter for the `.ts` and `.tsx` files.
- **Convert all .js files to .ts?** Automatically convert all your `.js` files to `.ts` files in the installation process.
- **Allow .js files to be compiled?** Activate the `tsconfig.json` flag to accept `.js` files in the compiler.

5. After choosing your options, click on **Finish the installation**.
6. Now, your project is a TypeScript Vue project, with all the files configured and ready to be coded:



```
<template>
  <div id="app">
    <img alt="Vue logo" src="./assets/logo.png">
    <HelloWorld msg="Welcome to Your Vue.js + TypeScript App"/>
  </div>
</template>

<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import HelloWorld from './components/HelloWorld.vue';

@Component({
  components: {
    HelloWorld,
  },
})
export default class App extends Vue {}
</script>
<style lang="stylus">
#app
  font-family 'Avenir', Helvetica, Arial, sans-serif
  -webkit-font-smoothing antialiased
  -moz-osx-font-smoothing grayscale
  text-align center
  color #2c3e50
```

How it works...

The Vue UI as a plugin manager will download the TypeScript package made for Vue, and install and configure it for you with the settings you choose.

Your project will be changed and modified according to your specifications, and will then be ready for development.

See also

Find more information about TypeScript ESLint at `https://github.com/typescript-eslint/typescript-eslint`

Find more information about `vue-class-component` at `https://github.com/vuejs/vue-class-component`.

Creating your first TypeScript Vue component with vue-class-component

As Vue components are object-based and have a strong relationship with the `this` keyword of the JavaScript object, it gets a little bit confusing to develop a TypeScript component.

The `vue-class-component` plugin uses the ECMAScript decorators proposal to pass the statically typed values directly to the Vue component and makes the process of the compiler understand what is happening more easily.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

First, we need to create our Vue CLI project. We can use the one we created in the last recipe or start a new one. To find how to create a Vue CLI project with TypeScript, please check the '*Adding TypeScript to a Vue CLI project*' recipe.

Follow the instructions to create your first Vue component with Typescript and `vue-class-component`:

1. Create a new file inside the `src/components` folder, called `Counter.vue`.
2. Now, let's start making the script part of the Vue component. We will make a class that will have data with a number, two methods—one for increasing and another for decreasing—and, finally, a computed property to format the final data:

```
<script lang="ts">
import Vue from 'vue';
import Component from 'vue-class-component';

@Component
export default class Counter extends Vue {
  valueNumber: number = 0;

  get formattedNumber() {
    return `Your total number is: ${this.valueNumber}`;
  }

  increase() {
    this.valueNumber += 1;
  }

  decrease() {
    this.valueNumber -= 1;
  }
}
</script>
```

3. It's time to create the template and rendering for this component. The process is the same as a JavaScript Vue file. We will add the buttons for increasing and decreasing the value and showing the formatted text:

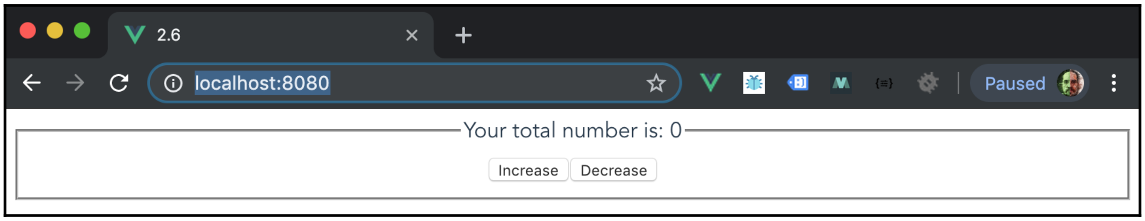
```
<template>
  <div>
    <fieldset>
      <legend>{{ formattedNumber }}</legend>
        <button @click="increase">Increase</button>
```

```
          <button @click="decrease">Decrease</button>
        </fieldset>
      </div>
    </template>
```

4. In the `App.vue` file, we need to import the component we just created:

```
<template>
  <div id="app">
    <Counter />
  </div>
</template>

<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import Counter from './components/Counter.vue';

@Component({
  components: {
    Counter,
  },
})
export default class App extends Vue {

}
</script>
<style lang="stylus">
  #app
    font-family 'Avenir', Helvetica, Arial, sans-serif
    -webkit-font-smoothing antialiased
    -moz-osx-font-smoothing grayscale
    text-align center
    color #2c3e50
    margin-top 60px
</style>
```

5. Now, when you run the `npm run serve` command on Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows), you will see your component running and executing on screen:



Your total number is: 0

How it works...

The `vue-class-component` plugin makes use of the new proposal of decorators to inject and pass some attributes to the classes on TypeScript.

This injection helps in the process of simplifying the development of a component with a syntax more aligned with TypeScript than with the Vue common object.

See also

Find more information about `vue-class-component` at `https://github.com/vuejs/vue-class-component`.

Creating a custom mixin with vue-class-component

In Vue, a `mixin` is a way to reuse the same code in other Vue objects, like mixing all the property of the `mixin` inside the component.

When using a mixin, Vue first declares the `mixin` property and then the component values, so the components will be always the last and valid values. This merge occurs in a deep mode and has a specific way already declared inside the framework, but it can be changed by a special config.

With the use of mixins, developers can write tiny pieces of code and reuse them in lots of components.

This approach simplifies your work and allows you to complete tasks quicker.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

First, we need to create our Vue CLI project. We can use the one we created in the last recipe or start a new one. To find how to create a Vue CLI project with TypeScript, please check the '*Creating your first TypeScript Vue component with vue-class-component*' recipe.

In this recipe, we will split it into two separate parts. First, we will create the counter component, and then we will use the code that is shared to create the mixin.

Creating the Counter component

Now, follow the instructions to create a custom mixin with `vue-class-component`:

1. We need to make a new component called `CounterByTen.vue` in the `src/components` folder.
2. Now, let's start making the script part of the Vue component. We will make a class that will have a variable with the type of a number and a default value of `0`; two methods, one for increasing by `10` and another for decreasing by `10`; and, finally, a computed property to format the final data:

```
<script lang="ts">
import Vue from 'vue';
import Component from 'vue-class-component';

@Component
export default class CounterByTen extends Vue {
  valueNumber: number = 0;

  get formattedNumber() {
    return `Your total number is: ${this.valueNumber}`;
  }

  increase() {
    this.valueNumber += 10;
  }

  decrease() {
    this.valueNumber -= 10;
  }
}
</script>
```

3. It's time to create the template and rendering for this component. The process is the same as for a JavaScript Vue file. We will add the buttons for increasing and decreasing the value and for showing the formatted text:

```
<template>
  <div>
    <fieldset>
      <legend>{{ this.formattedNumber }}</legend>
        <button @click="increase">Increase By Ten</button>
        <button @click="decrease">Decrease By Ten</button>
    </fieldset>
  </div>
</template>
```

4. In the `App.vue` file, we need to import the component we just created:

```
<template>
  <div id="app">
    <Counter />
    <hr />
    <CounterByTen />
  </div>
</template>

<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import Counter from './components/Counter.vue';
import CounterByTen from './components/CounterByTen.vue';

@Component({
  components: {
    Counter,
    CounterByTen,
  },
})
export default class App extends Vue {

}
</script>
<style lang="stylus">
  #app
    font-family 'Avenir', Helvetica, Arial, sans-serif
    -webkit-font-smoothing antialiased
    -moz-osx-font-smoothing grayscale
    text-align center
    color #2c3e50
    margin-top 60px
</style>
```

Extracting similar code for the mixin

With both of the components having similar code, we can extract this similar code and create a mixin. This mixin can be imported in both of the components and their behavior will be the same:

1. Create a file called `defaultNumber.ts` in the `src/mixins` folder.
2. To code our mixin, we will import the `Component` and `Vue` decorators from the `vue-class-component` plugin, to be the base of the mixin. We will need to take a similar code and place it inside the mixin:

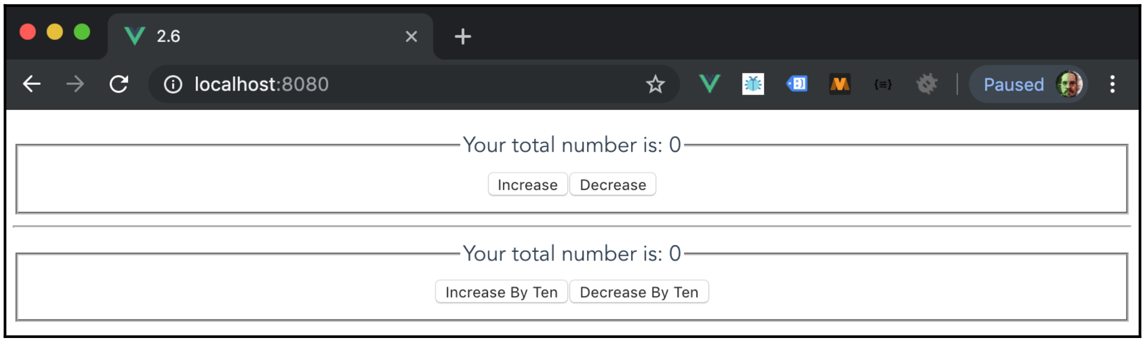
```
import Vue from 'vue';
import Component from 'vue-class-component';

@Component
export default class DefaultNumber extends Vue {
  valueNumber: number = 0;

  get formattedNumber() {
    return `Your total number is: ${this.valueNumber}`;
  }
}
```

3. With the mixin ready, open the `Counter.vue` component on the `src/components` folder and import it. To do this, we need to import a special export from the `vue-class-component` called `mixins` and extend it with the mixin we want to extend. This will remove the `Vue` and `Component` decorators because they are already declared on the mixin:

```
<template>
  <div>
    <fieldset>
      <legend>{{ this.formattedNumber }}</legend>
      <button @click="increase">Increase By Ten</button>
      <button @click="decrease">Decrease By Ten</button>
    </fieldset>
  </div>
</template>
```

```
<script lang="ts">
import Vue from 'vue';
import Component, { mixins } from 'vue-class-component';
import DefaultNumber from '../mixins/defaultNumber';

@Component
export default class CounterByTen extends mixins(DefaultNumber) {
  increase() {
    this.valueNumber += 10;
  }

  decrease() {
    this.valueNumber -= 10;
  }
}
</script>
```

4. Now, when you run the `npm run serve` command on Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows), you will see your component running and executing on screen:



How it works...

The process of using mixins with TypeScript is the same as with the Vue objects. The code that is shared can be split into smaller files and called in the components for easier coding.

When using TypeScript and `vue-class-component`, the `Vue` and `Component` decorators need to be declared on the mixins because the class that will be using the mixin will already have this extension, as it extends this mixin.

We took the same piece of code that works the same on both the components and placed it in a new file that is then called in both of the components.

See also

Find more about `vue-class-component` mixins at `https://github.com/vuejs/vue-class-component#using-mixins`.

Find more about Vue mixins at `https://v3.vuejs.org/guide/mixins.html`

Creating a custom function decorator with vue-class-component

Decorators were introduced in ECMAScript 2015. A decorator is a kind of high-order function that wraps a function with another function.

This brings a lot of new improvements to the code—along with greater productivity—because it takes the principle of functional programming and simplifies it.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

First, we need to create our Vue CLI project. To find how to create a Vue CLI project, please check the '*Creating your first project with Vue CLI*' recipe. We can use the one we created in the last recipe or start a new one.

Follow these steps to create your custom function decorator with `vue-class-component`:

1. Create a file called `componentMount.js` inside the `src/decorators` folder.
2. We need to import the `createDecorator` function from the `vue-class-component` to be able to use it on a `vue-class-component` based component, and to start coding our decorator:

```
import { createDecorator } from 'vue-class-component';
import componentMountLogger from './componentLogger';

export default createDecorator((options) => {
  options.mixins = [...options.mixins, componentMountLogger];
});
```

A `createDecorator` function is like an extension of the Vue vm *(View-Model)*, so it won't have the property of an ECMAScript decorator but will function as a Vue decorator.

3. We need to use the `componentLogger.js` file in our decorator. This function will take all the data values that are set in the `"decorated" component` and add a watcher to it. This watcher will log the new and old values whenever it changes. This function will only be executed with a debug data set to `true`:

```
export default {
  mounted() {
    if (this.debug) {
      const componentName = this.name || '';
      console.log(`The ${componentName} was mounted
                                successfully.`);

      const dataKeys = Object.keys(this.$data);

      if (dataKeys.length) {
        console.log('The base data are:');
        console.table(dataKeys);

        dataKeys.forEach((key) => {
          this.$watch(key, (newValue, oldValue) => {
            console.log(`The new value for ${key} is:
                        ${newValue}`);
            console.log(`The old value for ${key} is:
                        ${oldValue}`);
          }, {
            deep: true,
          });
          });
        }
      }
    },
  };
```

4. Now, we need to import the decorator to our `Counter.vue` component file located in the `src/components` folder and add the debugger data to it:

```
<template>
  <div>
    <fieldset>
      <legend>{{ this.formattedNumber }}</legend>
      <button @click="increase">Increase</button>
      <button@click="decrease">Decrease</button>
    </fieldset>
  </div>
</template>

<script lang="ts">
import Vue from 'vue';
import Component from 'vue-class-component';
import componentMount from '../decorators/componentMount';

@Component
@componentMount
export default class Counter extends Vue {
  valueNumber: number = 0;

  debug: boolean = true;

  get formattedNumber() {
    return `Your total number is: ${this.valueNumber}`;
  }

  increase() {
    this.valueNumber += 1;
  }

  decrease() {
    this.valueNumber -= 1;
  }
}
</script>
```

How it works...

The `createDecorator` function is a factory function that extends the Vue vm (View Model), which produces an extension of the Vue component, such as a Vue mixin. A Vue mixin is a property of the Vue component that can be used to share and reuse code between components.

When we call the mixin, it takes the current component as an option of the first argument (the key if it was attached to a property), and the index of it.

We added a dynamic debugger that is only attached when debug data exists and is set to `true`. This debugger will log the current data and set watchers for the changes in the data, showing the logs on the console each time the data is changed.

There's more...

When using linters, some rules can be a problem with decorators. So, it's wise to disable them only on the files that are having problems with the rules that are required for the code to work.

In an AirBnB style, for example, the `no-param-reassign` rule is required because the decorator uses the option as a reference to pass the value.

See also

Find more information about creating custom decorators with `vue-class-component` at `https://github.com/vuejs/vue-class-component#create-custom-decorators`.

Find more information about decorators on ECMAScript at `https://www.typescriptlang.org/docs/handbook/decorators.html`.

Adding custom hooks to vue-class-component

On Vue, it's possible to add hooks to its life cycle through the Plugins **application programming interface (API)**. The most basic example is the `vue-router` with the navigation guards, such as the `beforeRouterEnter` and `beforeRouterLeave` functions hooks.

The hooks, as the name implies, are little functions that are called each time something will happen.

You can take advantage of the hooks and make them more powerful, adding new functionalities to your components, such as checking for special security access, adding meta **search engine optimization** (**SEO**), or even pre-fetching data.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

First, we need to create our Vue CLI project. We can use the one we created in the last recipe or start a new one. To find how to create a Vue CLI project with TypeScript, please check the '*Adding TypeScript to a Vue CLI project*' recipe.

Now, follow these steps to add custom hooks to your Vue project using TypeScript and `vue-class-component`:

1. We need to add `vue-router` to the project. This can be done with the Vue CLI project creation or in the Vue UI interface after the project has been created.

   If prompted about the mode, the `vue-router` should run. Take note that selecting the **History** option will require special server configuration when it's time to deploy.

2. Open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the `npm run serve` command, and you will see that the `vue-router` is working and that there are two working routers: `home` and `about`.

3. Let's start creating and naming our hooks to register on the main application. To do this, we need to create a `vue-router.js` file inside the `src/classComponentsHooks` folder:

```
import Component from 'vue-class-component';

Component.registerHooks([
  'beforeRouteEnter',
  'beforeRouteLeave',
]);
```

4. We need to import this file to the `main.ts` file as it needs to be called before the application final build:

```
import './classComponentsHooks/vue-router';

import Vue from 'vue';
import App from './App.vue';
import router from './router';

Vue.config.productionTip = false;

new Vue({
 router,
 render: h => h(App),
}).$mount('#app');
```

5. We now have the hooks registered on the `vue-class-component` and they can be used inside the TypeScript components.

6. We need to create a new router location called `Secure.vue` in the `src/views` folder. The secure page will have a password to enter, `vuejs`. When the user enters this password, the router guard will grant permission, and the user can see the page. If the password is wrong, the user will be taken back to the home page. When they leave the page, an alert will show a message to the user:

```
<template>
  <div class="secure">
    <h1>This is an secure page</h1>
  </div>
</template>

<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import { Route, RawLocation } from 'vue-router';
```

```
type RouteNext = (to?: RawLocation | false | ((vm: Vue) => any) |
   void) => void;

@Component
export default class Home extends Vue {
  beforeRouteEnter(to: Route, from: Route, next: RouteNext) {
    const securePassword = 'vuejs';

    const userPassword = prompt('What is the password?');

    if (userPassword === securePassword) {
      next();
    } else if (!userPassword) {
      next('/');
    }
  }

  beforeRouteLeave(to: Route, from: Route, next: RouteNext) {
    alert('Bye!');
    next();
  }
}
</script>
```

7. Now with our page done, we need to add it to the `router.ts` file to be able to call it in the Vue application:

```
import Vue from 'vue';
import Router from 'vue-router';
import Home from './views/Home.vue';

Vue.use(Router);

export default new Router({
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home,
    },
    {
      path: '/about',
      name: 'about',
      component: () => import('./views/About.vue'),
    },
    {
      path: '/secure',
      name: 'secure',
      component: () => import('./views/Secure.vue'),
    },
  ],
});
```

8. With the route added and the view created, the final step is to add the link to the main `App.vue` file, and we will have a component with an integrated hook on it:

```
<template>
  <div id="app">
    <div id="nav">
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link> |
      <router-link to="/secure">Secure</router-link>
    </div>
    <router-view/>
  </div>
</template>
<style lang="stylus">
#app
  font-family 'Avenir', Helvetica, Arial, sans-serif
  -webkit-font-smoothing antialiased
  -moz-osx-font-smoothing grayscale
   text-align center
   color #2c3e50

#nav
  padding 30px
  a
    font-weight bold
    color #2c3e50
    &.router-link-exact-active
      color #42b983
</style>
```

How it works...

The class component needs to understand what are the navigation guards that are being added to the Vue prototype before executing the Vue application. Because of this, we needed to import the custom hooks on the first line of the `main.ts` file.

In the component, with the hooks registered, it's possible to add them as methods because the `vue-class-component` has made all those custom imports into base methods for the component decorator.

We used two of the `vue-router` navigation guards' hooks. Those hooks are called each time a route will enter or leave. The first two parameters we didn't use, the `to` and `from` parameters, are the ones that carry information about the future route and the past route.

The `next` function is always required because it executes a route change. If no argument is passed in the function, the route will continue with the one that was called, but if you want to change the route on the fly, it is possible to pass an argument to change where the user will go.

See also

Find more about vue-router navigation guards at `https://router.vuejs.org/guide/advanced/navigation-guards.html`.

Find more about the vue-class-component hooks at `https://github.com/vuejs/vue-class-component#adding-custom-hooks`.

Adding vue-property-decorator to vue-class-component

Some of the most important parts of Vue are missing in the `vue-class-component` in the form of TypeScript decorators. So, the community made a library called `vue-property-decorator` that is fully endorsed by the Vue core team.

This library brings some of the missing parts as ECMAScript proposal decorators, such as `props`, `watch`, `model`, `inject`, and so on.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

First, we need to create our Vue CLI project. We can use the one we created in the last recipe or start a new one. To find how to create a Vue CLI project with TypeScript, please check the '*Creating a custom mixin with vue-class-component*' recipe.

Follow these steps to add `vue-property-decorator` to a Vue `class-based component`:

1. We need to add the `vue-property-decorator` to our project.
   Open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > npm install -S vue-property-decorator
   ```

2. In the components mixin, we will add a decorator for receiving a prop, which will be a value for our number that is calculated:

   ```
   import {
     Vue,
     Component,
     Prop,
   } from 'vue-property-decorator';

   @Component
   export default class DefaultNumber extends Vue {
     valueNumber: number = 0;

     @Prop(Number) readonly value: number | undefined;

     get formattedNumber() {
       return `Your total number is: ${this.valueNumber}`;
     }
   }
   ```

3. With that number, we need to make the watchers emit the event to the parent component when the value changes, and update the value inside when the value is changed within the parent component. To do this, we need to create a new file called `numberWatcher.ts` inside the `src/mixins` folder:

```
import {
  Watch,
  Mixins,
} from 'vue-property-decorator';
import DefaultNumber from './defaultNumber';

export default class NumberWatchers extends Mixins(DefaultNumber) {
  @Watch('valueNumber')
  onValueNumberChanged(val: number) {
    this.$emit('input', val);
  }

  @Watch('value', { immediate: true })
  onValueChanged(val: number) {
    this.valueNumber = val;
  }
}
```

TIP

In Vue, the `v-model` directive works like a sugar syntax, as a combination of the Vue `$emit` function and the Vue `props` function. When the value is changed, the component needs to `$emit` with the `'input'` name, and the component needs to have in the `props` function a `value` key, which will be the value that will be passed down from the parent component to the child component.

4. With our mixin updated, our components need to be updated too. First, we will update the `Counter.vue` component, changing the imported mixin from the `defaultNumber.ts` file to `numberWatcher.ts`:

```
<template>
  <div>
    <fieldset>
      <legend>{{ this.formattedNumber }}</legend>
      <button @click="increase">Increase</button>
      <button @click="decrease">Decrease</button>
    </fieldset>
  </div>
</template>
```

```
<script lang="ts">
import Vue from 'vue';
import Component, { mixins } from 'vue-class-component';
import NumberWatcher from '../mixins/numberWatcher';

@Component
export default class Counter extends mixins(NumberWatcher) {
  increase() {
    this.valueNumber += 1;
  }

  decrease() {
    this.valueNumber -= 1;
  }
}
</script>
```

5. Now, we will update the `CounterByTen.vue` component, and add the newly created mixin:

```
<template>
  <div>
    <fieldset>
      <legend>{{ this.formattedNumber }}</legend>
      <button @click="increase">Increase By Ten</button>
      <button @click="decrease">Decrease By Ten</button>
    </fieldset>
  </div>
</template>

<script lang="ts">
import Vue from 'vue';
import Component, { mixins } from 'vue-class-component';
import NumberWatcher from '../mixins/numberWatcher';

@Component
export default class CounterByTen extends mixins(NumberWatcher) {
  increase() {
    this.valueNumber += 10;
  }

  decrease() {
    this.valueNumber -= 10;
  }
}
</script>
```

6. With everything settled, we just need to update the `App.vue` component. This time, we will store a variable in the component that will be passed down to both of the child components, and when the components emit the update events, this variable will change automatically, updating the other components too:

```
<template>
  <div id="app">
    <Counter
      v-model="amount"
    />
    <hr />
    <CounterByTen
      v-model="amount"
    />
  </div>
</template>

<script lang="ts">
import { Component, Vue } from 'vue-property-decorator';
import Counter from './components/Counter.vue';
import CounterByTen from './components/CounterByTen.vue';

@Component({
  components: {
    Counter,
    CounterByTen,
  },
})
export default class App extends Vue {
  amount: number = 0;
}
</script>
<style lang="stylus">
  #app
    font-family 'Avenir', Helvetica, Arial, sans-serif
    -webkit-font-smoothing antialiased
    -moz-osx-font-smoothing grayscale
    text-align center
    color #2c3e50
    margin-top 60px
</style>
```

How it works...

By injecting the decorators at the `vue-class-components`, the `vue-property-decorator` helps the TypeScript compiler check for the types and static analysis of your Vue code.

We used two of the decorators available, the `@Watch` and `@Prop` decorators.

As we took apart the common parts of our code in the form of mixins, the process implementation became easier.

The parent component passed down a property to the child component, passing the initial value and the subsequently updated value.

This value is checked and updated inside the child component, which is used to update a local variable used by the calculation functions. When the calculation is done and the value is changed, the watcher emits an event that is passed to the parent component, which updates the main variable, and the loop goes on.

There's more...

There is another library that works the same as the `vue-property-decorator`, but for the `vuex` plugin, called `vuex-class`.

This library uses the same process as `vue-property-decorator`. It creates an inject decorator in the component. Those decorators help the TypeScript compiler to check for types in the development process.

You can find more information about this library at `https://github.com/ktsn/vuex-class/`

See also

You can find more information about the `vue-property-decorator` at `https://github.com/kaorun343/vue-property-decorator`

3
Data Binding, Form Validations, Events, and Computed Properties

Data is the most valuable asset in the world right now, and knowing how to manage it is a must. In Vue, we have the power to choose how we can gather this data, manipulate it as we want, and deliver it to the server.

In this chapter, we will learn more about the process of data manipulation and data handling, form validations, data filtering, how to display this data to the user, and how to present it in a way that is different from what we then have inside our application.

We will learn how to use the `vue-devtools` to go deep inside the Vue components and see what is happening to our data and application.

In this chapter, we'll cover the following recipes:

- Creating the "hello world" component
- Creating an input form with two-way data binding
- Adding an event listener to an element
- Removing the v-model from the input
- Creating a dynamic to-do list
- Creating computed properties and exploring how they work
- Displaying cleaner data and text with custom filters
- Adding form validation with Vuelidate
- Creating filters and sorters for a list
- Creating conditional filtering to sort list data
- Adding custom styles and transitions
- Using `vue-devtools` to debug your application

Technical requirements

In this chapter, we will be using **Node.js** and **Vue CLI.**

Attention, Windows users—you need to install an `npm` package called `windows-build-tools` to be able to install the following required packages. To do this, open PowerShell as administrator and execute the following command:
`> npm install -g windows-build-tools`.

To install **Vue CLI**, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install -g @vue/cli @vue/cli-service-global
```

Creating the "hello world" component

A Vue application is a combination of various components, bound together and orchestrated by the Vue framework. Knowing how to make your component is important. Each component is like a brick in the wall and needs to be made in a way that, when placed, doesn't end up needing other bricks to be reshaped in different ways around it. We are going to learn how to make a base component, with some important principles that focus on organization and clean code.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can create our Vue project with Vue CLI as learned in the 'Creating your first project with Vue CLI' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*, or start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create my-component
```

The **command-line interface** (**CLI**) will ask some questions that will help with the creation of the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *spacebar* to select an option. Choose the `default` option:

```
? Please pick a preset: (Use arrow keys)
❯ default (babel, eslint)
  Manually select features
```

Let's create our first "hello world" component, following these steps:

1. Let's create a new file called `CurrentTime.vue` file in the `src/components` folder.
2. On this file, we will start with the `<template>` part of our component. It will be a shadowed-box card that will display the current date formatted:

```
<template>
  <div class='cardBox'>
    <div class='container'>
      <h2>Today is:</h2>
      <h3>{{ getCurrentDate }}</h3>
    </div>
  </div>
</template>
```

3. Now, we need to create the `<script>` part. We will start with the `name` property. This will be used when debugging our application with `vue-devtools` to identify our component and helps the **integrated development environment** (**IDE**) too. For the `getCurrentDate` computed property, we will create a `computed` property that will return the current date, formatted by the `Intl` browser function:

```
<script>
export default {
  name: 'CurrentTime',
  computed: {
    getCurrentDate() {
      const browserLocale =
        navigator.languages && navigator.languages.length
          ? navigator.languages[0]
          : navigator.language;
      const intlDateTime = new Intl.DateTimeFormat(
        browserLocale,
        {
          year: 'numeric',
          month: 'numeric',
          day: 'numeric',
          hour: 'numeric',
          minute: 'numeric'
        });

      return intlDateTime.format(new Date());
    }
  }
};
</script>
```

4. For styling our box, we need to create a `style.css` file in the `src` folder, then add the `cardBox` style to it:

```
.cardBox {
  box-shadow: 0 5px 10px 0 rgba(0, 0, 0, 0.2);
  transition: 0.3s linear;
  max-width: 33%;
  border-radius: 3px;
  margin: 20px;
}

.cardBox:hover {
  box-shadow: 0 10px 20px 0 rgba(0, 0, 0, 0.2);
}

.cardBox>.container {
  padding: 4px 18px;
}

[class*='col-'] {
  display: inline-block;
}

@media only screen and (max-width: 600px) {
  [class*='col-'] {
    width: 100%;
    }

    .cardBox {
      margin: 20px 0;
    }
  }

  @media only screen and (min-width: 600px) {
    .col-1 {width: 8.33%;}
    .col-2 {width: 16.66%;}
    .col-3 {width: 25%;}
    .col-4 {width: 33.33%;}
    .col-5 {width: 41.66%;}
    .col-6 {width: 50%;}
    .col-7 {width: 58.33%;}
    .col-8 {width: 66.66%;}
    .col-9 {width: 75%;}
    .col-10 {width: 83.33%;}
    .col-11 {width: 91.66%;}
    .col-12 {width: 100%;}
  }

  @media only screen and (min-width: 768px) {
    .col-1 {width: 8.33%;}
    .col-2 {width: 16.66%;}
    .col-3 {width: 25%;}
    .col-4 {width: 33.33%;}
    .col-5 {width: 41.66%;}
    .col-6 {width: 50%;}
    .col-7 {width: 58.33%;}
    .col-8 {width: 66.66%;}
    .col-9 {width: 75%;}
    .col-10 {width: 83.33%;}
    .col-11 {width: 91.66%;}
    .col-12 {width: 100%;}
  }

  @media only screen and (min-width: 992px) {
    .col-1 {width: 8.33%;}
    .col-2 {width: 16.66%;}
    .col-3 {width: 25%;}
    .col-4 {width: 33.33%;}
    .col-5 {width: 41.66%;}
    .col-6 {width: 50%;}
    .col-7 {width: 58.33%;}
    .col-8 {width: 66.66%;}
    .col-9 {width: 75%;}
    .col-10 {width: 83.33%;}
  .col-11 {width: 91.66%;}
  .col-12 {width: 100%;}
}

@media only screen and (min-width: 1200px) {
  .col-1 {width: 8.33%;}
  .col-2 {width: 16.66%;}
  .col-3 {width: 25%;}
  .col-4 {width: 33.33%;}
  .col-5 {width: 41.66%;}
  .col-6 {width: 50%;}
  .col-7 {width: 58.33%;}
  .col-8 {width: 66.66%;}
  .col-9 {width: 75%;}
  .col-10 {width: 83.33%;}
  .col-11 {width: 91.66%;}
  .col-12 {width: 100%;}
}
```

5. In the `App.vue` file, we need to import our component to be able to see it:

```
<template>
  <div id='app'>
    <current-time />
  </div>
</template>

<script>
import CurrentTime from './components/CurrentTime.vue';

export default {
  name: 'app',
  components: {
    CurrentTime
  }
}
</script>
```

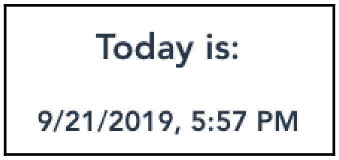
6. In the `main.js` file, we need to import the `style.css` file to be included in the Vue application:

```
import Vue from 'vue';
import App from './App.vue';
import './style.css';

Vue.config.productionTip = false

new Vue({
  render: h => h(App),
}).$mount('#app')
```

7. To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
    > npm run serve
```

Here is your component rendered and running:



Today is:

9/21/2019, 5:57 PM

How it works...

The Vue component works almost like the Node.js packages. To use it in your code, you need to import the component and then declare it inside the `components` property on the component you want to use.

Like a wall of bricks, a Vue application is made of components that call and use other components.

For our component, we used the `Intl.DateTimeFormat` function, a native function, which can be used to format and parse dates to declared locations. To get the local format, we used the `navigator` global variable.

See also

You can find more information about `Intl.DateTimeFormat` at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/DateTimeFormat`.

You can find more information about Vue components at `https://v3.vuejs.org/guide/single-file-component.html`

Creating an input form with two-way data binding

To gather data on the web, we use HTML form inputs. In Vue, it's possible to use a two-way data binding method, where the value of the input on the **Document Object Model** (**DOM**) is passed to the JavaScript—or vice versa.

This makes the web form more dynamic, giving you the possibility to manage, format, and validate the data before saving or sending the data back to the server.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can create our Vue project with Vue CLI, as learned in the '*Creating your first project with Vue CLI*' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*, or use the project from the '*Creating the "hello world" component*' recipe.

Now, let's follow these steps to create an input form with a two-way data binding:

1. Let's create a new file called `TaskInput.vue` in the `src/components` folder.
2. In this file, we're going to create a component that will have a text input and a display text. This text will be based on what is typed on the text input. At the `<template>` part of the component, we need to create an HTML input and a `mustache` variable that will receive and render the data:

```
<template>
  <div class='cardBox'>
    <div class='container tasker'>
      <strong>My task is: {{ task }}</strong>
```

```
        <input
          type='text'
          v-model='task'
          class='taskInput' />
      </div>
    </div>
</template>
```

3. Now, on the `<script>` part of the component, we will name it and add the task to the `data` property. As the data always needs to be a returned `Object`, we will use an arrow function to return an `Object` directly:

```
<script>
export default {
  name: 'TaskInput',
  data: () => ({
    task: '',
  }),
};
</script>
```

4. We need to add some style to this component. In the `<style>` part of the component, we need to add the `scoped` attribute so that the style remains only bound to the component and won't mix with other **Cascading Style Sheets** (**CSS**) rules:

```
<style scoped>
  .tasker{
    margin: 20px;
  }
  .tasker .taskInput {
    font-size: 14px;
    margin: 0 10px;
    border: 0;
    border-bottom: 1px solid rgba(0, 0, 0, 0.75);
  }
  .tasker button {
    border: 1px solid rgba(0, 0, 0, 0.75);
    border-radius: 3px;
    box-shadow: 0 1px 2px 0 rgba(0, 0, 0, 0.2);
  }
</style>
```

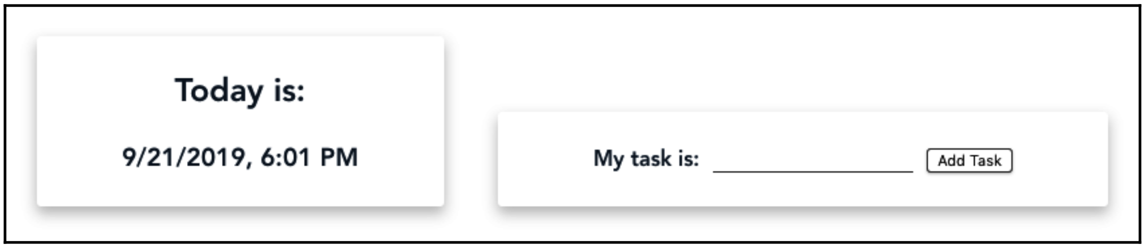
5. Now, we need to import this component into our `App.vue` file:

```
<template>
  <div id='app'>
    <current-time class='col-4' />
    <task-input class='col-6' />
  </div>
</template>

<script>
import CurrentTime from './components/CurrentTime.vue';
import TaskInput from './components/TaskInput';

export default {
  name: 'app',
  components: {
    CurrentTime,
    TaskInput,
  }
}
</script>
```

6. To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



How it works...

When you create an HTML `input` element and add a `v-model` to it, you are passing a directive, built into Vue, that checks the input type and gives us sugar syntax for the input. This handles the update of the value of the variable and the DOM.

This model is what is called two-way data binding. If the variable is changed by the code, the DOM will re-render, and if it's changed by the DOM via user input, such as the `input-form`, the JavaScript code can then execute a function.

See also

Find more information about the form input bindings at `https://v3.vuejs.org/guide/forms.html`

Adding an event listener to an element

The most common method of parent-child communication in Vue is through props and events. In JavaScript, it's common to add event listeners to elements of the DOM tree to execute functions on specific events. In Vue, it's possible to add listeners and name them as you wish, rather than sticking to the names that exist on the JavaScript engine.

In this recipe, we are going to learn how to create custom events and how to emit then.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can create our Vue project with Vue CLI, as learned in the '*Creating your first project with Vue CLI*' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*, or use the project from the '*Creating an input form with two-way data binding*' recipe.

Follow these steps to add an event listener in an element on Vue:

1. Create a new component or open the `TaskInput.vue` file.
2. At the `<template>` part, we are going to add a button element and add an event listener to the button click event with the `v-on` directive. We will remove the `{{ task }}` variable from the component, as from now on it will be emitted and won't be displayed on the component anymore:

```
<template>
  <div class='cardBox'>
    <div class='container tasker'>
      <strong>My task is:</strong>
      <input
        type='text'
        v-model='task'
        class='taskInput' />
      <button
        v-on:click='addTask'>
            Add Task
      </button>
    </div>
  </div>
</template>
```

3. On the `<script>` part of the component, we need to add a method to handle the click event. This method will be named `addTask`. The method will emit an event called `add-task` and send the task on the data. After that, the task on the component will be reset:

```
<script>
export default {
  name: 'TaskInput',
  data: () => ({
    task: '',
  }),
  methods: {
    addTask(){
      this.$emit('add-task', this.task);
      this.task = '';
    },
  }
};
</script>
```

4. On the `App.vue` file, we need to add an event listener bind on the component. This listener will be attached to the `add-task` event. We will use the shortened version of the `v-on` directive, `@`. When it's fired, the event will call the method, `addNewTask`, which will send an alert that a new task was added:

```
<template>
  <div id='app'>
    <current-time class='col-4' />
    <task-input
      class='col-6'
      @add-task='addNewTask'
    />
  </div>
</template>
```

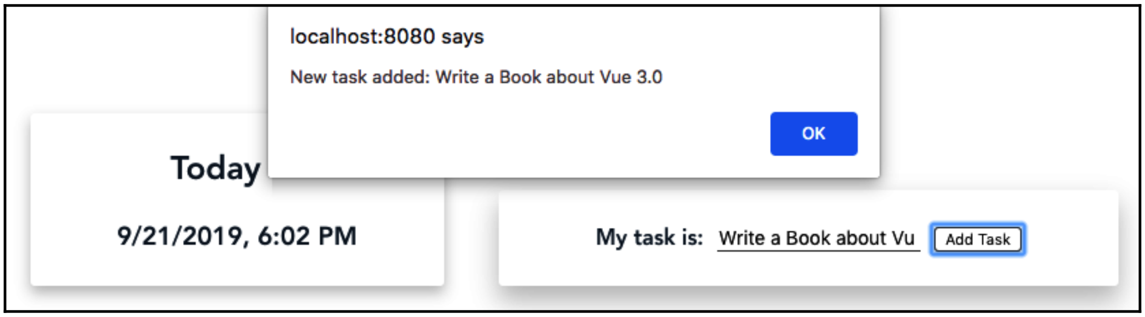
5. Now, let's create the `addNewTask` method. This will receive the task as a parameter and will show an alert to the user, displaying that the task was added:

```
<script>
import CurrentTime from './components/CurrentTime.vue';
import TaskInput from './components/TaskInput';

export default {
  name: 'app',
  components: {
    CurrentTime,
    TaskInput,
  },
  methods:{
    addNewTask(task){
      alert(`New task added: ${task}`);
    },
  },
}
</script>
```

6. To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:


localhost:8080 says

New task added: Write a Book about Vue 3.0

How it works...

The HTML events are read by Vue with the `v-on` event handling directive. When we attached the `v-on:click` directive to the button, we added a listener to the button so that a function will be executed when the user clicks on it.

The function is declared on the component methods. That function, when called, will emit an event, denoting that any component using this component as a child can listen to it with the `v-on` directive.

See also

You can find more information about event handling at `https://v3.vuejs.org/guide/events.html`

Removing the v-model from the input

What if I told you that behind the magic of the `v-model` there is a lot of code that makes our magic sugar syntax happen? What if I told you that the rabbit hole can go deep enough that you can control everything that can happen with the events and values of the inputs?

We will learn how to extract the sugar syntax of the `v-model` directive and transform it into the base syntax behind it.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can create our Vue project with Vue CLI, as learned in the '*Creating your first project with Vue CLI*' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*, or use the project from the '*Adding an event listener to an element'* recipe.

In the following steps, we will remove the `v-model` directive sugar syntax from the input:

1. Open the `TaskInput.vue` file.
2. At the `<template>` block of the component, find the `v-model` directive. We'll remove the `v-model` directive. Then, we need to add a new bind to the input called `v-bind:value` or the shortened version, `:value`, and an event listener to the HTML `input` element. We need to add an event listener to the `input` event with the `v-on:input` directive or the shortened version, `@input`. The input bind will receive the task value as a parameter and the event listener will receive a value attribution, where it will make the task variable equal the value of the event value:

```
<template>
  <div class='cardBox'>
    <div class='container tasker'>
      <strong>My task is:</strong>
      <input
        type='text'
        :value='task'
        @input='task = $event.target.value'
        class='taskInput'
      />
      <button v-on:click='addTask'>
        Add Task
      </button>
    </div>
  </div>
</template>
```

3. To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

How it works...

As a sugar syntax, the `v-model` directive does the magic of automatically declaring the bind and the event listener to the element for you, but the side effect is that you don't have full control over what can be achieved.

As we've seen, the bound value can be a variable, a method, a computed property, or a Vuex getter, for example. And for the event listener, it can be a function or a direct declaration of a variable assignment. When an event is emitted and passed to Vue, the `$event` variable is used to pass the event. In this case, as in normal JavaScript, to catch the value of an input, we need to use the `event.target.value` value.

See also

You can find more information about event handling at `https://v3.vuejs.org/guide/events.html`

Creating a dynamic to-do list

One of the first projects every programmer creates when learning a new language is a to-do list. Doing this allows us to learn more about the language process around the manipulation of states and data.

We are going to make our to-do list using Vue. We'll use what we have learned and created in the previous recipes.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

There are some basic principles involved in making a to-do application—it must have a list of tasks; those tasks can be marked as done and undone, and the list can be filtered and sorted. Now, we are going to learn how to take the tasks and add them to the task list.

To start our component, we can create our Vue project with Vue CLI, as learned in the '*Creating your first project with Vue CLI*' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*, or use the project from the '*Removing the v-model from the input*' recipe.

Now, follow these steps to create a dynamic to-do list with Vue and the previous recipes:

1. In the `App.vue` file, we will create our array of tasks. This task will be filled every time the `TaskInput.vue` component emits a message. We will add an object to this array with the task, and the current date when the task was created. The date when the task was finished will be undefined for now. To do this, in the `<script>` part of the component, we need to create a method that receives a task and add this task with the current date to the `taskList` array:

```
<script>
import CurrentTime from './components/CurrentTime.vue';
import TaskInput from './components/TaskInput';

export default {
  name: 'TodoApp',
  components: {
    CurrentTime,
    TaskInput,
  },
  data: () => ({
    taskList: [],
  }),
  methods:{
    addNewTask(task){
      this.taskList.push({
        task,
        createdAt: Date.now(),
        finishedAt: undefined,
      })
    },
  },
}
</script>
```

2. Now, we need to render this list on the `<template>` part. We will iterate the list of tasks using the `v-for` directive of Vue. This directive, when we use it with an array, gives us access to two properties—the item itself and the index of the item. We will use the item to render it and the index to make the key of the element for the rendering. We need to add a checkbox that, when marked, calls a function that changes the status of the task and the display when the task was done:

```
<template>
  <div id='app'>
    <current-time class='col-4' />
    <task-input class='col-6' @add-task='addNewTask' />
    <div class='col-12'>
      <div class='cardBox'>
        <div class='container'>
          <h2>My Tasks</h2>
          <ul class='taskList'>
            <li
              v-for='(taskItem, index) in taskList'
              :key='`${index}_${Math.random()}`'
            >
              <input type='checkbox'
                :checked='!!taskItem.finishedAt'
                @input='changeStatus(index)'
              />
              {{ taskItem.task }}
              <span v-if='taskItem.finishedAt'>
                {{ taskItem.finishedAt }}
              </span>
            </li>
          </ul>
        </div>
      </div>
    </div>
  </div>
</template>
```

It's always important to remember that the key in the iterator needs to be unique. This is needed because the render function needs to knows which elements were changed. In the example, we added the `Math.random()` function to the index to generate a unique key, because the index of the first elements of the array is always the same number when the number of elements is reduced.

3. We need to create the `changeStatus` function on the `methods` property of the `App.vue`. This function will receive the index of the task as a parameter, then go to the array of tasks and change the `finishedAt` property, which is our marker for when a task is done:

```
changeStatus(taskIndex){
  const task = this.taskList[taskIndex];
    if(task.finishedAt){
      task.finishedAt = undefined;
    } else {
      task.finishedAt = Date.now();
    }
}
```

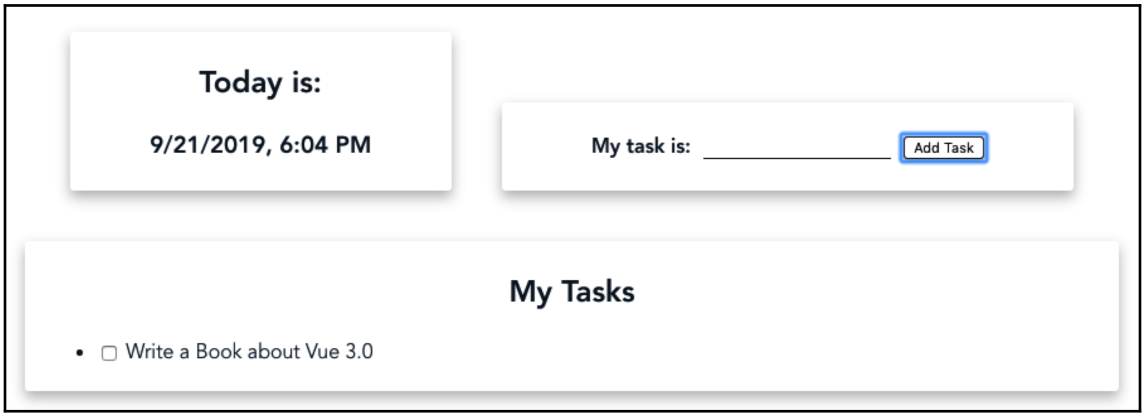
4. Now, we need to add the task text to the left-hand side of the screen. On the `<style>` part of the component, we will make it scoped and add the custom class:

```
<style scoped>
  .taskList li{
    text-align: left;
  }
</style>
```

5. To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



How it works...

As we received the emitted message from the component, we hydrated the message with more data and pushed it to a local array variable.

In the template we iterate this array, making it a list of tasks. This displays the tasks we need to do, the checkbox to mark when the task is done, and the time that a task was done.

When the user clicks on the checkbox, it executes a function, which marks the current task as done. If the task is already done, the function will set the `finishedAt` property as `undefined`.

See also

You can find more information about list rendering at `https://v3.vuejs.org/guide/list.html#mapping-an-array-to-elements-with-v-for`

You can find more information about conditional rendering at `https://v3.vuejs.org/guide/conditional.html#v-if`

You can find more information about `Math.random` at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random`.

Creating computed properties and understanding how they work

Imagine that every time you have to fetch manipulated data, you need to execute a function. Imagine you need to get specific data that needs to go through some process and you need to execute it through a function every time. This type of work would not be easy to maintain. Computed properties exist to solve these problems. Using computed properties makes it easier to obtain data that needs preprocessing or even caching without executing any other external memorizing function.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

We will continue our to-do list project or you can create a new Vue project with Vue CLI, as learned in the '*Creating your first project with Vue CLI*' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*.

Now, follow these steps to create a computed property and understand how it works:

1. On the `App.vue` file, at the `<script>` part, we will add a new property between `data` and `method`, called `computed`. This is where the `computed` properties will be placed. We will create a new computed property called `displayList`, which will be the one that will be used to render the final list on the template:

```
<script>
import CurrentTime from './components/CurrentTime.vue';
import TaskInput from './components/TaskInput';

export default {
  name: 'TodoApp',
  components: {
    CurrentTime,
    TaskInput
  },
  data: () => ({
    taskList: []
  }),
  computed: {
    displayList(){
      return this.taskList;
    },
  },
  methods: {
    addNewTask(task) {
      this.taskList.push({
        task,
        createdAt: Date.now(),
        finishedAt: undefined
      });
    },
    changeStatus(taskIndex){
      const task = this.taskList[taskIndex];
      if(task.finishedAt){
        task.finishedAt = undefined;
      } else {
        task.finishedAt = Date.now();
      }
    }
  }
};
</script>
```

The `displayList` property, for now, is just returning a cached value of the variable, and not the direct variable as itself.

2. Now, on the `<template>` part, we need to change where the list is being fetched:

```
<template>
  <div id='app'>
    <current-time class='col-4' />
    <task-input class='col-6' @add-task='addNewTask' />
    <div class='col-12'>
      <div class='cardBox'>
        <div class='container'>
          <h2>My Tasks</h2>
          <ul class='taskList'>
            <li
```

```
                v-for='(taskItem, index) in displayList'
                :key='`${index}_${Math.random()}`'
              >
                <input type='checkbox'
                  :checked='!!taskItem.finishedAt'
                  @input='changeStatus(index)'
                />
                {{ taskItem.task }}
                <span v-if='taskItem.finishedAt'>
                  {{ taskItem.finishedAt }}
                </span>
              </li>
            </ul>
          </div>
        </div>
      </div>
    </div>
  </template>
```

3. To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShel*l* (Windows) and execute the following command:

```
> npm run serve
```

How it works...

When using the `computed` property to pass a value to the template, this value is now cached. This means we will only trigger the rendering process when the value is updated. At the same time, we made sure that the template doesn't use the variable for rendering so that it can't be changed on the template, as it is a cached copy of the variable.

Using this process, we get the best performance because we won't waste processing time re-rendering the DOM tree for changes that have no effect on the data being displayed. This is because if something changes and the result is the same, the `computed` property caches the result and won't update the final result.

See also

You can find more information about computed properties at `https://v3.vuejs.org/guide/computed.html`.

Displaying cleaner data and text with custom filters

Sometimes you may find that the user, or even you, cannot read the Unix timestamp or other `DateTime` formats. How can we solve this problem? When rendering the data in Vue, it's possible to use what we call filters.

Imagine a series of pipes through which data flows. Data enters each pipe in one shape and exits in another. This is what filters in Vue look like. You can place a series of filters on the same variable, so it gets formatted, reshaped, and ultimately displayed with different data while the code remains the same. The code of the initial variable is immutable in those pipes.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

We will continue our to-do list project or you can create a new Vue project with Vue CLI, as learned in the '*Creating your first project with Vue CLI*' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*.

Follow these steps to create your first custom Vue filter:

1. In the `App.vue` file, at the `<script>` part, in the methods, create a `formatDate` function inside this property. This function will receive `value` as a parameter and enters the filter pipe. We can check if the value is a number because we know that our time is based on the Unix timestamp format. If it's a number, we will format based on the current browser location and return that formatted value. If the passed value is not a number, we just return the passed value:

```
<script>
  import CurrentTime from './components/CurrentTime.vue';
  import TaskInput from './components/TaskInput';

  export default {
    name: 'TodoApp',
    components: {
      CurrentTime,
      TaskInput
    },
    data: () => ({
      taskList: []
    }),
    computed: {
      displayList() {
        return this.taskList;
      }
    },
    methods: {
      formatDate(value) {
        if (!value) return '';
        if (typeof value !== 'number') return value;

        const browserLocale =
          navigator.languages && navigator.languages.length
            ? navigator.languages[0]
            : navigator.language;
        const intlDateTime = new Intl.DateTimeFormat(
          browserLocale,
          {
            year: 'numeric',
            month: 'numeric',
            day: 'numeric',
            hour: 'numeric',
            minute: 'numeric'
          });

        return intlDateTime.format(new Date(value));
```

```
          },
          addNewTask(task) {
            this.taskList.push({
              task,
              createdAt: Date.now(),
              finishedAt: undefined
            });
          },
          changeStatus(taskIndex) {
            const task = this.taskList[taskIndex];
            if (task.finishedAt) {
              task.finishedAt = undefined;
            } else {
              task.finishedAt = Date.now();
            }
          }
        }
      };
    </script>
```

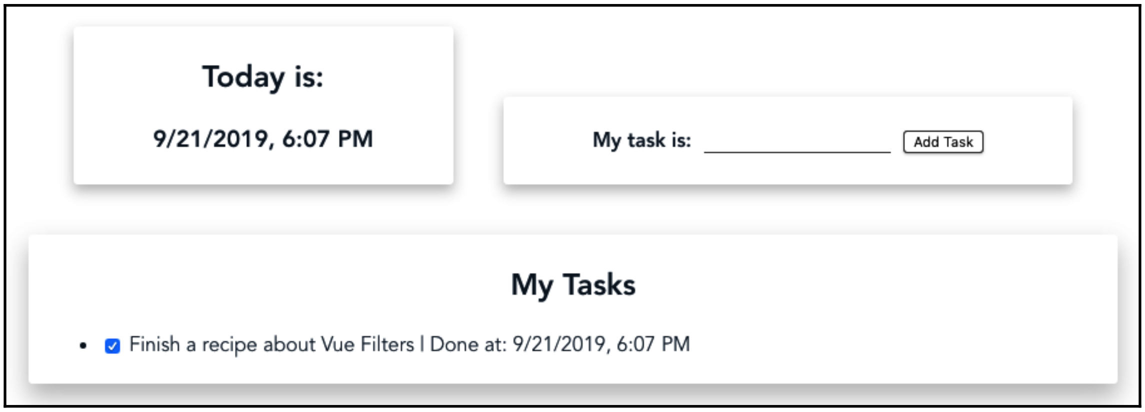
2. On the `<template>` part of the component, we need to pass the variable to the filter method. To do that, we need to find the `taskItem.finishedAt` property and make it the parameter of the `formatDate` method. We will add some text to denote that the task was `Done at:` at the beginning of the date:

```
    <template>
      <div id='app'>
        <current-time class='col-4' />
        <task-input class='col-6' @add-task='addNewTask' />
        <div class='col-12'>
          <div class='cardBox'>
            <div class='container'>
              <h2>My Tasks</h2>
              <ul class='taskList'>
                <li
                  v-for='(taskItem, index) in displayList'
                  :key='`${index}_${Math.random()}`'
                >
                  <input type='checkbox'
                    :checked='!!taskItem.finishedAt'
                    @input='changeStatus(index)'
                  />
                  {{ taskItem.task }}
                  <span v-if='taskItem.finishedAt'> |
                    Done at:
                    {{ formatDate(taskItem.finishedAt) }}
                  </span>
                </li>
              </ul>
            </div>
          </div>
        </div>
      </div>
    </template>
```

3. To run the server and see your component, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:


Today is:
9/21/2019, 6:07 PM
My task is:
Add Task
My Tasks
Finish a recipe about Vue Filters | Done at: 9/21/2019, 6:07 PM

How it works...

Filters are methods that receive a value and must return a value to be displayed on the `<template>` section of the file, or used in a Vue property.

When we pass the value to the `formatDate` method, we know that it's a valid Unix timestamp, so it was possible to invoke to a new `Date` class constructor, passing the `value` as a parameter because the Unix timestamp is a valid date constructor.

The code behind our filter is the `Intl.DateTimeFormat` function, a native function that can be used to format and parse dates to declared locations. To get the local format, we use the `navigator` global variable.

See also

You can find more information about `Intl.DateTimeFormat` at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/DateTimeFormat`.

Adding form validation with Vuelidate

Originally, JavaScript was used just for validating HTML forms before sending these to servers; we didn't have any JavaScript frameworks or the JavaScript ecosystem that we have today. However, one thing remains the same: form validation is to be done first by the JavaScript engine before sending the forms to the server.

We will learn how to use one of the most popular libraries on the Vue ecosystem to validate our input form before sending it.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

We will continue our to-do list project or you can create a new Vue project with Vue CLI, as learned in the '*Creating your first project with Vue CLI*' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*.

Now, follow these steps to add a form validation into your Vue project, and your form component:

1. To install **Vuelidate**, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > npm install vuelidate --save
   ```

2. To add the Vuelidate plugin to the Vue application, we need to import and add it to Vue in the `main.js` file in the `src` folder:

   ```
   import Vue from 'vue';
   import App from './App.vue';
   import Vuelidate from 'vuelidate';
   import './style.css';

   Vue.config.productionTip = false
   Vue.use(Vuelidate);

   new Vue({
     render: h => h(App),
   }).$mount('#app')
   ```

3. In the `TaskInput.vue` file, we will add a new property to the Vue object. This property is interpreted by the new plugin that was installed. At the end of the object, we will add the `validations` property, and inside that property, we will add the name of the model. The model is a direct name of the data or computed property that the plugin will check for validation:

   ```
   <script>
   export default {
     name: 'TaskInput',
     data: () => ({
       task: ''
     }),
     methods: {
       addTask() {
         this.$emit('add-task', this.task);
         this.task = '';
       }
     },
     validations: {
       task: {}
     }
   };
   </script>
   ```

4. Now, we need to import the rules that already exist on the plugins that we want to use—those will be `required` and `minLength`. After the import, we will add those rules to the model:

```
<script>
import { required, minLength } from 'vuelidate/lib/validators';

export default {
  name: 'TaskInput',
  data: () => ({
    task: ''
  }),
  methods: {
    addTask() {
      this.$emit('add-task', this.task);
      this.task = '';
    }
  },
  validations: {
    task: {
      required,
      minLength: minLength(5),
    }
  }
};
</script>
```

5. Now, we need to add the validation before emitting the event. We will use the `$touch` built-in function to tell the plugin that the field was touched by the user and check for validation. If there are any fields that had any interaction with the user, the plugin will set the flags accordingly. If there are no errors, we will emit the event and we will reset the validation with the `$reset` function. To do this, we will change the `addTask` method:

```
addTask() {
    this.$v.task.$touch();
    if (this.$v.task.$error) return false;

    this.$emit('add-task', this.task);
    this.task = '';
    this.$v.task.$reset();
    return true;
}
```

6. To alert the user that there are some errors on the field, we will make the input change the style to a complete red border and have a red text. To do this, we will need to make a conditional class on the input field. This will be attached directly to the model's `$error` property:

```
<template>
  <div class='cardBox'>
    <div class='container tasker'>
      <strong>My task is:</strong>
      <input
        type='text'
        :value='task'
        @input='task = $event.target.value'
        class='taskInput'
        :class="$v.task.$error ? 'fieldError' : ''"
      />
      <button v-on:click='addTask'>Add Task</button>
    </div>
  </div>
</template>
```

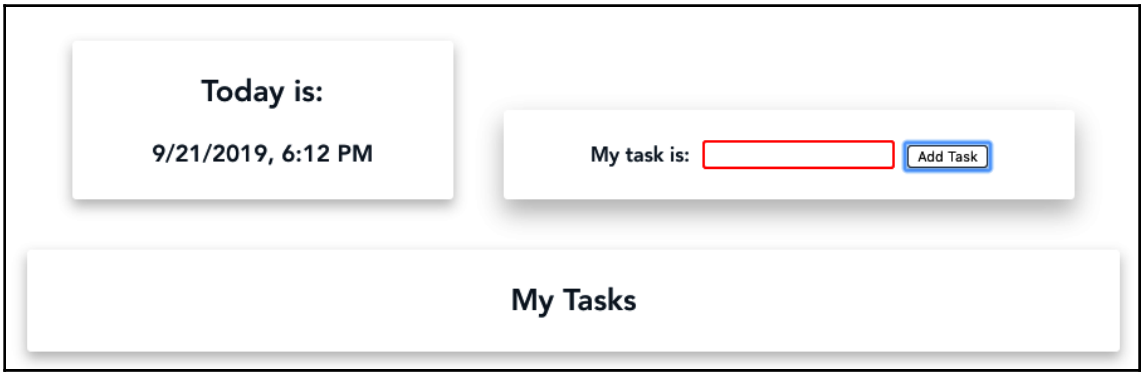
7. For the class, we can create a `fieldError` class in the `style.css` file in the `src` folder:

```
.fieldError {
  border: 2px solid red !important;
  color: red;
  border-radius: 3px;
}
```

8. To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



How it works...

Once installed, the Vuelidate plugin adds a new `$v` property to the Vue prototype and checks for a new object property in the Vue object, called `validations`. When this property is defined and has some rules, the plugins check for the model's rules on each update.

Using this new Vue prototype, we can check inside our code for the errors inside the rules we defined, and execute functions to tell the plugin that the field was touched by the user to flag as a dirty field or reset it. Using those features, we're able to add a new conditional class based on the rules that we defined on the task model.

The task model is required and has a minimum of five characters. If those rules are not met, the plugin will mark the model with an error. We take this error and use it to show the user that the task field has an active error. When the user fulfills the requirements, the display of the error disappears and the event can be emitted.

See also

You can find more information about Vuelidate at `https://vuelidate.netlify.com/`.

You can find more information about class and style bindings at `https://v3.vuejs.org/guide/class-and-style.html`

Creating filters and sorters for a list

When working with lists, it's common to find yourself with raw data. Sometimes, you need to get this data filtered so that it's readable for the user. To do this, we need a combination of the computed properties to form a final set of filters and sorters.

In this recipe, we will learn how to create a simple filter and sorter that will control our initial to-do task list.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

We will continue our to-do list project or you can create a new Vue project with Vue CLI, as learned in the '*Creating your first project with Vue CLI*' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*.

Follow these steps to add a set of filters and sorts into your list:

1. In the `App.vue` file, at the `<script>` part, we will add new computed properties; those will be for sorting and filtering. We will add three new computed properties, `baseList`, `filteredList`, and `sortedList`. The `baseList` property will be our first manipulation. We will add an `id` property to the task list via `Array.map`. As JavaScript arrays start at zero, we will add `1` to the index of the array. The `filteredList` property will filter the `baseList` property and return just the unfinished tasks, and the `sortedList` property will sort the `filteredList` property so that the last added `id` property will be the first displayed to the user:

```
<script>
import CurrentTime from "./components/CurrentTime.vue";
import TaskInput from "./components/TaskInput";

export default {
  name: "TodoApp",
  components: {
    CurrentTime,
    TaskInput
  },
  data: () => ({
    taskList: [],
  }),
  computed: {
    baseList() {
      return [...this.taskList]
        .map((t, index) => ({
            ...t,
            id: index + 1
          }));
    },
    filteredList() {
      return [...this.baseList]
            .filter(t => !t.finishedAt);
    },
    sortedList() {
      return [...this.filteredList]
          .sort((a, b) => b.id - a.id);
    },
    displayList() {
      return this.sortedList;
    }
  },
  methods: {
    formatDate(value) {
      if (!value) return "";
      if (typeof value !== "number") return value;

      const browserLocale =
        navigator.languages && navigator.languages.length
          ? navigator.languages[0]
          : navigator.language;
      const intlDateTime = new Intl.DateTimeFormat(browserLocale, {
        year: "numeric",
        month: "numeric",
        day: "numeric",
        hour: "numeric",
        minute: "numeric"
      });

      return intlDateTime.format(new Date(value));
    },
    addNewTask(task) {
      this.taskList.push({
        task,
        createdAt: Date.now(),
        finishedAt: undefined
      });
    },
    changeStatus(taskIndex) {
      const task = this.taskList[taskIndex];

      if (task.finishedAt) {
        task.finishedAt = undefined;
      } else {
        task.finishedAt = Date.now();
      }
    }
  }
};
</script>
```

2. On the `<template>` part, we will add the `Task ID` as an indicator and change how the `changeStatus` method sends the argument. Because now the index is mutable, we can't use it as a variable; it's just a temporary index on the array. We need to use the task `id`:

```
<template>
  <div id="app">
    <current-time class="col-4" />
    <task-input class="col-6" @add-task="addNewTask" />
```

```
        <div class="col-12">
          <div class="cardBox">
            <div class="container">
              <h2>My Tasks</h2>
              <ul class="taskList">
                <li
                  v-for="(taskItem, index) in displayList"
                  :key="`${index}_${Math.random()}`"
                >
                  <input type="checkbox"
                    :checked="!!taskItem.finishedAt"
                    @input="changeStatus(taskItem.id)"
                  />
                  #{{ taskItem.id }} - {{ taskItem.task }}
                  <span v-if="taskItem.finishedAt"> |
                    Done at:
                    {{ formatDate(taskItem.finishedAt) }}
                  </span>
                </li>
              </ul>
            </div>
          </div>
        </div>
      </div>
    </template>
```

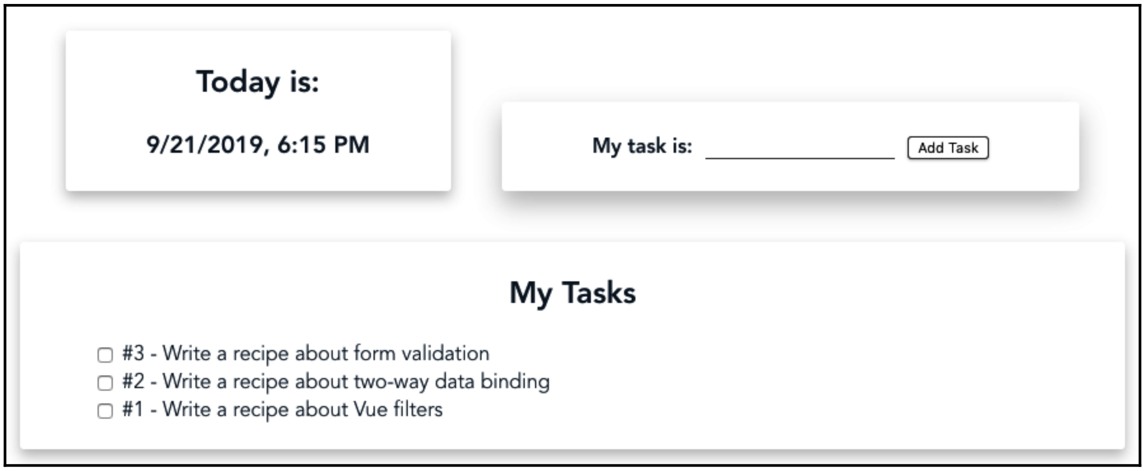
3. On the `changeStatus` method, we need to update our function too. As the index now starts at `1`, we need to decrease the index of the array by one to get the real index of the element before updating it:

```
changeStatus(taskId) {
    const task = this.taskList[taskId - 1];

    if (task.finishedAt) {
    task.finishedAt = undefined;
    } else {
    task.finishedAt = Date.now();
    }
}
```

4. To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



☐ #3 - Write a recipe about form validation
☐ #2 - Write a recipe about two-way data binding
☐ #1 - Write a recipe about Vue filters

How it works...

The `computed` properties worked together as a cache for the list and made sure there were no side effects on the manipulation of the elements:

1. At the `baseList` property, we created a new array with the same tasks but added a new `id` property to the task.
2. At the `filteredList` property, we took the `baseList` property and only returned the tasks that weren't finished.
3. At the `sortedList` property, we sorted the tasks on the `filteredList` property by their ID, in descending order.

When all the manipulation was done, the `displayList` property returned the result of the data that was manipulated.

See also

You can find more information about `Array.prototype.map` at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map`.

You can find more information about `Array.prototype.filter` at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter`.

You can find more information about `Array.prototype.sort` at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort`.

Creating conditional filters to sort list data

After completing the previous recipe, your data should be filtered and sorted, but you might need to check the filtered data or need to change how it was sorted. In this recipe, we will learn how to create conditional filters and sort the data on a list.

Using some basic principles, it's possible to gather information and display it in many different ways.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

We will continue our to-do list project or you can create a new Vue project with Vue CLI, as learned in the '*Creating your first project with Vue CLI*' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*.

Now, follow these steps to add a conditional filter to sort your list data:

1. In the `App.vue` file, at the `<script>` part, we will update the `computed` properties, `filteredList`, `sortedList`, and `displayList`. We need to add three new variables to our project, `hideDone`, `reverse`, and `sortById`. All three are going to be Boolean variables and will start with a default value of `false`. The `filteredList` property will check if the `hideDone` variable is `true`. If it is, it will have the same behavior, but if not, it will show the whole list with no filter. The `sortedList` property will check if the `sortById` variable is `true`. If it is, it will have the same behavior, but if not, it will sort the list by the finished date of the task. The `displayList` property will check if the `reverse` variable is `true`. If it is, it will reverse the displayed list, but if not, it will have the same behavior:

```
<script>
import CurrentTime from "./components/CurrentTime.vue";
import TaskInput from "./components/TaskInput";

export default {
  name: "TodoApp",
  components: {
    CurrentTime,
    TaskInput
  },
  data: () => ({
    taskList: [],
    hideDone: false,
    reverse: false,
    sortById: false,
  }),
  computed: {
    baseList() {
      return [...this.taskList]
        .map((t, index) => ({
            ...t,
            id: index + 1
          }));
    },
    filteredList() {
      return this.hideDone
        ? [...this.baseList]
            .filter(t => !t.finishedAt)
        : [...this.baseList];
    },
    sortedList() {
      return [...this.filteredList]
          .sort((a, b) => (
          this.sortById
            ? b.id - a.id
            : (a.finishedAt || 0) - (b.finishedAt || 0)
        ));
    },
    displayList() {
      const taskList = [...this.sortedList];

      return this.reverse
      ? taskList.reverse()
      : taskList;
    }
  },
  methods: {
    formatDate(value) {
      if (!value) return "";
      if (typeof value !== "number") return value;

      const browserLocale =
        navigator.languages && navigator.languages.length
          ? navigator.languages[0]
          : navigator.language;

      const intlDateTime = new Intl.DateTimeFormat(browserLocale, {
        year: "numeric",
        month: "numeric",
        day: "numeric",
        hour: "numeric",
        minute: "numeric"
      });

      return intlDateTime.format(new Date(value));
    },
    addNewTask(task) {
      this.taskList.push({
        task,
        createdAt: Date.now(),
        finishedAt: undefined
      });
    },
    changeStatus(taskId) {
      const task = this.taskList[taskId - 1];

      if (task.finishedAt) {
        task.finishedAt = undefined;
      } else {
        task.finishedAt = Date.now();
      }
    }
  }
};
</script>
```

2. On the `<template>` part, we need to add the controllers for those variables. We will create three checkboxes, linked directly to the variables via the `v-model` directive:

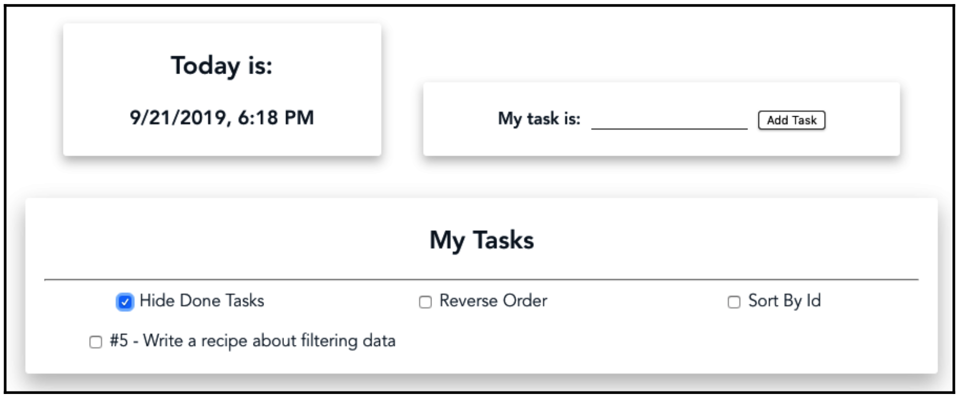
```
<template>
  <div id="app">
    <current-time class="col-4" />
    <task-input class="col-6" @add-task="addNewTask" />
    <div class="col-12">
      <div class="cardBox">
        <div class="container">
          <h2>My Tasks</h2>
          <hr />
          <div class="col-4">
            <input
              v-model="hideDone"
              type="checkbox"
              id="hideDone"
              name="hideDone"
            />
            <label for="hideDone">
              Hide Done Tasks
            </label>
          </div>
          <div class="col-4">
            <input
              v-model="reverse"
              type="checkbox"
              id="reverse"
              name="reverse"
            />
            <label for="reverse">
              Reverse Order
            </label>
          </div>
          <div class="col-4">
            <input
              v-model="sortById"
              type="checkbox"
              id="sortById"
              name="sortById"
            />
            <label for="sortById">
```

```
                Sort By Id
              </label>
            </div>
            <ul class="taskList">
              <li
                v-for="(taskItem, index) in displayList"
                :key="`${index}_${Math.random()}`"
              >
                <input type="checkbox"
                  :checked="!!taskItem.finishedAt"
                  @input="changeStatus(taskItem.id)"
                />
                #{{ taskItem.id }} - {{ taskItem.task }}
                <span v-if="taskItem.finishedAt"> |
                  Done at:
                  {{ formatDate(taskItem.finishedAt) }}
                </span>
              </li>
            </ul>
          </div>
        </div>
      </div>
    </div>
  </template>
```

3. To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



How it works...

The `computed` properties worked together as a cache for the list and made sure there weren't any side effects on the manipulation of the elements. With the conditional process, it was possible to change the rules of the filtering and sorting through a variable, and the display was updated in real-time:

1. At the `filteredList` property, we took the `baseList` property and returned just the tasks that weren't finished. When the `hideDone` variable was `false`, we returned the whole list without any filter.
2. At the `sortedList` property, we sorted the tasks on the `filteredList` property. When the `sortById` variable was `true`, the list was sorted by ID in descending order; when it was `false`, the sorting was done by the task finish time in ascending order.
3. At the `displayList` property, when the `reverse` variable was `true`, the final list was reversed.

When all the manipulation was done, the `displayList` property returned the result of the data that was manipulated.

Those `computed` properties were controlled by the checkboxes on the user screen, so the user had total control of what they could see and how they could see it.

See also

You can find more information about `Array.prototype.map` at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map`.

You can find more information about `Array.prototype.filter` at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter`.

You can find more information about `Array.prototype.sort` at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort`.

Adding custom styles and transitions

Adding styles in your components is a good practice, as it allows you to show your user what is happening more clearly. By doing this, you are able to show a visual response to the user and also give a better experience on your application.

In this recipe, we will learn how to add a new kind of conditional class binding. We will use CSS effects mixed with the re-rendering that comes with each new Vue update.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

We will continue our to-do list project or you can create a new Vue project with Vue CLI, as learned in the '*Creating your first project with Vue CLI*' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*.

Follow these steps to add custom styles and transitions to your component:

1. In the `App.vue` file, we will add a conditional class to the list items for the tasks that are done:

```
<template>
  <div id="app">
    <current-time class="col-4" />
    <task-input class="col-6" @add-task="addNewTask" />
    <div class="col-12">
      <div class="cardBox">
        <div class="container">
          <h2>My Tasks</h2>
          <hr />
          <div class="col-4">
            <input
              v-model="hideDone"
              type="checkbox"
              id="hideDone"
              name="hideDone"
            />
            <label for="hideDone">
```

```
              Hide Done Tasks
            </label>
          </div>
          <div class="col-4">
            <input
              v-model="reverse"
              type="checkbox"
              id="reverse"
              name="reverse"
            />
            <label for="reverse">
              Reverse Order
            </label>
          </div>
          <div class="col-4">
            <input
              v-model="sortById"
              type="checkbox"
              id="sortById"
              name="sortById"
            />
            <label for="sortById">
              Sort By Id
            </label>
          </div>
          <ul class="taskList">
            <li
              v-for="(taskItem, index) in displayList"
              :key="`${index}_${Math.random()}`"
              :class="!!taskItem.finishedAt ? 'taskDone' : ''"
            >
              <input type="checkbox"
                :checked="!!taskItem.finishedAt"
                @input="changeStatus(taskItem.id)"
              />
              #{{ taskItem.id }} - {{ taskItem.task }}
              <span v-if="taskItem.finishedAt"> |
                Done at:
                {{ formatDate(taskItem.finishedAt) }}
              </span>
            </li>
          </ul>
        </div>
      </div>
    </div>
  </div>
</template>
```

2. At the `<style>` part of the component, we will create the CSS style sheet classes for the `taskDone` CSS class. We need to make the list have a separator between the items; then, we will make the list have a striped style; and when they get marked as done, the background will change with an effect. To add the separator between the lines and the striped list or zebra style, we need to add a CSS style sheet rule that applies for each `even nth-child` of our list:

```
<style scoped>
  .taskList li {
    list-style: none;
    text-align: left;
    padding: 5px 10px;
    border-bottom: 1px solid rgba(0,0,0,0.15);
  }

  .taskList li:last-child {
    border-bottom: 0px;
  }

  .taskList li:nth-child(even){
    background-color: rgba(0,0,0,0.05);
  }
</style>
```

3. To add the effect on the background, when the task is done, at the end of the `<style>` part, we will add a CSS animation keyframe that indicates the background color change and apply this animation to the `.taskDone` CSS class:

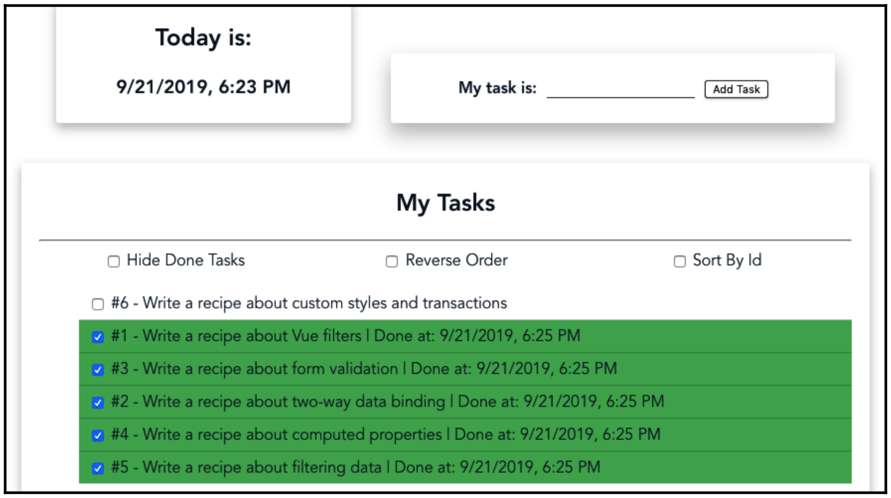
```
<style scoped>
  .taskList li {
    list-style: none;
    text-align: left;
    padding: 5px 10px;
    border-bottom: 1px solid rgba(0,0,0,0.15);
  }

  .taskList li:last-child {
    border-bottom: 0px;
  }

  .taskList li:nth-child(even){
    background-color: rgba(0,0,0,0.05);
  }

  @keyframes colorChange {
    from{
      background-color: inherit;
```

```
        }
        to{
          background-color: rgba(0, 160, 24, 0.577);
        }
      }

      .taskList li.taskDone{
        animation: colorChange 1s ease;
        background-color: rgba(0, 160, 24, 0.577);
      }
    </style>
```

4. To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:


Today is:

9/21/2019, 6:23 PM

My task is: Add Task

My Tasks

☐ Hide Done Tasks ☐ Reverse Order ☐ Sort By Id

☐ #6 - Write a recipe about custom styles and transactions

☑ #1 - Write a recipe about Vue filters | Done at: 9/21/2019, 6:25 PM

☑ #3 - Write a recipe about form validation | Done at: 9/21/2019, 6:25 PM

☑ #2 - Write a recipe about two-way data binding | Done at: 9/21/2019, 6:25 PM

☑ #4 - Write a recipe about computed properties | Done at: 9/21/2019, 6:25 PM

☑ #5 - Write a recipe about filtering data | Done at: 9/21/2019, 6:25 PM

How it works...

Each time a new item in our application is marked as done, the `displayList` property gets updated and triggers the re-rendering of the component.

Because of this, our `taskDone` CSS class has an animation attached to it that is executed on rendering, showing a green background.

See also

You can find more information about CSS animations at `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Animations/Using_CSS_animations`.

You can find more information about class and style bindings at `https://v3.vuejs.org/guide/class-and-style.html`

Using vue-devtools to debug your application

`vue-devtools` is a must for every Vue developer. This tool shows us the depths of the Vue components, routes, events, and vuex.

With the help of the `vue-devtools` extension, it's possible to debug our application, try new data before changing our code, execute functions without needing to call them in our code directly, and so much more.

In this recipe, we will learn more about how we can use the devtools to find more information on your application and how it can be used to help your debug process.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

You will need to install the `vue-devtools` extension in your browser:

- Chrome extension—`http://bit.ly/chrome-vue-devtools`
- Firefox extension—`http://bit.ly/firefox-vue-devtools`

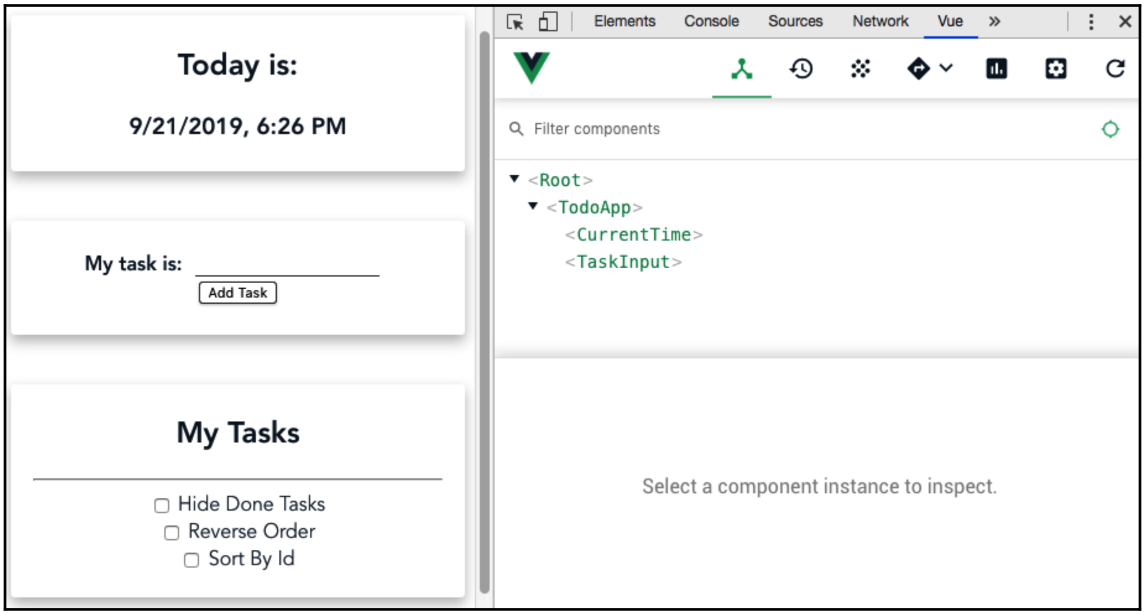
How to do it...

We will continue our to-do list project or you can create a new Vue project with Vue CLI, as learned in the '*Creating your first project with Vue CLI*' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*.

When developing any Vue application, it's always a good practice to develop with `vue-devtools` to hand.

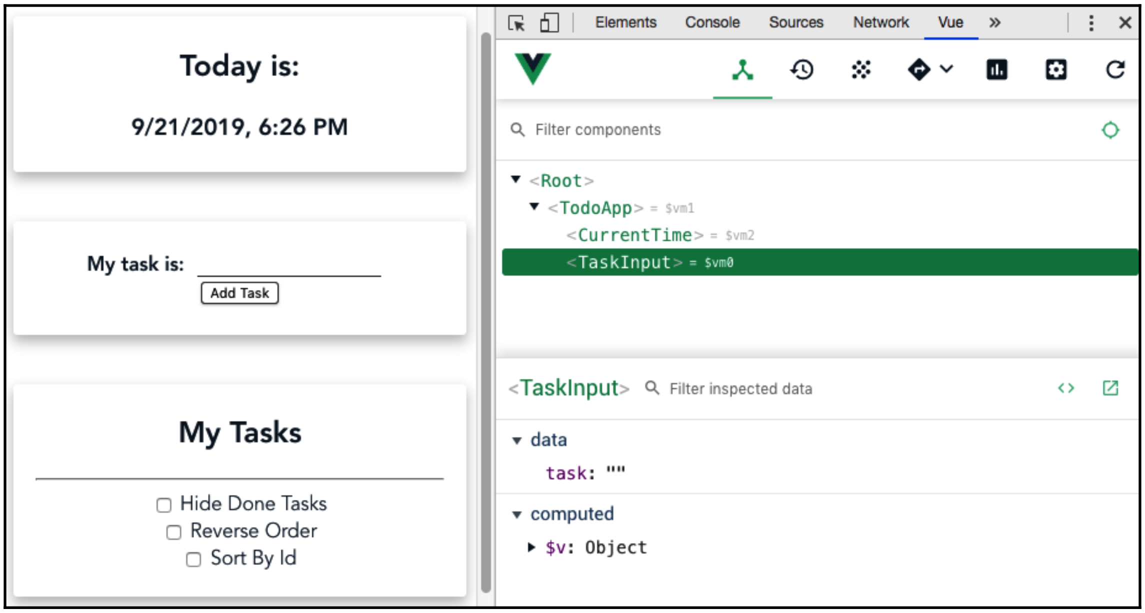
Follow these steps to understand how to use `vue-devtools` and how to properly debug a Vue application:

1. To enter `vue-devtools`, you need to have it installed in your browser first, so check the 'Getting ready' section of this recipe for the links to the extension for Chrome or Firefox. In your Vue development application, enter **browser developer inspector** mode. A new tab with the name **Vue** must appear:



```
▼ <Root>
  ▼ <TodoApp>
      <CurrentTime>
      <TaskInput>
```

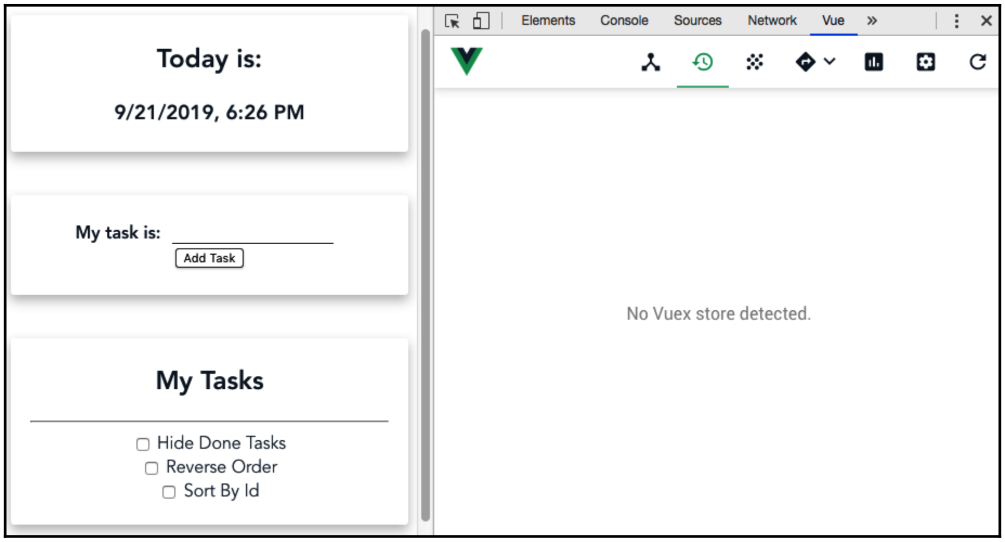
2. The first tab that you are presented with is the **Components** tab. This tab shows your application component tree. If you click on a component, you will be able to see all the available data, the computed property, extra data injected by plugins such as `vuelidate`, `vue-router`, or `vuex`. You can edit the data to see the changes in the application in real time:



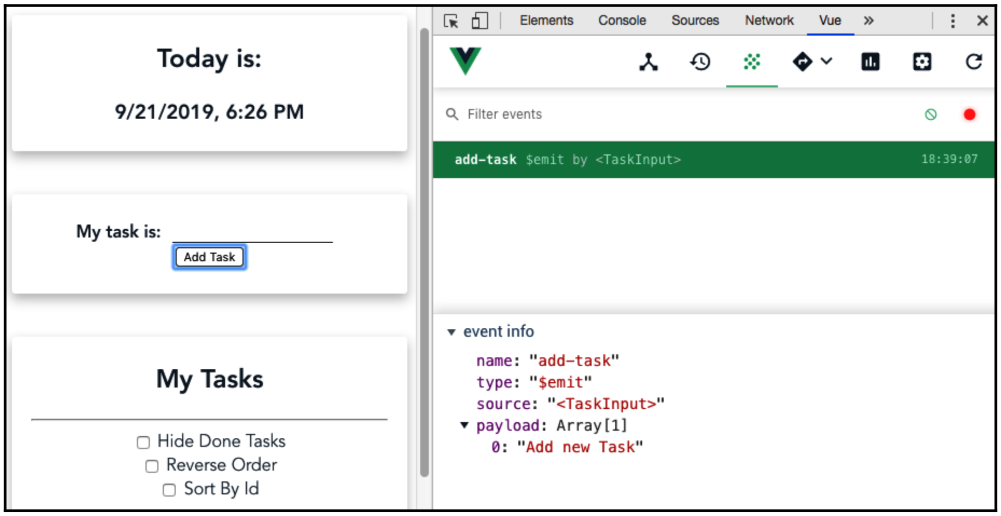
```
▼ <Root>
  ▼ <TodoApp> = $vm1
      <CurrentTime> = $vm2
      <TaskInput> = $vm0
```

```
<TaskInput>
▼ data
    task: ""
▼ computed
  ▶ $v: Object
```

3. The second tab is for **vuex development**. This tab will show the history of the mutations, the current state, and the getters. It's possible to check on each mutation the passed payload and do time-travel mutations, to "go back in time" in the vuex changes in the states:



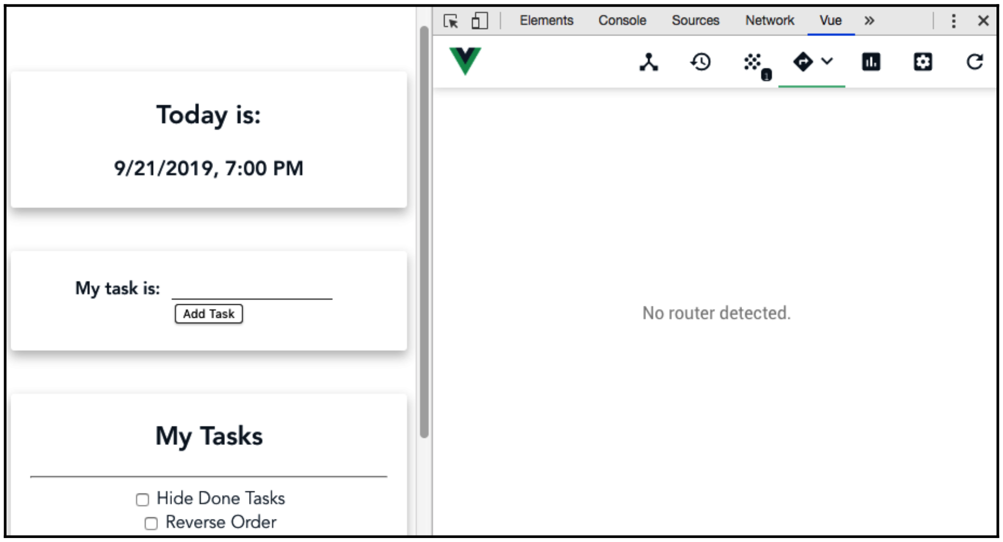
4. The third tab is dedicated to **event emitters** in the application. All events that are emitted in the application will be shown here. You can check the event that was emitted by clicking on it. You can see the name of the event, the type, who was the source of the event (in this case, it was a component), and the payload:



```
add-task $emit by <TaskInput>                18:39:07
```

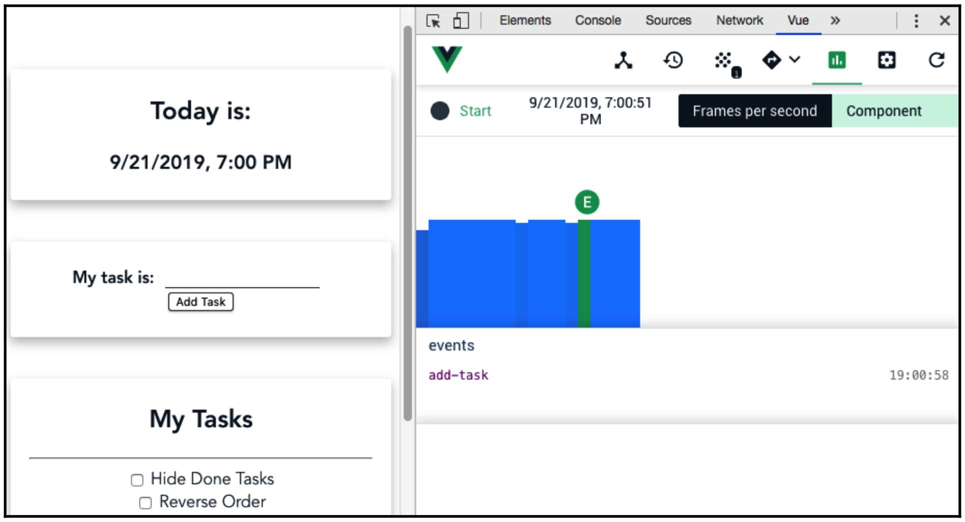
```
▾ event info
    name: "add-task"
    type: "$emit"
    source: "<TaskInput>"
  ▾ payload: Array[1]
      0: "Add new Task"
```

5. The fourth tab is dedicated to the **vue-router** plugin. There, you can see the navigation history, with all the metadata passed to the new route. You can check all the available routes in your application:



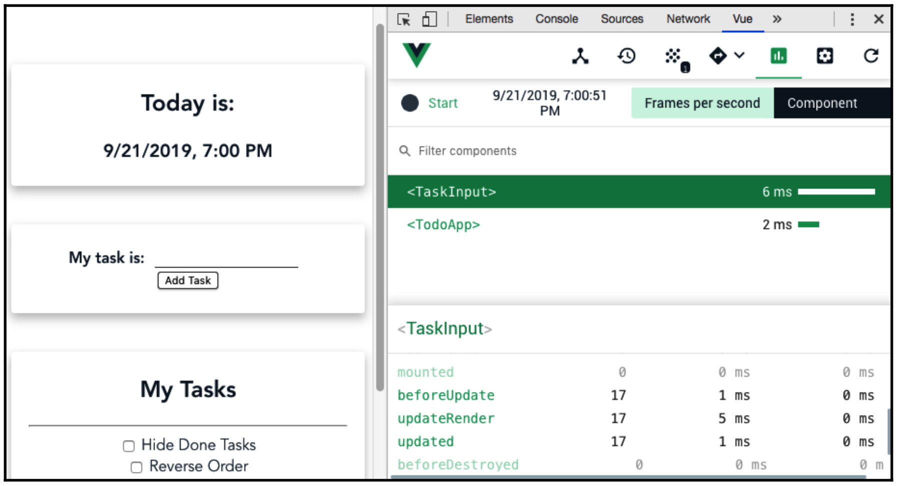
No router detected.

6. The fifth tab is a **Performance** tab. Here, you can check your component loading time, the frames per second that your application is running by the events that are happening in real time. This first screenshot shows the current frames per second of the current application, and for the selected component:



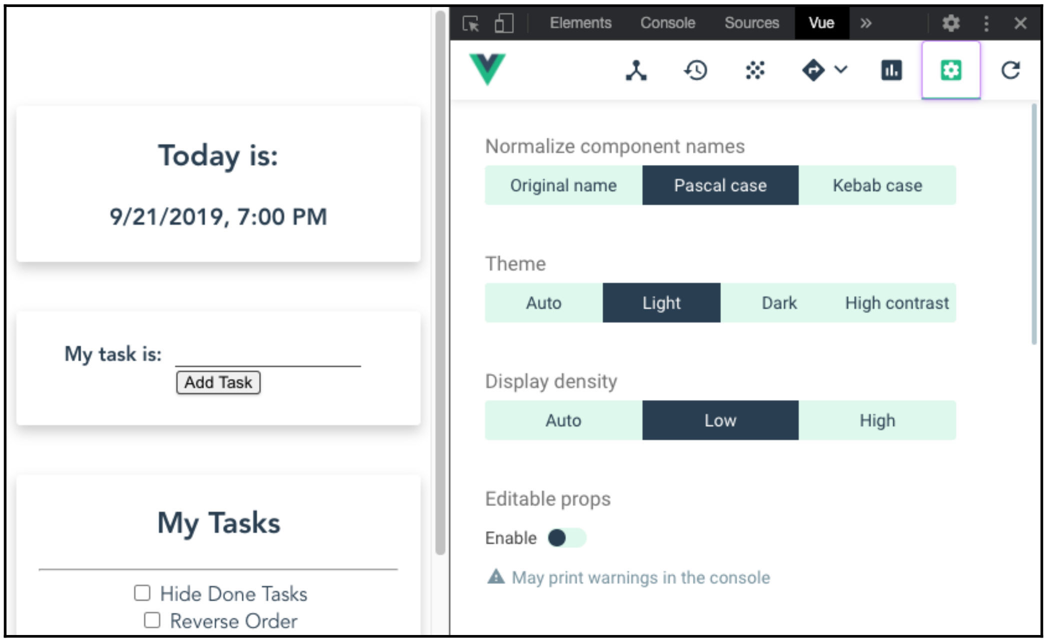
```
add-task                                                    19:00:58
```

This second screenshot shows the components lifecycle hooks performance and the time it took to execute each hook:



```
<TaskInput>          6 ms
<TodoApp>            2 ms

<TaskInput>
mounted          0     0 ms    0 ms
beforeUpdate    17     1 ms    0 ms
updateRender    17     5 ms    0 ms
updated         17     1 ms    0 ms
beforeDestroyed  0     0 ms    0 m
```

7. The sixth tab is your **Settings** tab; here, you can manage the extension, change how it looks, how it behaves internally, and how it will behave within the Vue plugins:



Normalize component names
Original name [✓ Pascal case] Kebab case

Theme
Auto [✓ Light] Dark High contrast

Display density
Auto [✓ Low] High

Editable props
Enable

May print warnings in the console

8. The last tab is a refresh button for the `vue-devtools`. Sometimes, when the `hot-module-reload` occurs or when some complex events happen in your application component tree, the extension can lose track of what is happening. This button forces the extension to reload and read the Vue application state again.

See also

You can find more information about `vue-devtools` at `https://github.com/vuejs/vue-devtools`.

4
Components, Mixins, and Functional Components

Building a Vue application is like putting a puzzle together. Each piece of the puzzle is a component, and each piece has a slot to fill.

Components play a big part in Vue development. In Vue, each part of your code will be a component—it could be a layout, page, container, or button, but ultimately, it's a component. Learning how to interact with them and reuse them is the key to cleaning up code and performance in your Vue application. Components are the code that will, in the end, render something on the screen, whatever the size might be.

In this chapter, we will learn about how we can make a visual component that can be reused in many places. We'll use slots to place data inside our components, create functional components for seriously fast rendering, implement direct communication between parent and child components, and finally, look at loading your components asynchronously.

Let's put these all those pieces together and create the beautiful puzzle that is a Vue application.

In this chapter, we'll cover the following recipes:

- Creating a visual template component
- Using slots and named slots to place data inside your components
- Passing data to your component and validating the data
- Creating functional components
- Accessing your children components data
- Creating a dynamic injected component
- Creating a dependency injection component

- Creating a component `mixin`
- Lazy loading your components

Technical requirements

In this chapter, we will be using **Node.js** and **Vue-CLI.**

Attention Windows users: you need to install an NPM package called `windows-build-tools` to be able to install the following required packages. To do so, open PowerShell as an administrator and execute the following command:

```
> npm install -g windows-build-tools
```

To install **Vue-CLI**, you need to open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install -g @vue/cli @vue/cli-service-global
```

Creating a visual template component

Components can be data-driven, stateless, stateful, or a simple visual component. But what is a visual component? A visual component is a component that has only one purpose: visual manipulation.

A visual component could have a simple Scoped CSS with some `div` HTML elements, or it could be a more complex component that can calculate the position of the element on the screen in real-time.

We will create a card wrapper component that follows the Material Design guide.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can use the Vue project with Vue-CLI, as we did in the 'Creating Your first project with Vue CLI' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*, or we can start a new one.

To start a new project, open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create visual-component
```

The CLI will ask some questions that will help with the creation of the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *spacebar* to select an option. Choose the `default` option:

```
? Please pick a preset: (Use arrow keys)
❯ default (babel, eslint)
  Manually select features
```

Now, let's follow these steps and create a visual template component:

1. Let's create a new file called `MaterialCardBox.vue` in the `src/components` folder.
2. In this file, we will start with the template of our component. We need to create the box for the card. By using the Material Design guide, this box will have a shadow and rounded corners:

```
<template>
 <div class="cardBox elevation_2">
 <div class="section">
 This is a Material Card Box
 </div>
 </div>
</template>
```

3. In the `<script>` part of our component, we will add just our basic name:

   ```
   <script>
     export default {
      name: 'MaterialCardBox',
     };
   </script>
   ```

4. We need to create our elevation CSS stylesheet rules. To do this, create a file named `elevation.css` in the `style` folder. There, we will create the elevations from `0` to `24`, to follow all the elevations on the Material Design guide:

   ```
   .elevation_0 {
       border: 1px solid rgba(0, 0, 0, 0.12);
   }

   .elevation_1 {
       box-shadow: 0 1px 3px rgba(0, 0, 0, 0.2),
           0 1px 1px rgba(0, 0, 0, 0.14),
           0 2px 1px -1px rgba(0, 0, 0, 0.12);
   }

   .elevation_2 {
       box-shadow: 0 1px 5px rgba(0, 0, 0, 0.2),
           0 2px 2px rgba(0, 0, 0, 0.14),
           0 3px 1px -2px rgba(0, 0, 0, 0.12);
   }

   .elevation_3 {
       box-shadow: 0 1px 8px rgba(0, 0, 0, 0.2),
           0 3px 4px rgba(0, 0, 0, 0.14),
           0 3px 3px -2px rgba(0, 0, 0, 0.12);
   }

   .elevation_4 {
       box-shadow: 0 2px 4px -1px rgba(0, 0, 0, 0.2),
           0 4px 5px rgba(0, 0, 0, 0.14),
           0 1px 10px rgba(0, 0, 0, 0.12);
   }

   .elevation_5 {
       box-shadow: 0 3px 5px -1px rgba(0, 0, 0, 0.2),
           0 5px 8px rgba(0, 0, 0, 0.14),
           0 1px 14px rgba(0, 0, 0, 0.12);
   }

   .elevation_6 {
       box-shadow: 0 3px 5px -1px rgba(0, 0, 0, 0.2),
   ```

```
        0 6px 10px rgba(0, 0, 0, 0.14),
        0 1px 18px rgba(0, 0, 0, 0.12);
}

.elevation_7 {
    box-shadow: 0 4px 5px -2px rgba(0, 0, 0, 0.2),
        0 7px 10px 1px rgba(0, 0, 0, 0.14),
        0 2px 16px 1px rgba(0, 0, 0, 0.12);
}

.elevation_8 {
    box-shadow: 0 5px 5px -3px rgba(0, 0, 0, 0.2),
        0 8px 10px 1px rgba(0, 0, 0, 0.14),
        0 3px 14px 2px rgba(0, 0, 0, 0.12);
}

.elevation_9 {
    box-shadow: 0 5px 6px -3px rgba(0, 0, 0, 0.2),
        0 9px 12px 1px rgba(0, 0, 0, 0.14),
        0 3px 16px 2px rgba(0, 0, 0, 0.12);
}

.elevation_10 {
    box-shadow: 0 6px 6px -3px rgba(0, 0, 0, 0.2),
        0 10px 14px 1px rgba(0, 0, 0, 0.14),
        0 4px 18px 3px rgba(0, 0, 0, 0.12);
}

.elevation_11 {
    box-shadow: 0 6px 7px -4px rgba(0, 0, 0, 0.2),
        0 11px 15px 1px rgba(0, 0, 0, 0.14),
        0 4px 20px 3px rgba(0, 0, 0, 0.12);
}

.elevation_12 {
    box-shadow: 0 7px 8px -4px rgba(0, 0, 0, 0.2),
        0 12px 17px 2px rgba(0, 0, 0, 0.14),
        0 5px 22px 4px rgba(0, 0, 0, 0.12);
}

.elevation_13 {
    box-shadow: 0 7px 8px -4px rgba(0, 0, 0, 0.2),
        0 13px 19px 2px rgba(0, 0, 0, 0.14),
        0 5px 24px 4px rgba(0, 0, 0, 0.12);
}

.elevation_14 {
    box-shadow: 0 7px 9px -4px rgba(0, 0, 0, 0.2),
        0 14px 21px 2px rgba(0, 0, 0, 0.14),
        0 5px 26px 4px rgba(0, 0, 0, 0.12);
}

.elevation_15 {
    box-shadow: 0 8px 9px -5px rgba(0, 0, 0, 0.2),
        0 15px 22px 2px rgba(0, 0, 0, 0.14),
        0 6px 28px 5px rgba(0, 0, 0, 0.12);
}

.elevation_16 {
    box-shadow: 0 8px 10px -5px rgba(0, 0, 0, 0.2),
        0 16px 24px 2px rgba(0, 0, 0, 0.14),
        0 6px 30px 5px rgba(0, 0, 0, 0.12);
}

.elevation_17 {
    box-shadow: 0 8px 11px -5px rgba(0, 0, 0, 0.2),
        0 17px 26px 2px rgba(0, 0, 0, 0.14),
        0 6px 32px 5px rgba(0, 0, 0, 0.12);
}

.elevation_18 {
    box-shadow: 0 9px 11px -5px rgba(0, 0, 0, 0.2),
        0 18px 28px 2px rgba(0, 0, 0, 0.14),
        0 7px 34px 6px rgba(0, 0, 0, 0.12);
}

.elevation_19 {
    box-shadow: 0 9px 12px -6px rgba(0, 0, 0, 0.2),
        0 19px 29px 2px rgba(0, 0, 0, 0.14),
        0 7px 36px 6px rgba(0, 0, 0, 0.12);
}

.elevation_20 {
    box-shadow: 0 10px 13px -6px rgba(0, 0, 0, 0.2),
        0 20px 31px 3px rgba(0, 0, 0, 0.14),
        0 8px 38px 7px rgba(0, 0, 0, 0.12);
}

.elevation_21 {
    box-shadow: 0 10px 13px -6px rgba(0, 0, 0, 0.2),
        0 21px 33px 3px rgba(0, 0, 0, 0.14),
        0 8px 40px 7px rgba(0, 0, 0, 0.12);
}

.elevation_22 {
    box-shadow: 0 10px 14px -6px rgba(0, 0, 0, 0.2),
            0 22px 35px 3px rgba(0, 0, 0, 0.14),
            0 8px 42px 7px rgba(0, 0, 0, 0.12);
    }

    .elevation_23 {
        box-shadow: 0 11px 14px -7px rgba(0, 0, 0, 0.2),
            0 23px 36px 3px rgba(0, 0, 0, 0.14),
            0 9px 44px 8px rgba(0, 0, 0, 0.12);
    }

    .elevation_24 {
        box-shadow: 0 11px 15px -7px rgba(0, 0, 0, 0.2),
            0 24px 38px 3px rgba(0, 0, 0, 0.14),
            0 9px 46px 8px rgba(0, 0, 0, 0.12);
    }
```

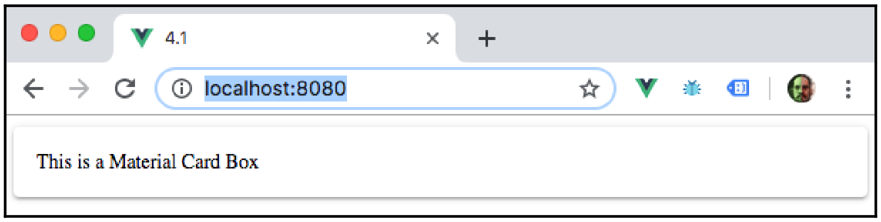
5. For styling our card in the `<style>` part of the component, we need to set the `scoped` attribute inside the `<style>` tag to make sure that the visual style won't interfere with any other components within our application. We will make this card follow the Material Design guide. We need to import the `Roboto` font family and apply it to all elements that will be wrapped inside this component:

```
<style scoped>
  @import
url('https://fonts.googleapis.com/css?family=Roboto:400,500,700&dis
play=swap');
  @import '../style/elevation.css';

  *{
    font-family: 'Roboto', sans-serif;
  }
  .cardBox{
      width: 100%;
  max-width: 300px;
    background-color: #fff;
    position: relative;
    display: inline-block;
    border-radius: 0.25rem;
  }
  .cardBox > .section {
    padding: 1rem;
    position: relative;
  }
</style>
```

6. To run the server and see your component, you need to open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



This is a Material Card Box

How it works...

A visual component is a component that will wrap any component and place the wrapped data with custom styles. As this component mixes with others, it can form a new component without the need to reapply or rewrite any style in your code.

See also

You can find more information about Scoped CSS at `https://vue-loader.vuejs.org/guide/scoped-css.html#child-component-root-elements`.

You can find more information about Material Design cards at `https://material.io/components/cards/`.

Check out the Roboto font family at `https://fonts.google.com/specimen/Roboto`.

Using slots and named slots to place data inside your components

Sometimes the pieces of the puzzle go missing, and you find yourself with a blank spot. Imagine that you could fill that empty spot with a piece that you crafted yourself, not the original one that came with the puzzle box. That's a rough analogy for what a Vue slot is.

Vue slots are like open spaces in your component that other components can fill with text, HTML elements, or other Vue components. You can declare where the slot will be and how it will behave in your component.

With this technique, you can create a component and, when needed, customize it without any effort at all.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can create our Vue project with Vue-CLI, as we did in the *Creating Your first project with Vue CLI* recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*, or use the project from the *Creating a visual template component* recipe.

Follow these instructions to create slots and named slots in components:

1. Let's open the file called `MaterialCardBox.vue` in the components folder.
2. In the `<template>` part of the component, we will need to add four main sections on the card. Those sections are based on the Material Design card anatomy and are the `header`, `media`, `main section`, and `action` areas. We will use the default slot for the `main section`, and the rest will all be named scopes. For some named slots, we will add a fallback configuration that will be displayed if the user doesn't choose any setting on the slot:

```
<template>
  <div class="cardBox elevation_2">
    <div class="header">
      <slot
        v-if="$slots.header"
        name="header"
```

```
          />
          <div v-else>
            <h1 class="cardHeader cardText">
              Card Header
            </h1>
            <h2 class="cardSubHeader cardText">
              Card Sub Header
            </h2>
          </div>
        </div>
        <div class="media">
          <slot
            v-if="$slots.media"
            name="media"
          />
          <img
            v-else
            src="https://via.placeholder.com/350x250"
          >
        </div>
        <div
          v-if="$slots.default"
          class="section cardText"
          :class="{
            noBottomPadding: $slots.action,
            halfPaddingTop: $slots.media,
          }"
        >
          <slot />
        </div>
        <div
          v-if="$slots.action"
          class="action"
        >
          <slot name="action" />
        </div>
      </div>
    </template>
```

3. Now, we need to create our text CSS stylesheet rules for the component. In the `style` folder, create a new file called `cardStyles.css`, and there we will add the rules for the card text and headers:

```
h1, h2, h3, h4, h5, h6{
    margin: 0;
}
.cardText{
    -moz-osx-font-smoothing: grayscale;
```

```
        -webkit-font-smoothing: antialiased;
        text-decoration: inherit;
        text-transform: inherit;
        font-size: 0.875rem;
        line-height: 1.375rem;
        letter-spacing: 0.0071428571em;
    }
    h1.cardHeader{
        font-size: 1.25rem;
        line-height: 2rem;
        font-weight: 500;
        letter-spacing: .0125em;
    }
    h2.cardSubHeader{
        font-size: .875rem;
        line-height: 1.25rem;
        font-weight: 400;
        letter-spacing: .0178571429em;
        opacity: .6;
    }
```

4. In the `<style>` part of the component, we need to create some CSS stylesheets to follow the rules of our design guide:

```
<style scoped>
@import
url("https://fonts.googleapis.com/css?family=Roboto:400,500,700&dis
play=swap");
@import "../style/elevation.css";
@import "../style/cardStyles.css";

* {
  font-family: "Roboto", sans-serif;
}

.cardBox {
  width: 100%;
  max-width: 300px;
  border-radius: 0.25rem;
  background-color: #fff;
  position: relative;
  display: inline-block;
  box-shadow: 0 1px 5px rgba(0, 0, 0, 0.2), 0 2px 2px rgba(0, 0, 0,
0.14),
    0 3px 1px -2px rgba(0, 0, 0, 0.12);
}
.cardBox > .header {
  padding: 1rem;
  position: relative;
  display: block;
}
.cardBox > .media {
  overflow: hidden;
  position: relative;
  display: block;
  max-width: 100%;
}
.cardBox > .section {
  padding: 1rem;
  position: relative;
  margin-bottom: 1.5rem;
  display: block;
}
.cardBox > .action {
  padding: 0.5rem;
  position: relative;
  display: block;
}
.cardBox > .action > *:not(:first-child) {
  margin-left: 0.4rem;
}
.noBottomPadding {
  padding-bottom: 0 !important;
}
.halfPaddingTop {
  padding-top: 0.5rem !important;
}
</style>
```

5. In the `App.vue` file, in the `src` folder, we need to add elements to those slots. Those elements will be added to each one of the named slots, and for the default slot. We will change the component in the `<template>` part of the file. To add a named slot, we need to use a directive called `v-slot:` and then the name of the slot we want to use:

```
<template>
  <div id="app">
    <MaterialCardBox>
      <template v-slot:header>
        <strong>Card Title</strong><br>
        <span>Card Sub-Title</span>
      </template>
      <template v-slot:media>
        <img src="https://via.placeholder.com/350x150">
      </template>
```

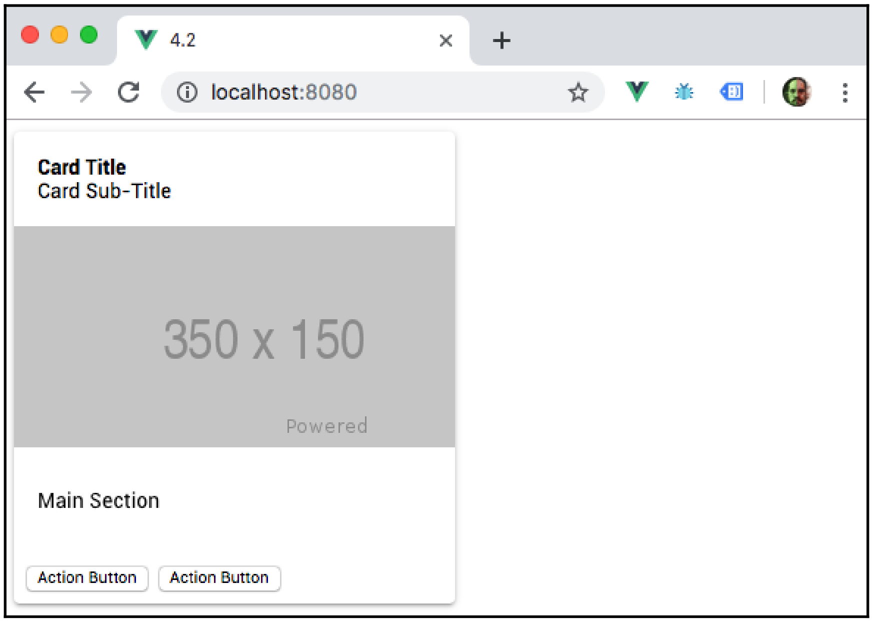
```
          <p>Main Section</p>
          <template v-slot:action>
            <button>Action Button</button>
            <button>Action Button</button>
          </template>
        </MaterialCardBox>
      </div>
    </template>
```

For the default slot, we don't need to use a directive; it just needs to be wrapped in the component to be placed in the `<slot />` part of the component.

6. To run the server and see your component, you need to open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



How it works...

Slots are places where you can put anything that can be rendered into the DOM. We choose the position of our slot and tell the component where to render when it receives any information.

In this recipe, we used named slots, which are designed to work with a component that requires more than one slot. To place any information in that component within the Vue single file (`.vue`) `<template>` part, you need to add the `v-slot:` directive so that Vue is able to know where to place the information that was passed down.

See also

You can find more information about Vue slots at `https://vuejs.org/v2/guide/components-slots.html`.

You can find more information about the Material Design card anatomy at `https://material.io/components/cards/#anatomy`.

Passing data to your component and validating the data

You now know how to place data inside your component through slots, but those slots were made for HTML DOM elements or Vue components. Sometimes, you need to pass data such as strings, arrays, Booleans, or even objects.

The whole application is like a puzzle, where each piece is a component. Communication between components is an important part of it. The possibility to pass data to a component is the first step to connect the puzzle, and then validating the data is the final step to connect the pieces.

In this recipe, we will learn how to pass data to a component and validate the data that was passed to the component.

Getting ready

The pre-requisite is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can create our Vue project with Vue-CLI, as we did in the recipe *Creating Your first project with Vue CLI* in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*, or use the project from the *Using slots and name slots to place data inside your components* recipe.

Follow these instructions to pass data to the component and validate it:

1. Let's open the file called `MaterialCardBox.vue` in the `src/components` folder.
2. In the `<script>` part of the component, we create a new property, called `props`. This property receives the component data, and that data can be used for visual manipulation, variables inside your code, or a function that needs to be executed. In this property, we need to declare the name of the attribute, the type, if it's required, and the validation function. This function will be executed at runtime to validate whether the passed attribute is a valid one:

```
<script>
export default {
  name: 'MaterialCardBox',
  inheritAttrs: false,
  props: {
    header: {
      type: String,
      required: false,
      default: '',
      validator: v => typeof v === 'string',
    },
    subHeader: {
      type: String,
      required: false,
      default: '',
      validator: v => typeof v === 'string',
    },
    mainText: {
      type: String,
      required: false,
      default: '',
      validator: v => typeof v === 'string',
    },
    showMedia: {
      type: Boolean,
      required: false,
      default: false,
      validator: v => typeof v === 'boolean',
    },
    imgSrc: {
      type: String,
      required: false,
      default: '',
      validator: v => typeof v === 'string',
    },
    showActions: {
      type: Boolean,
      required: false,
      default: false,
      validator: v => typeof v === 'boolean',
    },
    elevation: {
      type: Number,
      required: false,
      default: 2,
      validator: v => typeof v === 'number',
    },
  },
  computed: {},
};
</script>
```

3. In the `computed` property, in the `<script>` part of the component, we need to create a set of visual manipulation rules that will be used for rendering the card. Those rules will be `showMediaContent`, `showActionsButtons`, `showHeader`, and `cardElevation`. Each rule will check the received `props` and the `$slots` objects to see whether the relevant card part needs to be rendered:

```
computed: {
  showMediaContent() {
    return (this.$slots.media || this.imgSrc) && this.showMedia;
  },
  showActionsButtons() {
        return this.showActions && this.$slots.action;
      },
      showHeader() {
        return this.$slots.header || (this.header || this.subHeader);
      },
      showMainContent() {
        return this.$slots.default || this.mainText;
      },
      cardElevation() {
        return `elevation_${parseInt(this.elevation, 10)}`;
      },
    },
```

4. After adding the visual manipulation rules, we need to add the created rules to the `<template>` part of our component. They will affect the appearance and behavior of our card. For example, if there is no header slot defined, and there is a header property defined, we show the fallback header. That header is the data that was passed down via `props`:

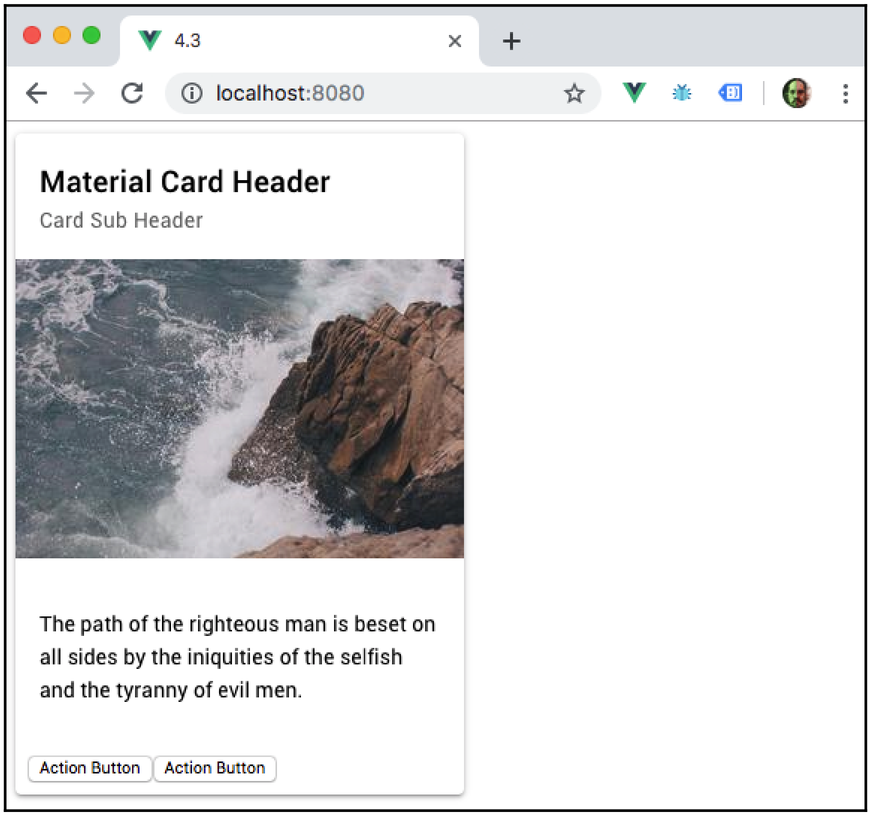
```
<template>
  <div
    class="cardBox"
    :class="cardElevation"
  >
    <div
      v-if="showHeader"
      class="header"
    >
      <slot
        v-if="$slots.header"
        name="header"
      />
      <div v-else>
        <h1 class="cardHeader cardText">
          {{ header }}
        </h1>
        <h2 class="cardSubHeader cardText">
          {{ subHeader }}
        </h2>
      </div>
    </div>
    <div
      v-if="showMediaContent"
      class="media"
    >
      <slot
        v-if="$slots.media"
```

```
          name="media"
        />
        <img
          v-else
          :src="imgSrc"
        >
      </div>
      <div
        v-if="showMainContent"
        class="section cardText"
        :class="{
          noBottomPadding: $slots.action,
          halfPaddingTop: $slots.media,
        }"
      >
        <slot v-if="$slots.default" />
        <p
          v-else
          class="cardText"
        >
          {{ mainText }}
        </p>
      </div>
      <div
        v-if="showActionsButtons"
        class="action"
      >
        <slot
          v-if="$slots.action"
          name="action"
        />
      </div>
    </div>
  </template>
```

5. To run the server and see your component, you need to open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



Material Card Header

Card Sub Header

The path of the righteous man is beset on all sides by the iniquities of the selfish and the tyranny of evil men.

How it works...

Each Vue component is a JavaScript object that has a render function. This render function is called when it is time to render it in the HTML DOM. A single file component is an abstraction of this object.

When we are declaring that our component has unique props that can be passed, it opens a tiny door for other components or JavaScript to place information inside our component. We are then able to use those values inside our component to render data, do some calculations, or make visual rules.

In our case, using the single file component, we are passing those rules as HTML attributes because `vue-template-compiler` will take those attributes and transform them into JavaScript objects.

When those values are passed to our component, Vue first checks whether the passed attribute matches the correct type, and then we execute our validation function on top of each value to see whether it matches what we'd expect.

After all of this is done, the component life cycle continues, and we can render our component.

See also

You can find more information about `props` at `https://vuejs.org/v2/guide/components-props.html`.

You can find more information about `vue-template-compiler` at `https://vue-loader.vuejs.org/guide/`.

Creating functional components

The beauty of functional components is their simplicity. They're a stateless component, without any data, computed property, or even a life cycle. They're just a render function that is called when the data that is passed changed.

You may be wondering how this can be useful. Well, a functional component is a perfect companion for UI components that don't need to keep any data inside them, or visual components that are just rendered components that don't require any data manipulation.

As the name implies, they are simple function components, and they have nothing more than the render function. They are a stripped-down version of a component used exclusively for performance rendering and visual elements.

Getting ready

The pre-requisite is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, create your Vue project with Vue-CLI, as we did in the recipe '*Creating Your first project with Vue CLI*' in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*, or use the project from the '*Passing data to your component and validating the data*' recipe.

Now, follow these instructions to create a Vue functional component:

1. Create a new file called `MaterialButton.vue` in the `src/components` folder.
2. In this component, we need to validate whether the prop we'll receive is a valid color. To do this, install in the project the `is-color` module. You'll need to open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > npm install --save is-color
   ```

3. In the `<script>` part of our component, we need to create the `props` object that the functional component will receive. As a functional component is just a render function with no state – it's stateless – the `<script>` part of the component is trimmed down to `props`, `injections`, and `slots`. There will be four `props` objects: `backgroundColor`, `textColor`, `isRound`, and `isFlat`. These won't be required when installing the component, as we will have a default value defined in `props`:

   ```
   <script>
     import isColor from 'is-color';

     export default {
       name: 'MaterialButton',
       props: {
   ```

```
          backgroundColor: {
            type: String,
            required: false,
            default: '#fff',
            validator: v => typeof v === 'string' && isColor(v),
          },
          textColor: {
            type: String,
            required: false,
            default: '#000',
            validator: v => typeof v === 'string' && isColor(v),
          },
          isRound: {
            type: Boolean,
            required: false,
            default: false,
          },
          isFlat: {
            type: Boolean,
            required: false,
            default: false,
          },
        },
      };
    </script>
```

4. In the `<template>` part of our component, we first need to add the `functional` attribute to the `<template>` tag to indicate to the `vue-template-compiler` that this component is a functional component. We need to create a button HTML element, with a basic `class` attribute button and a dynamic `class` attribute based on the `props` object received. Different from the normal component, we need to specify the `props` property in order to use the functional component. For the style of the button, we need to create a dynamic `style` attribute, also based on `props`. To emit all the event listeners directly to the parent, we can call the `v-on` directive and pass the `listeners` property. This will bind all the event listeners without needing to declare each one. Inside the button, we will add a `div` HTML element for visual enhancement, and add `<slot>` where the text will be placed:

```
    <template functional>
      <button
        tabindex="0"
        class="button"
        :class="{
          round: props.isRound,
          isFlat: props.isFlat,
      }"
      :style="{
        background: props.backgroundColor,
        color: props.textColor
      }"
      v-on="listeners"
    >
      <div
        tabindex="-1"
        class="button_focus_helper"
      />
      <slot/>
    </button>
  </template>
```

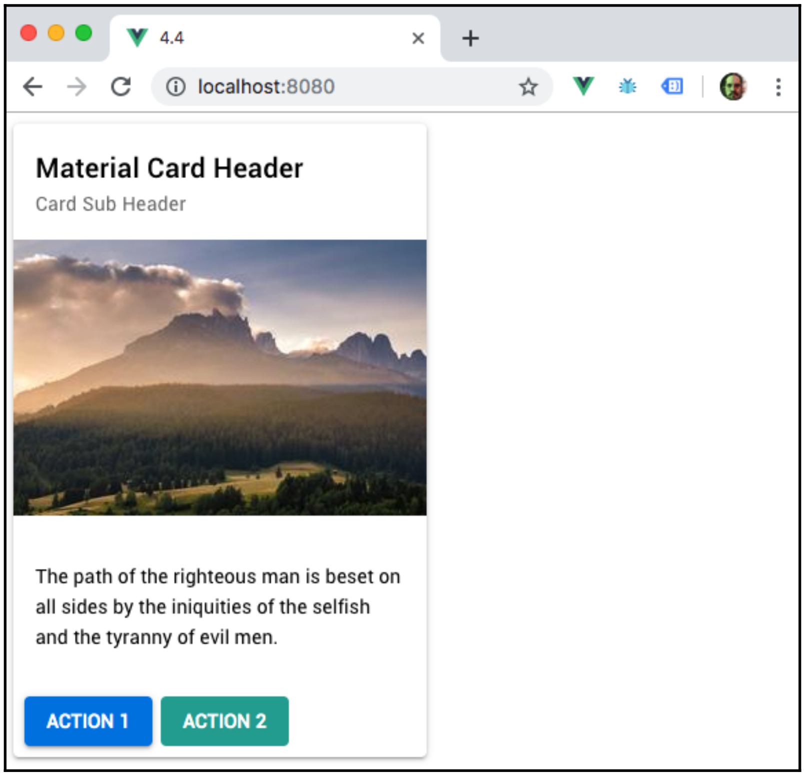
5. Now, let's make it pretty. In the `<style>` part of the component, we need to create all the CSS stylesheet rules for this button. We need to add the `scoped` attribute to `<style>` so that all the CSS stylesheet rules won't affect any other elements in our application:

```
<style scoped>
  .button {
    user-select: none;
    position: relative;
    outline: 0;
    border: 0;
    border-radius: 0.25rem;
    vertical-align: middle;
    cursor: pointer;
    padding: 4px 16px;
    font-size: 14px;
    line-height: 1.718em;
    text-decoration: none;
    color: inherit;
    background: transparent;
    transition: 0.3s cubic-bezier(0.25, 0.8, 0.5, 1);
    min-height: 2.572em;
    font-weight: 500;
    text-transform: uppercase;
  }
  .button:not(.isFlat){
    box-shadow: 0 1px 5px rgba(0, 0, 0, 0.2),
    0 2px 2px rgba(0, 0, 0, 0.14),
    0 3px 1px -2px rgba(0, 0, 0, 0.12);
  }

  .button:not(.isFlat):focus:before,
  .button:not(.isFlat):active:before,
  .button:not(.isFlat):hover:before {
    content: '';
    position: absolute;
    top: 0;
    right: 0;
    bottom: 0;
    left: 0;
    border-radius: inherit;
    transition: 0.3s cubic-bezier(0.25, 0.8, 0.5, 1);
  }

  .button:not(.isFlat):focus:before,
  .button:not(.isFlat):active:before,
  .button:not(.isFlat):hover:before {
    box-shadow: 0 3px 5px -1px rgba(0, 0, 0, 0.2),
    0 5px 8px rgba(0, 0, 0, 0.14),
    0 1px 14px rgba(0, 0, 0, 0.12);
  }

  .button_focus_helper {
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    pointer-events: none;
    border-radius: inherit;
    outline: 0;
    opacity: 0;
    transition: background-color 0.3s cubic-bezier(0.25, 0.8, 0.5,
1),
    opacity 0.4s cubic-bezier(0.25, 0.8, 0.5, 1);
  }

  .button_focus_helper:after, .button_focus_helper:before {
    content: '';
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    opacity: 0;
    border-radius: inherit;
    transition: background-color 0.3s cubic-bezier(0.25, 0.8, 0.5,
1),
    opacity 0.6s cubic-bezier(0.25, 0.8, 0.5, 1);
  }
  .button_focus_helper:before {
    background: #000;
  }

  .button_focus_helper:after {
    background: #fff;
  }

  .button:focus .button_focus_helper:before,
  .button:hover .button_focus_helper:before {
    opacity: .1;
  }

  .button:focus .button_focus_helper:after,
  .button:hover .button_focus_helper:after {
    opacity: .6;
  }

  .button:focus .button_focus_helper,
  .button:hover .button_focus_helper {
    opacity: 0.2;
  }

  .round {
    border-radius: 50%;
  }
</style>
```

6. To run the server and see your component, you need to open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



Material Card Header

Card Sub Header

The path of the righteous man is beset on all sides by the iniquities of the selfish and the tyranny of evil men.

How it works...

Functional components are as simple as a render function. They don't have any sort of data, function, or access to the outside world.

They were first introduced in Vue as a JavaScript object `render()` function only; later, they were added to `vue-template-compiler` for the Vue single file application.

A functional component works by receiving two arguments: `createElement` and `context`. As we saw in the single file, we only had access to the elements as they weren't in the `this` property of the JavaScript object. This occurs because as the context is passed to the render function, there is no `this` property.

A functional component provides the fastest rendering possible on Vue, as it doesn't depend on the life cycle of a component to check for the rendering; it just renders each time data is changed.

See also

You can find more information about functional components at `https://vuejs.org/v2/guide/render-function.html#Functional-Components`.

You can find more information about the `is-color` module at `https://www.npmjs.com/package/is-color`.

Accessing your children components data

Normally, parent-child communications are done via events or props. But sometimes, you need to access data, functions, or computed properties that exist in the child or the parent function.

Vue provides a way to interact in both ways, opening doors to communications and events, such as props and event listeners.

There is another way to access the data between the components: by using direct access. This can be done with the help of a special attribute in the template when using the single file component or a direct call of the object inside the JavaScript. This method is seen by some as a little lazy, but there are times when there really is no other way to do it than this.

Getting ready

The pre-requisite is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start your component, create your Vue project with Vue-CLI, as we did in the '*Creating Your first project with Vue CLI*' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*, or use the project from the '*Creating functional components*' recipe.

We're going to separate the recipe into four parts. The first three parts will cover the creation of new components – `StarRatingInput`, `StarRatingDisplay`, and `StarRating` – and the last part will cover the parent-child direct manipulation of the data and function access.

Creating the star rating input

We are going to create a star rating input, based on a five-star ranking system.

Follow these steps to create a custom star rating input:

1. Create a new file called `StarRatingInput.vue` in the `src/components` folder.
2. In the `<script>` part of the component, create a `maxRating` property in the `props` property that is a number, non-required, and has a default value of `5`. In the `data` property, we need to create our `rating` property, with the default value of `0`. In the `methods` property, we need to create three methods: `updateRating`, `emitFinalVoting`, and `getStarName`. The `updateRating` method will save the rating to the data, `emitFinalVoting` will call `updateRating` and emit the rating to the parent component through a `final-vote` event, and `getStarName` will receive a value and return the icon name of the star:

```
<script>
export default {
  name: 'StarRatingInput',
  props: {
    maxRating: {
      type: Number,
      required: false,
      default: 5,
    },
  },
  data: () => ({
    rating: 0,
  }),
  methods: {
    updateRating(value) {
      this.rating = value;
    },
    emitFinalVote(value) {
      this.updateRating(value);
      this.$emit('final-vote', this.rating);
    },
    getStarName(rate) {
      if (rate <= this.rating) {
        return 'star';
      }
      if (Math.fround((rate - this.rating)) < 1) {
        return 'star_half';
      }
      return 'star_border';
    },
  },
};
</script>
```

3. In the `<template>` part of the component, we need to create a `<slot>` component to place the text before the star rating. We'll create a dynamic list of stars based on the `maxRating` value that we received via the `props` property. Each star that is created will have a listener attached to it in the `mouseenter`, `focus`, and `click` events. `mouseenter` and `focus`, when fired, will call the `updateRating` method, and `click` will call `emitFinalVote`:

```
<template>
  <div class="starRating">
    <span class="rateThis">
      <slot />
    </span>
    <ul>
      <li
        v-for="rate in maxRating"
        :key="rate"
        @mouseenter="updateRating(rate)"
        @click="emitFinalVote(rate)"
        @focus="updateRating(rate)"
      >
        <i class="material-icons">
          {{ getStarName(rate) }}
        </i>
      </li>
    </ul>
  </div>
</template>
```

4. We need to import the Material Design icons into our application. Create a new styling file in the `styles` folder called `materialIcons.css`, and add the CSS stylesheet rules for `font-family`:

```
@font-face {
  font-family: 'Material Icons';
  font-style: normal;
  font-weight: 400;
  src:
url(https://fonts.gstatic.com/s/materialicons/v48/flUhRq6tzZclQEJ-
      Vdg-IuiaDsNcIhQ8tQ.woff2) format('woff2');
}

.material-icons {
  font-family: 'Material Icons' !important;
  font-weight: normal;
  font-style: normal;
  font-size: 24px;
  line-height: 1;
  letter-spacing: normal;
  text-transform: none;
  display: inline-block;
  white-space: nowrap;
  word-wrap: normal;
  direction: ltr;
  -webkit-font-feature-settings: 'liga';
  -webkit-font-smoothing: antialiased;
}
```

5. Open the `main.js` file and import the created stylesheet into it. The `css-loader` webpack will handle the processing of imported `.css` files in JavaScript files. This will help development because you don't need to re-import the file elsewhere:

```
import Vue from 'vue';
import App from './App.vue';
import './style/materialIcons.css';

Vue.config.productionTip = false;

new Vue({
  render: h => h(App),
}).$mount('#app');
```

6. To style our component, we will create a common styling file in the `src/style` folder called `starRating.css`. There we will add the common styles that will be shared between the `StarRatingDisplay` and `StarRatingInput` components:

```
.starRating {
  user-select: none;
  display: flex;
  flex-direction: row;
}
.starRating * {
  line-height: 0.9rem;
}
.starRating .material-icons {
  font-size: .9rem !important;
  color: orange;
}

ul {
  display: inline-block;
  padding: 0;
  margin: 0;
}

ul > li {
  list-style: none;
  float: left;
}
```

7. In the `<style>` part of the component, we need to create all the CSS stylesheet rules. Then, on the `StarRatingInput.vue` component file located in the `src/components` folder we need to add the `scoped` attribute to `<style>` so that all the CSS stylesheet rules won't affect any other elements in our application. Here, we will import the common styles that we created and add new ones for the input:

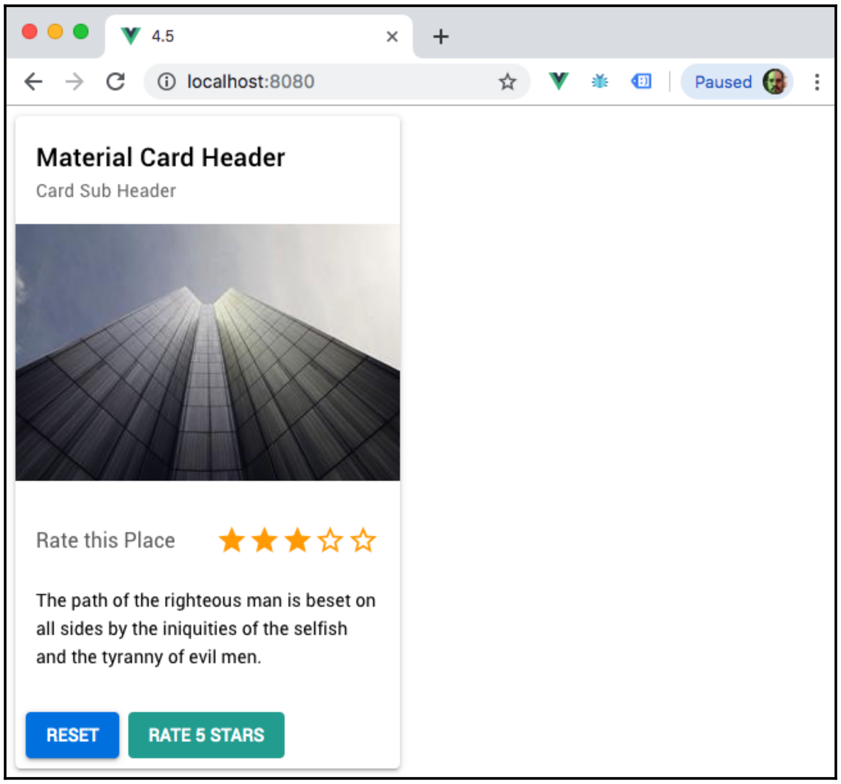
```
<style scoped>
  @import '../style/starRating.css';

  .starRating {
    justify-content: space-between;
  }

  .starRating * {
    line-height: 1.7rem;
  }
```

```
    .starRating .material-icons {
      font-size: 1.6rem !important;
    }

    .rateThis {
      display: inline-block;
      color: rgba(0, 0, 0, .65);
      font-size: 1rem;
    }
  </style>
```

8. To run the server and see your component, you need to open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
  > npm run serve
```

Here is your component rendered and running:



Material Card Header

Card Sub Header

Rate this Place

The path of the righteous man is beset on all sides by the iniquities of the selfish and the tyranny of evil men.

Creating the StarRatingDisplay component

Now that we have our input, we need a way to display the selected choice to the user. Follow these steps to create a `StarRatingDisplay` component:

1. Create a new component called `StarRatingDisplay.vue` in the `src/components` folder.
2. In the `<script>` part of the component, in the `props` property, we need to create three new properties: `maxRating`, `rating`, and `votes`. All three of them will be numbers and non-required and have a default value. In the `methods` property, we need to create a new method called `getStarName`, which will receive a value and return the icon name of the star:

```
<script>
export default {
  name: 'StarRatingDisplay',
  props: {
    maxRating: {
      type: Number,
      required: false,
      default: 5,
    },
    rating: {
      type: Number,
      required: false,
      default: 0,
    },
    votes: {
      type: Number,
      required: false,
      default: 0,
    },
  },
  methods: {
    getStarName(rate) {
      if (rate <= this.rating) {
        return 'star';
      }
      if (Math.fround((rate - this.rating)) < 1) {
        return 'star_half';
      }
      return 'star_border';
    },
  },
};
</script>
```

3. In `<template>`, we need to create a dynamic list of stars based on the `maxRating` value that we received via the `props` property. After the list, we need to display that we received votes, and if we receive any votes, we will display them too:

```
<template>
  <div class="starRating">
    <ul>
      <li
        v-for="rate in maxRating"
        :key="rate"
      >
        <i class="material-icons">
          {{ getStarName(rate) }}
        </i>
      </li>
    </ul>
    <span class="rating">
      {{ rating }}
    </span>
    <span
      v-if="votes"
      class="votes"
    >
      ({{ votes }})
    </span>
  </div>
</template>
```

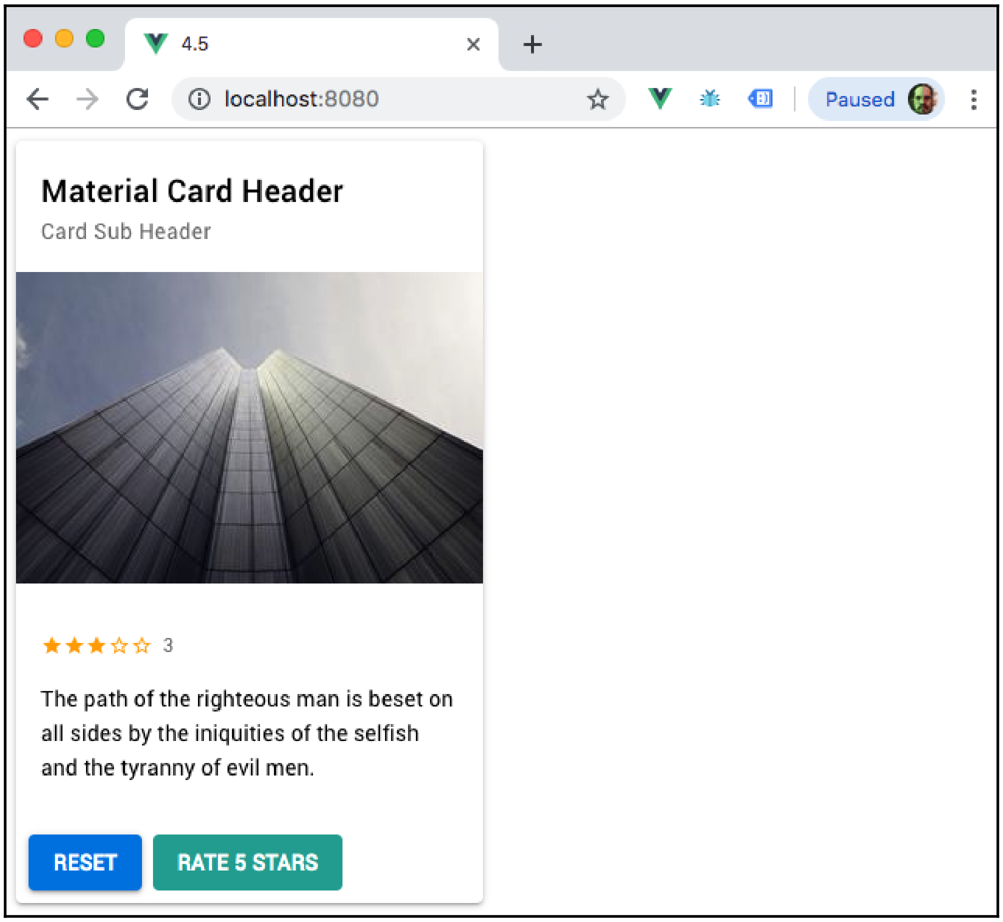
4. In the `<style>` part of the component, we need to create all the CSS stylesheet rules. We need to add the `scoped` attribute to `<style>` so that all the CSS stylesheet rules won't affect any other elements in our application. Here, we will import the common styles that we created and add new ones for the display:

```
<style scoped>
  @import '../style/starRating.css';

  .rating, .votes {
    display: inline-block;
    color: rgba(0,0,0, .65);
    font-size: .75rem;
    margin-left: .4rem;
  }
</style>
```

5. To run the server and see your component, you need to open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



Material Card Header

Card Sub Header

★★★☆☆ 3

The path of the righteous man is beset on all sides by the iniquities of the selfish and the tyranny of evil men.

Creating the StarRating component

After creating the input and the display, we need to join both together in a single component. This component will be the final component that we'll use in the application.

Follow these steps to create the final `StarRating` component:

1. Create a new file called `StarRating.vue` in the `src/components` folder.
2. In the `<script>` part of the component, we need to import the `StarRatingDisplay` and `StarRatingInput` components. In the `props` property, we need to create three new properties: `maxRating`, `rating`, and `votes`. All three of them will be numbers and non-required, with a default value. In the `data` property, we need to create our `rating` property, with a default value of `0`, and a property called `voted`, with a default value of `false`. In the `methods` property, we need to add a new method called `vote`, which will receive `rank` as an argument. It will define `rating` as the received value and define the inside variable of the `voted` component as `true`:

```
<script>
import StarRatingInput from './StarRatingInput.vue';
import StarRatingDisplay from './StarRatingDisplay.vue';

export default {
  name: 'StarRating',
  components: { StarRatingDisplay, StarRatingInput },
  props: {
    maxRating: {
      type: Number,
      required: false,
      default: 5,
    },
    rating: {
      type: Number,
      required: false,
      default: 0,
    },
    votes: {
      type: Number,
      required: false,
      default: 0,
    },
  },
  data: () => ({
    rank: 0,
    voted: false,
  }),
  methods: {
    vote(rank) {
      this.rank = rank;
      this.voted = true;
    },
  },
};
</script>
```

3. In the `<template>` part, we will place both the components, displaying the input of the rating:

```
<template>
  <div>
    <StarRatingInput
      v-if="!voted"
      :max-rating="maxRating"
      @final-vote="vote"
    >
      Rate this Place
    </StarRatingInput>
    <StarRatingDisplay
      v-else
      :max-rating="maxRating"
      :rating="rating || rank"
      :votes="votes"
    />
  </div>
</template>
```

Data manipulation on child components

Now that all of our components are ready, we need to add them to our application. The base application will access the child component, and it will set the rating to 5 stars.

Now, follow these steps to understand and manipulate the data in the child components:

1. In the `App.vue` file, in the `<template>` part of the component, remove the `main-text` attribute of the `MaterialCardBox` component and place it as the default slot of the component.

2. Before the placed text, we will add the `StarRating` component. We will add a `ref` attribute to it. This attribute will indicate to Vue to link this component directly to a special property in the `this` object of the component. In the action buttons, we will add the listeners for the click event—one for `resetVote` and another for `forceVote`:

```
<template>
  <div id="app">
    <MaterialCardBox
      header="Material Card Header"
      sub-header="Card Sub Header"
      show-media
      show-actions
      img-src="https://picsum.photos/300/200"
    >
      <p>
        <StarRating
          ref="starRating"
        />
      </p>
      <p>
        The path of the righteous man is beset on all sides by the
           iniquities of the selfish and the tyranny of evil men.
      </p>
      <template v-slot:action>
        <MaterialButton
          background-color="#027be3"
          text-color="#fff"
          @click="resetVote"
        >
          Reset
        </MaterialButton>
        <MaterialButton
          background-color="#26a69a"
          text-color="#fff"
          is-flat
          @click="forceVote"
        >
          Rate 5 Stars
        </MaterialButton>
      </template>
    </MaterialCardBox>
  </div>
</template>
```

3. In the `<script>` part of the component, we will create a `methods` property, and add two new methods: `resetVote` and `forceVote`. Those methods will access the `StarRating` component and reset the data or set the data to a 5-star vote, respectively:

```
<script>
import MaterialCardBox from './components/MaterialCardBox.vue';
import MaterialButton from './components/MaterialButton.vue';
import StarRating from './components/StarRating.vue';

export default {
  name: 'App',
  components: {
    StarRating,
    MaterialButton,
    MaterialCardBox,
  },
  methods: {
    resetVote() {
      this.$refs.starRating.rank = 0;
      this.$refs.starRating.voted = false;
    },
    forceVote() {
      this.$refs.starRating.rank = 5;
      this.$refs.starRating.voted = true;
    },
  },
};
</script>
```

How it works...

When the `ref` property is added to the component, Vue adds a link to the referenced element to the `$refs` property inside the `this` property object of JavaScript. From there, you have full access to the component.

This method is commonly used to manipulate HTML DOM elements without the need to call for document query selector functions.

However, the main function of this property is to give access to the Vue component directly, enabling you the ability to execute functions and see the computed properties, variables, and changed variables of the component—like full access to the component from the outside.

There's more...

In the same way that a parent can access a child component, a child can access a parent component by calling `$parent` on the `this` object. An event can access the root element of the Vue application by calling the `$root` property.

See also

You can find more information about parent-child communication at `https://vuejs.org/v2/guide/components-edge-cases.html#Accessing-the-Parent-Component-Instance`.

Creating a dynamic injected component

There are some cases where your component can be defined by the kind of variable you are receiving or the type of data that you have; then, you need to change the component on the fly, without the need to set a lot of Vue `v-if`, `v-else-if`, and `v-else` directives.

In those cases, the best thing to do is to use dynamic components, when a computed property or a function can define the component that will be used to be rendered, and the decision is taken in real time.

These decisions sometimes can be simple if there are two responses, but they can be more complex with a long switch case, where you may have a long list of possible components to be used.

Getting ready

The pre-requisite is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can create our Vue project with Vue-CLI, as we did in the '*Creating Your first project with Vue CLI*' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*, or use the project from the '*Accessing your children components data*' recipe.

Follow these steps to create a dynamic injected component:

1. Open the `StarRating.vue` component.
2. In the `<script>` part of the component, we need to create a `computed` property with a new computed value called `starComponent`. This value will check whether the user has voted. If they haven't, it will return the `StarRatingInput` component; otherwise, it will return the `StarRatingDisplay` component:

```
<script>
import StarRatingInput from './StarRatingInput.vue';
import StarRatingDisplay from './StarRatingDisplay.vue';

export default {
  name: 'StarRating',
  components: { StarRatingDisplay, StarRatingInput },
  props: {
    maxRating: {
      type: Number,
      required: false,
      default: 5,
    },
    rating: {
      type: Number,
      required: false,
      default: 0,
    },
    votes: {
      type: Number,
      required: false,
      default: 0,
    },
  },
  data: () => ({
    rank: 0,
    voted: false,
  }),
  computed: {
    starComponent() {
      if (!this.voted) return StarRatingInput;
      return StarRatingDisplay;
    },
  },
  methods: {
    vote(rank) {
      this.rank = rank;
      this.voted = true;
    },
  },
};
</script>
```

3. In the `<template>` part of the component, we will remove both of the existing components and replace them with a special component called `<component>`. This special component has a named attribute that you can point to anywhere that returns a valid Vue component. In our case, we will point to the computed `starComponent` property. We will take all the bind props that were defined from both of the other components and put them inside this new component, including the text that is placed in `<slot>`:

```
<template>
  <component
    :is="starComponent"
    :max-rating="maxRating"
    :rating="rating || rank"
    :votes="votes"
    @final-vote="vote"
  >
    Rate this Place
  </component>
</template>
```

How it works...

Using the Vue special `<component>` component, we declared what the component should render according to the rules set on the computed property.

Being a generic component, you always need to guarantee that everything will be there for each of the components that can be rendered. The best way to do this is by using the `v-bind` directive with the props and rules that need to be defined, but it's possible to define it directly on the component also, as it will be passed down as a prop.

See also

You can find more information about dynamic components at `https://vuejs.org/v2/guide/components.html#Dynamic-Components`.

Creating a dependency injection component

Accessing data directly from a child or a parent component without knowing whether they exist can be very dangerous.

In Vue, it's possible to make your component behavior like an interface and have a common and abstract function that won't change in the development process. The process of dependency injection is a common paradigm in the developing world and has been implemented in Vue also.

There are some pros and cons to using the internal Vue dependency injection, but it is always a good way to make sure that your children's components know what to expect from the parent component when developing it.

Getting ready

The pre-requisite is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can create our Vue project with Vue-CLI, as we did in the '*Creating Your first project with Vue CLI*' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*, or use the project from the '*Creating a dynamic injected component*' recipe.

Now, follow these steps to create a dependency injection component:

1. Open the `StarRating.vue` component.
2. In the `<script>` part of the component, add a new property called `provide`. In our case, we will just be adding a key-value to check whether the component is a child of the specific component. Create an object in the property with the `starRating` key and the `true` value:

```
<script>
import StarRatingInput from './StarRatingInput.vue';
import StarRatingDisplay from './StarRatingDisplay.vue';

export default {
  name: 'StarRating',
  components: { StarRatingDisplay, StarRatingInput },
  provide: {
    starRating: true,
  },
  props: {
    maxRating: {
      type: Number,
      required: false,
      default: 5,
    },
    rating: {
      type: Number,
      required: false,
      default: 0,
    },
    votes: {
      type: Number,
      required: false,
      default: 0,
    },
  },
  data: () => ({
    rank: 0,
    voted: false,
  }),
  computed: {
    starComponent() {
      if (!this.voted) return StarRatingInput;
      return StarRatingDisplay;
    },
  },
  methods: {
    vote(rank) {
      this.rank = rank;
      this.voted = true;
    },
  },
};
</script>
```

3. Open the `StarRatingDisplay.vue` file.
4. In the `<script>` part of the component, we will add a new property called `inject`. This property will receive an object with a key named `starRating`, and the value will be an object that will have a `default()` function. This function will log an error if this component is not a child of the `StarRating` component:

```
<script>
export default {
  name: 'StarRatingDisplay',
  props: {
    maxRating: {
      type: Number,
      required: false,
      default: 5,
    },
    rating: {
      type: Number,
      required: false,
      default: 0,
    },
    votes: {
      type: Number,
      required: false,
      default: 0,
    },
  },
  inject: {
    starRating: {
      default() {
        console.error('StarRatingDisplay need to be a child of
          StarRating');
      },
    },
  },
  methods: {
    getStarName(rate) {
      if (rate <= this.rating) {
        return 'star';
      }
```

```
          if (Math.fround((rate - this.rating)) < 1) {
            return 'star_half';
          }
          return 'star_border';
        },
      },
    };
    </script>
```

5. Open the `StarRatingInput.vue` file.
6. In the `<script>` part of the component, we will add a new property called `inject`. This property will receive an object with a key named `starRating`, and the value will be an object that will have a `default()` function. This function will log an error if this component is not a child of the `StarRating` component:

```
    <script>
    export default {
      name: 'StarRatingInput',
      props: {
        maxRating: {
          type: Number,
          required: false,
          default: 5,
        },
      },
      inject: {
        starRating: {
          default() {
            console.error('StarRatingInput need to be a child of
              StarRating');
          },
        },
      },
      data: () => ({
        rating: 0,
      }),
      methods: {
        updateRating(value) {
          this.rating = value;
        },
        emitFinalVote(value) {
          this.updateRating(value);
          this.$emit('final-vote', this.rating);
        },
        getStarName(rate) {
          if (rate <= this.rating) {
```

```
          return 'star';
        }
        if (Math.fround((rate - this.rating)) < 1) {
          return 'star_half';
        }
        return 'star_border';
      },
    },
  };
</script>
```

How it works...

At runtime, Vue will check for the injected property of `starRating` in the `StarRatingDisplay` and `StarRatingInput` components, and if the parent component does not provide this value, it will log an error on the console.

Using component injection is commonly used to maintain a way of a common interface between bounded components, such as a menu and an item. An item may need some function or data that is stored in the menu, or we may need to check whether it's a child of the menu.

The main downside of dependency injection is that there is no more reactivity on the shared element. Because of that, it's mostly used to share functions or check component links.

See also

You can find more information about component dependency injection at `https://vuejs.org/v2/guide/components-edge-cases.html#Dependency-Injection`.

Creating a component mixin

There are times where you find yourself rewriting the same code over and over. However, there is a way to prevent this and make yourself far more productive.

You can use what is called a `mixin`, a special code import in Vue that joins code parts from outside your component to your current component.

Getting ready

The pre-requisite is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can create our Vue project with Vue-CLI, as we did in the recipe '*Creating Your First Project with Vue CLI*' in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*, or use the project from the '*Creating a dependency injection component*' recipe.

Let's follow these steps to create a component mixin:

1. Open the `StarRating.vue` component.
2. In the `<script>` part, we need to extract the `props` property into a new file called `starRatingDisplay.js` that we need to create in the `mixins` folder. This new file will be our first `mixin`, and will look like this:

```
export default {
  props: {
    maxRating: {
      type: Number,
      required: false,
      default: 5,
    },
    rating: {
      type: Number,
      required: false,
      default: 0,
    },
    votes: {
      type: Number,
      required: false,
      default: 0,
    },
  },
};
```

3. Back in the `StarRating.vue` component, we need to import this newly created file and add it to a new property called `mixin`:

```
<script>
import StarRatingInput from './StarRatingInput.vue';
import StarRatingDisplay from './StarRatingDisplay.vue';
import StarRatingDisplayMixin from '../mixins/starRatingDisplay';

export default {
  name: 'StarRating',
  components: { StarRatingDisplay, StarRatingInput },
  mixins: [StarRatingDisplayMixin],
  provide: {
    starRating: true,
  },
  data: () => ({
    rank: 0,
    voted: false,
  }),
  computed: {
    starComponent() {
      if (!this.voted) return StarRatingInput;
      return StarRatingDisplay;
    },
  },
  methods: {
    vote(rank) {
      this.rank = rank;
      this.voted = true;
    },
  },
};
</script>
```

4. Now, we will open the `StarRatingDisplay.vue` file.

5. In the `<script>` part, we will extract the `inject` property into a new file called `starRatingChild.js`, which will be created in the `mixins` folder. This will be our `mixin` for the `inject` property:

```
export default {
  inject: {
    starRating: {
      default() {
        console.error('StarRatingDisplay need to be a child of
           StarRating');
      },
    },
  },
};
```

6. Back in the `StarRatingDisplay.vue` file, in the `<script>` part, we will extract the `methods` property into a new file called `starRatingName.js`, which will be created in the `mixins` folder. This will be our `mixin` for the `getStarName` method:

```
export default {
  methods: {
    getStarName(rate) {
      if (rate <= this.rating) {
        return 'star';
      }
      if (Math.fround((rate - this.rating)) < 1) {
        return 'star_half';
      }
      return 'star_border';
    },
  },
};
```

7. Back in the `StarRatingDisplay.vue` file, we need to import those newly created files and add them to a new property called `mixin`:

```
<script>
import StarRatingDisplayMixin from '../mixins/starRatingDisplay';
import StarRatingNameMixin from '../mixins/starRatingName';
import StarRatingChildMixin from '../mixins/starRatingChild';

export default {
  name: 'StarRatingDisplay',
  mixins: [
    StarRatingDisplayMixin,
    StarRatingNameMixin,
      StarRatingChildMixin,
    ],
  };
  </script>
```

8. Open the `StarRatingInput.vue` file.
9. In the `<script>` part, we remove the `inject` properties and extract the `props` property into a new file called `starRatingBase.js`, which will be created in the `mixins` folder. This will be our `mixin` for the `props` property:

```
export default {
  props: {
    maxRating: {
      type: Number,
      required: false,
      default: 5,
    },
    rating: {
      type: Number,
      required: false,
      default: 0,
    },
  },
};
```

10. Back in the `StarRatingInput.vue` file, we need to rename the `rating` data property to `rank`, and in the `getStarName` method, we need to add a new constant that will receive either the `rating` props or the `rank` data. Finally, we need to import the `starRatingChild` `mixin` and the `starRatingBase` `mixin`:

```
<script>
import StarRatingBaseMixin from '../mixins/starRatingBase';
import StarRatingChildMixin from '../mixins/starRatingChild';

export default {
  name: 'StarRatingInput',
  mixins: [
    StarRatingBaseMixin,
    StarRatingChildMixin,
  ],
  data: () => ({
    rank: 0,
  }),
  methods: {
    updateRating(value) {
      this.rank = value;
    },
    emitFinalVote(value) {
      this.updateRating(value);
      this.$emit('final-vote', this.rank);
    },
    getStarName(rate) {
      const rating = (this.rating || this.rank);
      if (rate <= rating) {
        return 'star';
      }
      if (Math.fround((rate - rating)) < 1) {
        return 'star_half';
      }
      return 'star_border';
    },
  },
};
</script>
```

How it works...

Mixins work as an object merge, but do make sure you don't replace an already-existing property in your component with an imported one.

The order of the `mixins` properties is important as well, as they will be checked and imported as a `for` loop, so the last `mixin` won't change any properties from any of their ancestors.

Here, we took a lot of repeated parts of our code and split them into four different small JavaScript files that are easier to maintain and improve productivity without needing to rewrite code.

See also

You can find more information about mixins at `https://vuejs.org/v2/guide/mixins.html`.

Lazy loading your components

`webpack` and Vue were born to be together. When using `webpack` as the bundler for your Vue project, it's possible to make your components load when they are needed or asynchronously. This is commonly known as lazy loading.

Getting ready

The pre-requisite is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can create our Vue project with Vue-CLI, as we did in the '*Creating Your first project with Vue CLI*' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*, or use the project from the '*Creating a component mixin*' recipe.

Now, follow these steps to import your component with a lazy loading technique:

1. Open the `App.vue` file.
2. In the `<script>` part of the component, we will take the imports at the top of the script and transform them into lazy load functions for each component:

```
<script>
export default {
  name: 'App',
  components: {
    StarRating: () => import('./components/StarRating.vue'),
    MaterialButton: () =>
import('./components/MaterialButton.vue'),
    MaterialCardBox: () =>
      import('./components/MaterialCardBox.vue'),
  },
  methods: {
    resetVote() {
      this.$refs.starRating.rank = 0;
      this.$refs.starRating.voted = false;
    },
    forceVote() {
      this.$refs.starRating.rank = 5;
      this.$refs.starRating.voted = true;
    },
  },
};
</script>
```

How it works...

When we declare a function that returns an `import()` function for each component, `webpack` knows that this import function will be code-splitting, and it will make the component a new file on the bundle.

The `import()` function was introduced as a proposal by the TC39 for module loading syntax. The base functionality of this function is to load any module that is declared asynchronously, avoiding the need to place all the files on the first load.

See also

You can find more information about async components at `https://vuejs.org/v2/guide/components-dynamic-async.html#Async-Components`.

You can find more information about the TC39 dynamic import at `https://github.com/tc39/proposal-dynamic-import`.

5 Fetching Data from the Web via HTTP Requests

Data is a part of everyday life nowadays. If it weren't for data, you wouldn't be reading this book or trying to learn more about Vue.

Knowing how to fetch and send your data inside an application is a requirement for a developer, not just an extra skill that's nice to have. The best way to learn it is by practicing it and finding out how it is done behind the scenes.

In this chapter, we will learn how to build our own API data manipulation with the Fetch API and the most popular API library in the web right now, `axios`.

In this chapter, we'll cover the following recipes:

- Creating a wrapper for the Fetch API as an HTTP client
- Creating a random cat image or GIF component
- Creating your local fake JSON API server with `MirageJS`
- Using `axios` as the new HTTP client
- Creating different `axios` instances
- Creating a request and response interceptor for `axios`
- Creating a CRUD interface with `axios` and `Vuesax`

Technical requirements

In this chapter, we will be using Node.js and Vue CLI.

Attention, Windows users! You need to install an NPM package called `windows-build-tools` to be able to install the following required packages. To do this, open PowerShell as administrator and execute the following command:

```
> npm install -g windows-build-tools
```

To install Vue CLI, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install -g @vue/cli @vue/cli-service-global
```

Creating a wrapper for the Fetch API as an HTTP client

The Fetch API is the child of the old `XMLHttpRequest`. It has an improved API and a new and powerful set of features completely based on `Promises`.

The Fetch API is both simple and based on a generic definition of two objects, `Request`, and `Response`, which allow it to be used everywhere in the browser. The browser Fetch API can be executed inside the `window` or the `service worker` as well. There is no limitation on the usage of this API.

In this recipe, we will learn how to create a wrapper around the Fetch API to make the API calls more simple.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can use the Vue project with Vue CLI we created in the '*Creating Your first project with Vue CLI*' recipe in `Chapter 2`, *Introducing TypeScript and the Vue Ecosystem*, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create http-project
```

The CLI will ask some questions that will help with the creation of the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *Spacebar* to select an option. Choose the `default` option:

```
? Please pick a preset: (Use arrow keys)
❯ default (babel, eslint)
  Manually select features
```

Creating the wrapper

First, we need to create a new API wrapper to be used in this recipe. This will be the main file we will use on all the HTTP methods.

Let's create the base wrapper by following these steps:

1. Create a new file called `baseFetch.js` in the `src/http` folder.
2. We will create an asynchronous function that will receive as an argument the three variables of `url`, `method`, and `options`. This will be a currying function, which the second function will receive as an argument, `type`:

```
export default async (url, method, options = {}) => {
  let httpRequest;
  if (method.toUpperCase() === 'GET') {
    httpRequest = await fetch(url, {
      cache: 'reload',
      ...options,
    });
  } else {
    httpRequest = fetch(url, {
      method: method.toUpperCase(),
      cache: 'reload',
      ...options,
    });
  }
```

```
            return (type) => {
              switch (type.toLocaleLowerCase()) {
                case 'json':
                  return httpRequest.json();
                case 'blob':
                  return httpRequest.blob();
                case 'text':
                  return httpRequest.text();
                case 'formdata':
                  return httpRequest.formData();
                default:
                  return httpRequest.arrayBuffer();
              }
            };
          };
```

Creating the API methods

Now we need to make our HTTP method function. These functions will use the wrapper to execute the browser Fetch API and return the response.

Follow these steps to create each one of the API method calls:

1. Let's create a new file called `fetchApi.js` in the `src/http` folder.
2. We need to import the `baseHttp` from the file we created in the first step:

```
import baseHttp from './baseFetch';
```

Now in the following parts, we will create each one of the HTTP methods that will be available in our wrapper.

GET method function

In these steps, we are going to create the *HTTP GET* method. Follow each of the following instructions to create the `getHttp` function:

1. Create a constant called `getHttp`.
2. Define that constant as an asynchronous function that receives three arguments, `url`, `type`, and `options`. The `type` argument will have the default value of `'json'`.

3. In this function return, we will execute the baseHttp function, passing the url that we received, 'get' as the second argument, the options as the third argument, and immediately execute the function with the type argument we received:

```
export const getHttp = async (url, type = 'json', options) =>
(await
   baseHttp(url, 'get', options))(type);
```

POST method function

In this part, we are creating the *HTTP POST* method. Follow these steps to create the postHttp function:

1. Create a constant called postHttp.
2. Assign to that constant an asynchronous function that receives four arguments, url, body, type, and options. The type argument will have the default value of 'json'.
3. In this function return, we will execute the baseHttp function, passing the url argument that we received, and 'post' as the second argument. In the third argument, we will pass an object with the body variable, and the deconstructed options argument that we received. Because of the currying property of baseHttp, we will execute the returned function with the type argument we received. The body is usually a JSON or a JavaScript object. If this request is going to be a file upload, body needs to be a FormData object:

```
export const postHttp = async (
  url,
  body,
  type = 'json',
  options,
) => (await baseHttp(url,
  'post',
  {
    body,
    ...options,
  }))(type);
```

PUT method function

Now we are creating an *HTTP PUT* method. Use the following steps to create the `putHttp` function:

1. Create a constant called `putHttp`.
2. Assign to that constant an asynchronous function that receives four arguments, `url`, `body`, `type`, and `options`. The `type` argument will have the default value of `'json'`.
3. In this function return, we will execute the `baseHttp` function, passing the `url` that we received, and `'put'` as the second argument. In the third argument, we will pass an object with the `body` variable, and the deconstructed `options` argument that we received. Because of the currying property of `baseHttp`, we will execute the returned function with the `type` argument we received. `body` is usually a JSON or a JavaScript object, but if this request is going to be a file upload, `body` needs to be a `FormData` object:

```
export const putHttp = async (
  url,
  body,
  type = 'json',
  options,
) => (await baseHttp(url,
  'put',
  {
    body,
    ...options,
  }))(type);
```

PATCH method function

It's time to create an *HTTP PATCH* method. Follow these steps to create the `patchHttp` function:

1. Create a constant called `patchHttp`.
2. Assign to that constant an asynchronous function that receives four arguments, `url`, `body`, `type`, and `options`. The `type` argument will have the default value of `'json'`.

3. In this function return, we will execute the baseHttp function, passing the url that we received, and 'patch' as the second argument. In the third argument, we will pass an object with the body variable, and the deconstructed options argument that we received. Because of the currying property of baseHttp, we will execute the returned function with the type we received. body is usually a JSON or a JavaScript object, but if this request is going to be a file upload, body needs to be a FormData object:

```
export const patchHttp = async (
  url,
  body,
  type = 'json',
  options,
) => (await baseHttp(url,
  'patch',
  {
    body,
    ...options,
  }))(type);
```

UPDATE method function

In this section, we are creating an *HTTP UPDATE* method. Follow these steps to create the updateHttp function:

1. Create a constant called updateHttp.
2. Assign to that constant an asynchronous function that receives four arguments, url, body, type, and options. The type argument will have the default value of 'json'.
3. In this function return, we will execute the baseHttp function, passing the url that we received, and 'update' as the second argument. In the third argument, we will pass an object with the body variable, and the deconstructed options argument that we received. Because of the currying property of baseHttp, we will execute the returned function with the type we received. body is usually a JSON or a JavaScript object, but if this request is going to be a file upload, body needs to be a FormData object:

```
export const updateHttp = async (
  url,
  body,
  type = 'json',
  options,
) => (await baseHttp(url,
  'update',
  {
    body,
    ...options,
  }))(type);
```

DELETE method function

In this final step, we will create a *DELETE HTTP* method. Follow these steps to create the `deleteHttp` function:

1. Create a constant called `deleteHttp`.
2. Assign to that constant an asynchronous function that receives four arguments, `url`, `body`, `type`, and `options`. The type argument will have the default value of `'json'`.
3. In this function return, we will execute the `baseHttp` function, passing the `url` that we received, and `'delete'` as the second argument. In the third argument, we will pass an object with the `body` variable, and the deconstructed `options` argument that we received. Because of the currying property of `baseHttp`, we will execute the returned function with the `type` we received. `body` is usually a JSON or a JavaScript object, but if this request is going to be a file upload, `body` needs to be a `FormData` object:

```
export const deleteHttp = async (
  url,
  body,
  type = 'json',
  options,
) => (await baseHttp(url,
  'delete',
  {
    body,
    ...options,
  }))(type);
```

How it works...

In this recipe, we created a wrapper for the `Fetch` API that is presented on the `window` element. This wrapper consists of a currying and closure function, where the first function receives the URL data, method, and options for the Fetch API, and the resulting function is the Fetch API response translator.

In the wrapper, the first part of the function will create our `fetch` request. There, we need to check whether it's a *GET* method, so we just need to execute it with the `url` parameter and omit the others. The second part of the function is responsible for the conversion of the `fetch` response. It will switch between the `type` parameter and execute the retrieving function according to the correct one.

To receive the final data for your request, you always need to call the response translator after the request, as in the following example:

```
getHttp('https://jsonplaceholder.typicode.com/todos/1',
          'json').then((response) => { console.log(response)); }
```

This will get the data from the URL and transform the response into a JSON/JavaScript object.

The second part we made was the methods translator. We made functions for each one of the REST verbs to be used more easily. The GET verb doesn't have the ability to pass any `body` but all the others are capable of passing a `body` in the request.

See also

You can find more information about the Fetch API at `https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API`.

You can find more information about FormData at `https://developer.mozilla.org/en-US/docs/Web/API/FormData/FormData`.

You can find more information about the Fetch response body at `https://developer.mozilla.org/en-US/docs/Web/API/Body/body`.

You can find more information about headers at `https://developer.mozilla.org/en-US/docs/Web/API/Headers`.

You can find more information about requests at `https://developer.mozilla.org/en-US/docs/Web/API/Request`.

Creating a random cat image or GIF component

It's common knowledge that the internet is made of many GIFs and videos of cats. I'm sure that if we took down all cat-related content, we would have a web blackout.

The best way to understand more about the Fetch API and how it can be used inside a component is to make a random cat image or GIF component.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can use the Vue project with Vue CLI that we used in the '*Creating a wrapper for the Fetch API as an HTTP client*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create http-project
```

The CLI will ask some questions that will help with the creation of the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *Spacebar* to select an option. Choose the `default` option:

```
? Please pick a preset: (Use arrow keys)
❯ default (babel, eslint)
  Manually select features
```

Creating the component

In this recipe, we will be using the components created in `Chapter 4`, *Components, Mixins, and Functional Components,* for visual elements. You can achieve the same results with simple HTML elements.

We will divide the creation of this component in three steps: `<script>`, `<template>`, and `<style>`.

Single file component <script> section

Follow these steps to create the `<script>` section of the single file component:

1. Create a new file called `RandomCat.vue` in the `src/components` folder and open it.
2. Import the `getHttp` function from the `fetchApi` wrapper we made in the '*Creating a wrapper for the Fetch API as an HTTP client*' recipe:

```
import { getHttp } from '../http/fetchApi';
```

3. Asynchronously import the `MaterialButton` and `MaterialCardBox` components in the `component` property:

```
components: {
  MaterialButton: () => import('./MaterialButton.vue'),
  MaterialCardBox: () => import('./MaterialCardBox.vue'),
},
```

4. In the `data` property, we need to create a new data value named `kittyImage`, which will be by default an empty string:

```
data: () => ({
  kittyImage: '',
}),
```

5. In the `methods` property, we need to create the `getImage` method, which will fetch the image as a `Blob` and return it as a `URL.createObjectURL`. We also need to create the `newCatImage` method that will fetch a new still image of a cat and the `newCatGif` method that will fetch a new cat GIF:

```
methods: {
    async getImage(url) {
      return URL.createObjectURL(await getHttp(url, 'blob'));
    },
    async newCatImage() {
      this.kittyImage = await
          this.getImage('https://cataas.com/cat');
    },
    async newCatGif() {
      this.kittyImage = await
          this.getImage('https://cataas.com/cat/gif');
    },
  },
```

6. In the `beforeMount` life cycle hook, we need to make it asynchronous and execute the `newCatImage` method:

```
async beforeMount() {
    await this.newCatImage();
  },
```

Single file component <template> section

Follow these steps to create the `<template>` section of the single file component:

1. First, we need to add the `MaterialCardBox` component with a header and sub-header, activate the `media` and `action` sections, and create the `<template>` named slots for `media` and `action`:

```
<MaterialCardBox
  header="Cat as a Service"
  sub-header="Random Cat Image"
  show-media
  show-actions
>
  <template
    v-slot:media>
  </template>
  <template v-slot:action>
  </template>
</MaterialCardBox>
```

2. In the `<template>` named slot for `media`, we need to add an `<img>` element that will receive a URI `Blob`, which will be displayed when there is any data in the `kittyImage` variable, or it will display a loading icon:

```
<img
  v-if="kittyImage"
  alt="Meow!"
  :src="kittyImage"
  style="width: 300px;"
>
 <p v-else style="text-align: center">
   <i class="material-icons">
     cached
   </i>
 </p>
```

3. At the `<template>` named slot for `action`, we will create two buttons, one for fetching cat images and another for fetching cat GIFs, and both will have an event listener on the `@click` directive that calls a function that fetches the corresponding image:

```
<MaterialButton
  background-color="#4ba3c7"
  text-color="#fff"
  @click="newCatImage"
>
  <i class="material-icons">
    pets
  </i> Cat Image
</MaterialButton>
<MaterialButton
  background-color="#005b9f"
  text-color="#fff"
  @click="newCatGif"
>
  <i class="material-icons">
    pets
  </i> Cat GIF
</MaterialButton>
```

Single file component <style> section

In the `<style>` part of the component, we need to set the `body font-size` for the CSS style calculation based on `rem` and `em`:

```
<style>
  body {
    font-size: 14px;
  }
</style>
```

Getting up and running with your new component

Follow these steps to add your component to your Vue application:

1. Open the `App.vue` file in the `src` folder.
2. In the `components` property, asynchronously import the `RandomCat.vue` component:

    ```
    <script>
    export default {
      name: 'App',
      components: {
        RandomCat: () => import('./components/RandomCat'),
      },
    };
    </script>
    ```

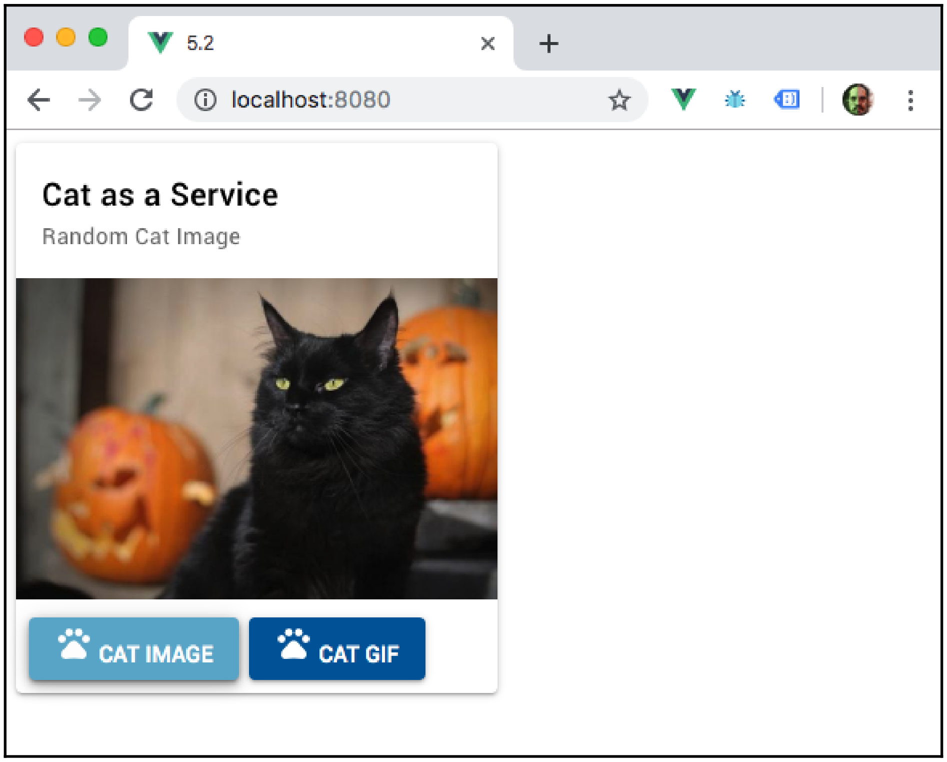
3. In the `<template>` section of the file, declare the imported component:

    ```
    <template>
      <div id="app">
        <random-cat />
      </div>
    </template>
    ```

To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



How it works...

Using the `getHttp` wrapper, the component was able to get the URL and retrieve it as a `Blob` type. With this response, we can use the `URL.createObjectUrl` navigator method and pass the `Blob` as an argument to get a valid image URL that can be used as the `src` attribute.

See also

You can find more information about `URL.createObjectUrl` at `https://developer.mozilla.org/en-US/docs/Web/API/URL/createObjectURL`.

You can find more information about the `Blob` response type at `https://developer.mozilla.org/en-US/docs/Web/API/Body/blob`.

Creating your fake JSON API server with MirageJS

Faking data for testing, developing, or designing is always a problem. You need to have a big JSON or make a custom server to handle any data changes when presenting the application at the development stage.

There is now a way to help developers and UI designers achieve this without needing to code an external server – a new tool called MirageJS, a server emulator that runs on the browser.

In this recipe, we will learn how to use the MirageJS as a mock server and execute HTTP requests on it.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can use the Vue project with Vue CLI that we did in the '*Creating a wrapper for the Fetch API as an HTTP client*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create visual-component
```

The CLI will ask some questions that will help with the creation of the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *Spacebar* to select an option. Choose the `default` option:

```
? Please pick a preset: (Use arrow keys)
❯ default (babel, eslint)
  Manually select features
```

Creating the mock server

In this recipe, we will be using the `getHttp` function from the `fetchApi` wrapper we made in the 'Creating a wrapper for the Fetch API as an HTTP client' recipe.

Work through the next steps and sections to create your `MirageJS` mock server:

Install the `MirageJS` server to your packages. You need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install --save miragejs
```

The version used in this recipe was 0.1.32. Watch for any changes to `MirageJS`, as there are no current LTS versions of the library yet.

Now in the following sections, we will create each one of the HTTP methods that will be mocked by the MirageJS server.

Creating the mock database

In this section, we will create a `MirageJS` database that will be used to store the temporary data. Follow these steps to create it:

1. Create a new file called `db.js` file in the `src/server` folder for the initial loaded data.

2. We need to make a JavaScript object the default export for this file, containing the initial data that we want the server to have:

```
export default {
  users: [
    {
      name: 'Heitor Ramon Ribeiro',
      email: 'heitor@example.com',
      age: 31,
      country: 'Brazil',
      active: true,
    },
  ],
};
```

Creating the GET route function

In this section, we are going to create the *HTTP GET* method that will be mocked by the `MirageJS` server. Follow these steps to create it:

1. For the *GET* methods, we need to create a new file called `get.js` in the `src/server` folder.
2. For this recipe, we will make a generic `getFrom` function that receives a key as an argument and returns a function. This returned function returns a direct point to the local database with the indicated key:

```
export const getFrom = key => ({ db }) => db[key];

export default {
  getFrom,
};
```

Creating the POST route function

In this section, we are going to create the *HTTP POST* method, that will be mocked by the `MirageJS` server. Follow these steps to create it:

1. For the *POST* methods, we need to create a new file called `post.js` in the `src/server` folder.

2. For this recipe, we will make a generic `postFrom` function that receives a key as an argument and returns a function. This returned function will parse the `data` property of the HTTP request body and returns an internal function of the server schema that inserts the data inside the database. Using the `key` argument, the schema knows which table we are handling:

```
export const postFrom = key => (schema, request) => {
  const { data } = typeof request.requestBody === 'string'
    ? JSON.parse(request.requestBody)
    : request.requestBody;

  return schema.db[key].insert(data);
};

export default {
  postFrom,
};
```

Creating the PATCH route function

In this section, we are going to create the *HTTP PATCH* method that will be mocked by the `MirageJS` server. Follow these steps to create it:

1. For the *PATCH* methods, we need to create a new file called `patch.js` in the `src/server` folder.
2. For this recipe, we will make a generic `patchFrom` function that receives a key as an argument and returns a function. This returned function will parse the `data` property of the HTTP request body and returns an internal function of the server schema that updates a specific object with the `id` property that was passed along with the data. Using the `key` argument, the schema knows which table we are handling:

```
export const patchFrom = key => (schema, request) => {
  const { data } = typeof request.requestBody === 'string'
    ? JSON.parse(request.requestBody)
    : request.requestBody;

  return schema.db[key].update(data.id, data);
};

export default {
  patchFrom,
};
```

Creating the DELETE route function

In this section, we are going to create the *HTTP DELETE* method that will be mocked by the `MirageJS` server. Follow these steps to create it:

1. For the *DELETE* methods, we need to create a new file called `delete.js` in the `src/server` folder.
2. For this recipe, we will make a generic `patchFrom` function that receives a key as an argument and returns a function. This returned function will parse the `data` property of the HTTP request body and return an internal function of the server schema that deletes a specific object with the `id` property, which was passed to the server via the route *REST* parameter. Using the `key` argument, the schema knows which table we are handling:

```
export const deleteFrom = key => (schema, request) =>
   schema.db[key].remove(request.params.id);

export default {
  deleteFrom,
};
```

Creating the server

In this section, we are going to create the `MirageJS` server and the routes that will be available. Follow these steps to create the server:

1. Create a new file called `server.js` inside the `src/server` folder.
2. Next, we need to import the `Server` class, the `baseData`, and the router methods:

```
import { Server } from 'miragejs';
import baseData from './db';
import { getFrom } from './get';
import { postFrom } from './post';
import { patchFrom } from './patch';
import { deleteFrom } from './delete';
```

3. Create a global variable to the `window` scope, called `server`, and set this variable as a new execution of the `Server` class:

   ```
   window.server = new Server({});
   ```

4. In the `Server` class construction options, add a new property called `seeds`. This property is a function that receives the server (`srv`) as an argument and executes the `srv.db.loadData` function passing the `baseDate` as a parameter:

   ```
   seeds(srv) {
     srv.db.loadData({ ...baseData });
   },
   ```

5. Now we need to add in the same construction options to a new property called `routes`, which will create the mock server routes. This property is a function and on the function body, we will need to set the `namespace` of the mock server and the delay in milliseconds within which the server will respond. There will be four routes. For the **Create** route, we will make a new route called `/users` that listen to the *POST* method. For the **Read** route, we will make a new route called `/users` that listen to the *GET* method. For the **Update** route, we will make a new route called `/users/:id` that listens to the *PATCH* method, and finally, for the **Delete** route, we will make a new route called `/users` that listen to the *DELETE* method:

   ```
   routes() {
       this.namespace = 'api';

       this.timing = 750;

       this.get('/users', getFrom('users'));

       this.post('/users', postFrom('users'));

       this.patch('/users/:id', patchFrom('users'));

       this.delete('/users/:id', deleteFrom('users'));
     },
   ```

Adding to the application

In this section, we will add the `MirageJS` server to the Vue application. Follow these steps to make the server available to your Vue application:

1. Open the `main.js` file in the `src` folder.
2. We need to declare the server as the first imported declaration, so it's available on the initial loading of the application:

```
import './server/server';
import Vue from 'vue';
import App from './App.vue';

Vue.config.productionTip = false;

new Vue({
  render: h => h(App),
}).$mount('#app');
```

Creating the component

Now that we have our server, we need to test it. To do so, we will create a simple application that will run each of the HTTP methods and show the results of each call.

In the following parts, we will create a simple Vue application.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these steps to create it:

1. Open the `App.vue` file in the `src` folder.
2. Import the `getHttp`, `postHttp`, `patchHttp`, and `deleteHTTP` methods from the `fetchHttp` wrapper that we made in the 'Creating a wrapper for the Fetch API as an HTTP client' recipe:

```
import {
  getHttp,
  postHttp,
  patchHttp,
  deleteHttp,
} from './http/fetchApi';
```

3. In the `data` property, we need to create three new properties to be used, `response`, `userData`, and `userId`:

    ```
    data: () => ({
        response: undefined,
        userData: '',
        userId: undefined,
      }),
    ```

4. In the `methods` property, we need to create four new methods, `getAllUsers`, `createUser`, `updateUser`, and `deleteUser`:

    ```
    methods: {
      async getAllUsers() {
      },
      async createUser() {
      },
      async updateUser() {
      },
      async deleteUser() {
      },
    },
    ```

5. In the `getAllUsers` method, we will set the response data property as the result of the `getHttp` function of the `api/users` route:

    ```
    async getAllUsers() {
      this.response = await
    getHttp(`${window.location.href}api/users`);
    },
    ```

6. In the `createUser` method, we will receive a `data` argument, which will be an object that we will pass to the `postHttp` on the `api/users` route, and after that, we will execute the `getAllUsers` method:

    ```
    async createUser(data) {
      await postHttp(`${window.location.href}api/users`, { data });
      await this.getAllUsers();
    },
    ```

7. For the `updateUser` method, we will receive a `data` argument, which will be an object that we will pass to the `patchHttp` on the `api/users/:id` route, using the `id` property on the object as the `:id` on the route. After that, we will execute the `getAllUsers` method:

   ```
   async updateUser(data) {
     await patchHttp(`${window.location.href}api/users/${data.id}`,
        { data });
     await this.getAllUsers();
   },
   ```

8. Finally, on the `deleteUser` method, we receive the user `id` as the argument, which is a number, then we pass it to the `deleteHttp` on the `api/users/:id` route, using the ID as `:id`. After that, we execute the `getAllUsers` method:

   ```
   async deleteUser(id) {
     await deleteHttp(`${window.location.href}api/users/${id}`, {},
        'text');
     await this.getAllUsers();
   },
   ```

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these steps to create it:

1. At the top of the template, we need to add the `response` property, wrapped in an `<pre>` HTML element:

   ```
   <h3>Response</h3>
   <pre>{{ response }}</pre>
   ```

2. For the creation and updating of a user, we need to create a `textarea` HTML input with a `v-model` directive bound to the `userData` property:

```
<hr/>
<h1> Create / Update User </h1>
<label for="userData">
 User JSON:
 <textarea
 id="userData"
 v-model="userData"
 rows="10"
 cols="40"
 style="display: block;"
 ></textarea>
</label>
```

3. To send this data, we need to create two buttons, both having an event listener bound on the click event with the `@click` directive targeting the `createUser` and `updateUser` respectively, and passing the `userData` in the execution:

```
<button
  style="margin: 20px;"
  @click="createUser(JSON.parse(userData))"
>
  Create User
</button>
<button
  style="margin: 20px;"
  @click="updateUser(JSON.parse(userData))"
>
  Update User
</button>
```

4. To execute the *DELETE* method, we need to create an input HTML element of type `number` with a `v-model` directive bound to the `userId` property:

```
<h1> Delete User </h1>
<label for="userData">
  User Id:
<input type="number" step="1" v-model="userId">
</label>
```

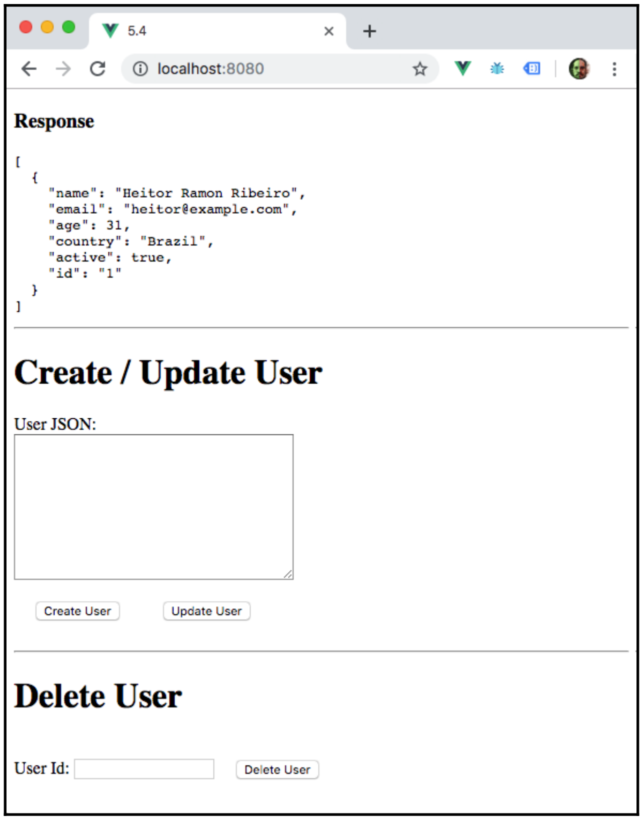
5. Finally, to execute this action we need to create a button that will have an event listener bound on the click event with the `@click` directive, targeting the `deleteUser` method and passing the `userId` property on the execution:

```
<button
  style="margin: 20px;"
  @click="deleteUser(userId)"
>
  Delete User
</button>
```

To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



Response

```
[
  {
    "name": "Heitor Ramon Ribeiro",
    "email": "heitor@example.com",
    "age": 31,
    "country": "Brazil",
    "active": true,
    "id": "1"
  }
]
```

Create / Update User

User JSON:

Delete User

User Id:

How it works...

`MirageJS` works like an interceptor of every HTTP request that happens on the application. The server intercepts all **XHR (XMLHttpRequest)** executions on the browsers and checks for the route to see whether it matches any one of the routes created on server creation. If it matches, the server will execute the function accordingly on the respective route.

Working as a simple REST server with basic CRUD functions, the server has a schema-like database structure that helps in the process of making a virtual database for storing the data.

See also

You can find more information about MirageJS at `https://github.com/miragejs/miragejs`.

Using axios as the new HTTP client

When you need a library for HTTP requests, there is no doubt that `axios` is the one you should go to. Used by more than 1.5 million open-source projects and countless closed ones, this library is the king of HTTP libraries.

It's built to work with most browsers and provides one of the most complete sets of options out there – you can customize everything in your request.

In this recipe, we will learn how to change our Fetch API wrapper to `axios` and start working around it.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can use the Vue project with Vue CLI that we made in the '*Creating your fake JSON API Server with MirageJS*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create http-project
```

The CLI will ask some questions that will help with the creation of the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *Spacebar* to select an option. Choose the `default` option:

```
? Please pick a preset: (Use arrow keys)
❯ default (babel, eslint)
  Manually select features
```

Changing from the Fetch API to Axios

In the next steps, we will change the Fetch API used in the HTTP wrapper for the `axios` library. Follow these steps to change it correctly:

1. Install `axios` in your packages. You need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > npm install --save axios
   ```

The version used in this recipe was 0.19.0. Watch for changes to `axios`, as there is no LTS version of the library yet.

2. Open the `baseFetch.js` file inside the `src/http` folder.
3. Simplify the method so that it will receive three arguments, `url`, `method`, and `options`, and return an `axios` method, calling the HTTP request with the methods passed as the constructor of the instance:

```
import axios from 'axios';

export default async (url, method, options = {}) => axios({
  method: method.toUpperCase(),
  url,
  ...options,
});
```

Changing the GET method function

In this part, we are changing the *HTTP GET* method. Follow these instructions to change the `getHttp` function:

1. Open the `fetchApi.js` file inside the `src/http` folder.
2. In the `getHttp` function, we will add a new argument param, and remove the currying functions:

```
export const getHttp = async (
  url,
  params,
  options,
) => baseHttp(url,
  'get',
  {
    ...options,
    params,
  });
```

Changing the POST method function

In this part, we are changing the *HTTP POST* method. Follow these instructions to change the `postHttp` function:

1. Open the `fetchApi.js` file inside the `http` folder.
2. In the `postHttp` function, we will change the `body` argument to `data`, and remove the currying functions:

```
export const postHttp = async (
  url,
  data,
  options,
) => baseHttp(url,
  'post',
  {
    data,
    ...options,
  });
```

Changing the PUT method function

In this part, we are changing the *HTTP PUT* method. Follow these instructions to change the `putHttp` function:

1. Open the `fetchApi.js` file inside the `http` folder.
2. In the `putHttp` function, we will change the `body` argument to `data`, and remove the currying functions:

```
export const putHttp = async (
  url,
  data,
  options,
) => baseHttp(url,
  'put',
  {
    data,
    ...options,
  });
```

Changing the PATCH method function

In this part, we are changing the *HTTP PATCH* method. Follow these instructions to change the `patchHttp` function:

1. Open the `fetchApi.js` file inside the `http` folder.
2. In the `patchHttp` function, we will change the `body` argument to `data`, and remove the currying functions:

```
export const patchHttp = async (
  url,
  data,
  options,
) => baseHttp(url,
  'patch',
  {
    data,
    ...options,
  });
```

Changing the UPDATE method function

In this part, we are changing the *HTTP UPDATE* method. Follow these instructions to change the `updateHttp` function:

1. Open the `fetchApi.js` file inside the `http` folder.
2. In the `updateHttp` function, we will add a new argument param, and remove the currying functions:

```
export const updateHttp = async (
  url,
  data,
  options,
) => baseHttp(url,
  'update',
  {
    data,
    ...options,
  });
```

Changing the DELETE method function

In this part, we are changing the *HTTP DELETE* method. Follow these instructions to change the `deleteHttp` function:

1. Open the `fetchApi.js` file inside the `http` folder.
2. On the `deleteHttp` function, we will change the `body` argument to `data`, and remove the currying functions:

```
export const deleteHttp = async (
  url,
  data,
  options,
) => baseHttp(url,
  'delete',
  {
    data,
    ...options,
  });
```

Changing the component

In this part, we will change how the component works with the new functions. Follow these instructions to change it correctly:

1. Open the `App.vue` file inside the `src` folder.
2. In the `getAllUsers` method, we will need to change the way the response is defined because `axios` gives us a completely different response object than the Fetch API:

```
async getAllUsers() {
  const { data } = await
getHttp(`${window.location.href}api/users`);
  this.response = data;
 },
```

3. In the `deleteUser` method, we can just pass the URL as the parameter:

```
async deleteUser(id) {
  await deleteHttp(`${window.location.href}api/users/${id}`);
  await this.getAllUsers();
 },
```

To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

How it works...

When we made the wrapper for the Fetch API, we used a technique of abstracting the API into another interface, which made it possible to change from the Fetch API to the `axios` library. By doing this we were able to improve the methods and simplify how the functions are called and handled. For example, the GET method can now receive a new argument called **params**, which are objects of URL query parameters that will be automatically injected into the URL.

We also had to change the way that the responses were interpreted because `axios` have a more robust and complete response object than the Fetch API, which returns just the fetched response itself.

See also

You can find more information about `axios` at `https://github.com/axios/axios`.

Creating different axios instances

When using `axios`, you can have multiple instances of it running with none of them interfering with each other. For example, you have an instance pointing to a user API that is on version 1 and another pointing to the payment API that is on version 2, both sharing the same namespace.

Here, we are going to learn how to create various `axios` instances, so you are able to work with as many API namespaces as you want without problems or interference.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can use the Vue project with Vue CLI that we did in the '*Using axios as the new HTTP client*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create http-project
```

The CLI will ask some questions that will help with the creation of the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *Spacebar* to select an option. Choose the `default` option:

```
? Please pick a preset: (Use arrow keys)
❯ default (babel, eslint)
  Manually select features
```

Changing the HTTP function

When creating multiple `axios` instances, the process of calling the `axios` library changes. Because of that, we need to change how our HTTP wrapper instantiates the `axios` library.

In the following parts, we will change how the HTTP wrapper works with the creation of a new `axios` instance, and make it available to the application.

Changing the HTTP Fetch wrapper

In the following steps, we will create a new custom `axios` instance that will be used in the HTTP wrapper. Follow these instructions to add the new instance to the application:

1. Open the `baseFetch.js` file in the `src/http` folder.
2. We need to create a new factory function called `createAxios` to generate a new `axios` instance each time it's executed:

```
export function createAxios(options = {}) {
  return axios.create({
    ...options,
  });
}
```

3. Now we need to create the `localApi` constant, the value of which will be the result of the execution of the `createAxios` factory:

```
const localApi = createAxios();
```

4. For the `JSONPlaceHolder` we will create a constant that will be exported, named `jsonPlaceholderApi`, the value of which will be the execution of the `createAxios` factory. We will also pass an object as an argument with the `baseURL` property defined:

```
export const jsonPlaceholderApi = createAxios({
  baseURL: 'https://jsonplaceholder.typicode.com/',
});
```

5. In the `export default` function, we need to change from `axios` to `localApi`:

```
export default async (url, method, options = {}) => localApi({
  method: method.toUpperCase(),
  url,
  ...options,
});
```

Changing the HTTP methods function

In this part, we will change how the HTTP methods will work with the new `axios` instances. Follow the instructions to do it correctly:

1. Open the `fetchApi.js` file in the `src/http` folder.
2. We will import the `jsonPlaceholderApi` function from `baseFetch` as an extra imported value:

```
import baseHttp, { jsonPlaceholderApi } from './baseFetch';
```

3. We need to create a new constant called `getTodos` that will be exported. This constant will be a function that will receive a `userId` as a parameter and return the GET function from `axios`, with the `userId` parameter we just received, inside a configuration object in a property called `params`:

```
export const getTodos = async userId =>
jsonPlaceholderApi.get('todos',
  {
    params: {
      userId,
    },
  });
```

Changing the MirageJS server

In this part, we will change how the `MirageJS` server works with the new `axios` instance that was created. Follow the instructions to do it correctly:

1. Open the `server.js` file in the `src/server` folder.
2. On the `routes` property in the constructor object, we need to add a `passthrough` declaration, which will indicate to the MirageJS that all the calls to that URL won't be intercepted:

```
import { Server } from 'miragejs';
import baseData from './db';
import { getFrom } from './get';
import { postFrom } from './post';
import { patchFrom } from './patch';
import { deleteFrom } from './delete';

window.server = new Server({
  seeds(srv) {
    srv.db.loadData({ ...baseData });
  },
  routes() {
    this.passthrough();
    this.passthrough('https://jsonplaceholder.typicode.com/**');

    this.namespace = 'api';

    this.timing = 750;

    this.get('/users', getFrom('users'));

    this.post('/users', postFrom('users'));

    this.patch('/users/:id', patchFrom('users'));

    this.delete('/users/:id', deleteFrom('users'));
  },
});
```

Changing the component

After the changes in the wrapper functions, the `MirageJS` server methods, and the HTTP methods, we need to change the component to the new library that was implemented.

In the following parts, we will change the component to match the new library that was implemented.

Single file component <script> section

In this part, we will change the `<script>` section of the single file component. Follow these steps to do it:

1. Open the `App.vue` file in the `src` folder.
2. We need to import the new `getTodos` function as follows:

```
import {
  getHttp,
  postHttp,
  patchHttp,
  deleteHttp,
  getTodos,
} from './http/fetchApi';
```

3. In the `data` property of the `Vue` object, we need to create a new property called `userTodo`, with the default value of an empty array:

```
data: () => ({
  response: undefined,
  userData: '',
  userId: undefined,
  userTodo: [],
}),
```

4. In the `methods` property, we need to create a new method called `getUserTodo` that receives the `userId` argument. This method will fetch the list of to-do items of the user and will attribute the response to the `userTodo` property:

```
async getUserTodo(userId) {
  this.userTodo = await getTodos(userId);
},
```

Single file component <template> section

In this part, we will change the `<template>` section of the single file component. Follow these steps to do it:

1. Open the `App.vue` file in the `src` folder.
2. At the bottom of the template, we need to create a new `input` HTML element, with the `v-model` directive bound to the `userId` property:

```
<h1> Get User ToDos </h1>
<label for="userData">
  User Id:
  <input type="number" step="1" v-model="userId">
</label>
```

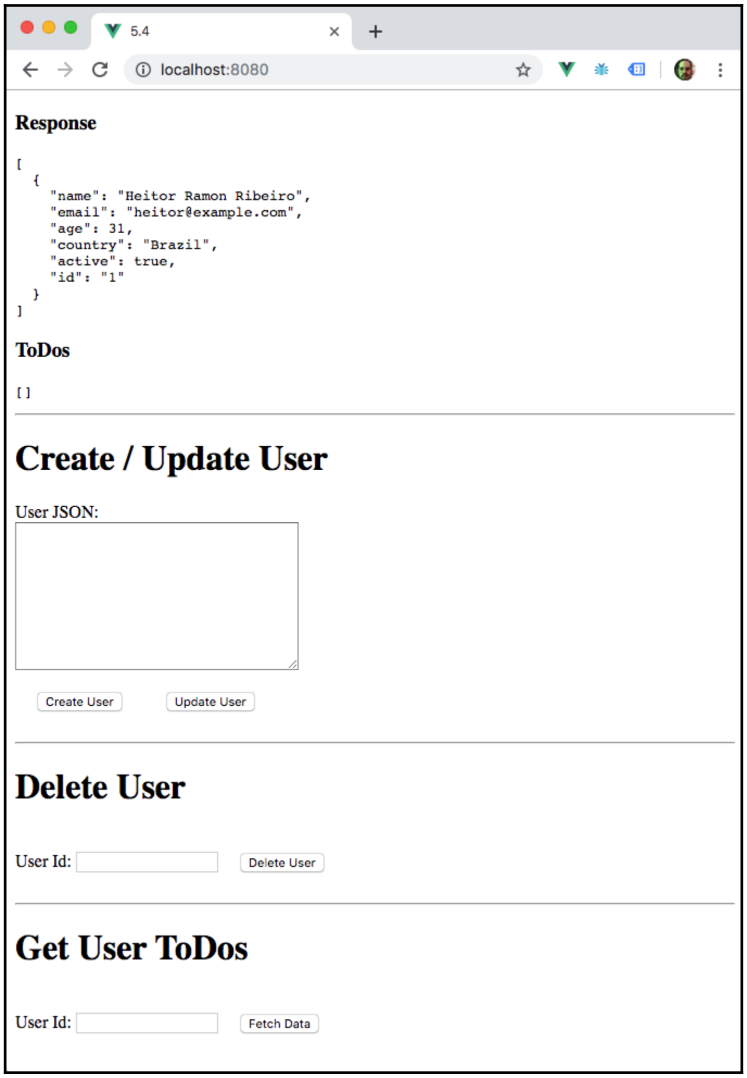
3. To fetch the list of items, we need to create a button with an event listener bound on the click event with the `@click` directive, targeting the `getUserTodo`, and passing the `userId` in the execution:

```
<button
  style="margin: 20px;"
  @click="getUserTodo(userId)"
>
  Fetch Data
</button>
```

To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



Response

```
[
  {
    "name": "Heitor Ramon Ribeiro",
    "email": "heitor@example.com",
    "age": 31,
    "country": "Brazil",
    "active": true,
    "id": "1"
  }
]
```

ToDos

```
[]
```

Create / Update User

User JSON:

Delete User

User Id:

Get User ToDos

User Id:

How it works...

When we create a new instance of `axios`, a new object is created, and new configurations, headers, interceptors, and manipulators are defined. This happens because `axios` declares the `create` functions as the same as `new Class`. It's the same interface but different objects.

Using this possibility, we were able to create two connection drivers, one for the local API and another for the `JSONPlaceHolder` API, which has a different `baseURL`.

Because of MirageJS server integration, all the HTTP requests are intercepted by MirageJS, so we needed to add a directive in the router constructor that indicates the routes that MirageJS won't intercept.

See also

You can find more information about the JSONPlaceHolder API at `https://jsonplaceholder.typicode.com/`.

You can find more information about `axios` instances at `https://github.com/axios/axios#creating-an-instance`.

You can find more information about MirageJS at `https://github.com/miragejs/miragejs`.

Creating a request and response interceptor for axios

Using `axios` as the main HTTP manipulator in our application allows us to use request and response interceptors. Those are used to manipulate the data before sending it to the server or when receiving the data, manipulating it before sending it back to the JavaScript code.

The most common way an interceptor is used is in JWT token validation and refreshing the requests that receive a specific error or API error manipulation.

In this recipe, we will learn how to create a request interceptor to check the *POST*, *PATCH*, and *DELETE* methods and a response error manipulator.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can use the Vue project with Vue CLI that we made in the '*Creating different axios instances*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create http-project
```

The CLI will ask some questions that will help with the creation of the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *Spacebar* to select an option. Choose the `default` option:

```
? Please pick a preset: (Use arrow keys)
❯ default (babel, eslint)
  Manually select features
```

Creating the interceptor

In the following steps, we will create an `axios` interceptor that will work as a middleware. Follow the instructions do it correctly:

1. Install the `Sweet Alert` package. To do this you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > npm install --save sweetalert2
   ```

2. Create a new file called `interceptors.js` in the `src/http` folder and open it.

3. Then, we import the Sweet Alert package:

```
import Swal from 'sweetalert2';
```

4. We need to create a constant with an array of the *POST* methods that will be intercepted:

```
const postMethods = ['post', 'patch'];
```

5. We need to create a function named `requestInterceptor` and export it. This function will receive one argument, `config`, which is an `axios` configuration object. We need to check whether the request method is included in the array we created earlier and whether the `data` property of the data body has an `id` property. If any of the checks didn't pass, we will throw an `Error`, otherwise, we will return `config`:

```
export function requestInterceptor(config) {
  if (
    postMethods.includes(config.method.toLocaleLowerCase()) &&
    Object.prototype.hasOwnProperty.call('id', config.data.data) &&
      !config.data.data.id)
    {
      throw new Error('You need to pass an ID for this request');
    }

 return config;
}
```

6. For the response interceptor, we need to create a new function called `responseInterceptor` that returns the response, as we won't change anything in this interceptor:

```
export function responseInterceptor(response) {
  return response;
}
```

7. For catching the error, we need to create an `errorInterceptor` function, which will be exported. This function receives an `error` as an argument and will display a `sweetalert2` alert error message and return a `Promise.reject` with the `error`:

```
export function errorInterceptor(error) {
  Swal.fire({
    type: 'error',
    title: 'Error!',
    text: error.message,
  });

  return Promise.reject(error);
}
```

Adding the interceptors to the HTTP methods functions

In the following steps, we will add the `axios` interceptor to the HTTP method functions. Follow these steps to do it correctly:

1. Open the `baseFetch.js` file in the `src/http` folder.
2. We need to import the three interceptors we just created:

```
import {
  errorInterceptor,
  requestInterceptor,
  responseInterceptor,
} from './interceptors';
```

3. After the creation of the `localApi` instance, we declare the use of the request and response interceptor:

```
localApi.interceptors
  .request.use(requestInterceptor, errorInterceptor);

localApi.interceptors
  .response.use(responseInterceptor, errorInterceptor);
```

4. After the creation of the `jsonPlaceholderApi` instance, we declare the use of the request and response interceptor:

```
jsonPlaceholderApi.interceptors
  .request.use(requestInterceptor, errorInterceptor);

jsonPlaceholderApi.interceptors
  .response.use(responseInterceptor, errorInterceptor);
```

How it works...

Each request that `axios` do passes through each of any one of the interceptors in the set. The same thing happens for the response. If any error is thrown on the interceptor, it will automatically be passed to the error manipulator, so the request won't be executed at all, or the response will be sent to the JavaScript code as an error.

We checked each request that was done for the *POST*, *PATCH*, and *DELETE* method to see if there was an `id` property in the body data. If there wasn't, we threw an error to the user, saying that they need to pass an ID for the request.

See also

You can find more information about Sweet Alert 2 at `https://sweetalert2.github.io`.

You can find more information about the `axios` request interceptor at `https://github.com/axios/axios#interceptors`.

Creating a CRUD interface with Axios and Vuesax

When dealing with data, there is something that we will always need to do: a CRUD process. Regardless of what kind of application you are developing, a CRUD interface is needed in order to input and manipulate any data on the server, the administrator panel, the backend of your application, or even the client side.

Here, we will learn how to create a simple CRUD interface using the `Vuesax` framework for the UI and `axios` for the HTTP request.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, use the Vue project with Vue CLI that we used in the '*Creating a request and response interceptor for axios*' recipe, or start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create http-project
```

The CLI will ask some questions that will help with the creation of the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *Spacebar* to select an option. Choose the `default` option:

```
? Please pick a preset: (Use arrow keys)
❯ default (babel, eslint)
  Manually select features
```

Adding Vuesax to the application

In the following steps, we will cover how to add the `Vuesax` UI library to your Vue application. Follow these instructions to do it correctly:

1. Open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > npm install --save vuesax material-icons
   ```

2. Create a file called `style.css` in the `src` folder and open it.
3. Import the `vuesax`, `material-icon`, and `Open Sans` font stylesheets:

   ```
   @import
   url('https://fonts.googleapis.com/css?family=Open+Sans:300,300i,400
   ,400i,600,600i,700,700i,800,
       800i&display=swap');
   @import url('~vuesax/dist/vuesax.css');
   @import url('~material-icons/iconfont/material-icons.css');

   * {
     font-family: 'Open Sans', sans-serif;
   }
   ```

4. Open the `main.js` file in the `src` folder.
5. Import the `style.css` file and `Vuesax`. After that, you need to inform Vue to use `Vuesax`:

```
import './server/server';
import Vue from 'vue';
import App from './App.vue';
import Vuesax from 'vuesax';
import './style.css';

Vue.use(Vuesax);

Vue.config.productionTip = false;

new Vue({
  render: h => h(App),
}).$mount('#app');
```

Creating the component routing

We will continue the recipe in five parts: `List`, `Create`, `Read`, `Update`, and `Delete`. Our application will be a dynamic component application, so we will create five components, one for each part. Those components will be like our pages.

First, we need to change `App.vue` to be our main route manager and create a mixin for changing the component.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. Open `App.vue` in the `src` folder.
2. Import each one of the components that will be created here:

```
import List from './components/list';
import Create from './components/create';
import View from './components/view';
import Update from './components/update';
```

3. In the `data` property, create two new values: `componentIs` with a default value of `'list'`, and `userId` with a default value of `0`:

```
data: () => ({
  componentIs: 'list',
  userId: 0,
}),
```

4. We need to add a new property to the Vue object, called `provide`. This property will be a function, so the provided values to the components can be reactive:

```
provide () {
  const base = {};

  Object.defineProperty(base, 'userId', {
    enumerable: true,
    get: () => Number(this.userId),
  });

  return base;
},
```

5. In the `computed` properties, we need to create a new property called `component`. This will be a switch case that will return our component, based on the `componentIs` property:

```
computed: {
  component() {
    switch (this.componentIs) {
      case 'list':
        return List;
      case 'create':
        return Create;
      case 'view':
        return View;
      case 'edit':
        return Update;
      default:
        return undefined;
    }
  }
},
```

6. Finally, in the methods, we need to create a `changeComponent` method that will update the current component to a new one:

```
methods: {
  changeComponent(payload) {
    this.componentIs = payload.component;
    this.userId = Number(payload.userId);
  },
},
```

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these instructions to create the component correctly:

1. In the `div#app` HTML element, we need to add a `vs-row` component:

```
<div id="app">
    <vs-row></vs-row>
</div>
```

2. In the `vs-row` component, we need to add a `vs-col` component with the following attributes: `vs-type` defined as `flex`, `vs-justify` defined as `left`, `vs-align` defined as `left`, and `vs-w` defined as `12`:

```
<vs-col
  vs-type="flex"
  vs-justify="left"
  vs-align="left"
  vs-w="12">
</vs-col>
```

3. Finally, inside the `vs-col` component, we will add a dynamic component that has an `is` attribute to the computed property `component` and point the event listener at the `"change-component"` event that will execute the `changeComponent` method:

```
<component
 :is="component"
 @change-component="changeComponent"
/>
```

Creating the route mixin

In this part, we will create the component mixin to be re-used in other components. Follow these instructions to create the component correctly:

1. Create a new file called `changeComponent.js` in the `src/mixin` folder and open it.
2. This mixin will have a method called `changeComponent`, which will emit a `'change-component'` event with the name of the new component that needs to be rendered, and the `userId`:

```
export default {
  methods: {
    changeComponent(component, userId = 0) {
      this.$emit('change-component', { component, userId });
    },
  }
}
```

Creating the list component

The list component will be the index component. It will list the users in the application and have all the links for the other CRUD actions.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. Create a new file called `list.vue` in the `src/components` folder and open it.
2. Import the `getHttp` and `deleteHttp` from `fetchApi` and the `changeComponent` mixin:

```
import {
    getHttp,
    deleteHttp,
  } from '../http/fetchApi';
  import changeComponent from '../mixin/changeComponent';
```

3. In the component `mixins` property, we need to add the imported `changeComponent` mixin:

```
mixins: [changeComponent],
```

4. In the `data` property of the component, we add a new property named `userList`, with a default empty array:

```
data: () => ({
  userList: [],
}),
```

5. For the methods, we create `getAllUsers` and `deleteUsers` methods. In the `getAllUsers` method, we fetch the user lists and set the `userList` value as the response from the `getHttp` function execution. The `deleteUser` method will execute the `deleteHttp` function, and then execute the `getAllUsers` method:

```
methods: {
  async getAllUsers() {
    const { data } = await
getHttp(`${window.location.href}api/users`);
    this.userList = data;
  },
  async deleteUser(id) {
    await deleteHttp(`${window.location.href}api/users/${id}`);
    await this.getAllUsers();
  },
}
```

6. Lastly, we make the `beforeMount` life cycle hook asynchronous, calling the `getAllUsers` method:

```
async beforeMount() {
  await this.getAllUsers();
},
```

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these instructions to create the component correctly:

1. Create a `vs-card` component with the `style` attribute defined as `margin: 20px`:

    ```
    <vs-card
        style="margin: 20px;"
      >
    </vs-card>
    ```

2. Inside the `vs-card` component, create a dynamic `<template>` named slot for `header`, with an `<h3>` tag and your title:

    ```
    <template slot="header">
      <h3>
        Users
      </h3>
    </template>
    ```

3. After that, create a `vs-row` component with a `vs-col` component inside of it, with the following attributes: `vs-type` defined as `flex`, `vs-justify` defined as `left`, `vs-align` defined as `left`, and `vs-w` defined as `12`:

    ```
    <vs-row>
      <vs-col
        vs-type="flex"
        vs-justify="left"
        vs-align="left"
        vs-w="12">
      </vs-col>
    </vs-row>
    ```

4. Inside the `vs-col` component, we need to create a `vs-table` component. This component will have the `data` attribute pointed to the `userList` variable, and will have the `search`, `stripe`, and `pagination` attributes defined as true. The `max-items` attribute will be defined as `10` and the `style` attribute will have the value of `width: 100%; padding: 20px;`:

```
<vs-table
  :data="userList"
  search
  stripe
  pagination
  max-items="10"
  style="width: 100%; padding: 20px;"
></vs-table>
```

5. For the table header, we need to create a dynamic `<template>` named slot `thead`, and create for each column a `vs-th` component with the `sort-key` attribute defined as the respective object key property and the display as the name you want:

```
<template slot="thead">
  <vs-th sort-key="id">
    #
  </vs-th>
  <vs-th sort-key="name">
    Name
  </vs-th>
  <vs-th sort-key="email">
    Email
  </vs-th>
  <vs-th sort-key="country">
    Country
  </vs-th>
  <vs-th sort-key="phone">
    Phone
  </vs-th>
  <vs-th sort-key="Birthday">
    Birthday
  </vs-th>
  <vs-th>
    Actions
  </vs-th>
</template>
```

6. For the table body, we need to create a dynamic `<template>` with a `slot-scope` attribute defined as the `data` property. Inside this `<template>` we need to create a `vs-tr` component that will iterate the data property and have a `vs-td` component for each column that you set on the head of the table. Each `vs-td` component has a data property set to the respective column data object property, and the content will be the same data rendered. The final column that is the actions column will have three buttons, one for **Read**, another for **Update**, and the last for **Delete**. The **Read** button will have an event listener on the `"click"` event pointing to the `changeComponent`, and the same goes for the **Update** button. The **Delete** button `"click"` event listener will be pointing to the `deleteUser` method:

```
<template slot-scope="{data}">
  <vs-tr :key="index" v-for="(tr, index) in data">
    <vs-td :data="data[index].id">
      {{data[index].id}}
    </vs-td>
    <vs-td :data="data[index].name">
      {{data[index].name}}
    </vs-td>
    <vs-td :data="data[index].email">
      <a :href="`mailto:${data[index].email}`">
        {{data[index].email}}
      </a>
    </vs-td>
    <vs-td :data="data[index].country">
      {{data[index].country}}
    </vs-td>
    <vs-td :data="data[index].phone">
      {{data[index].phone}}
    </vs-td>
    <vs-td :data="data[index].birthday">
      {{data[index].birthday}}
    </vs-td>
    <vs-td :data="data[index].id">
      <vs-button
        color="primary"
        type="filled"
        icon="remove_red_eye"
        size="small"
        @click="changeComponent('view', data[index].id)"
      />
      <vs-button
        color="success"
        type="filled"
        icon="edit"
```

```
          size="small"
          @click="changeComponent('edit', data[index].id)"
        />
        <vs-button
          color="danger"
          type="filled"
          icon="delete"
          size="small"
          @click="deleteUser(data[index].id)"
        />
      </vs-td>
    </vs-tr>
  </template>
```

7. Finally, in the card footer, we need to create a dynamic `<template>` named slot for `footer`. Inside this `<template>` we will add a `vs-row` component with the `vs-justify` attribute defined as `flex-start` and insert a `vs-button` with the `color` attribute defined as `primary`, `type` defined as `filled`, `icon` defined as `fiber_new`, and `size` defined as `small`. The `@click` event listener will target the `changeComponent` method with the parameters `'create'` and `0`:

```
<template slot="footer">
  <vs-row vs-justify="flex-start">
    <vs-button
      color="primary"
      type="filled"
      icon="fiber_new"
      size="small"
      @click="changeComponent('create', 0)"
    >
      Create User
    </vs-button>
  </vs-row>
</template>
```

Single file component <style> section

In this part, we will create the `<style>` section of the single file component. Follow these instructions to create the component correctly:

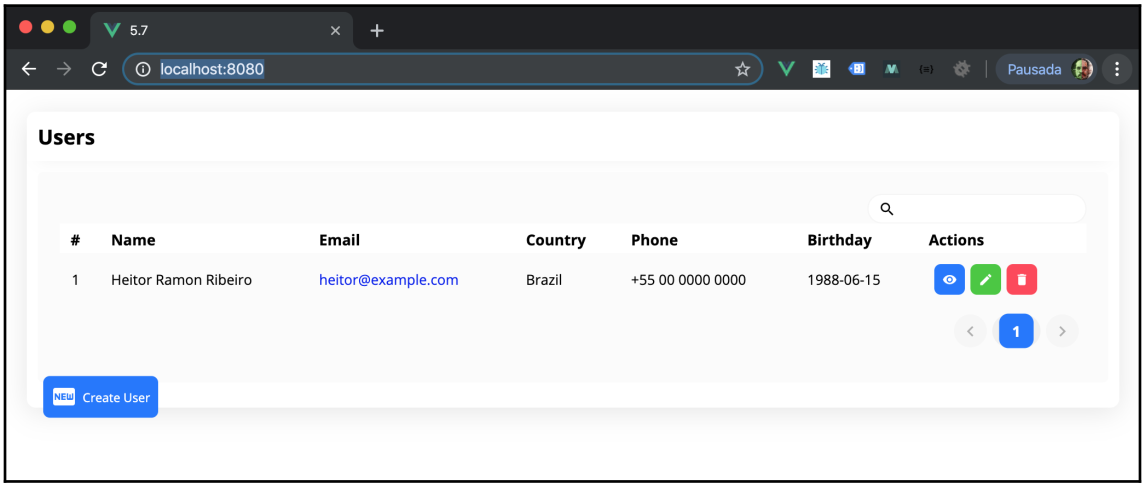
1. Create a declaration of margin to the `vs-button` component class:

```
<style scoped>
  .vs-button {
    margin-left: 5px;
  }
</style>
```

2. To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



Users

#	Name	Email	Country	Phone	Birthday	Actions
1	Heitor Ramon Ribeiro	heitor@example.com	Brazil	+55 00 0000 0000	1988-06-15	

Creating a generic user form component

In the following parts, we will create a generic user form component that will be used by other components. This component is considered generic because it is a component that can be used by anyone.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. Create a new file called `userForm.vue` in the `src/components` folder and open it.
2. In the `props` Vue property, create two new properties called `value` and `disabled`, both being objects and having the three properties of `type`, `required`, and `default`. For the `value` property, the `type` will be `Object`, `required` will be `false`, and `default` will be a factory returning an object. For the `disabled` property, the `type` will be `Boolean`, `required` will be `false`, and the `default` will also be `false`:

```
props: {
  value: {
    type: Object,
    required: false,
    default: () => {
    },
  },
  disabled: {
    type: Boolean,
    required: false,
    default: false,
  }
},
```

3. In the `data` property, we need to add a new value of `tmpForm`, with the default value of an empty object:

```
data: () => ({
  tmpForm: {},
}),
```

4. In the Vue `watch` property, we need to create the handler for the `tmpForm` and the `value`. For the `tmpForm` watcher, we will add a `handler` function that will emit an `'input'` event on each change with the new `value`, and add the `deep` property to `true`. Finally, on the `value` watcher, we will add a `handler` function that will set the value of the `tmpForm` as the new `value`. We also need to define the `deep` and `immediate` properties as `true`:

```
watch: {
  tmpForm: {
    handler(value) {
      this.$emit('input', value);
    },
    deep: true,
  },
  value: {
    handler(value) {
      this.tmpForm = value;
    },
    deep: true,
    immediate: true,
  }
},
```

When using watchers, declaring the `deep` property makes the watcher checks for deep changes on arrays or objects, and the `immediate` property executes the watcher as soon as the component is created.

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these instructions to create the component correctly:

1. For the inputs wrapper, we need to create a `vs-row` component. Inside the `vs-row` component, we will create each input for our user form:

   ```
   <vs-row></vs-row>
   ```

2. For the name input, we need to create a `vs-col` component, with the attributes of `vs-type` defined as `'flex'`, `vs-justify` defined as `'left'`, `vs-align` defined as `'left'`, and `vs-w` defined as `'6'`. Inside of the `vs-col` component, we need to create a `vs-input` component, with the `v-model` directive bound to `tmpForm.name`, the attributes of `disabled` bound to the `disabled` props, `label` defined as `'Name'`, `placeholder` defined as `'User Name'`, and `class` defined as `'inputMargin full-width'`:

   ```
   <vs-col
     vs-type="flex"
     vs-justify="left"
     vs-align="left"
     vs-w="6">
     <vs-input
       v-model="tmpForm.name"
       :disabled="disabled"
       label="Name"
   ```

```
      placeholder="User Name"
      class="inputMargin full-width"
    />
  </vs-col>
```

3. For the email input, we need to create a `vs-col` component, with the attributes of `vs-type` defined as `'flex'`, `vs-justify` defined as `'left'`, `vs-align` defined as `'left'`, and `vs-w` defined as `'6'`. Inside of the `vs-col` component, we need to create a `vs-input` component, with the `v-model` directive bound to `tmpForm.email`, the `disabled` attributes bound to the `disabled` props, `label` defined as `'Email'`, `placeholder` defined as `'User Email'`, and `class` defined as `'inputMargin full-width'`:

```
  <vs-col
    vs-type="flex"
    vs-justify="left"
    vs-align="left"
    vs-w="6">
    <vs-input
      v-model="tmpForm.email"
      :disabled="disabled"
      label="Email"
      placeholder="User Email"
      class="inputMargin full-width"
    />
  </vs-col>
```

4. For the country input, we need to create a `vs-col` component, with the attributes of `vs-type` defined as `'flex'`, `vs-justify` defined as `'left'`, `vs-align` defined as `'left'`, and `vs-w` defined as `'6'`. Inside of the `vs-col` component, we need to create a `vs-input` component, with the `v-model` directive bound to `tmpForm.country`, the `disabled` attributes bound to the `disabled` props, `label` defined as `'Country'`, `placeholder` defined as `'User Country'`, and `class` defined as `'inputMargin full-width'`:

```
  <vs-col
    vs-type="flex"
    vs-justify="left"
    vs-align="left"
    vs-w="6">
    <vs-input
      v-model="tmpForm.country"
      :disabled="disabled"
      label="Country"
      placeholder="User Country"
      class="inputMargin full-width"
    />
  </vs-col>
```

5. For the phone input, we need to create a `vs-col` component, with the attributes of `vs-type` defined as `'flex'`, `vs-justify` defined as `'left'`, `vs-align` defined as `'left'`, and `vs-w` defined as `'6'`. Inside of the `vs-col` component, we need to create a `vs-input` component, with the `v-model` directive bound to `tmpForm.phone`, the `disabled` attributes bound to the `disabled` props, `label` defined as `'Phone'`, `placeholder` defined as `'User Phone'`, and `class` defined as `'inputMargin full-width'`:

```
  <vs-col
    vs-type="flex"
    vs-justify="left"
    vs-align="left"
    vs-w="6">
    <vs-input
      v-model="tmpForm.phone"
      :disabled="disabled"
      label="Phone"
      placeholder="User Phone"
      class="inputMargin full-width"
    />
  </vs-col>
```

6. For the birthday input, we need to create a `vs-col` component, with the attributes of `vs-type` defined as `'flex'`, `vs-justify` defined as `'left'`, `vs-align` defined as `'left'`, and `vs-w` defined as `'6'`. Inside of the `vs-col` component, we need to create a `vs-input` component, with the `v-model` directive bound to `tmpForm.birthday`, the `disabled` attributes bound to the `disabled` props, `label` defined as `'Birthday'`, `placeholder` defined as `'User Birthday'`, and `class` defined as `'inputMargin full-width'`:

```
  <vs-col
    vs-type="flex"
    vs-justify="left"
    vs-align="left"
    vs-w="6">
    <vs-input
      v-model="tmpForm.birthday"
      :disabled="disabled"
      label="Birthday"
      placeholder="User Birthday"
      class="inputMargin full-width"
```

```
            />
          </vs-col>
```

Single file component <style> section

In this part, we will create the `<style>` section of the single file component. Follow these instructions to create the component correctly:

Create a new scoped class called `inputMargin` with the `margin` property defined as `15px`:

```
<style>
  .inputMargin {
    margin: 15px;
  }
</style>
```

Creating the create user component

To start our process with user manipulation, we need to create an initial base user form to be shared between the `View`, `Create`, and `Update` components.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. Create a new file called `create.vue` in the `src/components` folder and open it.
2. Import the `UserForm` component, the `changeComponent` mixin, and `postHttp` from `fetchApi`:

   ```
   import UserForm from './userForm';
   import changeComponent from '../mixin/changeComponent';
   import { postHttp } from '../http/fetchApi';
   ```

3. It the `data` property, we will add a `userData` object with the `name`, `email`, `birthday`, `country`, and `phone` properties all defined as empty strings:

   ```
   data: () => ({
     userData: {
       name: '',
       email: '',
       birthday: '',
       country: '',
       phone: '',
   ```

```
      },
    }),
```

4. In the Vue `mixins` property, we need to add the `changeComponent`:

   ```
   mixins: [changeComponent],
   ```

5. In the Vue `components` property, add the `UserForm` component:

   ```
   components: {
     UserForm,
   },
   ```

6. In the `methods` property, we need to create the `createUser` method that will use the data on the `userData` property and will create a new user on the server and then redirect the user to the users lists:

   ```
   methods: {
     async createUser() {
       await postHttp(`${window.location.href}api/users`, {
         data: {
           ...this.userData,
         }
       });
       this.changeComponent('list', 0);
     },
   },
   ```

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these instructions to create the component correctly:

1. Create a `vs-card` component with the `style` attribute defined as `margin: 20px`:

   ```
   <vs-card
       style="margin: 20px;"
     >
   </vs-card>
   ```

2. Inside the `vs-card` component, create a dynamic `<template>` named slot for `header`, with an `<h3>` tag and your title:

   ```
   <template slot="header">
     <h3>
       Create User
   ```

```
  </h3>
</template>
```

3. After that, create a `vs-row` component with a `vs-col` component inside of it, with the attributes of `vs-type` defined as `flex`, `vs-justify` defined as `left`, `vs-align` defined as `left`, and `vs-w` defined as `12`:

```
<vs-row>
  <vs-col
    vs-type="flex"
    vs-justify="left"
    vs-align="left"
    vs-w="12">
  </vs-col>
</vs-row>
```

4. Inside the `vs-col` component, we will add the `user-form` component with the `v-model` directive bound to `userData`:

```
<user-form
  v-model="userData"

/>
```

5. Finally, in the card footer, we need to create a dynamic `<template>` named slot for `footer`. Inside this `<template>` we will add a `vs-row` component with the `vs-justify` attribute defined as `flex-start` and insert two `vs-button` components. The first will be for creating the user and will have the attributes of `color` defined as `success`, `type` defined as `filled`, `icon` defined as `save`, and `size` defined as `small`. The `@click` event listener will target the `createUser` method and the second `vs-button` component will be for canceling this action and returning to the users lists. It will have the attributes of `color` defined as `danger`, `type` defined as `filled`, `icon` defined as `cancel`, `size` defined as `small`, `style` defined as `margin-left: 5px`, and the `@click` event listener target to the `changeComponent` method with the `'list'` and `0` parameters:

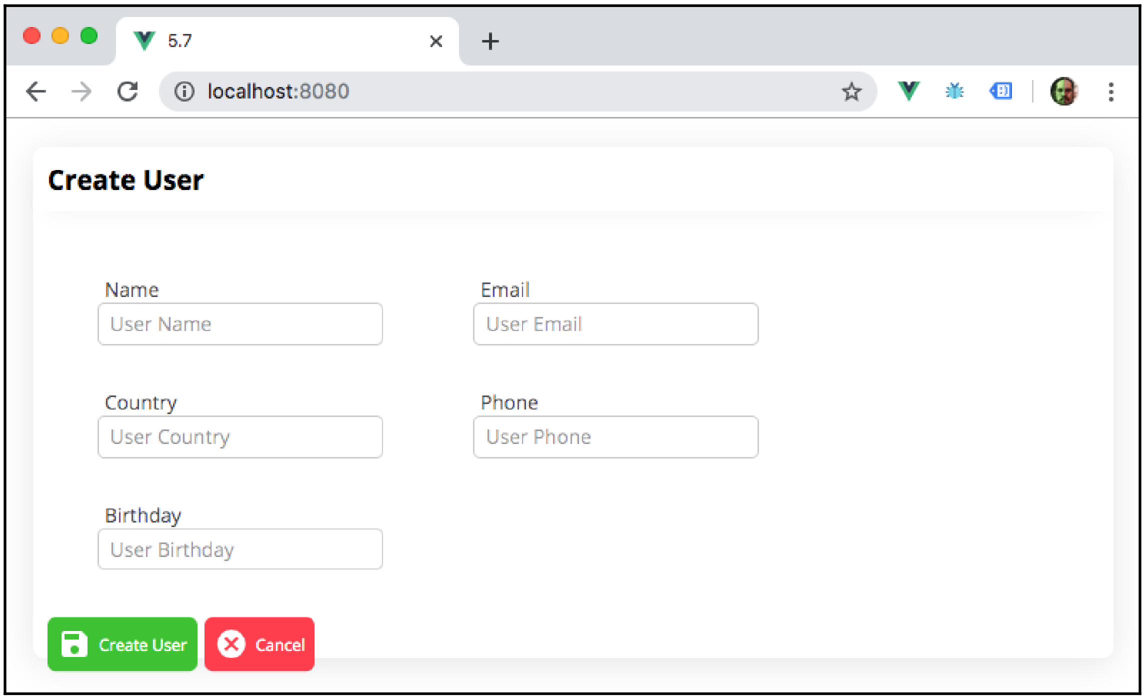
```
<template slot="footer">
  <vs-row vs-justify="flex-start">
    <vs-button
      color="success"
      type="filled"
      icon="save"
      size="small"
      @click="createUser"
    >
```

```
          Create User
        </vs-button>
        <vs-button
          color="danger"
          type="filled"
          icon="cancel"
          size="small"
          style="margin-left: 5px"
          @click="changeComponent('list', 0)"
        >
          Cancel
        </vs-button>
      </vs-row>
    </template>
```

To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



View component

In the following parts, we will create the visualization component. This component will be used for viewing the information of the user only.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. Create a file named `view.vue` in the `src/components` folder and open it.
2. Import the `UserForm` component, the `changeComponent` mixin, and the `getHttp` from `fetchApi`:

```
import {
  getHttp,
} from '../http/fetchApi';
import UserForm from './userForm';
import changeComponent from '../mixin/changeComponent';
```

3. In the `data` property, we will add a `userData` object with the `name`, `email`, `birthday`, `country`, and `phone` properties all defined as empty strings:

```
data: () => ({
  userData: {
    name: '',
    email: '',
    birthday: '',
    country: '',
    phone: '',
  },
}),
```

4. In the Vue `mixins` property, we need to add the `changeComponent` mixin:

```
mixins: [changeComponent],
```

5. In the Vue `inject` property, we need to declare the `'userId'` property:

```
inject: ['userId'],
```

6. In the Vue `components` property, add the `UserForm` component:

```
components: {
  UserForm,
},
```

7. For the methods, we will create the `getUserById` method. This method will fetch the user data by the current ID and set the `userData` value as the response from the `getHttp` function execution:

```
methods: {
  async getUserById() {
    const { data } = await
getHttp(`${window.location.href}api/users/${this.userId}`);
    this.userData = data;
  },
}
```

8. In the `beforeMount` life cycle hook, we will make it asynchronous, calling the `getUserById` method:

```
async beforeMount() {
  await this.getUserById();
},
```

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these instructions to create the component correctly:

1. Create a `vs-card` component with the `style` attribute defined as `margin: 20px`:

```
<vs-card
    style="margin: 20px;"
  >
</vs-card>
```

2. Inside the `vs-card` component, create a dynamic `<template>` named slot for `header`, with an `<h3>` tag and your title:

```
<template slot="header">
  <h3>
    View User
  </h3>
</template>
```

3. After that, create a `vs-row` component with a `vs-col` component inside of it, with the attributes of `vs-type` defined as `flex`, `vs-justify` defined as `left`, `vs-align` defined as `left`, and `vs-w` defined as `12`:

```
<vs-row>
  <vs-col
    vs-type="flex"
    vs-justify="left"
    vs-align="left"
    vs-w="12">
  </vs-col>
</vs-row>
```

4. Inside the `vs-col` component, we will add the `UserForm` component with the `v-model` directive bound to `userData` and the `disabled` attribute set to `true`:

```
<user-form
  v-model="userData"
  disabled
/>
```

5. Finally, in the card footer, we need to create a dynamic `<template>` named slot for `footer`. Inside this `<template>` we will add a `vs-row` component with the `vs-justify` attribute defined as `flex-start` and insert two `vs-button` components. The first will be for canceling this action and returning to the users lists. It will have the attributes of `color` defined as `danger`, `type` defined as `filled`, `icon` defined as `cancel`, `size` defined as `small`, and the`@click` event listener target to the `changeComponent` method with the `'list'` and `0` parameters. The second `vs-button` component will be for the editing the user and will have the attributes of `color` defined as `success`, `type` defined as `filled`, `icon` defined as `save`, `size` defined as `small` `style` defined as `margin-left: 5px`, and the `@click` event listener target to the `changeComponent` method with the `'list'` parameter and the injected `userId`:

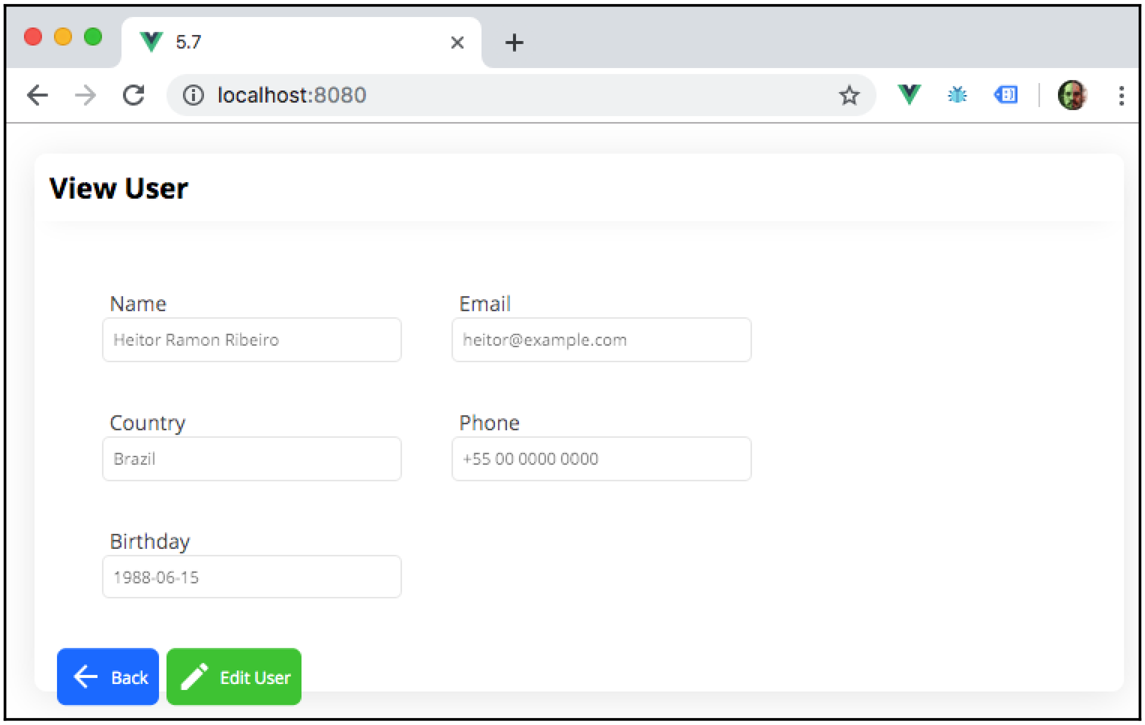
```
<template slot="footer">
  <vs-row vs-justify="flex-start">
    <vs-button
      color="primary"
      type="filled"
      icon="arrow_back"
      size="small"
      style="margin-left: 5px"
      @click="changeComponent('list', 0)"
    >
      Back
```

```
            </vs-button>
            <vs-button
              color="success"
              type="filled"
              icon="edit"
              size="small"
              style="margin-left: 5px"
              @click="changeComponent('edit', userId)"
            >
              Edit User
            </vs-button>
          </vs-row>
        </template>
```

To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



5.7

localhost:8080

View User

Name: Heitor Ramon Ribeiro

Email: heitor@example.com

Country: Brazil

Phone: +55 00 0000 0000

Birthday: 1988-06-15

Updating the user component

We just viewed the user data, and now we want to update it. We need to make a new component that is almost the same as the view component but has the method of updating the user and has the form enabled.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. Create a file named `update.vue` in the `src/components` folder and open it.
2. Import the `UserForm` component, the `changeComponent` mixin, and the `getHttp` and `patchHttp` functions from `fetchApi`:

   ```
   import UserForm from './userForm';
   import changeComponent from '../mixin/changeComponent';
   import {
     getHttp,
     patchHttp,
   } from '../http/fetchApi';
   ```

3. In the `data` property, we will add a `userData` object, with the `name`, `email`, `birthday`, `country`, and `phone` properties, all defined as empty strings:

   ```
   data: () => ({
     userData: {
       name: '',
       email: '',
       birthday: '',
       country: '',
       phone: '',
     },
   }),
   ```

4. In the Vue `mixins` property, we need to add the `changeComponent` mixin:

   ```
   mixins: [changeComponent],
   ```

5. In the Vue `inject` property, we need to declare the `'userId'` property:

   ```
   inject: ['userId'],
   ```

6. In the Vue `components` property, add the `UserForm` component:

    ```
    components: {
      UserForm,
    },
    ```

7. For the methods, we will create two: `getUserById` and `updateUser`. The `getUserById` method will fetch the user data by the current ID and set the `userData` value as the response from the `getHttp` function execution, and the `updateUser` will send the current `userDate` to the server via the `patchHttp` function and redirect back to the users list:

    ```
    methods: {
      async getUserById() {
        const { data } = await
          getHttp(`${window.location.href}api/users/${this.userId}`);
        this.userData = data;
      },
      async updateUser() {
        await patchHttp
          (`${window.location.href}api/users/${this.userData.id}`, {
          data: {
            ...this.userData,
          }
        });
        this.changeComponent('list', 0);
      },
    },
    ```

8. On the `beforeMount` life cycle hook, we will make it asynchronous, calling the `getUserById` method:

    ```
    async beforeMount() {
     await this.getUserById();
    },
    ```

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these instructions to create the component correctly:

1. Create a `vs-card` component with the `style` attribute defined as `margin: 20px`:

```
<vs-card
    style="margin: 20px;"
  >
</vs-card>
```

2. Inside the `vs-card` component, create a dynamic `<template>` named slot for `header`, with an `<h3>` tag and your title:

```
<template slot="header">
  <h3>
    Update User
  </h3>
</template>
```

3. After that, create a `vs-row` component with a `vs-col` component inside of it, with the attributes of `vs-type` defined as `flex`, `vs-justify` defined as `left`, `vs-align` defined as `left`, and `vs-w` defined as `12`:

```
<vs-row>
  <vs-col
    vs-type="flex"
    vs-justify="left"
    vs-align="left"
    vs-w="12">
  </vs-col>
</vs-row>
```

4. Inside the `vs-col` component, we will add the `UserForm` component with the `v-model` directive bound to `userData` and the `disabled` attribute set to `true`:

```
<user-form
  v-model="userData"
  disabled
/>
```

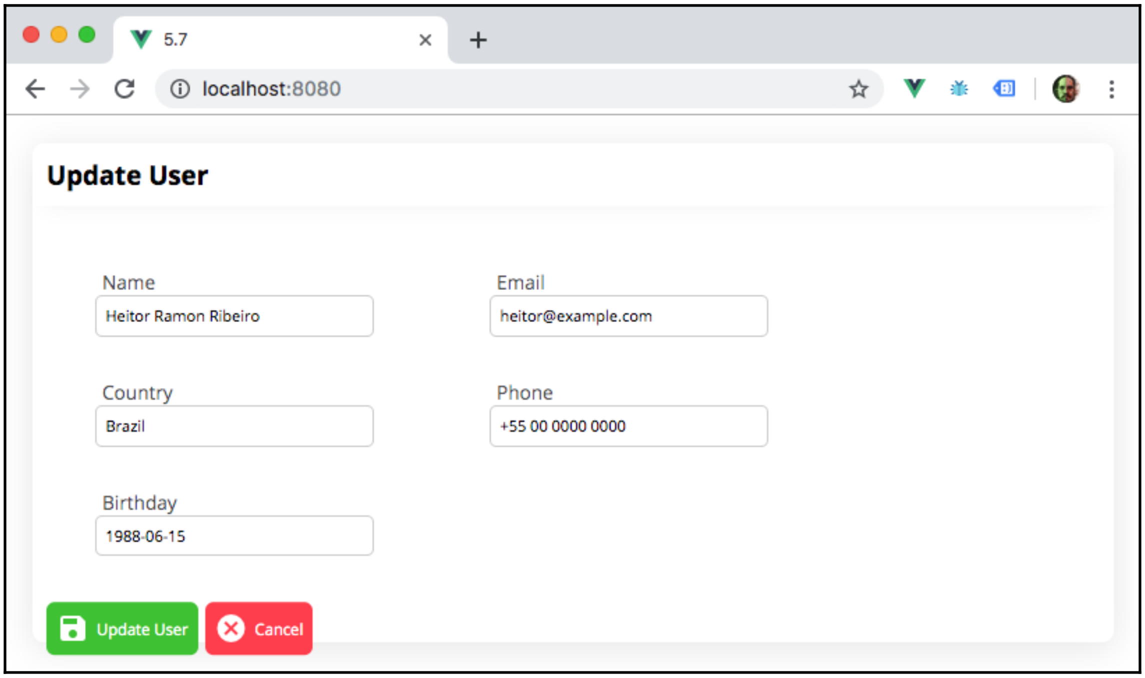
5. Finally, in the card footer, we need to create a dynamic `<template>` named slot for `footer`. Inside `<template>`, we will add a `vs-row` component with the `vs-justify` attribute defined as `flex-start` and insert two `vs-button` components. The first will be for creating the user and will have the attributes of `color` defined as `success`, `type` defined as `filled`, `icon` defined as `save`, `size` defined as `small`, and the `@click` event listener target to the `updateUser` method. The second `vs-button` component will be for canceling this action and returning to the users lists. It will have the attributes of `color` defined as `danger`, `type` defined as `filled`, `icon` defined as `cancel`, `size` defined as `small`, `style` defined as `margin-left: 5px`, and the `@click` event listener target to the `changeComponent` method with the `'list'` and `0` parameters:

```
<template slot="footer">
  <vs-row vs-justify="flex-start">
    <vs-button
      color="success"
      type="filled"
      icon="save"
      size="small"
      @click="updateUser"
    >
      Update User
    </vs-button>
    <vs-button
      color="danger"
      type="filled"
      icon="cancel"
      size="small"
      style="margin-left: 5px"
      @click="changeComponent('list', 0)"
    >
      Cancel
    </vs-button>
  </vs-row>
</template>
```

To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



Update User

Name: Heitor Ramon Ribeiro
Email: heitor@example.com
Country: Brazil
Phone: +55 00 0000 0000
Birthday: 1988-06-15

How it works...

The CRUD interface that we created works like a router application, with three routes, the index or list, the view, and the edit route. Each route has its own screen and components, with separated logic functions.

We created an abstract `UserForm` component that was used on the `View` and `Update` components. This abstract component can be used in many other components, as it does not require any base logic to work; it's like an input but made of several inputs.

Using the provide/inject API of Vue, we were able to pass the `userId` to each of the components in an observable way, which means that when the variable is updated, the component receives the updated variable. This is not achievable using the normal Vue API, so we had to use the `Object.defineProperty` and use the `provide` property as a factory function to return the final object.

See also

You can find more information about `Vuesax` at `https://lusaxweb.github.io/vuesax/`.

You can find more information about `Object.defineProperty` at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty`.

You can find more information about the Vue provide/inject API at `https://vuejs.org/v2/guide/components-edge-cases.html`.

6
Managing Routes with vue-router

One of the main parts of your application is router management. Here, it is possible to bring together infinite component combinations in a single place.

A router is capable of coordinating component rendering and dictating where the application should be, depending on the URL. There are many ways to increase the customization of `vue-router`. You can add route guards to check whether specific routes are navigatable by access level or fetch data before entering the route to manage errors on your application.

In this chapter, you will learn how to create application routes, dynamic routes, alias and credited routes, and nested router views. We'll also look at how to manage errors, create router guards, and lazy load your pages.

In this chapter, we'll cover the following recipes:

- Creating a simple route
- Creating a programmatic navigation
- Creating a dynamic router path
- Creating a route alias
- Creating a route redirect
- Creating a nested router view
- Creating a 404 error page
- Creating an authentication middleware
- Lazy loading your pages asynchronously

Technical requirements

In this chapter, we will be using **Node.js** and **Vue-CLI.**

Attention Windows users: you need to install an npm package called `windows-build-tools` to be able to install the following required packages. To do so, open the PowerShell as an administrator and execute the following command:

```
> npm install -g windows-build-tools
```

To install Vue-CLI, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install -g @vue/cli @vue/cli-service-global
```

Creating a simple route

In your application, you can create an infinite combination of routes that can lead to any number of pages and components.

`vue-router` is the maintainer of this combination. We need to use this to set instructions on how to create paths and lay down routes for our visitors.

In this recipe, we will learn how to create an initial route that will lead to a different component.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To create a Vue-CLI project, follow these steps:

1. We need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create initial-routes
```

2. The CLI will ask some questions that will help with the creation of the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *Spacebar* to select an option.

3. There are two methods for starting a new project. The default method is a basic Babel and ESLint project without any plugins or configuration, and the `Manually` mode, where you can select more modes, plugins, linters, and options. We will go for `Manually`:

```
? Please pick a preset: (Use arrow keys)
 default (babel, eslint)
❯ Manually select features
```

4. Now we are asked about the features that we will want on the project. Those features are some Vue plugins such as Vuex or Vue Router (Vue-Router), testers, linters, and more. Select `Babel`, `Router`, and `Linter / Formatter`:

```
? Check the features needed for your project: (Use arrow keys)
❯ Babel
  TypeScript
  Progressive Web App (PWA) Support
❯ Router
  Vuex
  CSS Pre-processors
❯ Linter / Formatter
  Unit Testing
  E2E Testing
```

5. Now Vue-CLI will ask if you want to use the history mode on the route management. We will choose `Y` (yes):

```
? Use history mode for router? (Requires proper server setup for
  index fallback in production) (Y/n) y
```

6. Continue this process by selecting a linter and formatted. In our case, we will select `ESLint + Airbnb config`:

   ```
   ? Pick a linter / formatter config: (Use arrow keys)
     ESLint with error prevention only
   ❯ ESLint + Airbnb config
     ESLint + Standard config
     ESLint + Prettier
   ```

7. After the linting rules are set, we need to define when they are applied to your code. They can be either applied on save or fixed on commit:

   ```
   ? Pick additional lint features: (Use arrow keys)
     Lint on save
   ❯ Lint and fix on commit
   ```

8. After all those plugins, linters, and processors are defined, we need to choose where the settings and configs are stored. The best place to store them is in a dedicated file, but it is also possible to store them in the `package.json`:

   ```
   ? Where do you prefer placing config for Babel, ESLint, etc.? (Use
   arrow keys)
   ❯ In dedicated config files
     In package.json
   ```

9. Now you can choose whether you want to make this selection a preset for future projects, so you don't need to reselect everything again:

   ```
   ? Save this as a preset for future projects? (y/N) n
   ```

Our recipe will be divided into five parts:

- Creating the `NavigationBar` component
- Creating the contact page
- Creating the about page
- Changing the application's main component
- Creating the routes

Let's get started.

Creating the NavigationBar component

Now we are going to create the `NavigationBar` component that will be used in our application.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. Create a `navigationBar.vue` file in the `src/components` folder and open it.
2. Create a default `export` object of the component, with the Vue property `name`:

```
<script>
export default {
  name: 'NavigationBar',
};
</script>
```

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these instructions to create the component correctly:

1. Create a `div` HTML element with the `id` attribute defined as `"nav"`, and inside of it, create three `RouterLink` components. Those components will point to the `Home`, `About`, and `Contact` routes. In the `RouterLink` component, we will add a `to` attribute that will be defined as the route for each component, respectively, and define the text content as the name of the menu:

```
<div id="nav">
  <router-link to="/">
    Home
  </router-link> |
  <router-link to="/about">
    About
  </router-link> |
  <router-link to="/contact">
    Contact
  </router-link>
</div>
```

Creating the contact page

We need to make sure the contact page gets rendered when the user enters the `/contact` URL. To do so, we need to create a single file component to be used as the contact page.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. In the `src/views` folder, create a new file called `contact.vue` and open it.
2. Create a default `export` object of the component, with the Vue property `name`:

```
<script>
export default {
  name: 'ContactPage',
};
</script>
```

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these instructions to create the component correctly:

1. Create a `div` HTML element, with the `class` attribute defined as `"contact"`.
2. Inside of the `<h1>` HTML element, add a text context displaying the current page:

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

Creating the about page

We need to make the contact page be rendered when the user enters the `/about` URL. In the following subsections, we will create the Single File component for the about page.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. In the `src/views` folder, create a new file called `About.vue` and open it.
2. Create a default export object of the component, with the Vue property `name`:

```
<script>
export default {
  name: 'AboutPage',
};
</script>
```

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these instructions to create the component correctly:

1. Create a `div` HTML element with the `class` attribute defined as `"about"` .
2. Inside of it, place an `<h1>` element with a text context displaying the current page:

```
<template>
  <div class="about">
    <h1>This is an about page</h1>
  </div>
</template>
```

Changing the application's main component

After creating the pages and the navigation bar, we need to change the application's main component to be able to render the routes and have the navigation bar at the top.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. Open `App.vue` in the `src` folder.
2. Import the `NavigationBar` component:

```
import NavigationBar from './components/navigationBar.vue';
```

3. In the Vue `components` property, declare the imported `NavigationBar`:

   ```
   export default {
     components: { NavigationBar },
   };
   ```

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Inside the `div` HTML element, add the `NavigationBar` component and the `RouterView` component:

```
<template>
  <div id="app">
    <navigation-bar />
    <router-view/>
  </div>
</template>
```

Creating the routes

Now we need to make the routes available in the application. To do so, first, we need to declare the routes and the components that the routes will render. Follow these steps to create your Vue application router correctly:

1. In the `src/router` folder, open the `index.js` file.
2. Import the `Contact` component page:

   ```
   import Vue from 'vue';
   import VueRouter from 'vue-router';
   import Home from '../views/Home.vue';
   import Contact from '../views/contact.vue';
   ```

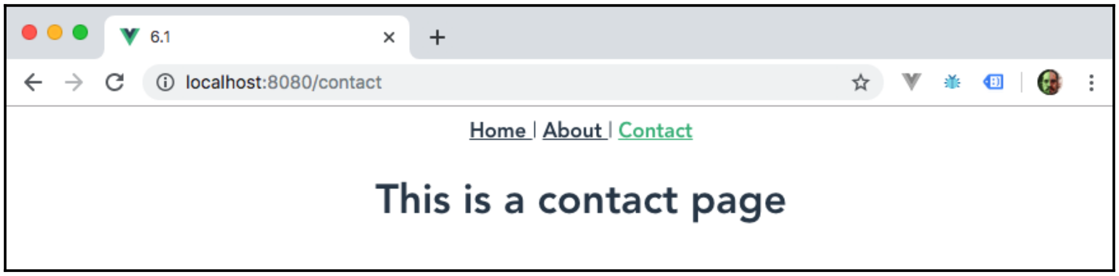
3. In the `routes` array, we need to create a new `route` object. This object will have the `path` property defined as `'/contact'`, `name` defined as `'contact'`, and `component` pointing to the imported `Contact` component:

   ```
   {
     path: '/contact',
     name: 'contact',
     component: Contact,
   },
   ```

To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



This is a contact page

How it works...

When `vue-router` is added to Vue as a plugin, it starts to watch for changes on `window.location.pathname`, and other URL properties, to check the weight of the current URL on the browser against the list of URLs on your router configurations.

In this particular case, we are using a direct URL and a non-dynamic URL. Because of that, the `vue-router` plugin only needs to check direct matches of the URL paths and doesn't need to weigh the possible matches against a regex validator.

After a URL is matched, the `router-view` component acts as a **dynamic component** and renders the component we defined on the `vue-router` configuration.

See also

You can find more information about `vue-router` at `https://router.vuejs.org/`.

You can find more information about Vue CLI at `https://cli.vuejs.org/`.

Creating a programmatic navigation

When using `vue-router`, it is also possible to change the current route of the application through function execution, without the need for special `vue-router` components for creating links.

Using programmatic navigation, you can make sure all the route redirections can be executed anywhere in your code. Using this method enables the usage of special route directions, such as passing parameters and navigation with the route name.

In this recipe, we will learn how to execute a programmatic navigation function, using the extra possibilities it provides.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we can use the Vue project with Vue-CLI that we created in the '*Creating a simple route*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create route-project
```

Choose the manual features and add the `Router` as a needed feature, as indicated in the '*How to do it...*' section in the '*Creating a simple route*' recipe.

Our recipe will be divided into two parts:

- Changing the application's main component
- Changing the contact view

Let's get started.

Changing the application's main component

We will start with the `App.vue` file. We will add a programmatic navigation function to be executed after a timeout, which will be added to the component life cycle hook.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. Open `App.vue` in the `src` folder.
2. Add a `mounted` property:

```
mounted() {}
```

3. In the `mounted` property, we need to add a `setTimeout` function, which will execute the `$router.push` function. When executed, this function receives a JavaScript object as an argument, with two properties, `name`, and `params`:

```
mounted() {
  setTimeout(() => {
    this.$router.push({
      name: 'contact',
      params: {
        name: 'Heitor Ribeiro',
        age: 31,
      },
    });
  }, 5000);
},
```

Changing the contact view

On the contact view, we need to add an event listener, which will grab the route change and execute an action.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. Open `contact.vue` in the `src/views` folder.
2. Add a new `mounted` property:

   ```
   mounted() {}
   ```

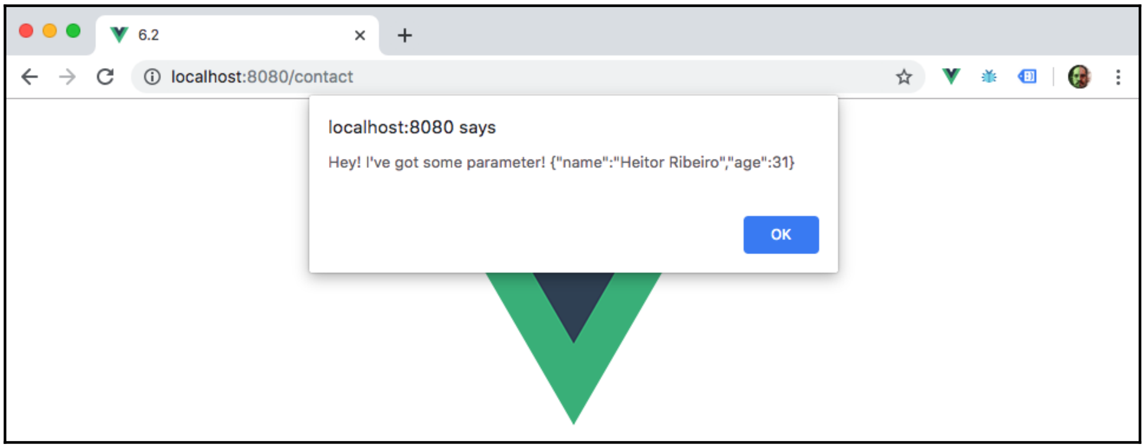
3. In this property, we will add a verification that will check whether there are any parameters on the `$route.params` object and display an `alert` with that `$route.params`:

   ```
   mounted() {
     if (Object.keys(this.$route.params).length) {
       alert(`Hey! I've got some parameter!
          ${JSON.stringify(this.$route.params)}`);
     }
   },
   ```

To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



localhost:8080 says

Hey! I've got some parameter! {"name":"Heitor Ribeiro","age":31}

How it works...

The `$router.push` function, when executed, tells `vue-router` to change where the application is, and in this process, you are passing down some parameters to the new router that will replace the current route. In those parameters, there's a property called `params`, which sends a group of parameters to the new router.

When entering this new router, all the parameters that we will have called from within the router will be available in the `$route.params` object; there, we can use it in our view or component.

There's more...

In the programmatic navigation, it's possible to navigate through the routers, adding them to the browser history with the `$router.push` function, but there are other functions that can be used too.

The `$router.replace` function will replace the user navigation history for a new one, making it unable to go back to the last page.

`$router.go` is used to move the user navigation history in steps. To go forward, you need to pass positive numbers and to go backward, you will need to pass negative numbers.

See also

You can find more information about `vue-router` programmatic navigation at `https://router.vuejs.org/guide/essentials/navigation.html`.

Creating a dynamic router path

Adding a route to your application is a must, but sometimes you need more than just simple routes. In this recipe, we'll take a look at how dynamic routes come into play. With dynamic routes, you can define custom variables that can be set via the URL, and your application can start with those variables already defined.

In this recipe, we will learn how to use a dynamic router path on a CRUD list.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we will use the Vue project with Vue-CLI that we completed in the '*Creating a CRUD interface with axios and Vuesax*' recipe in `Chapter 5`, *Fetching Data from the Web via HTTP Requests*. In the following steps, we will add `vue-router` to the project through the Vue UI dashboard:

1. First, you will need to open `vue ui`. To do this, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue ui
```

2. There, you will need to import the project by locating the project folder. After importing `vue ui`, you will be redirected to the dashboard.
3. Add `vue-router` to the plugins by going to the plugins management page and clicking on the **Add vue-router** button. Then, click on the **Continue** button.
4. The Vue-CLI will automatically install and configure vue-router on the project for us. We now need to create each view for the **List**, **View,** and **Edit** pages.

To start view development, we will go first by the user list route. In each route, we will deconstruct the old component that we had made, and recreate it as a view.

Our recipe will be divided into eight parts:

- Changing the application's main component
- Changing the route mixin
- Axios instance configuration
- User list view

- User create view
- User information view
- User update view
- Creating dynamic routes

Let's get started.

Changing the application's main component

After adding the vue-router plugin, `App.vue` will change. We need to revert the changes made by the installation of the `vue-router`. This is needed because when `vue-ui` adds the `vue-router` plugin, it will change `App.vue`, adding an example code that we don't need.

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these instructions to create the component correctly:

1. Open `App.vue` in the `src` folder.
2. Remove everything, leaving just the `div#app` HTML element and the `router-view` component:

```
<template>
  <div id="app">
    <router-view/>
  </div>
</template>
```

Changing the route mixin

In the previous recipe, we used a `changeComponent` mixin. Now that we are going to work with routes, we need to change this mixin to a `changeRoute` mixin and alter its behavior. In the following steps, we will change how the mixin works, to be able to change the route instead of the component:

1. In the `src/mixin` folder, rename `changeComponent.js` to `changeRoute.js` and open it.

2. We will remove the `changeComponent` method and create a new one called `changeRoute`. This new method will receive two arguments, `name` and `id`. The `name` argument is the route name, as set in the `vue-router` configuration and the `id` will be the user id that we will pass a parameter in the route change. This method will execute `$router.push`, passing those arguments as the parameters:

```
export default {
  methods: {
    async changeRoute(name, id = 0) {
      await this.$router.push({
        name,
        params: {
          id,
        },
      });
    },
  }
}
```

Axios instance configuration

To fetch the data in the MirageJS server, we will need to define some options in our axios instance. Now, in the following steps, we will configure the axios instance to work with the new routering system:

1. In the `src/http` folder, open the `baseFetch.js` file.
2. At the creator of the `localApi` instance of `axios`, we will need to add an `options` object, passing the `baseURL` property. This `baseURL` will be the current browser navigation URL:

```
const localApi = createAxios({
  baseURL:
`${document.location.protocol}//${document.location.host}`,
});
```

User list view

To create our view, we will extract the code from the `list.vue` component and reshape it as a page view.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. Move the `list.vue` file from `components` to the `views` folder, and rename it `List.vue`.
2. Remove the old `changeComponent` mixin import and import the new `changeRoute` mixin:

```
import changeRouteMixin from '@/mixin/changeRoute';
```

3. At the Vue `mixins` property, we need to replace `changeComponent` with `changeRoute`:

```
mixins: [changeRouteMixin],
```

4. In the `getAllUsers` and `deleteUser` methods, we need to remove `${window.location.href}` from the `getHttp` and `deleteHttp` function parameters:

```
methods: {
  async getAllUsers() {
    const { data } = await getHttp(`api/users`);
    this.userList = data;
  },
  async deleteUser(id) {
    await deleteHttp(`api/users/${id}`);
    await this.getAllUsers();
  },
}
```

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these instructions to create the component correctly:

1. We need to wrap the `VsCard` component and its child contents with a `VsRow` and `VsCol` component. The `VsCol` component will have the `vs-type` attribute defined as `'flex'`, `vs-justify` defined as `'left'`, `vs-align` defined as `'left'`, and `vs-w` defined as `12`:

```
<template>
  <vs-row>
    <vs-col
```

```
        vs-type="flex"
        vs-justify="left"
        vs-align="left"
        vs-w="12">
        <vs-card... />
      </vs-col>
    </vs-row>
  </template>
```

2. On the actions buttons, we will change the `changeComponent` functions to `changeRoute`:

```
<vs-td :data="data[index].id">
  <vs-button
    color="primary"
    type="filled"
    icon="remove_red_eye"
    size="small"
    @click="changeRoute('view', data[index].id)"
  />
  <vs-button
    color="success"
    type="filled"
    icon="edit"
    size="small"
    @click="changeRoute('edit', data[index].id)"
  />
  <vs-button
    color="danger"
    type="filled"
    icon="delete"
    size="small"
    @click="deleteUser(data[index].id)"
  />
</vs-td>
```

3. At the `VsCard` footer, we need to change the actions buttons, `changeComponent` method to the `changeRoute` method:

```
<template slot="footer">
  <vs-row vs-justify="flex-start">
    <vs-button
      color="primary"
      type="filled"
      icon="fiber_new"
      size="small"
      @click="changeRoute('create')"
    >
```

```
        Create User
      </vs-button>
    </vs-row>
  </template>
```

User create view

To create our view, we will extract the code from the `create.vue` component and reshape it as a page view.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. Move the `create.vue` file from `components` to the `views` folder, and rename it `Create.vue`.
2. Remove the old `changeComponent` mixin import and import the new `changeRoute` mixin:

   ```
   import changeRouteMixin from '@/mixin/changeRoute';
   ```

3. At the Vue `mixins` property, we need to replace `changeComponent` with `changeRoute`:

   ```
   mixins: [changeRouteMixin],
   ```

4. On the `getUserById` method, we need to remove `${window.location.href}` from the `postHttp` function URL and change the `changeComponent` functions to `changeRoute`:

   ```
   async createUser() {
     await postHttp(`/api/users`, {
       data: {
         ...this.userData,
       }
     });
     this.changeRoute('list');
   },
   ```

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these instructions to create the component correctly:

1. We need to wrap the `VsCard` component and its child contents with a `VsRow` and `VsCol` component. The `VsCol` component will have the `vs-type` attribute defined as `'flex'`, `vs-justify` defined as `'left'`, `vs-align` defined as `'left'`, and `vs-w` defined as `12`:

```
<template>
  <vs-row>
    <vs-col
      vs-type="flex"
      vs-justify="left"
      vs-align="left"
      vs-w="12">
      <vs-card... />
    </vs-col>
  </vs-row>
</template>
```

2. On the `VsCard` footer, we need to change the `Cancel` button's `changeComponent` functions to `changeRoute`:

```
<vs-button
  color="danger"
  type="filled"
  icon="cancel"
  size="small"
  style="margin-left: 5px"
  @click="changeRoute('list')"
>
  Cancel
</vs-button>
```

User information view

To create our view, we will extract the code from the `view.vue` component and reshape it as a page view.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. Move the `view.vue` file from `src/components` to the `src/views` folder and rename it as `View.vue`.
2. Remove the old `changeComponent` mixin import and import the new `changeRoute` mixin:

```
import changeRouteMixin from '@/mixin/changeRoute';
```

3. At the Vue `mixins` property, we need to replace `changeComponent` with `changeRoute`:

```
mixins: [changeRouteMixin],
```

4. Create a new `computed` property in the `component` object, with the property `userId`, which will return `$route.params.id`:

```
computed: {
  userId() {
    return this.$route.params.id;
  },
},
```

5. On the `getUserById` method, we need to remove `${window.location.href}` from the `getHttp` function URL:

```
methods: {
  async getUserById() {
    const { data } = await getHttp(`api/users/${this.userId}`);
    this.userData = data;
  },
}
```

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these instructions to create the component correctly:

1. We need to wrap the `VsCard` component and its child contents with a `VsRow` and `VsCol` component. The `VsCol` component will have the `vs-type` attribute defined as `'flex'`, `vs-justify` defined as `'left'`, `vs-align` defined as `'left'`, and `vs-w` defined as `12`:

```
<template>
  <vs-row>
    <vs-col
      vs-type="flex"
      vs-justify="left"
      vs-align="left"
      vs-w="12">
      <vs-card... />
    </vs-col>
  </vs-row>
</template>
```

2. On the `VsCard` footer, we need to change the back button `changeComponent` functions to `changeRoute`:

```
<vs-button
  color="primary"
  type="filled"
  icon="arrow_back"
  size="small"
  style="margin-left: 5px"
  @click="changeRoute('list')"
>
  Back
</vs-button>
```

User update view

To create our view, we will extract the code from the `update.vue` component and reshape it as a page view.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. Move the `update.vue` file from `src/components` to the `src/views` folder and rename it `Edit.vue`.
2. Remove the old `changeComponent` mixin import and import the new `changeRoute` mixin:

   ```
   import changeRouteMixin from '@/mixin/changeRoute';
   ```

3. At the Vue `mixins` property, we need to replace `changeComponent` with `changeRoute`:

   ```
   mixins: [changeRouteMixin],
   ```

4. Create a new `computed` property in the `component` object, with the `userId` property , which will return `$route.params.id`:

   ```
   computed: {
     userId() {
       return this.$route.params.id;
     },
   },
   ```

5. On the `getUserById` and `updateUser` methods, we need to remove `${window.location.href}` from the `getHttp` and `patchHttp` function URLs and change the `changeComponent` functions to `changeRoute`:

   ```
   methods: {
     async getUserById() {
       const { data } = await getHttp(`api/users/${this.userId}`);
       this.userData = data;
     },
     async updateUser() {
       await patchHttp(`api/users/${this.userData.id}`, {
         data: {
           ...this.userData,
         }
       });
       this.changeRoute('list');
     },
   },
   ```

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these instructions to create the component correctly:

1. We need to wrap the `VsCard` component and its child contents with a `VsRow` and `VsCol` component. The `VsCol` component will have the `vs-type` attribute defined as `'flex'`, `vs-justify` defined as `'left'`, `vs-align` defined as `'left'`, and `vs-w` defined as `12`:

```
<template>
  <vs-row>
    <vs-col
      vs-type="flex"
      vs-justify="left"
      vs-align="left"
      vs-w="12">
      <vs-card... />
    </vs-col>
  </vs-row>
</template>
```

2. On the `VsCard` footer, we need to change the `Cancel` button's change `Component` functions to `changeRoute`:

```
<vs-button
  color="danger"
  type="filled"
  icon="cancel"
  size="small"
  style="margin-left: 5px"
  @click="changeRoute('list')"
>
  Cancel
</vs-button>
```

Creating dynamic routes

Now, with our page views created, we need to create our routes and make them accept parameters, transforming them into dynamic routes. In the following steps, we will create the dynamic routes of the application:

1. Open `index.js` in the `src/router` folder.
2. First, we need to import the four new pages – `List`, `View`, `Edit`, `Create`, and `Update`:

   ```
   import List from '@/views/List.vue';
   import View from '@/views/View.vue';
   import Edit from '@/views/Edit.vue';
   import Create from '@/views/Create.vue';
   ```

3. On the `routes` array, we will add a new route object for each one of the pages that were imported. In this object, there will be three properties: `name`, `path`, and `component`.
4. For the `list` route, we will define `name` as `'list'`, `path` as `'/'`, and `component` as the imported `List` component:

   ```
   {
     path: '/',
     name: 'list',
     component: List,
   },
   ```

5. On the `view` route, we will define `name` as `'view'`, `path` as `'/view/:id'`, and `component` as the imported `View` component:

   ```
   {
     path: '/view/:id',
     name: 'view',
     component: View,
   },
   ```

6. In the `edit` route, we will define `name` as `'edit'`, `path` as `'/edit/:id'`, and `component` as the imported `Edit` component:

   ```
   {
     path: '/edit/:id',
     name: 'edit',
     component: Edit,
   },
   ```

7. Finally, at the `create` route, we will define `name` as `'create'`, `path` as `'/create'`, and `component` as the imported `Create` component:

```
{
  path: '/create',
  name: 'create',
  component: Create,
},
```

8. When the `VueRouter` is created, we will add the `mode` options property and set it as `'history'`:

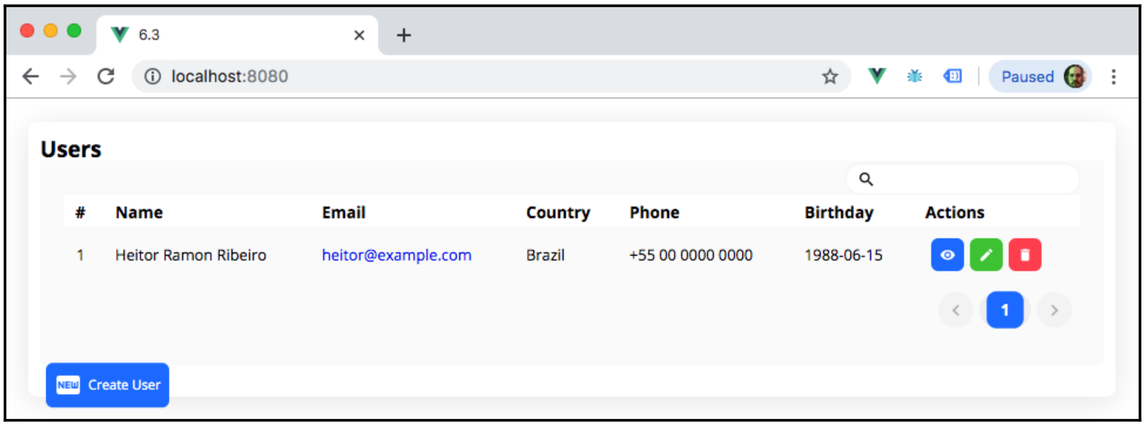
```
const router = new VueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes
});
```

To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

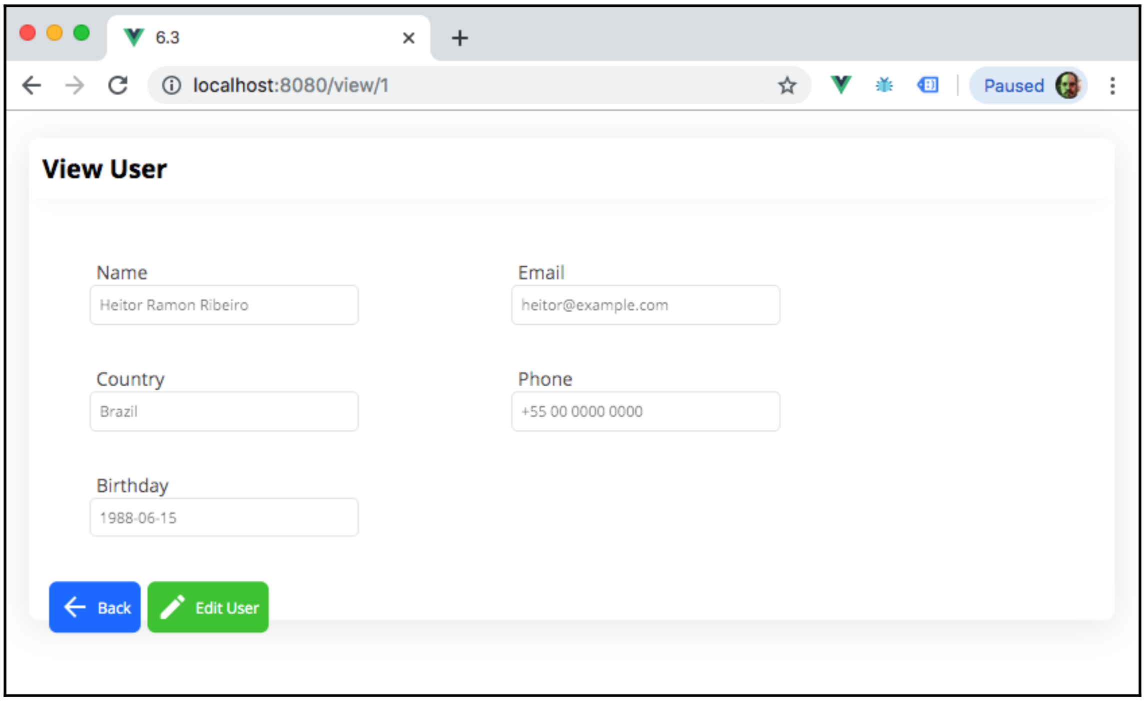
Here is your component rendered and running:

- **List View Route -** `/` will be your user list page, containing a list of all the users in your application and buttons to view, edit, and delete it, as well as a button to create a new user:



#	Name	Email	Country	Phone	Birthday	Actions
1	Heitor Ramon Ribeiro	heitor@example.com	Brazil	+55 00 0000 0000	1988-06-15	

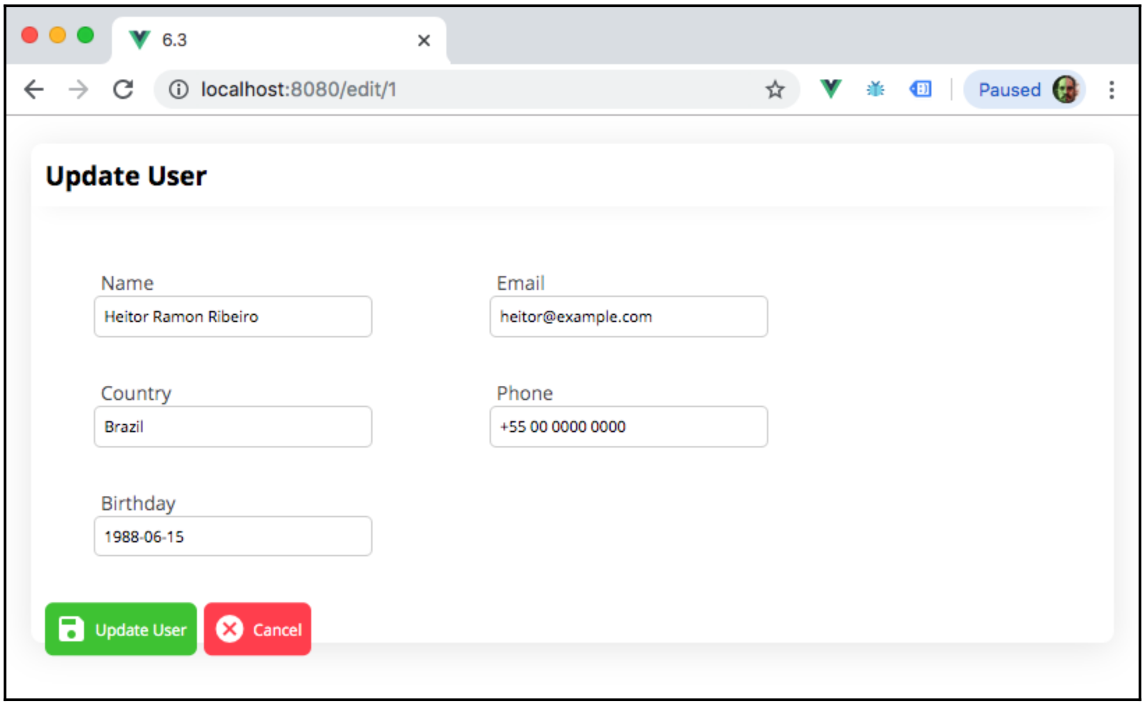
- **User View Route -** `/view/:id` will be your user view page, where it's possible to view your user information, such as the user's name, email, country, birthday, and phone number:



View User

Name: Heitor Ramon Ribeiro	Email: heitor@example.com
Country: Brazil	Phone: +55 00 0000 0000
Birthday: 1988-06-15	

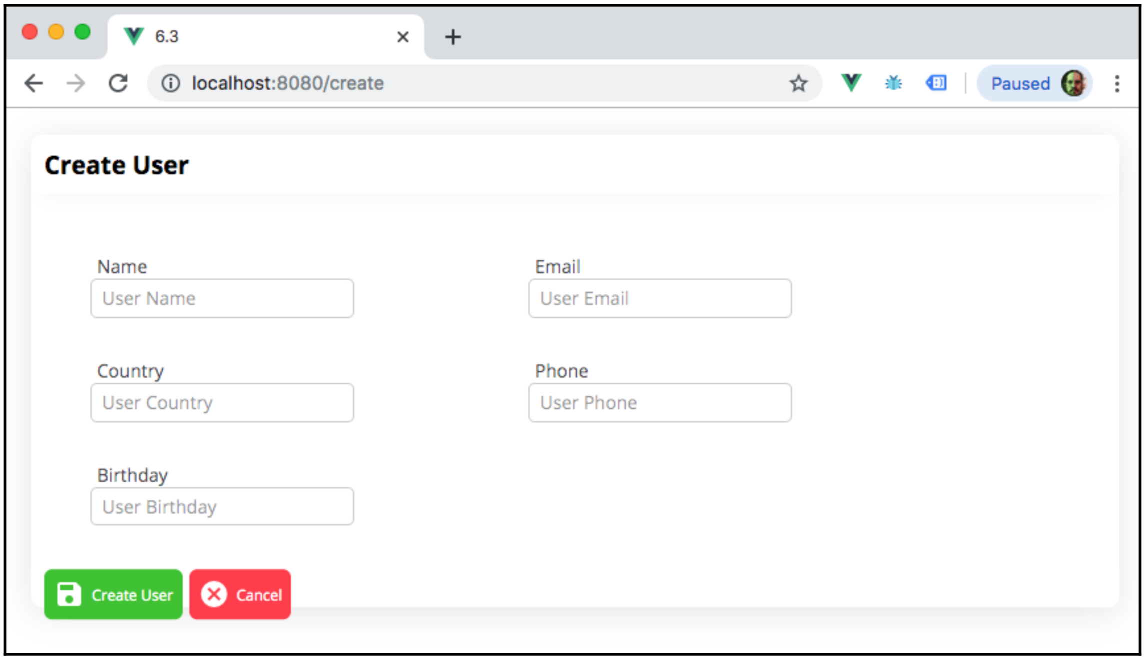
- **User Edit Route -** `/update/:id` will be your user edit page, where it's possible to edit your user's information, changing the user's name, email, country, birthday, and phone number:



Update User

Name: Heitor Ramon Ribeiro	Email: heitor@example.com
Country: Brazil	Phone: +55 00 0000 0000
Birthday: 1988-06-15	

- **Create User Route -** `/update/:id` will be your user creation page, where it's possible to create a new user on the system:



How it works...

When `vue-router` is created, and the route is passed for matching, the router analysis check for the best match for the route based on a RegEx for defining a weight on each route.

When a route is defined and has a variable in its path, you need to add a `:` before the variable parameter. This parameter is passed down to the component in the `$route.params` property.

See also

You can find more information about the dynamic router matching at `https://router.vuejs.org/guide/essentials/dynamic-matching.html`.

Creating a route alias

Every application is a living organism – it evolves, mutates, and changes day by day. Sometimes these evolutions can come through the form of a router change, for better naming or for a deprecated service. In `vue-router`, it's possible to make all those changes invisible to the user, so when they use old links, they still can access the application.

In this recipe, we will learn how to create a route alias for our application and use it.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we will use the Vue project with Vue-CLI that we completed in the '*Creating a dynamic router path*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create http-project
```

Choose the manual features and add `router` as a required feature, as indicated in the '*How to do it...*' section of the '*Creating a simple route*' recipe.

Now, in the following steps, we will create the router alias:

1. Open `index.js` in the `src/router` folder.
2. At the `list` object, we will change the `path` property from `'/'` to `'/user'` and for the `alias` property, we will set `'/'`:

```
{
  path: '/user',
  name: 'list',
  alias: '/',
  component: List,
},
```

3. In the `view` object, we will change the `path` property from `'/view/:id'` to `'/user/:id'` and we will set the `alias` property to `'/view/:id'`:

```
{
  path: '/user/:id',
  name: 'view',
  alias: '/view/:id',
  component: View,
},
```

4. In the `edit` object, we will change the `path` property from `'/edit/:id'` to `'/user/edit/:id'` and set the `alias` property to `'/edit/:id'`:

```
{
  path: '/user/edit/:id',
  name: 'edit',
  alias: '/edit/:id',
  component: Edit,
},
```

5. Finally, in the `create` object, we will change the `path` property from `'/create'` to `'/user/create'` and set the `alias` property to `'/create'`:

```
{
  path: '/user/create',
  name: 'create',
  alias: '/create',
  component: Create,
},
```

How it works...

When the user enters your application, `vue-router` will try to match paths to the one that the user is trying to access. If there is a property called `alias` in the route object, this property will be used by the `vue-router` to maintain the old route under the hood and use the alias route instead. If an alias is found, the component of that alias is rendered, and the router remains as the alias, not showing the user the change, making it transparent.

In our scenario, we made a transformation for our application to now handle all the users called on the `/user` namespace, but still maintaining the old URL structure so that if an old visitor tries to access the website, they will be able to use the application normally.

See also

You can find more information about the `vue-router` alias at `https://router.vuejs.org/guide/essentials/redirect-and-alias.html#alias`.

Creating route redirects

Router redirect works almost the same as the router alias, but the main difference is that the user is truly redirected to the new URL. Using this process, you are able to manage how the new route can be loaded.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we will use the Vue project with Vue-CLI that we completed in the '*Creating a route alias*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create http-project
```

Choose the manual features and add `Router` as a required feature, as indicated in the '*How to do it...*' steps in the '*Creating a simple route*' recipe.

Now, in these steps, we will create the router redirect rules:

1. Open `index.js` in the `src/router` folder.
2. Insert a new route object at the end of the `routes` array. This object will have two properties, `path` and `redirect`. In the `path` property, we need to define the path that the user will enter, `'/create-new-user'`, and in `redirect`, the path that the user will be redirected to, in this case, `'/user/create'`:

```
{
  path: '/create-new-user',
  redirect: '/user/create',
},
```

3. Create a new object, and this object will have two properties, `path` and `redirect`. In the `path` property, we need to define the path that the user will enter, `'/users'`, and in the `redirect`, we will create an object with a property called `name` and will put the value as `'list'`:

```
{
  path: '/users',
  redirect: {
    name: 'list',
  },
},
```

4. Create a new object. This object will have two properties, `path` and `redirect`. In the `path` property, we need to define the path that the user will enter, `'/my-user/:id?'`, and in the `redirect`, we will create a function, which will receive an argument, `to`, which is an object of the current route. We need to check whether the user ID is present in the route, to redirect the user to the edit page. Otherwise, we will redirect them to the user list:

```
{
  path: '/my-user/:id?',
  redirect(to) {
    if (to.params.id) {
      return '/user/:id';
    }
    return '/user';
  },
},
```

5. Finally, in the end, we will create a route object with two properties, `path` and `redirect`. In the `path` property, we need to define the path that the user will enter, `'/*'`, and in the `redirect`, we need to define the `redirect` property as `'/'`:

```
{
  path: '*',
  redirect: '/',
},
```

Remember that the last route with the `'*'` will always be the route that will be rendered when there is no match in the URL that your user is trying to enter.

How it works...

As we define `redirect` as a new route, it works similar to the alias, but the `redirect` property can receive three types of arguments: a string when redirecting for the route itself, objects when redirecting with other parameters such as the name of the route, and last but not least, the function type, which `redirect` can handle and return one of the first two objects so the user can be redirected.

See also

You can find more information about the `vue-router` redirect at `https://router.vuejs.org/guide/essentials/redirect-and-alias.html#redirect`.

Creating a nested router view

In `vue-router`, nested routes are like a namespace for your routes, where you can have multiple levels of routes inside the same route, use a base view as the main view, and have the nested routes rendered inside.

In a multi-module application, this is used to handle routes like CRUD, where you will have a base route, and the children will be the CRUD views.

In this recipe, you will learn how to create a nested route.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we will use the Vue project with Vue-CLI that we used in the '*Creating a dynamic router path*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create http-project
```

Choose the manual features and add `Router` as a required feature, as indicated in the '*How to do it...*' section in the '*Creating a simple route*' recipe.

Our recipe will be divided into two parts:

- Creating the `router-view` on the layout
- Changing the router files

Let's get started.

Creating the router-view on the layout

When using `vue-router` with children's routes, we need to create the main view, which will have a special component called `RouterView`. This component will render the current router inside the layout or page you are rendering.

Now, in the following steps, we will create the layout for the pages:

1. In the `src/views` folder, we need to create a new folder called `user` and move the `Create`, `Edit`, `List`, and `View` pages to this new folder.
2. Create a new file called `Index.vue` in the `user` folder and open it.
3. In the single file component `<template>` section, add a `router-view` component:

```
<template>
  <router-view/>
</template>
<script>
  export default {
    name: 'User',
  }
</script>
```

Changing the router files

We will create a new file that will manage the user's specific routes, which will help us to maintain the code and make it cleaner.

User routes

In the following steps, we will create routes for the user:

1. Create a new file called `user.js` in the `src/router` folder.
2. Import the `Index`, `List`, `View`, `Edit`, and `Create` views:

```
import Index from '@/views/user/Index.vue';
import List from '@/views/user/List.vue';
import View from '@/views/user/View.vue';
import Edit from '@/views/user/Edit.vue';
import Create from '@/views/user/Create.vue';
```

3. Create an array and make it the default export of the file. In this array, add a `route` object, with four properties – `path`, `name`, `component`, and `children`. Set the `path` property as `'/user'`, define the `name` property as `'user'`, define `component` as the imported `Index` component, and finally, define the `children` property as an empty array:

```
export default [
  {
    path: '/user',
    name: 'user',
    component: Index,
    children: [],
  },
]
```

4. In the `children` property, add a new route object with three properties – `path`, `name`, and `component`. Define `path` as `''`, `name` as `'list'`, and finally, define the `component` property as the imported `List` component:

```
{
  path: '',
  name: 'list',
  component: List,
},
```

5. Create a route object for the view route and use the same structure as the last `route` object. Define the `path` property as `':id'`, define `name` as `'view'`, and define `component` as the imported `View` component:

```
{
  path: ':id',
  name: 'view',
  component: View,
},
```

6. Create a route object for the `edit` route and use the same structure as the last `route` object. Define the `path` property as `'edit/:id'`, define `name` as `'edit'`, and define `component` as the imported `Edit` component:

```
{
  path: 'edit/:id',
  name: 'edit',
  component: Edit,
},
```

7. Create a route object for the `create` route, using the same structure as the last `route` object. Define the `path` property as `'create'`, define `name` as `'create'` , and define `component` as the imported `Create` component:

```
{
  path: 'create',
  name: 'create',
  component: Create,
},
```

Router manager

In the following steps, we will create the router manager that will control all the routes on the application:

1. Open the `index.js` in the `src/router` folder.
2. Import the newly created `user.js` file in the `src/router` folder:

```
import Vue from 'vue';
import VueRouter from 'vue-router';
import UserRoutes from './user';
```

3. In the `routes` array, add the imported `UserRoutes` as a destructed array:

```
const routes = [
  ...UserRoutes,
  {
    path: '*',
    redirect: '/user',
  },
];
```

How it works...

`vue-router` provides the ability to use child routes as internal components of a current view or layout. This gives the possibility to create an initial route with a special layout file, and render the child component inside this layout through the `RouterView` component.

This technique is commonly used for defining a layout in an application and setting a namespace for the modules where the parent route can have a set of specific orders that will be available for every one of its children.

See also

You can find more information about nested routes at `https://router.vuejs.org/guide/essentials/nested-routes.html#nested-routes`.

Creating a 404 error page

There will be some occasions when your user may try to enter an old link or enter a typo and won't get to the correct route, and this should lead them directly to a not found error.

In this recipe, you will learn how to handle a 404 error in `vue-router`.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we will use the Vue project with Vue-CLI that we used in the '*Creating a nested router view*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create http-project
```

Choose the manual features and add `Router` as a required feature, as indicated in the '*How to do it...*' section in the '*Creating a simple route*' recipe.

Our recipe will be divided into two parts:

- Creating the `NotFound` view
- Changing the router files

Let's get started.

Creating the NotFound view

We need to create a new view to be displayed for the user when there is no matching route on the application. This page will be a simple, generic page.

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these instructions to create the component correctly:

1. In the `src/views` folder, create a new file called `NotFound.vue` and open it.

2. Create a `VsRow` component and inside of it create four `VsCol` components. All of those components will have the attribute `vs-w` defined as `12` and `class` as `text-center`:

   ```
   <vs-row>
     <vs-col vs-w="12" class="text-center">
     <!-- Icon -->
     </vs-col>
     <vs-col vs-w="12" class="text-center">
     <!-- Title -->
     </vs-col>
     <vs-col vs-w="12" class="text-center">
     <!-- Text -->
     </vs-col>
     <vs-col vs-w="12" class="text-center">
     <!-- Button -->
     </vs-col>
   </vs-row>
   ```

3. On the first `VsCol` component, we will add a `VsIcon` component, and set the attribute icon as `sentiment_dissatisfied` and define the `size` as `large`:

   ```
   <vs-icon
     icon="sentiment_dissatisfied"
     size="large"
   />
   ```

4. In the second `VsCol` component, we will add a title for the page:

   ```
   <h1>Oops!</h1>
   ```

5. In the third `VsCol` component, we need to create the text that will be placed on the page:

   ```
   <h3>The page you are looking for are not here anymore...</h3>
   ```

6. Finally, on the fourth `VsCol` component, we will add the `VsButton` component. This button will have the attribute `type` defined as `relief` and `to` defined as `'/'`:

   ```
   <vs-button
     type="relief"
     to="/"
   >
     Back to Home...
   </vs-button>
   ```

Single file component <style> section

In this part, we will create the `<style>` section of the single file component. Follow these instructions to create the component correctly:

1. Add the `scoped` tag to the `<style>` tag:

   ```
   <style scoped>
   </style>
   ```

2. Create a new rule named `.text-center`, with the `text-align` property defined as `center` and `margin-bottom` defined as `20px;`:

   ```
   .text-center {
     text-align: center;
     margin-bottom: 20px;
   }
   ```

Changing the router files

After we have created the view, we need to add it to the router and make it available to the user. To do it, we will need to add the view route into the router manager.

In these steps, we will change the router manager, to add the new error page:

1. Open `index.js` in the `src/router` folder.
2. Import the `NotFound` component:

   ```
   import Vue from 'vue';
   import VueRouter from 'vue-router';
   import UserRoutes from './user';
   import NotFound from '@/views/NotFound';
   ```

3. In the `routes` array, after `UserRoutes`, add a new `route` object with two properties, `path` and `redirect`. Define the `path` property as `'/'` and the `redirect` property as `'/user'`:

   ```
   {
     path: '/',
     redirect: '/user'
   },
   ```

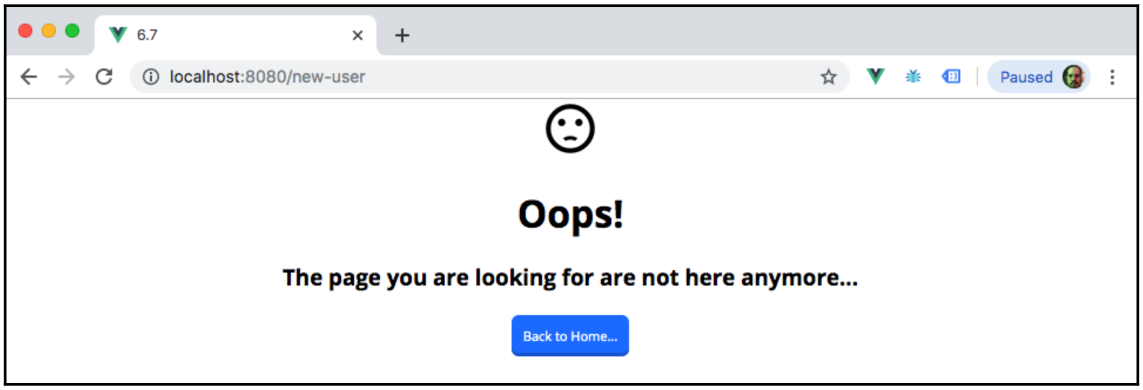
4. For the not found page, we need to create a new route object that needs to be placed in the last position in the `routes` array. This route object will have two properties, `path`, and `component`. The `path` property will be defined as `'*'` and `component` as the imported `NotFound` view:

```
{
  path: '*',
  component: NotFound,
},
```

To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



Oops!

The page you are looking for are not here anymore...

How it works...

`vue-router` tries to find the best match for the URL that the user wants to access; if there isn't a match, `vue-router` will use the `'*'` path as the default value for these scenarios, where the `*` represents any value that the user has entered that is not in the router lists.

Because the process of matching in `vue-router` is determined by the weight of the route, we need to place the error page at the very bottom, so `vue-router` needs to pass in every possible route before actually calling the `NotFound` route.

See also

You can find more information about handling 404 errors in the vue-router history mode at `https://router.vuejs.org/guide/essentials/history-mode.html#caveat`.

Creating and applying authentication middleware

In `vue-router`, it's possible to create router guards – functions that run each time a router is changed. Those guards are used as middleware in the router management process. It's common to use them as an authentication middleware or session validators.

In this recipe, we will learn how to create authentication middleware, add metadata to our routes to make them restricted, and create a login page.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we will use the Vue project with Vue-CLI that we used in the '*Creating a 404 error page*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create http-project
```

Choose the manual features and add `Router` as a required feature, as indicated in the '*How to do it...*' section in the '*Creating a simple route*' recipe.

Our recipe will be divided into three parts:

- Creating the authentication middleware
- Adding the metadata and the middleware to the router
- Attaching the middleware to vue-router and creating the login page

Let's get started.

Creating the login view

The login view will be the page that the user will see if they are not authenticated. We will construct a simple page with two inputs inside – a card and a button.

Single file component <script> section

In this part, we will create the `<script>` section of the single file component. Follow these instructions to create the component correctly:

1. In the `src/views` folder, create a new file called `Login.vue` and open it.
2. Create a `data` property, containing `username`, `password`, and `error`:

```
data: () => ({
  username: '',
  password: '',
  error: false,
}),
```

3. Then create the `methods` property with a method called `userSignIn`. This method will validate if the `username` and `password` data are complete. If it is, it will create a new key called `'auth'` in `sessionStorage`, with encrypted stringified JSON of the `username` data. Then, set `error` to `false` and execute `$router.replace` to redirect the user to the user list `'/user'`. If any of the fields do not pass in any of the validations, the method will define the error as `true` and return `false`:

```
methods: {
  userSignIn() {
    if (this.username && this.password) {
      window.sessionStorage.setItem('auth',
        window.btoa(JSON.stringify({
            username: this.username
          })
        )
      );
      this.error = false;
      this.$router.replace('/user');
    }
    this.error = true;
    return false;
  },
}
```

Single file component <template> section

In this part, we will create the `<template>` section of the single file component. Follow these instructions to create the component correctly:

1. Create a `div.container` HTML element with a `VsRow` component inside. The `VsRow` component will have the attribute `vs-align` defined as `"center"` and `vs-justify` defined as `"center"`:

```
<div class="container">
  <vs-row
    vs-align="center"
    vs-justify="center"
  >
  </vs-row>
</div>
```

2. Inside the `VsRow` component, add a `VsCol` component with the attribute `vs-lg` defined as `4`, `vs-sm` defined as `6`, and `vs-xs` defined as `10`. Then, inside the `VsCol` component, we will create a `VsCard` component with the `style` attribute defined as `margin: 20px;`:

```
<vs-col
  vs-lg="4"
  vs-sm="6"
  vs-xs="10"
>
  <vs-card
    style="margin: 20px;"
  >
  </vs-card>
</vs-col>
```

3. Inside the `VsCard` component, create a dynamic `<template>` with the `slot` named `header`, an `h3` HTML element, and your title:

   ```
   <template slot="header">
     <h3>
       User Login
     </h3>
   </template>
   ```

4. After that, create a `VsRow` component with the attribute `vs-align` defined as `"center"`, `vs-justify` defined as `"center"`, and two `VsCol` components inside of it, with the attribute `vs-w` defined as `12`:

   ```
   <vs-row
     vs-align="center"
     vs-justify="center"
   >
     <vs-col vs-w="12">
     </vs-col>
     <vs-col vs-w="12">
     </vs-col>
   </vs-row>
   ```

5. On the first `VsCol` component, we will add a `VsInput` component, with the attribute `danger` defined as the data `error` value, `danger-text` defined as the text that will display on error, `label` defined as `"Username"`, `placeholder` defined as `"Username or e-mail"`, and the `v-model` directive bound to `username`:

   ```
   <vs-input
     :danger="error"
     danger-text="Check your username or email"
     label="Username"
     placeholder="Username or e-mail"
     v-model="username"
   />
   ```

6. In the second `VsCol` component, we will add a `VsInput` component, with the attribute `danger` defined as the data `error` value, `danger-text` defined as the text that will display on error, `label` defined as `"Password"`, `type` defined as `password`, `placeholder` defined as `"Your password"`, and the `v-model` directive bound to `password`:

   ```
   <vs-input
     :danger="error"
     label="Password"
   ```

```
    type="password"
    danger-text="Check your password"
    placeholder="Your password"
    v-model="password"
  />
```

7. Finally, in the card footer, we need to create a dynamic `<template>` with the slot named `footer`. Inside this `<template>`, we will add a `VsRow` component with the `vs-justify` attribute defined as `flex-start` and insert a `VsButton` with the attribute `color` defined as `success`, `type` defined as `filled`, `icon` defined as `account_circle`, `size` defined as `small` and the `@click` event listener targeted to the `userSignIn` method:

```
<template slot="footer">
  <vs-row vs-justify="flex-start">
    <vs-button
      color="success"
      type="filled"
      icon="account_circle"
      size="small"
      @click="userSignIn"
    >
      Sign-in
    </vs-button>
  </vs-row>
</template>
```

Single file component <style> section

In this part, we will create the `<style>` section of the single file component. Follow these instructions to create the component correctly:

1. First, we need to make this section scoped, so the CSS rules won't affect any other component of the application:

```
<style scoped></style>
```

2. Then, we need to add the rules for the `container` class and the `VsInput` component:

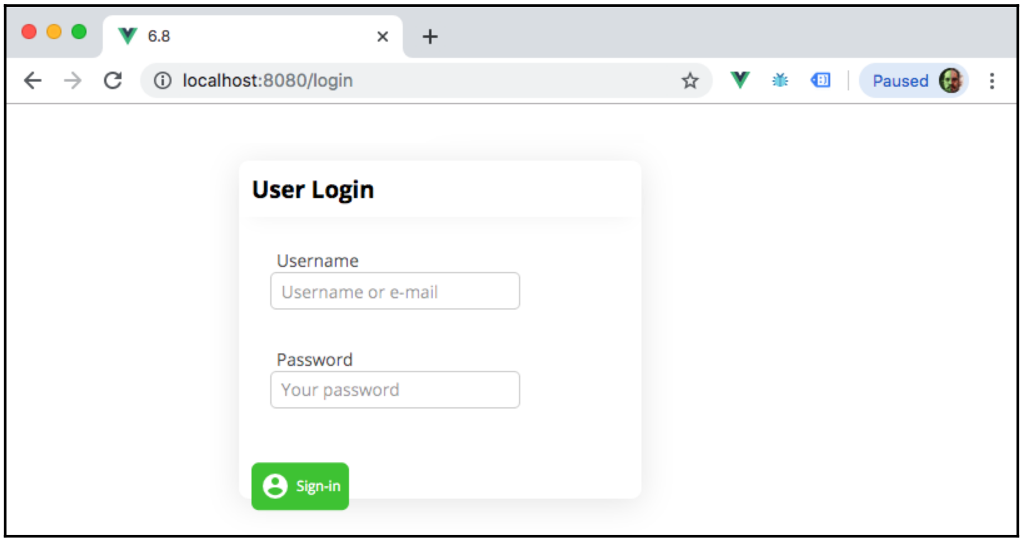
```
<style scoped>
  .container {
    height: 100vh;
    display: flex;
    flex-wrap: wrap;
    justify-content: center;
```

```
      align-content: center;
    }

    .vs-input {
      margin: 5px;
    }
  </style>
```

To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



Creating the middleware

All `vue-router` middleware can also be referred to as navigation guards, and they can be attached to the application route changes. Those changes have some hooks that you can apply to your middleware. The authentication middleware takes place before the router changes, so we can handle everything and send the user to the correct route.

1. In the `src/router` folder, create a new folder called `middleware`, then create and open a new file called `authentication.js`.

2. In this file, we will create a default `export` function that will have three function parameters – `to`, `from`, and `next`. The `to` and `from` parameters are objects, and the `next` parameter is a callback function:

   ```
   export default (to, from, next) => {
   };
   ```

3. We need to check whether the route that we are being redirected to has an authenticated `meta` property set to `true` and whether we have a `sessionStorage` item with the `'auth'` key. If we pass those validations, we can execute the `next` callback:

   ```
   if (to.meta.authenticated && sessionStorage.getItem('auth')) {
     return next();
   }
   ```

4. Then, if the first validation didn't pass, we need to check whether the router that we are redirecting the user to has the authenticated `meta` property and check whether it's a `false` value. If the validation did pass, we will execute the `next` callback:

   ```
   if (!to.meta.authenticated) {
     return next();
   }
   ```

5. Finally, if none of our validations pass, execute the `next` callback, passing `'/login'` as an argument:

   ```
   next('/login');
   ```

Adding the metadata and the middleware to the router

After creating our middleware, we need to define which routes will be authenticated and which routes won't. Then we have to import the middleware to the router and define it when it is executed:

1. Open `user.js` in the `src/router` folder.
2. In each `route` object, add a new property called `meta`. This property will be an object with an authenticated `key` and a `value` defined as `true`. We need to do this to every route – even the children's routes:

   ```
   import Index from '@/views/user/Index.vue';
   import List from '@/views/user/List.vue';
   import View from '@/views/user/View.vue';
   ```

```
import Edit from '@/views/user/Edit.vue';
import Create from '@/views/user/Create.vue';

export default [
  {
    path: '/user',
    name: 'user',
    component: Index,
    meta: {
      authenticated: true,
    },
    children: [
      {
        path: '',
        name: 'list',
        component: List,
        meta: {
          authenticated: true,
        },
      },
      {
        path: ':id',
        name: 'view',
        component: View,
        meta: {
          authenticated: true,
        },
      },
      {
        path: 'edit/:id',
        name: 'edit',
        component: Edit,
        meta: {
          authenticated: true,
        },
      },
      {
        path: 'create',
        name: 'create',
        component: Create,
        meta: {
          authenticated: true,
        },
      },
    ],
  },
]
```

3. Open `index.js` in the `src/router` folder.
4. Import the newly created middleware and the `Login` view component:

```
import Vue from 'vue';
import VueRouter from 'vue-router';
import UserRoutes from './user';
import NotFound from '@/views/NotFound';
import Login from '@/views/Login';
import AuthenticationMiddleware from './middleware/authentication';
```

5. Create a new `route` object for the login page view. This route object will have `path` set to `'/login'`, `name` defined as `'login'`, `component` defined as `Login`, and the `meta` property will have the `authenticated` key with the value set to `false`:

```
{
  path: '/login',
  name: 'login',
  component: Login,
  meta: {
    authenticated: false,
  },
},
```

6. On the error handling route, we'll define the `meta` property `authenticated` as `false` because the login view is a public route:

```
{
  path: '*',
  component: NotFound,
  meta: {
    authenticated: false,
  },
},
```

7. Finally, after the creation of the `router` constructor, we need to inject the middleware in the `beforeEach` execution:

```
router.beforeEach(AuthenticationMiddleware);
```

How it works...

Router guards work as middleware; they have a hook that is executed in each life cycle of the `vue-router` process. For the purposes of this recipe, we chose the `beforeEach` hook to add our middleware.

In this hook, we checked whether the user was authenticated and whether the user needed authentication to navigate the route or not. After checking these variables, we continued the process by sending the user to the route they needed.

See also

You can find more information about vue-router router guards at `https://router.vuejs.org/guide/advanced/navigation-guards.html#global-before-guards`.

Lazy loading your pages asynchronously

Components can be loaded when needed, and so can routes. Using lazy loading techniques with `vue-router` allows more code-splitting and smaller final bundles in your application.

In this recipe, we will learn how to transform routes in order to load them asynchronously.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we will use the Vue project with Vue-CLI that we used in the '*Creating an authentication middleware*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create http-project
```

Choose the manual features and add `Router` as a required feature, as indicated in the '*How to do it...*' section in the '*Creating a simple route*' recipe.

Our recipe will be divided into two parts:

- Updating the router manager
- Updating the user routes

Let's get started.

Updating the router manager

To update the router manager, follow these instructions:

1. Open the `index.js` file in the `src/router` folder.
2. In each route that has a `component` property, we will transform the direct attribution of the component to a new function. This will be an arrow function returning the `import()` method of webpack:

```
{
  path: '/login',
  name: 'login',
  component: () => import('@/views/Login'),
  meta: {
    authenticated: false,
  },
},
```

3. Repeat the process on each one of the `route` objects that has a `component` property.

Updating the user routes

To update the user routes, follow these instructions:

1. Open the `user.js` file in the `src/router` folder.
2. In each route that has a `component` property, we will transform the direct attribution of the component to a new function. This will be an arrow function returning the `import()` method of webpack:

```
{
  path: '/user',
  name: 'user',
  component: () => import('@/views/user/Index.vue'),
  meta: {
    authenticated: true,
  },
  children: [],
},
```

3. Repeat the process on each one of the `route` objects that has a `component` property.

How it works...

In ECMAScript, `export` and `import` are objects with predefined values when we use the `export default` method. This means that when we `import` a new component, this component is already being pointed to the `default export` of that file.

To carry out the lazy loading process, we need to pass a function that will be executed at runtime, and the return of that function will be the part of the code that webpack divides in the bundling process.

When we call this function in `vue-router`, instead of the direct component import, `vue-router` does a validation check that the present component import is a function and needs to be executed. After the execution of the function, the response is used as the component that will be displayed on the user's screen.

Because the webpack `import()` method is asynchronous, this process can happen alongside other code execution, without tempering or blocking the main thread of the JavaScript VM.

See also

You can find more information about `vue-router` lazy loading at `https://router.vuejs.org/guide/advanced/lazy-loading.html`.

You can find more information about `webpack` code-splitting at `https://webpack.js.org/guides/code-splitting/`.

You can find more information about the ECMAScript dynamic import proposal at `https://github.com/tc39/proposal-dynamic-import`.

7
Managing the Application State with Vuex

Transferring data between sibling components can be very easy, but imagine making a tree of components react to any data change. You will need to trigger an event in an event bus or send the event through all the parent components until it reaches over the top of the event chain and then gets sent all the way down to the desired component; this process can be very tedious and painful. If you are developing a large-scale application, this process is not sustainable.

Flux libraries were developed to help with this process, with the idea of bringing the reactivity outside of the component bounds, as Vuex is capable of maintaining one single source of truth of your data and, at the same time, is the place for you to have your business rules.

In this chapter, we will learn how to use Vuex, develop our store, apply it to our components, and namespace it so we can have different modules of Vuex inside the same store.

In this chapter, we'll cover the following recipes:

- Creating a simple Vuex store
- Creating and understanding the Vuex state
- Creating and understanding the Vuex mutations
- Creating and understanding the Vuex actions
- Creating and understanding the Vuex getters
- Creating a dynamic component with Vuex
- Adding hot module reload for development
- Creating a Vuex module

Technical requirements

In this chapter, we will be using **Node.js** and **Vue-CLI**.

Attention, Windows users, you need to install an NPM package called `windows-build-tools`, to be able to install the following required packages. To do it, open PowerShell as administrator and execute the following command:

```
> npm install -g windows-build-tools
```

To install Vue-CLI, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install -g @vue/cli @vue/cli-service-global
```

Creating a simple Vuex store

Creating a single source of truth in your application gives you the power to simplify the flow of your data, enabling the reactivity of the data to flow into another perspective, where you are not tied to a parent-child relationship anymore. The data can now be stored in a single place and everyone can fetch or request data.

In this recipe, we will learn how to install the Vuex library and create our first single store, and how we can manipulate it with reactive actions and data getters.

Getting ready

The prerequisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To create a Vue-CLI project, follow these steps:

1. We need to open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create initial-vuex
```

2. The CLI will ask some questions that will help with the creation of the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *Spacebar* to select an option.

3. There are two methods for starting a new project. The default method is a basic `babel` and `eslint` project without any plugin or configuration, and the `Manually` mode, where you can select more modes, plugins, linters, and options. We will go for `Manually`:

```
? Please pick a preset: (Use arrow keys)
  default (babel, eslint)
❯ Manually select features
```

4. Now we are asked about the features that we will want in the project. Those features are some Vue plugins such as Vuex or Router (`Vue-Router`), testers, linters, and more. Select `Babel`, `Router`, `Vuex`, and `Linter / Formatter`:

```
? Check the features needed for your project: (Use arrow keys)
❯ Babel
  TypeScript
  Progressive Web App (PWA) Support
❯ Router
❯ Vuex
  CSS Pre-processors
❯ Linter / Formatter
  Unit Testing
  E2E Testing
```

5. Continue this process by selecting a linter and formatter. In our case, we will select the `ESLint + Airbnb` config:

```
? Pick a linter / formatter config: (Use arrow keys)
  ESLint with error prevention only
❯ ESLint + Airbnb config
  ESLint + Standard config
  ESLint + Prettier
```

6. After the linting rules are set, we need to define when they are applied to your code. They can be either applied `on save` or fixed `on commit`:

   ```
   ? Pick additional lint features: (Use arrow keys)
     Lint on save
   ❯ Lint and fix on commit
   ```

7. After all those plugins, linters, and processors are defined, we need to choose where the settings and configs are stored. The best place to store them is on a dedicated file, but it is also possible to store them in the `package.json` file:

   ```
   ? Where do you prefer placing config for Babel, ESLint, etc.? (Use
     arrow keys)
   ❯ In dedicated config files
     In package.json
   ```

8. Now you can choose if you want to make this selection a preset for future projects, so you don't need to reselect everything again:

   ```
   ? Save this as a preset for future projects? (y/N) n
   ```

Our recipe will be divided into two parts:

- Creating the store
- Creating the reactive component with Vuex

Let's get started.

Creating the store

Now you have the project with the Vuex library, and we need to create our first store. In the following steps, we will create the Vuex store:

1. Open the `index.js` from the `src/store` folder.
2. In the `state` property, add a new key called `counter` and set the value to `0`:

   ```
   state: {
     counter: 0,
   },
   ```

3. In the `mutations` property, add two new functions, `increment` and `decrement`. Both of the functions will have a `state` argument, which is the current Vuex `state` object. The `increment` function will increment the `counter` by `1` and the `decrement` function will decrement the `counter` by `1`:

```
mutations: {
  increment: (state) => {
    state.counter += 1;
  },
  decrement: (state) => {
    state.counter -= 1;
  },
},
```

4. Finally, in the `actions` property, add two new functions, `increment` and `decrement`. Both of the functions will have a deconstructed argument, `commit`, which is a function to call the Vuex mutation. In each function, we will execute the `commit` function, passing as a parameter the name of the current function as a string:

```
actions: {
 increment({ commit }) {
 commit('increment');
 },
 decrement({ commit }) {
 commit('decrement');
 },
},
```

Creating the reactive component with Vuex

Now that you have your Vuex store defined, you need to interact with it. We will create a reactive component that will display the current state `counter` on the screen, and show two buttons, one for incrementing the `counter`, and another for decrementing the `counter`.

Single file component <script> section

Here we are going to write the `<script>` section of the single file component:

1. Open the `App.vue` file from the `src` folder.
2. Create the `<script>` section in the file, with an `export default` object:

```
<script>
  export default {};
</script>
```

3. In the newly created object, add the Vue `computed` property with a property called `counter`. In this property we need to return the current `$store.state.counter`:

```
computed: {
  counter() {
    return this.$store.state.counter;
  },
},
```

4. Finally, create a Vue `methods` property with two functions, `increment` and `decrement`. Both of the functions will execute a `$store.dispatch` with a parameter being the function name as a string:

```
methods: {
  increment() {
    this.$store.dispatch('increment');
  },
  decrement() {
    this.$store.dispatch('decrement');
  },
},
```

Single file component <template> section

Let's code the `<template>` section of the single file component:

1. Open the `App.vue` file in the `src` folder.
2. In the `<template>` section, remove everything inside the `div#app`.
3. Create an `h1` HTML element with the counter variable inside of it.
4. Create a button with an event listener on the `@click` directive that calls the `increment` function, and have + as a label:

```
<button @click="increment">+</button>
```

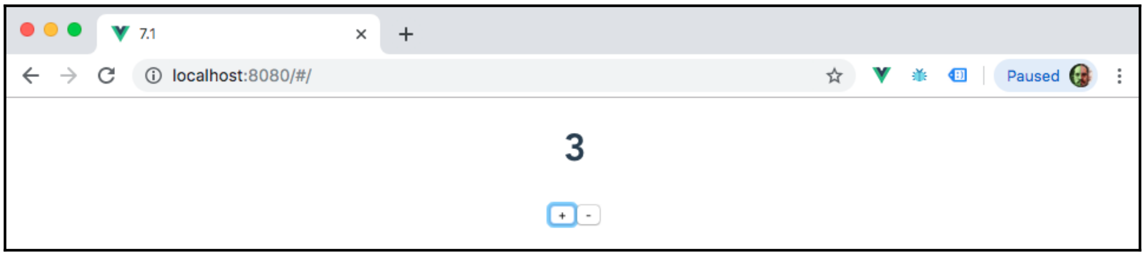
5. Create a button with an event listener on the `@click` directive that calls the `decrement` function, and `-` as a label:

   ```
   <button @click="decrement">-</button>
   ```

To run the server and see your component, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run serve
```

Here is your component rendered and running:



How it works...

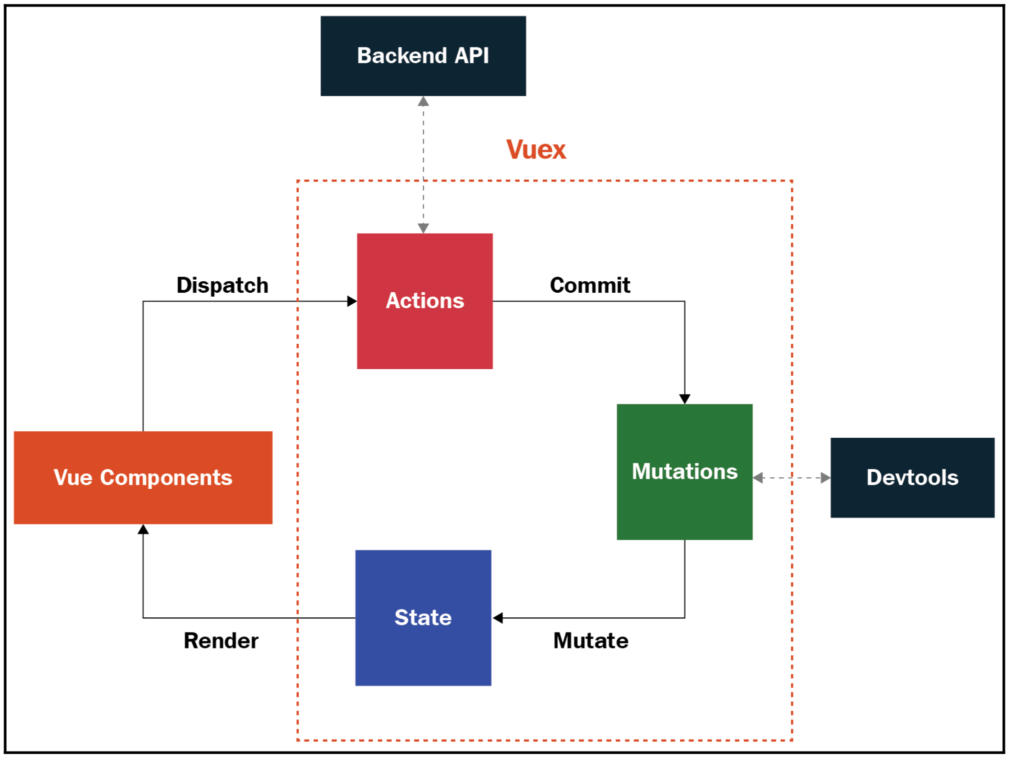
When you declare your Vuex store, you need to create three main properties, `state`, `mutations`, and `actions`. Those properties act as a single structure, bounded to the Vue application through the `$store` injected prototype or the exported `store` variable.

A `state` is a centralized object that holds your information and makes it available to be used by the `mutation`, `actions`, or the components. Changing the `state` always requires a synchronous function executed through a `mutation`.

A `mutation` is a synchronous function that can change the `state` and is traceable, so when developing, you can time travel through all the executed `mutations` in the Vuex store.

An `action` is an asynchronous function, which can be used to hold business logic, API calls, dispatch other `actions`, and execute `mutations`. Those functions are the common entrance point of any change in a Vuex store.

A simple representation of a Vuex store can be seen in this chart:



See also

You can find more information about Vuex at `https://vuex.vuejs.org/`.

Creating and understanding the Vuex state

The Vuex state can seem straightforward to understand. However, as the data gets more in-depth and nested, its complexity and maintainability can get more complicated.

In this recipe, we will learn how to create a Vuex state that can be used in the scenarios of both a **Progressive Web Application (PWA)**/ **Single Page Application (SPA)** and a **Server Side Rendering (SSR)**, without any problems.

Getting ready

The prerequisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we will use the Vue project with Vue-CLI that we used in the '*Lazy Loading your pages asynchronously*' recipe in `Chapter 6`, *Managing Routes with vue-router*, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create vuex-store
```

Choose the manual features, add `Router` and `Vuex` as required features, as indicated in the '*How to do it...*' section of the '*Creating a simple Vuex store*' recipe.

Our recipe will be divided into two parts:

- Adding Vuex via the `vue ui`
- Creating the `Vuex` state

Let's get started.

Adding Vuex via the vue ui

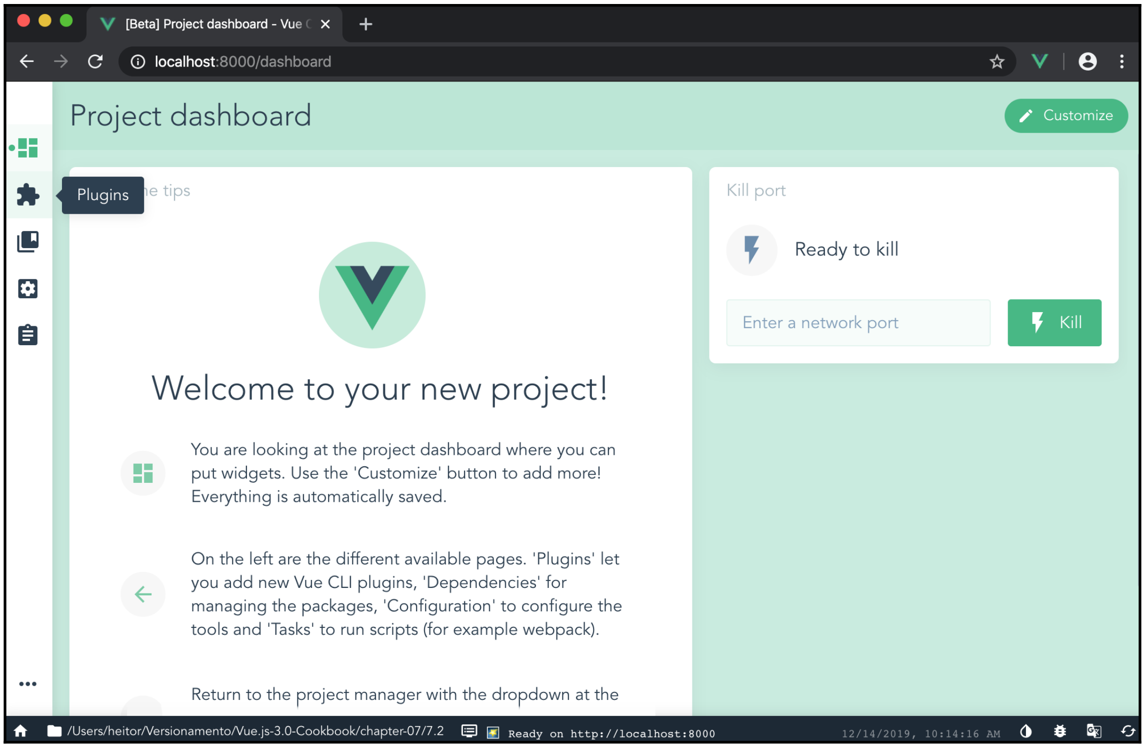
When importing an old project that was created via the Vue-CLI, it is possible to automatically add Vuex through the `vue ui` interface without any effort at all. We will learn how to add the Vuex library to the old project, so we can continue developing the recipe.

In the following steps, we will add the Vuex with the `vue ui` interface:

1. In the project folder, open the `vue ui` by executing the following command on Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows):

```
> vue ui
```

2. Select the correct project that you are working on. In the right sidebar, click on the **Plugins** menu icon:



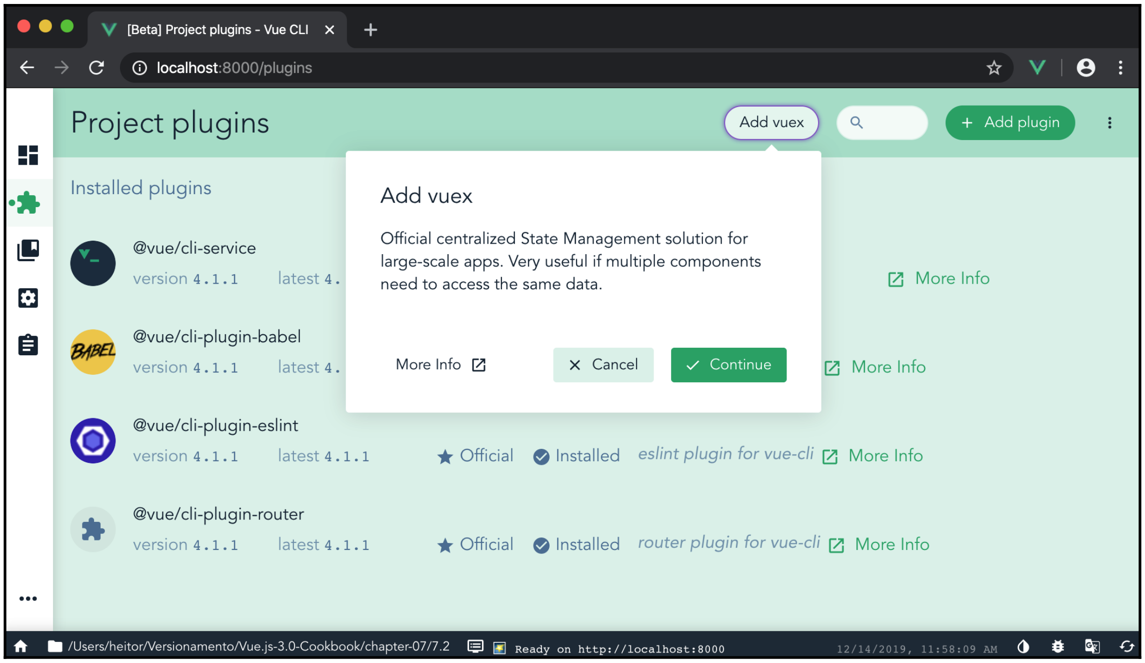
Welcome to your new project!

You are looking at the project dashboard where you can put widgets. Use the 'Customize' button to add more! Everything is automatically saved.

On the left are the different available pages. 'Plugins' let you add new Vue CLI plugins, 'Dependencies' for managing the packages, 'Configuration' to configure the tools and 'Tasks' to run scripts (for example webpack).

Return to the project manager with the dropdown at the

3. On the plugins page, on the top bar, click on the **Add vuex** button. This will trigger a pop-up modal, then click on the **Continue** button to finish the installation of Vuex on the application:



Add vuex

Official centralized State Management solution for large-scale apps. Very useful if multiple components need to access the same data.

More Info | Cancel | Continue

@vue/cli-service version 4.1.1 latest 4. More Info

@vue/cli-plugin-babel version 4.1.1 latest 4. More Info

@vue/cli-plugin-eslint version 4.1.1 latest 4.1.1 Official Installed *eslint plugin for vue-cli* More Info

@vue/cli-plugin-router version 4.1.1 latest 4.1.1 Official Installed *router plugin for vue-cli* More Info

/Users/heitor/Versionamento/Vue.js-3.0-Cookbook/chapter-07/7.2 Ready on http://localhost:8000 12/14/2019, 11:58:09 AM

4. Adding the Vuex to our application will change the structure of the application. First, we will notice that there is a new folder called `store` in the `src` folder, and in the `main.js` file, it was added to the imports and the injection of the `store` in the Vue application:

```
import './server/server';
import Vue from 'vue';
import App from './App.vue';
import Vuesax from 'vuesax';
import './style.css';
import router from './router'
import store from './store'

Vue.use(Vuesax);

Vue.config.productionTip = false;

new Vue({
  router,
  store,
  render: h => h(App)
}).$mount('#app');
```

Creating the Vuex state

In order to save the data inside of Vuex, you need to have an initial state that is loaded with the application and defined as the default one when the user enters your application. Here, we are going to learn how to create the Vuex state and use it as a singleton, so that Vuex can be used in an SPA and an SSR page:

Now we will create a Vuex store that can be used in an SSR and an SPA:

1. In the `src/store` folder, create a new folder called `user`, and inside this folder create a new file named `state.js`.
2. Create a new `generateState` function. This function will return a JavaScript object, with three main properties, `data`, `loading`, and `error`. The `data` property will be a JavaScript object, with a property called `usersList` defined as an empty array as default, and a property called `userData` with the default object of a user. The `loading` property will be a boolean, set to `false` by default, and `error` will have a default value initializing to `null`:

```
const generateState = () => ({
  data: {
    usersList: [],
    userData: {
      name: '',
      email: '',
      birthday: '',
      country: '',
      phone: '',
    },
  },
  loading: false,
  error: null,
});
```

3. After creating the function, we will create an `export default` object at the end of the file, which will be a JavaScript object, and we will destruct the return of the `generateState` function:

```
export default { ...generateState() };
```

4. Create a new file named `index.js` in the `user` folder and open it.
5. Import the newly created `state`:

```
import state from './state';
```

6. At the end of the file, create an `export default` file as a JavaScript object. In this object, we will add the imported `state`:

```
export default {
  state,
};
```

7. Open the `index.js` file from the `src/store` folder.
8. Import the `index.js` file from the `user` folder:

```
import Vue from 'vue';
import Vuex from 'vuex';
import UserStore from './user';
```

9. In the `export default` function, which creates a new Vuex store, we will remove all the properties inside of it, and put the imported `UserStore` deconstructed object inside the `Vuex.Store` parameter:

```
export default new Vuex.Store({
  ...UserStore,
})
```

How it works...

When using the `vue ui` to add Vuex as a plugin, the `vue ui` will automatically add the required files, and import everything that is needed. This is the initial phase of the creation of a Vuex `store`.

First is the creation of an exclusive file for managing the `state` that we can use to separate, from the `store`, the process of how the state begins and how it can be initialized.

In this case of this `state`, we used a function to generate a completely new `state` every time it's called. This is a good practice, because in an SSR environment, the `state` of the server will always be the same, and we need to create a new `state` for each new connection.

After the creation of the `state`, we needed to create the default file for exporting the Vuex files that will be created in the `user` folder. This file is a simple import of all the files that will be created in the folder, `state`, `actions`, `mutation`, and `getters`. After the import, we export an object with the name of the required Vuex properties, `state`, `actions`, `mutations`, and `getters`.

Finally, in the Vuex `store`, we import the file that aggregates everything and deconstructs it into our store to initialize it.

There's more...

The `Vuex` state is a single source of truth in your application, it works like a global data manager, and it should not be changed directly. This is because we need to prevent the mutation of data with a concurrent mutation of the same data. To avoid that, we always need to change our state through the mutations, because the functions are synchronous and controlled by Vuex.

See also

Find more information about the Vuex state at `https://vuex.vuejs.org/guide/state.html`.

Creating and understanding the Vuex mutations

When there is a change in Vuex, we need a way to execute this change in asynchronous form and keep track of it so it won't execute over another change before the first change finishes.

For this case, we need the mutations, which are functions that are only responsible for changing the state of your application.

In this recipe, we will learn how to create Vuex mutations and the best practices by which to do it.

Getting ready

The prerequisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we will use the Vue project with Vue-CLI that we used in the '*Creating and understanding the Vuex state*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create vuex-store
```

Choose the manual features, add `Router` and `Vuex` as required features, as indicated in the '*How to do it...*' section of the '*Creating a simple Vuex store*' recipe.

Now we create a Vuex mutation and base type for the mutations:

1. Create a new file called `types.js` in the `user` folder inside the `src/store` folder, and open it.
2. In this file, we will create an `export default` JavaScript object, with a group of keys that will be the names of our mutations. Those keys will be `LOADING`, `ERROR`, `SET_USER_LIST`, `SET_USER_DATA`, `UPDATE_USER`, and `REMOVE_USER`:

```
export default {
  LOADING: 'LOADING',
  ERROR: 'ERROR',
  SET_USER_LIST: 'SET_USER_LIST',
  SET_USER_DATA: 'SET_USER_DATA',
  UPDATE_USER: 'UPDATE_USER',
  REMOVE_USER: 'REMOVE_USER',
}
```

3. Create a new file called `mutations.js` in the `user` folder, and open it.
4. Import the newly created `types.js` file:

```
import MT from './types';
```

5. Create a new function called `setLoading`, which will receive the Vuex `state` as an argument and will define the loading property of the state to `true` when executed:

   ```
   const setLoading = state => {
     state.loading = true;
   };
   ```

6. Create a new function called `setError`, which will receive the Vuex `state` as an argument and `payload`. This function will set the `loading` property of the `state` to `false`, and the `error` property to the received `payload` argument:

   ```
   const setError = (state, payload) => {
     state.loading = false;
     state.error = payload;
   };
   ```

7. Create a new function called `setUserList`, which will receive the Vuex `state` and `payload` as an argument. This function will define the `usersList` property of the `state.data` to the received `payload` argument, set the `loading` property of the `state` to `false`, and the `error` property to `null`:

   ```
   const setUserList = (state, payload) => {
     state.data.usersList = payload;
     state.loading = false;
     state.error = null;
   };
   ```

8. Create a new function called `setUserData`, which will receive the Vuex `state` and `payload` as arguments. This function will define the `userData` property of the `state.data` to the received `payload` argument, set the `loading` property of the `state` to `false`, and the `error` property to `null`:

   ```
   const setUserData = (state, payload) => {
     state.data.userData = payload;
     state.loading = false;
     state.error = null;
   };
   ```

9. Create a new function called `updateUser`, which will receive the Vuex `state` and `payload` as an argument. This function will update the user data in the `usersList` property of the `state.data`, define the `loading` property of the `state` to `false`, and the `error` property to `null`:

```
const updateUser = (state, payload) => {
  const userIndex = state.data.usersList.findIndex(u => u.id ===
     payload.id);
  if (userIndex > -1) {
    state.data.usersList[userIndex] = payload;
  }
  state.loading = false;
  state.error = null;
};
```

10. Create a new function called `removeUser`, which will receive the Vuex `state` and `payload` as an argument. This function will remove the user data from the `usersList` property of the `state.data`, define the `loading` property of the `state` to `false`, and the `error` property to `null`:

```
const removeUser = (state, payload) => {
  const userIndex = state.data.usersList.findIndex(u => u.id ===
     payload);
  if (userIndex > -1) {
    state.data.usersList.splice(userIndex, 1);
  }
  state.loading = false;
  state.error = null;
};
```

11. Finally, create an `export default` object, with the keys being the types we created in the `types.js` file, and define each of the keys to the functions we created:

```
export default {
  [MT.LOADING]: setLoading,
  [MT.ERROR]: setError,
  [MT.SET_USER_LIST]: setUserList,
  [MT.SET_USER_DATA]: setUserData,
  [MT.UPDATE_USER]: updateUser,
  [MT.REMOVE_USER]: removeUser,
}
```

12. Open the `index.js` file in the `user` folder.
13. Import the newly created `mutations.js` file, and add it to the `export default` JavaScript object:

```
import state from './state';
import mutations from './mutations';

export default {
  state,
  mutations,
};
```

How it works...

Each `mutation` is a function that will be called as a `commit`, and will have an *identifier* in the Vuex store. This identifier is the `mutation` key in the exported JavaScript object. In this recipe, we created a file that holds all the identifiers as an object value so that it can be used as a constant inside our code.

This pattern helps us in the development of Vuex `actions`, which need to know each `mutation` name.

When exporting the `mutation` JavaScript object, we use the constant as the key and the corresponding function as its value, so the Vuex store can execute the correct function when called.

See also

Find more information about Vuex mutations at `https://vuex.vuejs.org/guide/mutations.html`.

Creating and understanding the Vuex getters

Accessing data from `Vuex` can be done through the state itself, which can be very dangerous, or via the getters. The getters are like data that can be preprocessed and delivered without touching or messing with the Vuex store state.

The whole idea behind getters is the possibility to write custom functions that can extract data from your state in a single place when you need it, so that you get just the data you need.

In this recipe, we will learn how to create a Vuex getter and a dynamic getter that can be used as a high-order function.

Getting ready

The prerequisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we will use the Vue project with Vue-CLI that we used in the 'Creating and understanding the Vuex mutations' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or
Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create vuex-store
```

Choose the manual features, and add Router and `Vuex` as needed features, as indicated in the 'How to do it...' section of the 'Creating a simple Vuex store' recipe.

In the following steps, we will create the Vuex getters:

1. Create a new file called `getters.js` in the `src/store/user` folder.
2. Create a new function called `getUsersList`, and return the `state.data.usersList` property:

```
function getUsersList(state) {
  return state.data.usersList;
}
```

In a `getter` function, the first argument that the function will receive will be always the current `state` of the Vuex `store`.

3. Create a new function called `getUserData`, and return the `state.data.userData` property:

```
function getUserData(state) {
  return state.data.userData;
}
```

4. Create a new function called `getUserById`, and return another function that receives `userId` as an argument. This returning function will return the result of a search of `state.data.usersList` that matches the same `id` as the `userId` received:

```
function getUserById(state) {
  return (userId) => {
    return state.data.usersList.find(u => u.id === userId);
  }
}
```

5. Create a new function called `isLoading`, and return the `state.loading` property:

```
function isLoading(state) {
  return state.loading;
}
```

6. Create a new function called `hasError`, and return the `state.error` property:

```
function hasError(state) {
  return state.error;
}
```

7. Finally, create an `export default` JavaScript object, with all the created functions as properties:

```
export default {
  getUsersList,
  getUserData,
  getUserById,
  isLoading,
  hasError,
};
```

8. Open the `index.js` file in the `src/store/user` folder.
9. Import the newly created `getters.js` file, and add it to the export default JavaScript object:

```
import state from './state';
import mutations from './mutations';
import getters from './getters';

export default {
  state,
  mutations,
  getters,
};
```

How it works...

Getters are like a GET function from an object and are static cached functions – they only change the returned value when the `state` has changed. But if you add the return as a high-order function, you can give it more power to use a more sophisticated algorithm and provide specific data.

In this recipe, we created two types of getters: the most basic, with simple data return, and the high-order function, which needs to be called as a function to retrieve the value you want.

There's more...

Using getters with business logic is a good way to gather more data on the state. This is a good pattern because, on larger projects, it helps other developers to understand more what is going on in each of the GET functions and how it works behind the curtain.

You always need to remember that getters are synchronous functions and reactive to the state change, so the data on the getters is memoized and cached until the single source of truth receives a commit and changes it.

See also

You can find more information about Vuex getters at `https://vuex.vuejs.org/guide/getters.html`.

Creating and understanding the Vuex actions

You have all your state ready, your dataset, and now you need to fetch new data from an outside source or change this data inside your application. Here comes the part where actions do their job.

Actions are responsible for orchestrating the process in this communication between the application and the outside world. Controlling when the data need to be mutated on the state and returned to the caller of the action.

Usually, the action is a dispatch through a component or a view, but there are some occasions where actions can dispatch another action to create a chain of actions in your application.

In this recipe, we will learn how to create the actions needed in our application to define a user's list, update a user, and remove a user.

Getting ready

The prerequisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we will use the Vue project with Vue-CLI that we used in the '*Creating and understanding the Vuex getters*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create vuex-store
```

Choose the manual features, and add `Router` and `Vuex` as required features, as indicated in the '*How to do it...* section of the '*Creating a simple Vuex store*' recipe.

Now follow these steps to create the Vuex actions:

1. Create a new file called `actions.js` in the `src/store/user` folder.
2. Import the mutation types files (`types.js`), and the `getHttp`, `patchHttp`, `postHttp`, and `deleteHttp` functions from the `fetchApi` wrapper:

```
import {
  getHttp,
  patchHttp,
  deleteHttp,
  postHttp,
} from '@/http/fetchApi';
import MT from './types';
```

3. Create a new `asynchronous` function called `createUser`, which receives as the first argument a deconstructed JavaScript object, with the `commit` property, and `userData` as the second argument, which will be used to create the user. Add a `try/catch` statement, in the `try` context. First, we execute `commit(MT.LOADING)`, then we fetch the users lists from the API, and finally, `commit(MT.SET_USER_DATA, data)`, passing the users lists to be mutated. If we receive an exception and get into the `Catch` statement, we will execute `commit(MT.ERROR, error)`, passing the error that we receive to the `state`:

```
async function createUser({ commit }, userData) {
  try {
    commit(MT.LOADING);
    await postHttp(`/api/users`, {
      data: {
        ...userData,
      }
    });
    commit(MT.SET_USER_DATA, userData);
  } catch (error) {
    commit(MT.ERROR, error);
  }
}
```

4. Create a new `asynchronous` function called `fetchUsersList`, which receives as the first argument a deconstructed JavaScript object, with the `commit` property. Add a `try/catch` statement in the `try` context. We execute `commit(MT.LOADING)`, then we fetch the users lists from the API, and finally, `commit(MT.SET_USER_LIST, data)`, passing the users lists to be mutated. If we receive an exception and get into the `catch` statement, we will execute a mutation of `commit(MT.ERROR, error)`, passing the error that we receive to the `state`:

```
async function fetchUsersList({ commit }) {
  try {
    commit(MT.LOADING);
    const { data } = await getHttp(`api/users`);
    commit(MT.SET_USER_LIST, data);
  } catch (error) {
    commit(MT.ERROR, error);
  }
}
```

5. Create a new `asynchronous` function called `fetchUsersData`, which receives as the first argument a deconstructed JavaScript object, with the `commit` property, and the second argument the `userId` that will be fetched. Add a `try/catch` statement, in the `try` context. We execute `commit(MT.LOADING)`, then we fetch the users lists from the API, and finally, `commit(MT.SET_USER_DATA, data)`, passing the users lists to be mutated. If we receive an exception and get into the `catch` statement, we will execute a mutation of `commit(MT.ERROR, error)`, passing the error that we receive to the `state`:

```
async function fetchUserData({ commit }, userId) {
  try {
    commit(MT.LOADING);
    const { data } = await getHttp(`api/users/${userId}`);
    commit(MT.SET_USER_DATA, data);
  } catch (error) {
    commit(MT.ERROR, error);
  }
}
```

6. Create a new `asynchronous` function called `updateUser`, which receives as the first argument a deconstructed JavaScript object, with the `commit` property, and `payload` as the second argument. Add a `try/catch` statement, in the `try` context. We execute `commit(MT.LOADING)`, then we patch the user data to the API and finally `commit(MT.UPDATE_USER, payload)`, passing the user new data to be mutated. If we receive an exception and get into the `catch` statement, we will execute a mutation of `commit(MT.ERROR, error)`, passing the error that we received to the `state`:

```
async function updateUser({ commit }, payload) {
  try {
    commit(MT.LOADING);
    await patchHttp(`api/users/${payload.id}`, {
      data: {
        ...payload,
      }
    });
    commit(MT.UPDATE_USER, payload);
  } catch (error) {
    commit(MT.ERROR, error);
  }
}
```

7. Create a new `asynchronous` function called `removeUser`, which receives as the first argument a deconstructed JavaScript object, with the `commit` property, and `userId` as the second argument. Add a `try/catch` statement, in the `try` context. We execute `commit(MT.LOADING)`, then we delete the user data from the API and finally, `commit(MT.REMOVE_USER, userId)`, passing the `userId` to be used in the mutation. If we receive an exception and get into the `Catch` statement, we will execute a mutation of `commit(MT.ERROR, error)`, passing the error that we receive to the `state`:

```
async function removeUser({ commit }, userId) {
  try {
    commit(MT.LOADING);
    await deleteHttp(`api/users/${userId}`);
    commit(MT.REMOVE_USER, userId);
  } catch (error) {
    commit(MT.ERROR, error);
  }
}
```

8. Finally, we will create an export default JavaScript object, with all the created functions as properties:

```
export default {
  createUser,
  fetchUsersList,
  fetchUserData,
  updateUser,
  removeUser,
}
```

9. Import the newly created `actions.js` file in the `index.js` in the `src/store/user` folder, and add it to the `export default` JavaScript object:

```
import state from './state';
import mutations from './mutations';
import getters from './getters';
import actions from './actions';

export default {
  state,
  mutations,
  getters,
  actions,
};
```

How it works...

Actions are the initializers of all the Vuex life cycle changes. When dispatched, the action can execute a mutation commit, or another action dispatch, or even an API call to the server.

In our case, we took our API calls and put it inside the actions, so when the asynchronous function returns, we can execute the commit and set the state to the result of the function.

See also

Find more information about Vuex actions at `https://vuex.vuejs.org/guide/actions.html`.

Creating a dynamic component with Vuex

Combining Vuex with Vue components, it's possible to employ the reactivity between multiple components without the need for direct parent-child communication, and split the responsibilities of the components.

Using this method allows the developer to enhance the scale of the application, where there is no need to store the state of the data inside the components itself, but using a single source of truth as a store for the whole application.

In this recipe, we will use the last recipes to improve an application, where it was using parent-child communication and making it as a single source of truth available in the whole application.

Getting ready

The prerequisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To create our dynamic component, we will transform the components from stateful to stateless, and will extract some parts that can be made into new components as well.

We will use the Vue project with Vue-CLI that we used in the '*Creating and understanding the Vuex actions*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create vuex-store
```

Choose the manual features, and add `Router` and `Vuex` as required features, as indicated in the '*How to do it...*' section of the '*Creating a simple Vuex store*' recipe.

Our recipe will be divided into five parts:

- Creating the user list component
- Editing the user list page
- Editing the user view page
- Editing the user view page
- Editing the user create page

Let's get started.

Creating the user list component

Because Vuex gives us the ability to have a single source of truth on our application, we can create a new component for our application that will handle the user listing and triggers the Vuex action that fetches the users list from the server. This component can be stateless and execute the `Vuex` actions by itself.

Single file component <script> section

Let's code the `<script>` section of the single file component:

1. Create a new file called `userList.vue` in the `src/components` folder.
2. Import the `changeRouterMixin` from the `src/mixin` folder:

   ```
   import changeRouteMixin from '@/mixin/changeRoute';
   ```

3. Create an `export default` JavaScript object, and add a new Vue property called `mixin`, with a default value of an array. To this array, add the imported `changeRouteMixin`:

   ```
   mixins: [changeRouteMixin],
   ```

4. Create a new Vue property called `computed`. In this property, create a new value called `userList`. This property will be a function returning the Vuex store getter `getUsersList`:

   ```
   computed: {
     userList() {
       return this.$store.getters.getUsersList;
     },
   },
   ```

Single file component <template> section

Here, we are going to code the `<template>` section of the single file component:

1. Open the `List.vue` file from the `users` folder inside the `views` folder and copy the content and component from the `VsTable` component.
2. Open the `userList.vue` file from the `src/components` folder.
3. Paste the content you'd copied from the `List.vue` file inside the `<template>` section:

```
<template>
  <vs-table
    :data="userList"
    search
    stripe
    pagination
    max-items="10"
    style="width: 100%; padding: 20px;"
  >
    <template slot="thead">
      <vs-th sort-key="name">
        #
      </vs-th>
      <vs-th sort-key="name">
        Name
      </vs-th>
      <vs-th sort-key="email">
        Email
      </vs-th>
      <vs-th sort-key="country">
        Country
      </vs-th>
      <vs-th sort-key="phone">
        Phone
      </vs-th>
      <vs-th sort-key="Birthday">
        Birthday
      </vs-th>
      <vs-th>
        Actions
      </vs-th>
    </template>
    <template slot-scope="{data}">
      <vs-tr :key="index" v-for="(tr, index) in data">
        <vs-td :data="data[index].id">
          {{data[index].id}}
        </vs-td>
        <vs-td :data="data[index].name">
          {{data[index].name}}
        </vs-td>
        <vs-td :data="data[index].email">
          <a :href="`mailto:${data[index].email}`">
            {{data[index].email}}
          </a>
        </vs-td>
        <vs-td :data="data[index].country">
          {{data[index].country}}
        </vs-td>
        <vs-td :data="data[index].phone">
          {{data[index].phone}}
        </vs-td>
        <vs-td :data="data[index].birthday">
          {{data[index].birthday}}
        </vs-td>
        <vs-td :data="data[index].id">
          <vs-button
            color="primary"
            type="filled"
            icon="remove_red_eye"
            size="small"
            @click="changeRoute('view', data[index].id)"
          />
          <vs-button
            color="success"
            type="filled"
            icon="edit"
            size="small"
            @click="changeRoute('edit', data[index].id)"
          />
          <vs-button
            color="danger"
            type="filled"
            icon="delete"
            size="small"
            @click="deleteUser(data[index].id)"
          />
        </vs-td>
      </vs-tr>
    </template>
  </vs-table>
</template>
```

Editing the user list page

Now that we have extracted the user list into a new component, we need to import this component and remove the old VsTable that was cluttering our view.

Single file component <script> section

In this step, we are going to write the `<script>` section of the single file component:

1. Open the `List.vue` file in the `users` folder inside the `views` folder.
2. Import the newly created Users List component, from the `components` folder:

```
import changeRouteMixin from '@/mixin/changeRoute';
import UserTableList from '@/components/userList';
```

3. In the `export default` JavaScript object, add a new property called `components`. Declare the property as a JavaScript object, and add the imported `UserTableList` component to the object:

```
components: { UserTableList },
```

4. In the `methods` property, at the `getAllUsers` function, we need to change the content to execute a Vuex dispatch when called. This method will perform the `fetchUsersList` Vuex action:

```
async getAllUsers() {
  this.$store.dispatch('fetchUsersList');
},
```

5. Finally, in the `deleteUser` function, we need to change the content to execute a Vuex dispatch when called. This method will perform the `removeUser` Vuex action, passing the `userId` as the argument:

```
async deleteUser(id) {
  this.$store.dispatch('removeUser', id);
  await this.getAllUsers();
},
```

Single file component <template> section

Let's code the `<template>` section of the single file component:

1. Open the `List.vue` file in the `users` folder inside the `view` folder.
2. Replace the `VsTable` component and its contents with the newly imported `UserTableList`:

```
<vs-col
  vs-type="flex"
  vs-justify="left"
  vs-align="left"
  vs-w="12">
  <user-table-list />
</vs-col>
```

Editing the user view page

Now we can add the Vuex to the user view page. We will add the Vuex actions and getters to manipulate the data, and extract from the page the responsibility of managing it.

Single file component <script> section

Now you are going to create the `<script>` section of the single file component:

1. Open the `View.vue` file from the `users` folder inside the `view` folder.
2. Remove the Vue `data` property.
3. Inside the Vue `computed` property, add the `userData`, returning a Vuex getter, `getUserData`:

```
userData() {
  return this.$store.getters.getUserData;
},
```

4. Finally, in the `getUserById` method, change the content to dispatch a Vuex action, `fetchUserData`, passing the computed `userId` property as a parameter:

```
async getUserById() {
  await this.$store.dispatch('fetchUserData', this.userId);
},
```

Single file component <template> section

It's time to write the `<template>` section of the single file component:

1. Open the `View.vue` file in the `users` folder inside the `view` folder.
2. In the UserForm component, change the `v-model` directive to a `:value` directive:

```
<user-form
  :value="userData"
  disabled
/>
```

When using a read-only value, or you need to remove the syntactic sugar of the `v-model` directive, you can declare the input value as a `:value` directive and the value change event to an `@input` event listener.

Editing the user edit page

We need to edit our user. In the last recipe, we used a stateful page and executed everything within the page. We will transform the state into a temporary state, and execute the API calls on the Vuex actions.

Single file component <script> section

Here, we are going to create the `<script>` section of the single file component:

1. Open the `Edit.vue` file in the `users` folder inside the `view` folder.
2. In the Vue `data` property, change the name of the data from `userData` to `tmpUserData`:

```
data: () => ({
  tmpUserData: {
    name: '',
    email: '',
    birthday: '',
    country: '',
    phone: '',
  },
}),
```

3. In the Vue `computed` property, add a new property called `userData`, which will return the Vuex getter `getUserData`:

```
userData() {
  return this.$store.getters.getUserData;
}
```

4. Add a new Vue property named `watch`, and add a new property, `userData`, which will be a JavaScript object. In this object, add three properties, `handler`, `immediate`, and `deep`. The `handler` property will be a function that receives an argument called `newData`, which will set `tmpUserData` to this argument. The `immediate` and `deep` properties are both boolean properties set to `true`:

```
watch: {
  userData: {
    handler(newData) {
      this.tmpUserData = newData;
    },
    immediate: true,
    deep: true,
  }
},
```

5. In the Vue `methods` property, we need to change the contents of `getUserById` to dispatch a Vuex action named `fetchUserData`, passing the `computed` property `userId` as a parameter:

```
async getUserById() {
  await this.$store.dispatch('fetchUserData', this.userId);
},
```

6. In the `updateUser` method, change the content to dispatch a Vuex action named `updateUser`, passing `tmpUserData` as a parameter:

```
async updateUser() {
  await this.$store.dispatch('updateUser', this.tmpUserData);
  this.changeRoute('list');
},
```

Single file component <template> section

In this part, we are going to write the `<template>` section of the single file component:

1. Open the `Edit.vue` in the `users` folder inside the `view` folder.
2. Change the target of the `v-model` directive of the `UserForm` component to `tmpUserData`:

```
<vs-col
  vs-type="flex"
  vs-justify="left"
  vs-align="left"
  vs-w="12"
  style="margin: 20px"
>
  <user-form
    v-model="tmpUserData"
  />
</vs-col>
```

Editing the user create page

For the user create page, the changes will be minimal, as it only executes an API call. We need to add the Vuex action dispatch.

Single file component <script> section

Here, we are going to create the `<script>` section of the single file component:

1. Open the `Create.vue` file in the `users` folder inside the `view` folder.
2. Change the content of the `createUser` method to dispatch a Vuex action named `createUser`, passing `userData` as the parameter:

```
async createUser() {
  await this.$store.dispatch('createUser', this.userData);
  this.changeRoute('list');
},
```

How it works...

In all four pages, we made changes that removed the business logic or API calls from the page to the Vuex store and tried making it less responsible for maintaining the data.

Because of that, we could place a piece of code into a new component that can be placed anywhere in our application, and will show the current users lists without any limitations from the container that is instantiating it.

This pattern helps us in the development of more prominent applications, where there is a need for components that are less business-oriented and more focused on their tasks.

See also

You can find more information about Vuex application structures at `https://vuex.vuejs.org/guide/structure.html`.

Adding hot-module-reload for development

The **hot-module-reload** (**HMR**) is a technique used for the faster development of the application, where you don't need to refresh the whole page to get the new code you have just changed on the editor. The HMR will change and refresh only the part that were updated by you on the editor.

In all the Vue-CLI projects or Vue-based frameworks, such as Quasar Framework, the HMR is present in the presentation of the application. So each time you change any file that is a Vue component and it's rendered, the application will replace the old code for the new one on the fly.

In this recipe, we will learn how to add HMR to a Vuex store and be able to change the Vuex store without the need to refresh our entire application.

Getting ready

The prerequisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we will use the Vue project with Vue-CLI that we used in the '*Creating a dynamic component with Vuex*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or
Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create vuex-store
```

Choose the manual features, add `Router` and `Vuex` as required features, as indicated in the '*How to do it...*' section of the '*Creating a simple Vuex store*' recipe.

In the following steps, we will add HMR to Vuex:

1. Open the `index.js` file in the `src/store` folder.
2. Transform the `export default` into a constant called `store`, and make it exportable:

```
export const store = new Vuex.Store({
  ...UserStore,
});
```

3. Check if the webpack `hot-module-reload` plugin is active:

```
if (module.hot) {}
```

4. Create a new constant called `hmr`, which is an array containing the direction to the `index.js`, `getters.js`, `actions.js`, and `mutations.js` files of the `user` folder:

```
const hmr = [
  './user',
  './user/getters',
  './user/actions',
  './user/mutations',
];
```

5. Create a new function called `reloadCallback`. In this function, create three constants, `getters`, `actions`, and `mutations`. Each constant will point to the equivalent file inside the `user` folder, and call the `store.hotUpdate` function, passing an object as an argument with the values for the constants you created:

```
const reloadCallback = () => {
  const getters = require('./user/getters').default;
  const actions = require('./user/actions').default;
```

```
    const mutations =  require('./user/mutations').default;

    store.hotUpdate({
      getters,
      actions,
      mutations,
    })
  };
```

Because of the Babel output of the files, you need to add the `.default` in the end of the files that you are dynamically importing with the webpack `require` function.

6. Execute the webpack HMR `accept` function, passing as the first argument the `hmr` constant and `reloadCallback` as the second argument:

```
module.hot.accept(hmr, reloadCallback);
```

7. Finally, default export the created `store`:

```
export default store;
```

How it works...

The Vuex store supports HMR with the API of the webpack HMR plugin.

When it's available, we create a list of possible files that can be updated, so that webpack can be aware of any updates to those files. When any of those files are updated, a special callback that you created is executed. This callback is the one that enables Vuex to update or change the behavior of the updated file entirely.

See also

You can find more information about Vuex hot reloading at `https://vuex.vuejs.org/guide/hot-reload.html`.

You can find more information about webpack HMR at `https://webpack.js.org/guides/hot-module-replacement/`.

Creating a Vuex module

As our application grows, working in a single object can be very risky. The maintainability of the project and the risks that it can generate on every change get worse each time.

Vuex has an approach called modules that helps us to separate our store into different branches of stores. These branches, or modules, have on each one of them a different set of state, mutation, getter, and action. This pattern helps with development and cuts the risk of adding new features to the application.

In this recipe, we will learn how to create a module and how to work with it, separating it into dedicated branches.

Getting ready

The prerequisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To start our component, we will use the Vue project with Vue-CLI that we used in the '*Creating a dynamic component with Vuex*' recipe, or we can start a new one.

To start a new one, open Terminal (macOS or Linux) or
Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create vuex-store
```

Choose the manual features and add `Router` and `Vuex` as required features, as indicated in the '*How to do it...*' section of the '*Creating a simple Vuex store*' recipe.

Our recipe will be divided into two parts:

- Creating the new authentication module
- Adding modules to Vuex

Let's get started.

Creating the new authentication module

To start, we need to create a new `Vuex` module. This example module will be called `authentication`, and will store the credentials data for the user.

In these steps, we will create the `authentication` module for `Vuex`:

1. Create a new folder called `authentication` in the `src/store` folder.
2. In this newly created folder, create a new file called `state.js`, and open it.
3. Create a function called `generateState` that will return a JavaScript object with the properties of `data.username`, `data.token`, `data.expiresAt`, `loading`, and `error`:

```
const generateState = () => ({
  data: {
    username: '',
    token: '',
    expiresAt: null,
  },
  loading: false,
  error: null,
});
```

4. Create an `export default` object at the end of the file. This object will be a JavaScript object. We will destruct the return of the `generateState` function:

```
export default { ...generateState() };
```

5. Create a new file called `index.js` in the `authentication` folder inside the `src/store` folder, and open it.

6. Import the newly created `state.js` file:

```
import state from './state';
```

7. Create an `export default` object at the end of the file. This object will be a JavaScript object. Add a new property called `namespaced` with the value set to `true`, and add the imported `state`:

```
export default {
  namespaced: true,
  state,
};
```

Adding the modules to Vuex

Now that we've created our modules, we will add them to the Vuex store. We can integrate the new modules with our old code. This is not a problem because Vuex will handle the new module as a namespaced object, with a completely separate Vuex store.

Now in these steps, we will add the created modules to the Vuex:

1. Open the `index.js` file in the `src/store` folder.
2. Import the `index.js` file from the `authentication` folder:

```
import Vue from 'vue';
import Vuex from 'vuex';
import UserStore from './user';
import Authentication from './authentication';
```

3. In the `Vuex.Store` function, add a new property called `modules`, which is a JavaScript object. Then add the imported `User` and `Authentication` modules:

```
export default new Vuex.Store({
  ...UserStore,
  modules: {
    Authentication,
  }
})
```

How it works...

Modules work like separate Vuex stores but in the same Vuex single source of truth. This helps in the development of larger-scale applications because you can maintain and work with a more complex structure without the need to check for problems in the same file.

In the meantime, it's possible to work with modules and the plain Vuex store, migrating from legacy applications so you don't have to re-write everything from the ground up to be able to use the module structure.

In our case, we added a new module named `authentication` with just a state present in the store, and continued with the old user Vuex store, so that in the future we can refactor the user store into a new module and separate it off into a more specific, domain-driven architecture.

See also

You can find more information about Vuex modules at `https://vuex.vuejs.org/guide/modules.html`.

8
Animating Your Application with Transitions and CSS

To have a more dynamic application and have the full attention of the user, using animation is crucial. Today, CSS animations are present in toasts, banners, notifications, and even input fields.

There are some cases where you need to create special animations, known as transitions, and have full control of what is happening on your page. To do this, you must use custom components and have the framework to help you with rendering your application.

With Vue, we can use two custom components that can help us create animations and transitions in our application with the help of CSS classes. Those components are `Transition` and `TransitionGroup`.

In this chapter, we will learn how to create a CSS animation, use the Animate.css framework to create a custom transition, use the `Transition` component hook to execute custom functions, create animations that execute on the render of the component, create animations and transitions for groups and lists, create reusable custom transition components, and create seamless transitions between components.

In this chapter, we'll cover the following recipes:

- Creating your first CSS animation
- Creating a custom transition class with Animate.css
- Creating transactions with custom hooks
- Creating animations on page render
- Creating animations for lists and groups
- Creating a custom transition component
- Creating a seamless transition between elements

Technical requirements

In this chapter, we will be using **Node.js** and **Vue-CLI.**

Attention Windows users! You need to install an NPM package called `windows-build-tools` to be able to install the following required packages. To do so, open PowerShell as an Administrator and execute the `> npm install -g windows-build-tools` command.

To install **Vue-CLI**, you need to open Terminal (macOS or Linux) or a Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install -g @vue/cli @vue/cli-service-global
```

Creating the base project

In this chapter, we will use this project as the base for each recipe. Here, I will guide you through how to create the base project:

1. Open Terminal (macOS or Linux) or a Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create {replace-with-recipe-name}
```

2. Vue-CLI will ask for you to choose a preset; select `Manually select features` using the *spacebar*:

```
? Please pick a preset: (Use arrow keys)
  default (babel, eslint)
❯ Manually select features
```

3. Now, Vue-CLI will ask for what features you wish to install. You will need to select `CSS Pre-processors` as an additional feature on top of the default ones:

```
? Check the features needed for your project: (Use arrow keys)
❯ Babel
  TypeScript
  Progressive Web App (PWA) Support
  Router
  Vuex
❯ CSS Pre-processors
❯ Linter / Formatter
  Unit Testing
  E2E Testing
```

4. Continue this process by selecting a linter and formatter. In our case, we will select `ESLint + Airbnb config`:

```
? Pick a linter / formatter config: (Use arrow keys)
  ESLint with error prevention only
❯ ESLint + Airbnb config
  ESLint + Standard config
  ESLint + Prettier
```

5. Choose the additional features of the linter. In our case, select the `Lint on save` and `Lint and fix on commit`:

```
? Pick additional lint features: (Use arrow keys)
  Lint on save
❯ Lint and fix on commit
```

6. Select where you want to place the linter and formatter configuration files. In our case, we will select `In dedicated config files`:

```
? Where do you prefer placing config for Babel, ESLint, etc.? (Use
arrow keys)
❯ In dedicated config files
  In package.json
```

7. Finally, the CLI will ask you whether you want to save the settings for future projects; select `N`. After that, Vue-CLI will create the folder and install the dependencies for you:

```
? Save this as a preset for future projects? (y/N) n
```

8. From the created project, open the `App.vue` file, which is located in the `src` folder. In the `<script>` section of the single file component, remove the `HelloWorld` component. Add a `data` property and define it as a singleton function that's returning a JavaScript object with a property named `display`, and with a default value of `true`:

```
<script>
export default {
  name: 'App',
  data: () => ({
    display: true,
  }),
};
</script>
```

9. In the `<template>` section of the single file component, remove the `HelloWorld` component and add a `button` HTML element with the text `Toggle`. In the `img` HTML element, add a `v-if` directive bounded to the `display` variable. Finally, in the `button` HTML element, add a `click` event. In the event listener, define the value as an anonymous function that sets the `display` variable as the negation of the `display` variable:

```
<template>
  <div id="app">
    <button @click="display = !display">
      Toggle
    </button>
    <img
      v-if="display"
      alt="Vue logo" src="./assets/logo.png">
  </div>
</template>
```

With these instructions, we can create a base project for each recipe in this chapter.

Creating your first CSS animation

With the help of CSS, we can animate our application without the need to manually program the changes of DOM elements through JavaScript. Using special CSS properties dedicated exclusively to controlling animations, we can achieve beautiful animations and transitions.

To use the animations that are available in Vue, we need to use a component called `Transition` when an animation is being applied to a single element or a component called `TransitionGroup` when dealing with a list of components.

In this recipe, we will learn how to create a CSS animation and apply this animation to a single element on the Vue application.

Getting ready

The following are the prerequisites for this recipe:

- Node.js 12+
- A Vue-CLI base project called `cssanimation`

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

Using the base project, create a new project for this recipe called `cssanimation` and open the project folder. Now, follow these steps:

1. Open the `App.vue` file. In the `<template>` section of the single file component, wrap the `img` HTML element with a `Transaction` component. In the `Transaction` component, add a `name` attribute with a value of `"image"`:

```
<transition name="image">
  <img
    v-if="display"
    alt="Vue logo" src="./assets/logo.png">
</transition>
```

2. In the `<style>` section of the single file component, create an `.image-enter-active` class with an `animation` property that has a value of `bounce-in .5s`. Then, create an `.image-leave-active` class with an `animation` property that has a value of `bounce-in .5s reverse`:

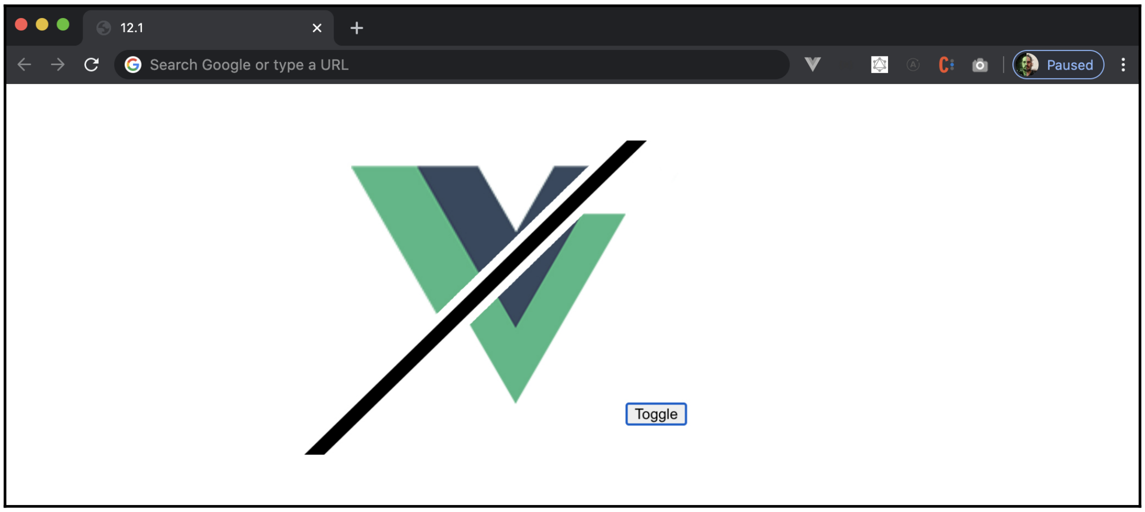
```
.image-enter-active {
  animation: bounce-in .5s;
}
.image-leave-active {
  animation: bounce-in .5s reverse;
}
```

3. Finally, create a `@keyframes bounce-in` CSS rule. Inside it, do the following:
    - Create a `0%` rule with a property transform and a value of `scale(0)`.
    - Create a `50%` rule with a property transform and a value of `scale(1.5)`.
    - Create a `100%` rule with a property transform and a value of `scale(1)`:

```
@keyframes bounce-in {
  0% {
    transform: scale(0);
  }
```

```
    50% {
      transform: scale(1.5);
    }
    100% {
      transform: scale(1);
    }
  }
```

After doing this, your image will scale up and disappear when the toggle button is pressed for the first time. When pressed again, it will scale up and stay in the correct scale after the animation has finished:



How it works...

First, we added the Vue animation wrapper to the element we wanted to add the transition to, and then added the name of the CSS class that will be used on the transition.

The `Transition` component uses pre-made namespaces for the CSS class that are required to be present. These are `-enter-active`, for when the component enters the screen, and `-leave-active`, for when the component leaves the screen.

Then, we create the CSS classes in `<style>` for the transition of the element to leave and enter the screen, and the `keyframe` ruleset for the `bounce-in` animation in order to define how it will behave.

See also

You can find more information about class-based animation and transitions with Vue classes at `https://v3.vuejs.org/guide/transitions-overview.html#class-based-animations-transitions`.

You can find more information about CSS keyframes at `https://developer.mozilla.org/en-US/docs/Web/CSS/@keyframes`.

Creating a custom transition class with Animate.css

In the `Transition` component, it is possible to define the CSS classes that will be used in each transition step. By using this property, we can make the `Transition` component use Animate.css in the transition animations.

In this recipe, we will learn how to use the Animate.css classes with the `Transition` component in order to create custom transitions in our components.

Getting ready

The following are the prerequisites for this recipe:

- Node.js 12+
- A Vue-CLI base project called `animatecss`

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

Using the base project, create a new project for this recipe called `animatecss` and open the project folder. Now, follow these steps:

1. Inside the project folder, open Terminal (macOS or Linux) or a Command Prompt/PowerShell (Windows) and execute the following command to install the Animate.css framework:

   ```
   > npm install animate.css@3.7.2
   ```

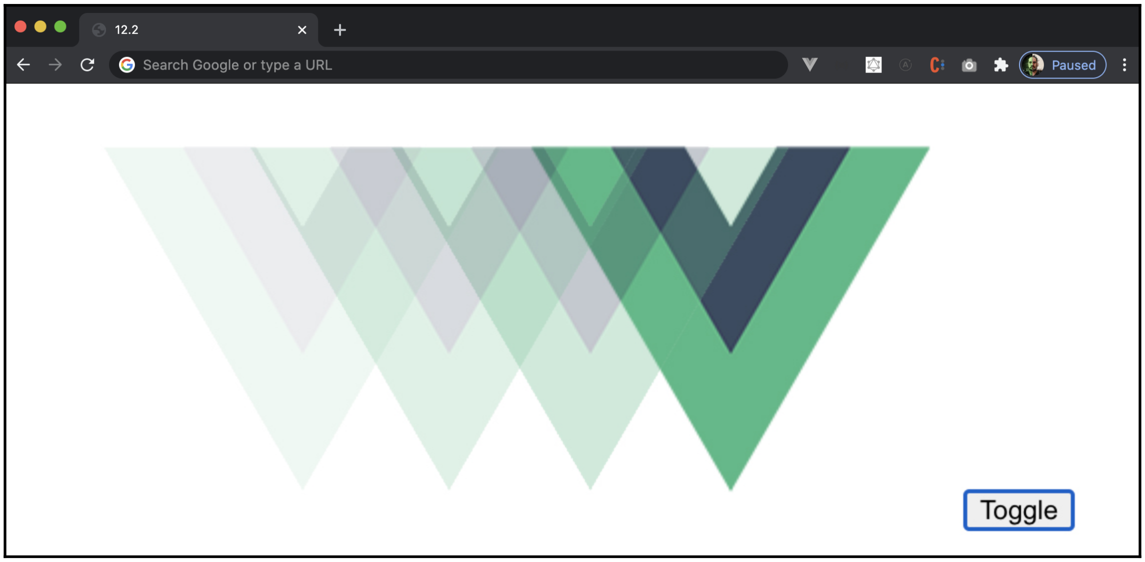
2. Open the `main.js` file in the `src` folder and import the Animate.css framework:

   ```
   import Vue from 'vue';
   import App from './App.vue';
   import 'animate.css';
   ```

3. Open the `App.vue` file in the `src` folder and add a `Transition` component as a wrapper for the `img` HTML element. In the `Transition` component, add an attribute called `enter-active-class` and define it as `"animated bounceInLeft"`. Then, add another attribute called `leave-active-class` and define it as `"animated bounceOutLeft"`:

   ```
   <transition
     enter-active-class="animated bounceInLeft"
     leave-active-class="animated bounceOutLeft"
   >
     <img
       v-if="display"
       alt="Vue logo" src="./assets/logo.png">
   </transition>
   ```

After doing this, your image will slide out to the left and disappear when the toggle button is pressed for the first time. When pressed again, it will slide in from the left:



How it works...

The `Transition` component can receive up to six props that can set up custom classes for each step of the transaction. Those props are `enter-class`, `enter-active-class`, `enter-to-class`, `leave-class`, `leave-active-class`, and `leave-to-class`. In this recipe, we used `enter-active-class` and `leave-active-class`; these props defined the custom classes for when the element is visible on the screen or leaves the screen.

To use custom animations, we used the Animate.css framework, which provides custom CSS animations that have been pre-made and ready for use. We used `bounceInLeft` and `bounceOutLeft` in order to make the element slide in and out from the screen.

There's more...

You can try to change the classes of the `enter-active-class` and `leave-active-class` props for any of the props available on Animate.css and see how the CSS animation behaves on the browser.

You can find the full list of available classes in the Animate.css documentation at `https://animate.style/`.

See also

You can find more information about class-based animation and transitions with Vue classes at `https://v3.vuejs.org/guide/transitions-overview.html#class-based-animations-transitions`.

You can find more information about Animate.css at `https://animate.style/`.

Creating transactions with custom hooks

The `Transaction` component has custom event emitters for each animation life cycle. These can be used to attach custom functions and methods to be executed when the animations cycle is completed.

We can use these methods to execute data fetches after the page transaction completes or a button animation ends, thus chaining animations in a specific order that need to be executed one after another based on dynamic data.

In this recipe, we will learn how to use the custom event emitters of the `Transaction` component to execute custom methods.

Getting ready

The following are the prerequisites for this recipe:

- Node.js 12+
- A Vue-CLI base project called `transactionhooks`

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

Using the base project, create a new project for this recipe called `transactionhooks` and open the project folder. Now, follow these steps:

1. Inside the project folder, open Terminal (macOS or Linux) or a Command Prompt/PowerShell (Windows) and execute the following command to install the Animate.css framework:

```
> npm install animate.css@3.7.2
```

2. Open the `main.js` file in the `src` folder and import the Animate.css framework:

```
import Vue from 'vue';
import App from './App.vue';
import 'animate.css';
```

3. Open the `App.vue` file in the `src` folder. In the `<script>` section of the single file component, in the data property, in the singleton function, add a new property called `status` with the value defined as `"appeared"`:

```
data: () => ({
  display: true,
  status: 'appeared',
}),
```

4. Create a `methods` property and define it as a JavaScript object. Inside the object, add two properties called `onEnter` and `onLeave`. In the `onEnter` property, define it as a function, and inside of it, set the data `status` variable to`"appeared"`. In the `onLeave` property, define it as a function, and inside of it set the data `status` variable to `"disappeared"`:

```
methods: {
  onEnter() {
    this.status = 'appeared';
  },
  onLeave() {
    this.status = 'disappeared';
  },
},
```

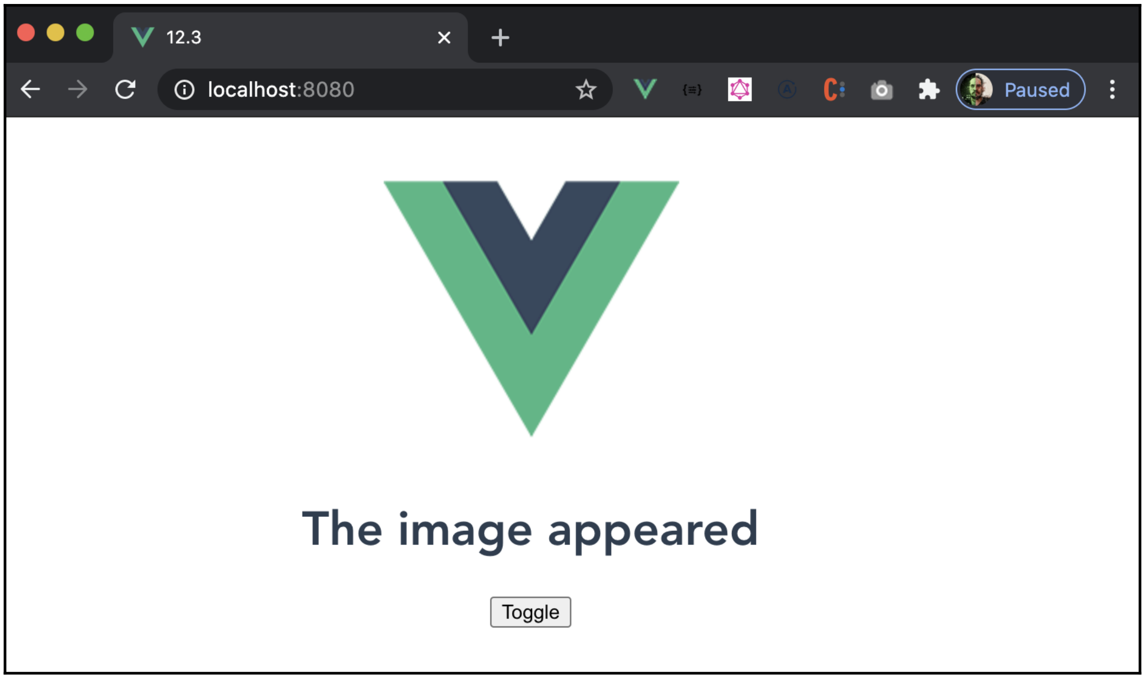
5. In the `<template>` section of the single file component, add a `Transition` component as a wrapper for the `img` HTML element. In the `Transition` component, do the following:
   - Add an attribute called `enter-active-class` and define it as `"animated rotateIn"`.
   - Add another attribute called `leave-active-class` and define it as `"animated rotateOut"`.
   - Add an event listener `after-enter` bind and attach it to the `onEnter` method.
   - Add an event listener `after-leave` bind and attach it to the `onLeave` method:

```
<transition
  enter-active-class="animated rotateIn"
  leave-active-class="animated rotateOut"
  @after-enter="onEnter"
  @after-leave="onLeave"
>
  <img
    v-if="display"
    alt="Vue logo" src="./assets/logo.png">
</transition>
```

6. Create an `h1` HTML element as a sibling of the `Transition` component and add the text `The image {{ status }}`:

```
<h1>The image {{ status }}</h1>
```

Now, when the button is clicked, the text will change when the animation finishes. It will show **The image appeared** when the animation finishes entering and **The image disappeared** when the animation has finished leaving:



How it works...

The `Transition` component has eight custom hooks. These hooks are triggered by the CSS animations and when they are triggered, they emit custom events, which can be used by the parent component. These custom events are `before-enter`, `enter`, `after-enter`, `enter-cancelled`, `before-leave`, `leave`, `after-leave`, and `leave-cancelled`.

When using the `after-enter` and `after-leave` hooks, when the CSS animations have finished, the text on the screen changes accordingly to the functions that have been defined on the event listeners for each hook.

See also

You can find more information about transition hooks at `https://v3.vuejs.org/guide/transitions-enterleave.html#javascript-hooks`.

You can find more information about Animate.css at `https://animate.style/`.

Creating animations on page render

Using page transition animations or custom animations that are displayed on the render of a page is common and sometimes needed to catch the attention of the user of an application.

It's possible to create this effect in a Vue application without the need to refresh the page or re-render all the elements on the screen. You can do this using the `Transition` component or the `TransitionGroup` component.

In this recipe, we will learn how to use the `Transition` component so that the animation is triggered when the page is being rendered.

Getting ready

The following are the prerequisites for this recipe:

- Node.js 12+
- A Vue-CLI base project called `transactionappear`

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

Using the base project, create a new project for this recipe called `transactionappear` and open the project folder. Now, follow these steps:

1. Inside the project folder, open Terminal (macOS or Linux) or a Command Prompt/PowerShell (Windows) and execute the following command to install the Animate.css framework:

   ```
   > npm install animate.css@3.7.2
   ```

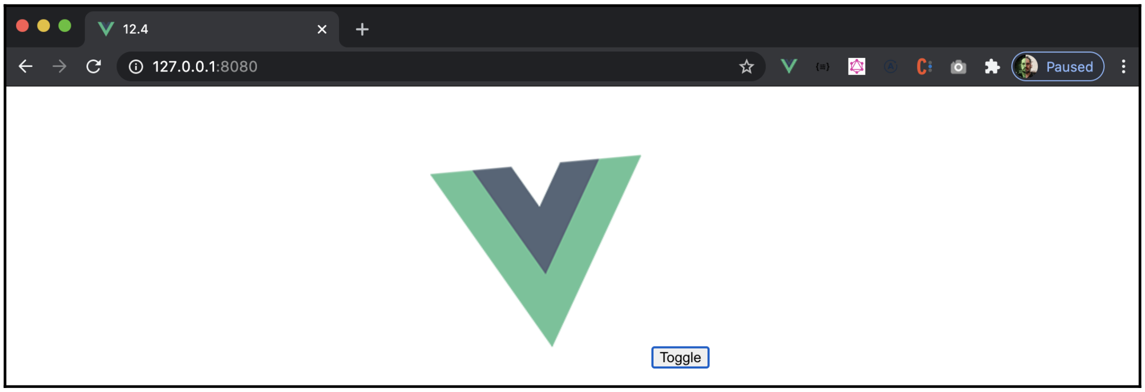
2. Open the `main.js` file in the `src` folder and import the Animate.css framework:

   ```
   import Vue from 'vue';
   import App from './App.vue';
   import 'animate.css';
   ```

3. Open the `App.vue` file in the `src` folder and add a `Transition` component as a wrapper for the `img` HTML element. In the `Transition` component, do the following:
    - Add an attribute called `appear-active-class` and define it as `"animated jackInTheBox"`.
    - Add an attribute called `enter-active-class` and define it as `"animated jackInTheBox"`.
    - Add another attribute called `leave-active-class` and define it as `"animated rollOut"`.
    - Add the `appear` attribute and define it as `true`:

```
<transition
  appear
  appear-active-class="animated jackInTheBox"
  enter-active-class="animated jackInTheBox"
  leave-active-class="animated rollOut"
>
  <img
    v-if="display"
    alt="Vue logo" src="./assets/logo.png">
</transition>
```

When the page opens, the Vue logo will shake like a jack-in-the-box and will be static after the animation has finished running:



How it works...

The `Transition` component has a special property called `appear` that, when enabled, makes the element trigger an animation when it is rendered on the screen. This property comes with three properties for controlling the animation CSS classes, which are called `appear-class`, `appear-to-class`, and `appear-active-class`.

There are four custom hooks that are executed with this property as well, which are called `before-appear`, `appear`, `after-appear`, and `appear-cancelled`.

In our case, we made the component execute the `jackInTheBox` animation from the Animate.css framework when the component gets rendered on-screen.

See also

You can find more information about transitions on initial render at `https://v3.vuejs.org/guide/transitions-enterleave.html#transitions-on-initial-render`.

You can find more information about Animate.css at `https://animate.style/`.

Creating animations for lists and groups

There are some animations that need to be executed within a group of elements or a list. These animations need to be wrapped in a `TransitionGroup` element in order to work.

This component has some properties that are the same as the ones in the `Transition` component, but to get it working, you have to define a set of special instructions for the child elements and the components that are specific to this component.

In this recipe, we will create a dynamic list of images that will be added when the user clicks on the respective button. This will execute the animation when the image appears on the screen.

Getting ready

The following are the prerequisites for this recipe:

- Node.js 12+
- A Vue-CLI base project called `transactiongroup`

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

Using the base project, create a new project for this recipe called `transactiongroup` and open the project folder. Now, follow these steps:

1. Inside the project folder, open Terminal (macOS or Linux) or a Command Prompt/PowerShell (Windows) and execute the following command to install the Animate.css framework:

```
> npm install animate.css@3.7.2
```

2. Open the `main.js` file in the `src` folder and import the Animate.css framework:

```
import Vue from 'vue';
import App from './App.vue';
import 'animate.css';
```

3. Open the `App.vue` file in the `src` folder. In the `<script>` section of the single file component, on the `data` singleton, return a property called `count` with a value of `0`:

```
data: () => ({
  count: 0,
}),
```

4. In the `<template>` section of the single file component, remove everything inside the `div#app` HTML element. Then, as a child of the `div#app` HTML element, create a `TransitionGroup` component with an attribute called `tag` defined as `"ul"` and an attribute called `enter-active-class` defined as `"animated zoomIn"`:

```
<div id="app">
  <transition-group
    tag="ul"
    enter-active-class="animated zoomIn"
  ></transition-group>
</div>
```

5. As a child of the `TransitionGroup` component, create a `li` HTML element with the `v-for` directive, iterating over the `count` variable as `i in count`. Add a variable attribute called `key` defined as `i` and a `style` attribute defined as `"float: left"`. As a child of the `li` HTML component, create an `img` HTML component with the `src` attribute defined as `"https://picsum.photos/100"`:

```
<li
  v-for="i in count"
  :key="i"
  style="float: left"
>
  <img src="https://picsum.photos/100"/>
</li>
```

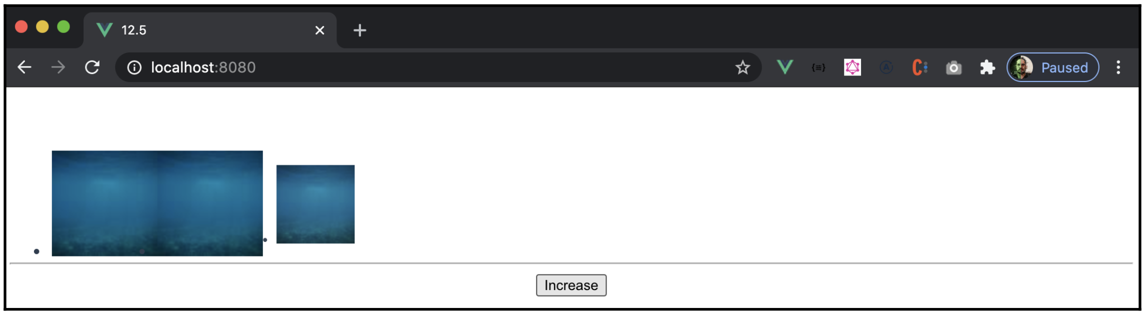
6. Then, as a sibling element of the `TransitionGroup` component, create a `hr` HTML element with the `style` attribute defined as `"clear: both"`:

```
<hr style="clear: both"/>
```

7. Finally, as a sibling of the `hr` HTML element, create a `button` HTML element with the `click` event, adding `1` to the current `count` variable and setting the text to `Increase`:

```
<button
  @click="count = count + 1"
>
  Increase
</button>
```

Now, when the user clicks the respective button to increase the list, it will add a new item to the list and the zooming in animation will trigger:



How it works...

The `TransitionGroup` element creates a wrapper element with the tag you declared in the `tag` property. This will manage the custom elements that will trigger the animation by checking the unique identity of the child elements by their unique keys. Because of this, all the child elements inside the `TransitionGroup` component need to have a `key` declared and have to be unique.

In our case, we created an HTML list using a combination of `ul` and `li` HTML elements, where `TransitionGroup` was defined with the `ul` tag and the child elements were defined with the `li` HTML elements. Then, we created a virtual iteration over a number. This means there will be a list of items and display images on-screen according to the number of items on that list.

To increase our list, we created a `button` HTML element that increased the count of the `count` variable by one each time it was pressed.

See also

You can find more information about transition groups at `https://v3.vuejs.org/guide/transitions-list.html#reusable-transitions`.

You can find more information about Animate.css at `https://animate.style/`.

Creating a custom transition component

Using a framework to create an application is good because you can make reusable components and shareable code. Using this pattern is great for simplifying the development of the application.

Creating a reusable transition component is the same as creating a reusable component and can have a simpler approach as it can be used with functional rendering instead of the normal rendering method.

In this recipe, we will learn how to create a reusable functional component that can be used in our application.

Getting ready

The following are the prerequisites for this chapter:

- Node.js 12+
- A Vue-CLI base project called `customtransition`

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

Using the base project, create a new project for this recipe called `customtransition` and open the project folder. Now, follow these steps:

1. Inside the project folder, open Terminal (macOS or Linux) or a Command Prompt/PowerShell (Windows) and execute the following command to install the Animate.css framework:

   ```
   > npm install animate.css@3.7.2
   ```

2. Open the `main.js` file in the `src` folder and import the Animate.css framework:

   ```
   import Vue from 'vue';
   import App from './App.vue';
   import 'animate.css';
   ```

3. Create a new file named `CustomTransition.vue` in the `src/components` folder and open it. In the `<template>` section of the single file component, add the `functional` attribute to enable the functional rendering of the component. Then, create a `Transition` component, with the `appear` variable attribute defined as `props.appear`. Define the `enter-active-class` attribute as `"animated slideInLeft"` and the `leave-active-class` attribute as `"animated slideOutRight"`. Finally, inside the `Transition` component, add a `<slot>` placeholder:

   ```
   <template functional>
     <transition
       :appear="props.appear"
       enter-active-class="animated slideInLeft"
       leave-active-class="animated slideOutRight"
   ```

```
    >
      <slot />
    </transition>
  </template>
```

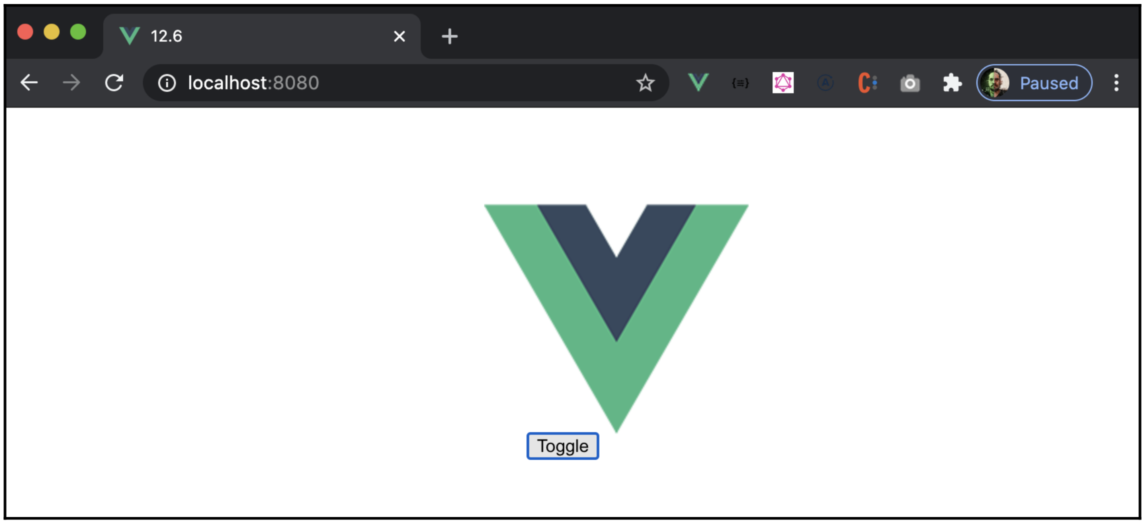
4. Open the `App.vue` file in the `src` folder. In the `<script>` section of the single file component, import the newly created `CustomTransition` component. On the Vue object, add a new property called `components`, define it as a JavaScript object, and add the imported `CustomTransition` component:

```
<script>
import CustomTransition from './components/CustomTransition.vue';

export default {
  name: 'App',
  components: {
    CustomTransition,
  },
  data: () => ({
    display: true,
  }),
};
</script>
```

5. Finally, in the `<template>` section of the single file component, wrap the `img` HTML element with the `CustomTransition` component:

```
<custom-transition>
  <img
    v-if="display"
    alt="Vue logo" src="./assets/logo.png">
</custom-transition>
```

With this custom component, it's possible to reuse the transition without the need to redeclare the `Transition` component and the transition CSS classes on the component:



How it works...

First, we created a custom component using the functional component method, where there is no need to declare the `<script>` section of the single file component.

In this custom component, we used the `Transaction` component as the base component. Then, we defined the `appear` attribute with the injected functional context, `prop.appear`, and added the animations classes for the transition to slide in from the left when the component is rendered and slide out from the right when it's destroyed.

Then, in the main application, we used this custom component to wrap the `img` HTML element and make it work as the `Transition` component.

See also

You can find more information about reusable transition components at `https://v3.vuejs.org/guide/transitions-list.html#reusable-transitions`.

You can find more information about Animate.css at `https://animate.style/`.

Creating a seamless transition between elements

When there are animations and transitions between two components, they need to be seamless so that the user won't see the DOM shaking and redrawing itself when the components are being placed on the screen. To achieve this, we can use the `Transition` component and the transition mode property to define how the transition will occur.

In this recipe, we will create a transition between images using the `Transition` component and the transition mode attribute to create a seamless animation.

Getting ready

The following are the prerequisites for this chapter:

- Node.js 12+
- A Vue-CLI base project called `seamlesstransition`

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

Using the base project, create a new project for this recipe called `seamlesstransition` and open the project folder. Now, follow these steps:

1. Open the `App.vue` file in the `src` folder. In the `<style>` section of the single file component, create a property called `.rotate-enter-active,.rotate-leave-active` and define the `transition` CSS style property as `transform .8s ease-in-out;`. Then, create a property called `.rotate-enter,.rotate-leave-active` and define the `transform` CSS style property as `rotate( -180deg );` and `transition` as `.8s ease-in-out;`:

```
.rotate-enter-active,
.rotate-leave-active {
  transition: transform .8s ease-in-out;
}
```

```
.rotate-enter,
.rotate-leave-active {
  transform: rotate( -180deg );
  transition: transform .8s ease-in-out;
}
```

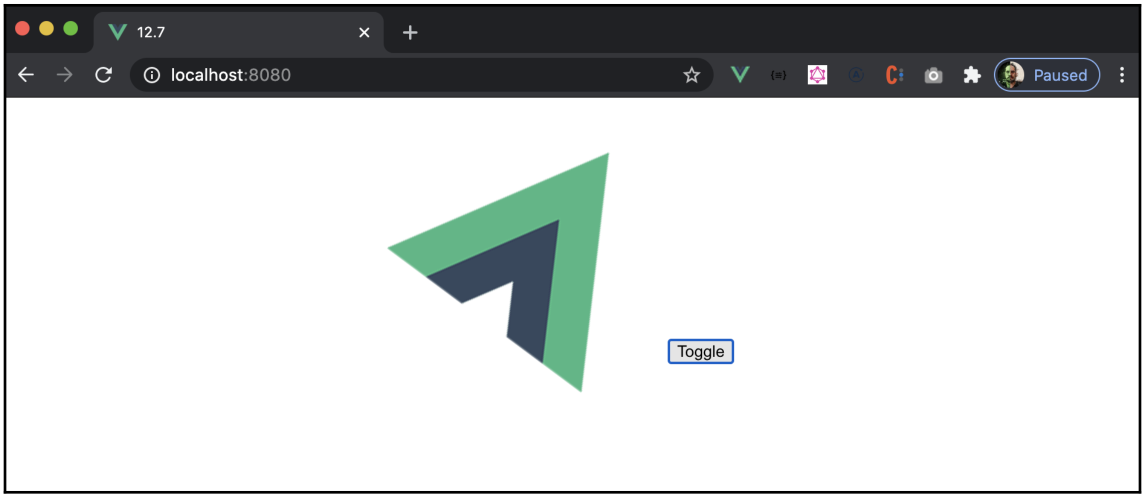
2. In the `<template>` section of the single file component, wrap the `img` HTML element with a `Transition` component. Then, define the `name` attribute as `rotate` and the `mode` attribute as `out-in`:

```
<transition
  name="rotate"
  mode="out-in"
></transition>
```

3. Inside the `Transition` component, in the `img` HTML element, add a `key` attribute and define it as `up`. Then, add another `img` HTML element and add a `v-else` directive. Add a `key` attribute and define it as `down`, add an `src` attribute and define it as `"./assets/logo.png"`, and finally add a `style` attribute and define it as `"transform: rotate(180deg)"`:

```
<img
  v-if="display"
  key="up"
  src="./assets/logo.png">
<img
  v-else
  key="down"
  src="./assets/logo.png"
  style="transform: rotate(180deg)"
>
```

When the user toggles the element, the leaving animation will be executed, and then after it has finished, the entering animation will start with no delay. This makes for a seamless transition between the old element and the new one:



How it works...

The `Transition` component has a special attribute called `mode`, where it is possible to define the behavior of the element's transition animation. This behavior will create a set of rules that controls how the animation steps will occur inside the `Transition` component. It's possible to use `"in-out"` or `"out-in"` mode in the component:

- In the `"in-out"` behavior, the new element transition will occur first, and when it's finished, the old element transition will start.
- In the `"out-in"` behavior, the old element transition will occur first, and then the new element transition will start.

In our case, we created an animation that rotates the Vue logo upside down. Then, to handle this seamless change, we used `"out-in"` mode so that the new element will only show up after the old one has finished the transition.

See also

You can find more information about transition modes at `https://v3.vuejs.org/guide/transitions-enterleave.html`.

9
Creating Beautiful Applications Using UI Frameworks

Using UI frameworks and libraries is a good way to increase productivity and help the development of your application. You can focus more on the code and less on the design.

Learning how to use such frameworks means that you know how these frameworks behave and work. This will help you in the process of developing an application or a framework in the future.

Here, you will learn more about the usage of frameworks when creating user registration forms and all the components that are needed for a page. In this chapter, we will learn how to create a layout, a page, and a form using Buefy, Vuetify, and Ant-Design.

In this chapter, we'll cover the following recipes:

- Creating a page, a layout, and a user form with Buefy
- Creating a page, a layout, and a user form with Vuetify
- Creating a page, a layout, and a user form with Ant-Design

Technical requirements

In this chapter, we will be using Node.js and Vue-CLI.

Attention Windows users: you need to install an `npm` package called `windows-build-tools`. To do so, open PowerShell as administrator and execute the following command:

```
> npm install -g windows-build-tools
```

To install `Vue-CLI`, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install -g @vue/cli @vue/cli-service-global
```

Creating a page, a layout, and a user form with Buefy

Bulma was one of the first frameworks to be used for rapid prototyping and web development that didn't require a JavaScript library attached to it. All the special components that needed to be coded were the responsibility of the developer using the framework.

With the advent of JavaScript frameworks and the community that was created around the Bulma framework, a wrapper for Vue was created. This wrapper takes all the responsibility of JavaScript component development and delivers a complete solution for developers to use the Bulma framework within their applications, without the need to re-invent the wheel.

In this recipe, we will learn how to use the Buefy framework with Vue and how to create a layout, a page, and a user registration form.

Getting ready

The pre-requisites for this recipe are as follows:

- Node.js 12+
- A Vue-CLI project

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

To create a Vue-CLI project with the Buefy framework, we need to create a Vue-CLI project first and then add the Buefy framework to the project. We will divide this recipe into four parts: creating the Vue-CLI project, adding the Buefy framework to the project, creating the layout and the page, and finally creating the user registration form.

Creating the Vue-CLI project

Here we will create the Vue-CLI project to be used in this recipe. This project will have custom settings to be able to work with the Buefy framework:

1. We need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > vue create bulma-vue
   ```

2. Vue-CLI will ask for you to choose a preset – select `Manually select features`:

   ```
   ? Please pick a preset: (Use arrow keys)
     default (babel, eslint)
   ❯ Manually select features
   ```

3. Now the Vue-CLI will ask for the features, and you will need to select `CSS Pre-processors` as an additional feature on top of the default ones:

   ```
   ? Check the features needed for your project: (Use arrow keys)
   ❯ Babel
     TypeScript
     Progressive Web App (PWA) Support
     Router
     Vuex
   ❯ CSS Pre-processors
   ❯ Linter / Formatter
     Unit Testing
     E2E Testing
   ```

4. Here the Vue-CLI will ask which CSS pre-processor you want to use; select `Sass/SCSS (with node-sass)`:

    ```
    ? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules
    are supported by default): (Use arrow keys)
      Sass/SCSS (with dart-sass)
    ❯ Sass/SCSS (with node-sass)
      Less
      Stylus
    ```

5. Continue this process by selecting a linter and formatted. In our case, we will select the `ESLint + Airbnb` config:

    ```
    ? Pick a linter / formatter config: (Use arrow keys)
      ESLint with error prevention only
    ❯ ESLint + Airbnb config
      ESLint + Standard config
      ESLint + Prettier
    ```

6. Choose the additional features of the linter (here, `Lint and fix on commit`):

    ```
    ? Pick additional lint features: (Use arrow keys)
      Lint on save
    ❯ Lint and fix on commit
    ```

7. Select where you want to place the linter and formatter configuration files (here, `In dedicated config files`):

    ```
    ? Where do you prefer placing config for Babel, ESLint, etc.? (Use
    arrow keys)
    ❯ In dedicated config files
      In package.json
    ```

8. Finally, the Vue-CLI will ask you whether you want to save the settings for future projects; you should select `N`. After that, Vue-CLI will create the folder and install the dependencies for you:

    ```
    ? Save this as a preset for future projects? (y/N) n
    ```

Adding Buefy to the Vue-CLI project

To use Bulma in a Vue project, we are going to use the Buefy UI library. This library is a wrapper around the Bulma framework and provides all the components that are available with the original framework and some additional components to use:

1. In the folder that you created for your Vue-CLI project, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > vue add buefy
   ```

2. Vue-CLI will ask whether you want to choose a style to work with Buefy; we will go with `scss`:

   ```
   ? Add Buefy style? (Use arrow keys)
     none
     css
   ❯ scss
   ```

3. Then, Vue-CLI will ask whether you want to include the Material Design icons; for this project, we won't use them:

   ```
   ? Include Material Design Icons? (y/N) n
   ```

4. Now Vue-CLI will ask whether you want to include Font Awesome icons; we will add them to the project:

   ```
   ? Include Font Awesome Icons? (y/N) y
   ```

Creating the layout and a page with Buefy

To create a page, we need to create a layout structure and the base components for the page, such as a header menu, a footer, and the hero section of the page.

Creating the header menu component

In our design, we will have a header menu, with a basic combination of links and call-to-action buttons:

1. Create a new file named `top-menu.vue` in the `src/components` folder and open it.

2. In the `<script>` section of the single file component, we will export a `default` JavaScript object, with the `name` property defined as `TopMenu`:

   ```
   <script>
   export default {
     name: 'TopMenu',
   };
   </script>
   ```

3. In the `<template>` section of the single file component, create a `section` HTML element with the `section` class, and add a child `div` HTML element with the `container` class:

   ```
   <section class="section">
     <div class="container">
     </div>
   </section>
   ```

4. Now create a `b-navbar` component as a child of the `div.container` HTML element, and add as a child a `template` placeholder component, with the named slot as `brand`. Inside that, add a `b-navbar-item` component with the `href` attribute defined as `#` and add an `img` HTML element as a child:

   ```
   <b-navbar>
     <template slot="brand">
       <b-navbar-item href="#">
         <img src="https://raw.githubusercontent.com/buefy/buefy/dev
                  /static/img/buefy-logo.png"
           alt="Lightweight UI components for Vue.js based on Bulma"
         >
       </b-navbar-item>
     </template>
   </b-navbar>
   ```

5. After this `template` placeholder, create another `template` placeholder with the named slot as `start`. Inside it, create two `b-navbar-item` components with the `href` attribute defined as `#`. Create, as a sibling component, a `b-navbar-dropdown` component with the `label` attribute defined as `Info`. In this component, add two `b-navbar-item` components as children with the `href` attribute defined as `#`:

   ```
   <template slot="start">
     <b-navbar-item href="#">
       Home
     </b-navbar-item>
     <b-navbar-item href="#">
   ```

```
      Documentation
    </b-navbar-item>
    <b-navbar-dropdown label="Info">
      <b-navbar-item href="#">
        About
      </b-navbar-item>
      <b-navbar-item href="#">
        Contact
      </b-navbar-item>
    </b-navbar-dropdown>
  </template>
```

6. Finally, create another `template` placeholder with the named slot as `end`. Create a `b-navbar-item` component as a child component with the `tag` attribute defined as `div`, and add a `div` HTML element as a child of this component with the `buttons` class. In the `div` HTML element, create an `a` HTML element with the `button is-primary` class, and another `a` HTML element with the `button is-light` class:

```
<template slot="end">
  <b-navbar-item tag="div">
    <div class="buttons">
      <a class="button is-primary">
        <strong>Sign up</strong>
      </a>
      <a class="button is-light">
        Log in
      </a>
    </div>
  </b-navbar-item>
</template>
```

Creating the hero section component

We will create a hero section component. A hero component is a full-width banner that provides visual information on the page to the user:

1. Create a new file named `hero-section.vue` in the `src/components` folder and open it.
2. In the `<script>` section of the single file component, we will export a `default` JavaScript object, with the `name` property defined as `HeroSection`:

```
<script>
export default {
  name: 'HeroSection',
};
</script>
```

3. In the `<template>` section of the single file component, create a `section` HTML element with the `hero is-primary` class, then add a `div` HTML element as a child, with the `hero-body` class:

```
<section class="hero is-primary">
  <div class="hero-body">
  </div>
</section>
```

4. Inside the `div.hero-body` HTML element, create a child `div` HTML element with the `container` class. Then, add an `h1` HTML element as a child with the `title` class and an `h2` HTML element with the `subtitle` class:

```
<div class="container">
  <h1 class="title">
    user Registration
  </h1>
  <h2 class="subtitle">
    Main user registration form
  </h2>
</div>
```

Creating the footer component

The final component that we are going to use in our layout is the footer component. This will be displayed as the footer of our page:

1. Create a new file named `Footer.vue` in the `src/components` folder and open it.
2. In the `<script>` section of the single file component, we will export a `default` JavaScript object, with the `name` property defined as `FooterSection`:

```
<script>
export default {
  name: 'FooterSection',
};
</script>
```

3. In the `<template>` section of the single file component, create a `footer` HTML element with the `footer` class, and then add a `div` HTML element with the `content has-text-centered` class:

```
<footer class="footer">
  <div class="content has-text-centered">
  </div>
</footer>
```

4. Inside the `div.content` HTML element, create a `p` HTML element for the initial footer line, and create a second `p` HTML element for the second line:

```
<p>
  <strong>Bulma</strong> by <a href="https://jgthms.com">Jeremy
    Thomas</a>
  | <strong>Buefy</strong> by
      <a href="https://twitter.com/walter_tommasi">Walter
         Tommasi</a>
</p>
<p>
  The source code is licensed
   <a href="http://opensource.org/licenses/mit-
license.php">MIT</a>.
  The website content is licensed
   <a href="http://creativecommons.org/licenses/by-nc-sa/4.0/">CC
       BY NC SA 4.0</a>.
</p>
```

Creating the layout component

To create the layout component, we are going to use all the created components, and add a slot that will hold the page content:

1. Create a new folder called `layouts` in the `src` folder, and create a new file named `Main.vue` and open it.
2. In the `<script>` section of the single file component, import the `footer-section` component and the `top-menu` component:

```
import FooterSection from '../components/Footer.vue';
import TopMenu from '../components/top-menu.vue';
```

3. Then, we will export a `default` JavaScript object, with the `name` property defined as `Mainlayout`, and define the `components` property with the imported components:

    ```
    export default {
      name: 'Mainlayout',
      components: {
        TopMenu,
        FooterSection,
      },
    };
    ```

4. Finally, in the `<template>` section of the single file component, create a `div` HTML element with the child `top-menu` component, a `slot` placeholder, and the `footer-section` component:

    ```
    <template>
      <div>
        <top-menu />
        <slot/>
        <footer-section />
      </div>
    </template>
    ```

Creating the user registration form with Buefy

Now we are going to create the user registration form and make the final page. In this step, we will join the outputs of all the other steps into a single page:

1. Open the `App.vue` file in the `src` folder. In the `<script>` section of the single file component, import the `main-layout` component and the `hero-section` component:

    ```
    import Mainlayout from './layouts/main.vue';
    import HeroSection from './components/heroSection.vue';
    ```

2. Then, we will export a `default` JavaScript object with the `name` property defined as `App`, then define the `components` property with the imported components. Add the `data` property to the JavaScript object, with the `name`, `username`, `password`, `email`, `phone`, `cellphone`, `address`, `zipcode`, and `country` properties:

    ```
    export default {
      name: 'App',
    ```

```
  components: {
    HeroSection,
    Mainlayout,
  },
  data: () => ({
    name: '',
    username: '',
    password: '',
    email: '',
    phone: '',
    cellphone: '',
    address: '',
    zipcode: '',
    country: '',
  }),
};
```

3. In the `<template>` section of the single file, add the imported `main-layout` component and add `hero-section` as a child component:

```
<template>
  <main-layout>
    <hero-section/>
  </main-layout>
</template>
```

4. After the `hero-section` component, create a sibling `section` HTML element, with the `section` class, and add a child `div` HTML element with the `container` class. In this `div` HTML element, create a `h1` HTML element with the `title is-3` class and a sibling `hr` HTML element:

```
<section class="section">
  <div class="container">
    <h1 class="title is-3">Personal Information</h1>
    <hr/>
  </div>
</section>
```

5. Then, create a `b-field` component as a sibling of the `hr` HTML element, with `Name` for `label`, and add a child `b-input` with the `v-model` directive bound to `name`. Do the same for the `email` field, changing `label` to `Email`, and the `v-model` directive bound to `email`. In the email `b-input`, add a `type` attribute defined as `email`:

```
<b-field label="Name">
  <b-input
```

```
      v-model="name"
    />
  </b-field>
  <b-field
    label="Email"
  >
    <b-input
      v-model="email"
      type="email"
    />
  </b-field>
```

6. Create a `b-field` component as a sibling of the `b-field` component, with the `grouped` attribute. Then, as child components, create the following:
    - A `b-field` component with the `expanded` attribute and `label` defined as `Phone`. Add a child `b-input` component with the `v-model` directive bound to `phone` and `type` as `tel`.
    - A `b-field` component with the `expanded` attribute and `label` defined as `Cellphone`. Add a child `b-input` component with the `v-model` directive bound to `cellphone` and `type` as `tel`:

```
<b-field grouped>
  <b-field
    expanded
    label="Phone"
  >
    <b-input
      v-model="phone"
      type="tel"
    />
  </b-field>
  <b-field
    expanded
    label="Cellphone"
  >
    <b-input
      v-model="cellphone"
      type="tel"
    />
  </b-field>
</b-field>
```

7. Then, create an `h1` HTML element as a sibling of the `b-field` component with the `title is-3` class, and add an `hr` HTML element as a sibling. Create a `b-field` component with `label` defined as `Address`, and add a `b-input` component with the `v-model` directive bound to `address`:

   ```
   <h1 class="title is-3">Address</h1>
   <hr/>
   <b-field
     label="Address"
   >
     <b-input
       v-model="address"
     />
   </b-field>
   ```

8. Create a `b-field` component as a sibling of the `b-field` component, with the `grouped` attribute. Then, as child components, create the following:
   - A child `b-field` component with the `expanded` attribute and `label` defined as `Zipcode`. Add a `b-input` component with the `v-model` directive bound to `zipcode` and the `type` attribute defined as `tel`.
   - A child `b-field` component with the `expanded` attribute and `label` defined as `Country`. Add a `b-input` component with the `v-model` directive bound to `country` and the `type` attribute defined as `tel`:

     ```
     <b-field grouped>
       <b-field
         expanded
         label="Zipcode"
       >
         <b-input
           v-model="zipcode"
           type="tel"
         />
       </b-field>
       <b-field
         expanded
         label="Country"
       >
         <b-input
           v-model="country"
         />
       </b-field>
     </b-field>
     ```

9. Then, create an `h1` HTML element as a sibling of the `b-field` component, with the `title is-3` class, and add an `hr` HTML element as a sibling. Create a `b-field` component with the `grouped` attribute. Create a child `b-field` component with the `expanded` attribute and `label` defined as `username`, and add a `b-input` component with the `v-model` directive bound to `username`. Do the same for the `Password` input, changing `label` to `Password`, in the `b-input` component defining the `v-model` directive as bound to `password`, and adding the `type` attribute as `password`:

```
<h1 class="title is-3">user Information</h1>
<hr/>
<b-field grouped>
  <b-field
    expanded
    label="username"
  >
    <b-input
      v-model="username"
    />
  </b-field>
  <b-field
    expanded
    label="Password"
  >
    <b-input
      v-model="password"
      type="password"
    />
  </b-field>
</b-field>
```

10. Finally, create a `b-field` component as a sibling of the `b-field` component, with the `position` attribute defined as `is-right` and the `grouped` attribute. Then, create two `div` HTML element with the `control` class. In the first `div` HTML element, add a `button` HTML element as a child with the `button is danger is-light` class, and in the second `div` HTML element, create a child `button` HTML element with the `button is-success` class:

```
<b-field
  position="is-right"
  grouped
>
  <div class="control">
    <button class="button is-danger is-light">Cancel</button>
  </div>
```

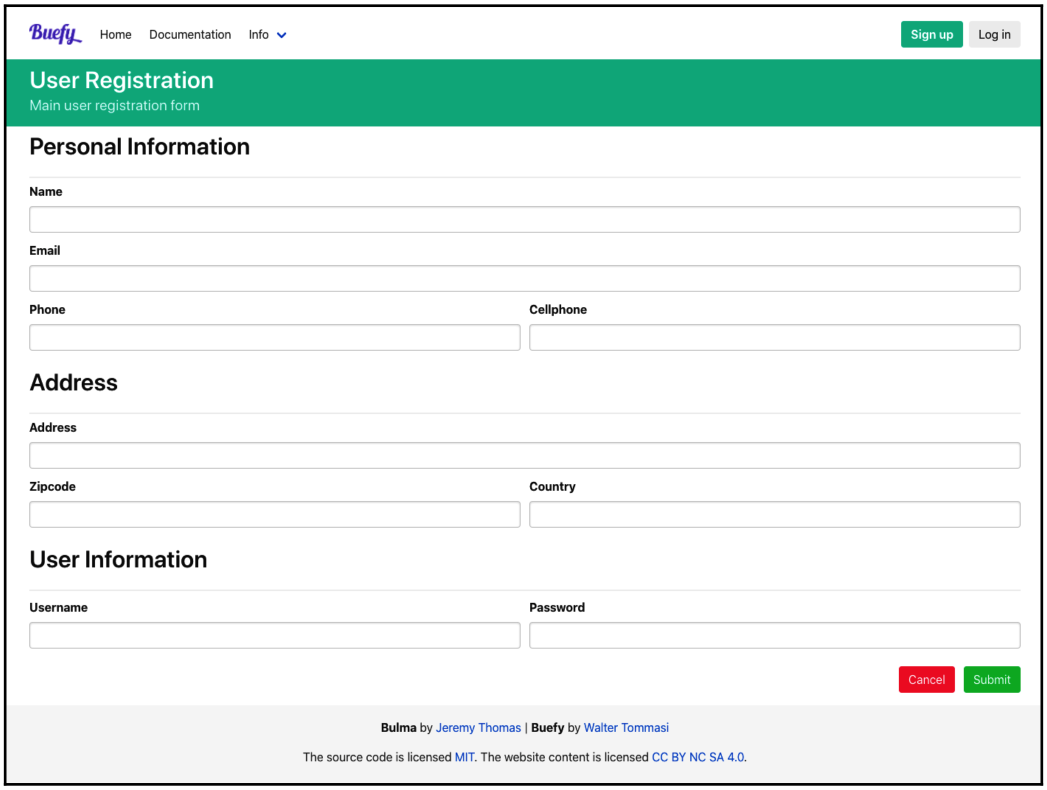
```
          <div class="control">
            <button class="button is-success">Submit</button>
          </div>
        </b-field>
```

How it works...

First, we create a Vue-CLI project, with the basic configurations, and the additional CSS pre-processor `node-sass`. Then, we were able to add the Buefy framework to our project, using Vue-CLI and the Buefy plugin. Using the Buefy framework, we created a layout page component, with a header menu component and a footer component.

For the page, we used the Bulma CSS container structure to define our page, and place our user registration form on a default grid size. Then, we added the hero section component, for the page identification, and finally, we created the user registration form and inputs.

Here is a screenshot of the final project up and running:



User Registration

Main user registration form

Personal Information

Name

Email

Phone

Cellphone

Address

Address

Zipcode

Country

User Information

Username

Password

Bulma by Jeremy Thomas | **Buefy** by Walter Tommasi

The source code is licensed MIT. The website content is licensed CC BY NC SA 4.0.

See also

Find more information about Bulma at `http://bulma.io/`.

Find more information about Buefy at `https://buefy.org/`.

Creating a page, a layout, and a user form with Vuetify

Vuetify is on the top three list of the most well-known UI frameworks for Vue. Based on Material Design by Google, this framework was initially designed by John Leider and is now gathering ground in the Vue ecosystem as the go-to UI framework for the initial development of an application.

In this recipe, we will learn how to use Vuetify to create a layout component wrapper, a page, and a user registration form.

Getting ready

The pre-requisites for this recipe are as follows:

- Node.js 12+
- A Vue-CLI project

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

We will divide this recipe into four main sections. The first and second sections are dedicated to the creation of the project and the installation of the framework and the last two sections are dedicated to the creation of the user registration page and the components needed to create it.

Creating the Vue-CLI project

To use Vuetify with a Vue-CLI project, we need to create a custom Vue-CLI project with pre-defined configurations, so that we are able to take full advantage of the framework and the styling options it provides:

1. We need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create vue-vuetify
```

2. First, Vue-CLI will ask for you to choose a preset; select `Manually select features` using the space bar:

```
? Please pick a preset: (Use arrow keys)
  default (babel, eslint)
❯ Manually select features
```

3. Now Vue-CLI will ask for the features, and you will need to select `CSS Pre-processors` as an additional feature on top of the default ones:

```
? Check the features needed for your project: (Use arrow keys)
❯ Babel
  TypeScript
  Progressive Web App (PWA) Support
  Router
  Vuex
❯ CSS Pre-processors
❯ Linter / Formatter
  Unit Testing
  E2E Testing
```

4. Here, Vue-CLI will ask which `CSS pre-processor` you want to use; select `Sass/SCSS (with node-sass)`:

```
? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules
  are supported by default): (Use arrow keys)
  Sass/SCSS (with dart-sass)
❯ Sass/SCSS (with node-sass)
  Less
  Stylus
```

5. Continue this process by selecting a linter and formatted. In our case, we will select the `ESLint + Airbnb` config:

```
? Pick a linter / formatter config: (Use arrow keys)
  ESLint with error prevention only
❯ ESLint + Airbnb config
  ESLint + Standard config
  ESLint + Prettier
```

6. Choose the additional features of the linter (here, `Lint and fix on commit`):

```
? Pick additional lint features: (Use arrow keys)
  Lint on save
❯ Lint and fix on commit
```

7. Select where you want to place the linter and formatter configuration files (here, `In dedicated config files`):

```
? Where do you prefer placing config for Babel, ESLint, etc.? (Use
arrow keys)
❯ In dedicated config files
  In package.json
```

8. Finally, Vue-CLI will ask you whether you want to save the settings for future projects; you will select `N`. After that, Vue-CLI will create a folder and install the dependencies for you:

```
? Save this as a preset for future projects? (y/N) n
```

Adding Vuetify to the Vue-CLI project

To use Vuetify in a Vue project, we will use the Vue-CLI plugin installation of the framework:

1. In the folder that you created your Vue-CLI project in, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue add vuetify
```

2. Vue-CLI will ask whether you want to choose an installation preset. Choose the default preset. Then, Vue-CLI will finish the installation of the framework on the project:

```
? Choose a preset: (Use arrow keys)
❯ Default (recommended)
  Prototype (rapid development)
  Configure (advanced)
```

3. After the installation is finished, Vuetify will have configured the files inside your project to load the framework. Now you are ready to use it.

Creating the layout with Vuetify

Using Vuetify as the UI framework, we are using the Material Design guide as a base, because by using Material Design, we can follow a design guideline to create our design structure, which will mean a more appealing structure. You can find the Material Design guidelines at `https://material.io/design/introduction#goals`.

Creating the top bar component

In this part, we will create the `top-bar` component that will be used in the layout of our page:

1. In the `src/components` folder, create a file named `TopBar.vue` and open it.
2. In the `<script>` section of the single file component, we will export a `default` JavaScript object, with the `name` property defined as `TopBar`:

```
<script>
export default {
  name: 'TopBar',
};
</script>
```

3. Inside the `<template>` section, create a `v-app-bar` component with the `app`, `dark`, and `dense` attributes defined as `true`, and the `color` attribute defined as `primary`:

    ```
    <v-app-bar
     color="primary"
      app
      dark
      dense
    ></v-app-bar>
    ```

4. Inside the component, create a `v-app-bar-nav-icon` component with an event listener on the `click` event, sending an event `'open-drawer'` when the button is clicked:

    ```
    <v-app-bar-nav-icon
      @click="$emit('open-drawer')"
    />
    ```

5. Finally, as a sibling of the `v-app-bar-nav-icon` component, add a `v-toolbar-title` component with the title of the page or application:

    ```
    <v-toolbar-title>Vue.JS 3 Cookbook - Packt</v-toolbar-title>
    ```

Creating the drawer menu component

Inside a Material Design application, we have a drawer menu that pops up over the page. This menu will be opened when the user clicks on the button we have just created in the `TopBar` component:

1. In the `src/components` folder, create a file named `DrawerMenu.vue` and open it.
2. In the `<script>` section of the single file component, we will export a `default` JavaScript object with three properties:
    - The `name` property, defined as `DrawerMenu`.
    - The `props` property, defined as a JavaScript object, with a property called `value`. This property will be another JavaScript object, with the `type`, `required`, and `default` properties. The `type` property is defined as `Boolean`, the `required` property as `true`, and the `default` property as `false`.

- The `data` property, as a singleton function returning a JavaScript object. This object will have a `menu` property, which we will define as an array of the menu items that will be used. The array will contain Javascript objects with the `name`, then `link`, and `icon` properties. The first menu item will have `name` defined as `Home`, then `link` defined as `#`, and `icon` defined as `mdi-home`. The second menu item will have `name` defined as `Contact`, then `link` defined as `#`, and `icon` defined as `mdi-email`. Finally, the third menu item will have `name` defined as `Vuetify`, then `link` defined as `#`, and `icon` defined as `mdi-vuetify`:

```
<script>
export default {
  name: 'DrawerMenu',
  props: {
    value: {
      type: Boolean,
      required: true,
      default: false,
    },
  },
  data: () => ({
    menu: [
      {
        name: 'Home',
        link: '#',
        icon: 'mdi-home',
      },
      {
        name: 'Contact',
        link: '#',
        icon: 'mdi-email',
      },
      {
        name: 'Vuetify',
        link: '#',
        icon: 'mdi-vuetify',
      },
    ],
  }),
};
</script>
```

3. In the `<template>` section, create a `v-navigation-drawer` component with the `value` attribute as a variable attribute bound to the `value` props, the `app` attribute defined as `true`, and the event listener on the `click` event, sending an `'input'` event:

   ```
   <v-navigation-drawer
     :value="value"
     app
     @input="$emit('input', $event)"
   ></v-navigation-drawer>
   ```

4. Create a `v-list` component with the `dense` attribute defined as `true`. As a child element, create a `v-list-item` component and define the following:
   - The `v-for` directive iterating over the `menu` items.
   - The `key` attribute with `index` of the item menu.
   - The `link` attribute defined as `true`.
   - Inside `v-list-item`, create `v-list-item-action` with a `VIcon` child, with the inner text as `item.icon`.
   - Create, as a sibling of `v-list-item-action`, a `v-list-item-content` component with `v-list-item-title` as a child element, with `item.name` as the inner text:

     ```
     <v-list dense>
       <v-list-item
         v-for="(item, index) in menu"
         :key="index"
         link>
         <v-list-item-action>
           <v-icon>{{ item.icon }}</v-icon>
         </v-list-item-action>
         <v-list-item-content>
           <v-list-item-title>{{ item.name }}</v-list-item-
                                                      title>
         </v-list-item-content>
       </v-list-item>
     </v-list>
     ```

Creating the layout component

To create the layout component, we are going to use all the created components and add a slot that will hold the page content:

1. In the `src/components` folder, create a new file named `Layout.vue` and open it.
2. In the `<script>` section of the single file component, import the `top-bar` component and the `drawer-menu` component:

```
import TopBar from './TopBar.vue';
import DrawerMenu from './DrawerMenu.vue';
```

3. Then, we will export a `default` JavaScript object, with the `name` property defined as `Layout`, then create the `components` property with the imported components. Finally, add the `data` property as a singleton function returning a JavaScript object, with the `drawer` property with the value defined as `false`:

```
export default {
  name: 'Layout',
  components: {
    DrawerMenu,
    TopBar,
  },
  data: () => ({
    drawer: false,
  }),
};
```

4. Inside the `<template>` section, create a `v-app` component. As the first child, add the `top-bar` component, with the event listener on the `open-drawer` event listener, changing the `drawer` data property as the negation of the `drawer` property. Then, as a sibling of `top-bar`, create a `drawer-menu` component with the `v-model` directive bound to `drawer`. Finally, create a `v-content` component with a child `<slot>` element:

```
<template>
  <v-app>
    <top-bar
      @open-drawer="drawer = !drawer"
    />
    <drawer-menu
      v-model="drawer"
    />
```

```
        <v-content>
          <slot/>
        </v-content>
      </v-app>
    </template>
```

Creating the user registration form with Vuetify

Now, with the layout component ready, we will create the user registration form. Because Vuetify has built-in validation in forms, we will be using that to validate some fields in our form.

Single file component <script> section

Here, we will create the `<script>` section of the single file component:

1. In the `src` folder, open the `App.vue` file and clear its contents.
2. Import the `layout` component:

```
import Layout from './components/Layout.vue';
```

3. Then, we will export a `default` JavaScript object, with a `name` property defined as `App`, then define the `components` property with the imported component. Define the `computed` and `methods` properties as an empty JavaScript object. Then create a `data` property defined as a singleton function returning a JavaScript object. In the `data` property, create the following:
    - A `valid` property with the value defined as `false`;
    - A `name`, `username`, `password`, `email`, `phone`, `cellphone`, `address`, `zipcode`, and `country` properties defined as empty strings:

```
export default {
  name: 'App',

  components: {
    Layout,
  },

  data: () => ({
    valid: true,
    name: '',
    username: '',
    password: '',
    email: '',
    phone: '',
    cellphone: '',
    address: '',
    zipcode: '',
    country: '',
  }),
  computed: {},
  methods: {},
};
```

4. In the `computed` property, create a property called `nameRules`; this property is a function that returns an array, with an anonymous function that receives a value and returns the verification of the value or the error text. Do the same for the `passwordRules` and `emailRules` properties. In the `emailRules` property, add another anonymous function to the returned array that checks whether the value is a valid email through a regular expression, and if the value is not a valid email it returns the error message:

```
computed: {
  nameRules() {
    return [
      (v) => !!v || 'Name is required',
    ];
  },
  passwordRules() {
    return [
      (v) => !!v || 'Password is required',
    ];
  },
  emailRules() {
    return [
      (v) => !!v || 'E-mail is required',
      (v) => /.+@.+\..+/.test(v) || 'E-mail must be valid',
    ];
  },
},
```

5. Finally, inside the `methods` property, create a new property named `register` that is a function that calls the `$refs.form.validate` function. Also, create another property named `cancel` that is another function that calls the `$refs.form.reset` function:

```
methods: {
  register() {
    this.$refs.form.validate();
  },
      cancel() {
        this.$refs.form.reset();
      },
    },
```

Single file component <template> section

It's time to create the `<template>` section of the single file component:

1. In the `src` folder, open the `App.vue` file.
2. In the `<template>` section, create a `layout` component element, and add a `v-container` component as a child with the `fluid` attribute defined as `true`:

```
<layout>
  <v-container
    fluid
  >
  </v-container>
</layout>
```

3. Inside the `v-container` component, create a child HTML `h1` element with the page title and a sibling `v-subheader` component with the page description:

```
<h1>user Registration</h1>
<v-subheader>Main user registration form</v-subheader>
```

4. After that, create a `v-form` component with the `ref` attribute defined as `form` and the `lazy-validation` attribute as `true`. Then, the `v-model` directive of the component gets bound to the `valid` variable. Create a child `v-container` component with the `fluid` attribute defined as `true`:

```
<v-form
  ref="form"
  v-model="valid"
  lazy-validation
>
  <v-container
    fluid
  >
  </v-container>
</v-form>
```

5. Inside the `v-container` component, create a `v-row` component, and then add a `v-col` component as a child with the `cols` attribute defined as `12`. Inside the `v-col` component, create a `v-card` component with the `outlined` attribute and `flat` defined as `true`, and `class` defined as `mx-auto`:

```
<v-row>
  <v-col
    cols="12"
  >
    <v-card
      outlined
      flat
      class="mx-auto"
    >
    </v-card>
  </v-col>
</v-row>
```

6. As a child element of the `v-card` component, create a `v-card-title` component with the card title, then as a sibling element create a `v-divider` component. After that, create a `v-container` element with the `fluid` attribute defined as `true`. Inside the `v-container` element, create a child `v-row` component:

```
<v-card-title>
  Personal Information
</v-card-title>
<v-divider/>
<v-container
  fluid
>
  <v-row>
  </v-row>
</v-container>
```

7. Inside the `v-row` component, create a child `v-col` element with the `cols` attribute defined as `12`. Then inside the `v-col` component, create `v-text-field` with the `v-model` directive bound to the `name` variable, the `rules` variable attribute defined as the `nameRules` computed property, the `label` attribute defined as `Name`, and finally, the `required` attribute defined as `true`:

```
<v-col
  cols="12"
>
  <v-text-field
    v-model="name"
    :rules="nameRules"
    label="Name"
    required
  />
</v-col>
```

8. As a sibling of the `v-col` component, create another `v-col` component with the `cols` attribute defined as `12`. Then, add the `v-text-field` component as a child, with the `v-model` directive bound to the `email` variable, the `rules` variable attribute defined as the `emailRules` computed property, the `type` attribute as `email`, the `label` attribute as `E-mail`, and finally, the `required` attribute defined as `true`:

```
<v-col
  cols="12"
>
  <v-text-field
    v-model="email"
    :rules="emailRules"
    type="email"
    label="E-mail"
    required
  />
</v-col>
```

9. Create a `v-col` component as a sibling of the `v-col` component, and define the `cols` attribute as `6`. Then, add as a child component the `v-text-field` component, with the `v-model` directive bound to the `phone` variable and the `label` attribute defined as `Phone`. Do the same for the `Cellphone` input; you must change the `v-model` directive bound to the `cellphone` variable and the `label` to `Cellphone`*:*

```
<v-col
  cols="6"
>
  <v-text-field
    v-model="phone"
    label="Phone"
  />
</v-col>
<v-col
  cols="6"
>
  <v-text-field
    v-model="cellphone"
    label="Cellphone"
  />
</v-col>
```

10. Now we will create the `Address` card, as a sibling of `v-col` in the `v-row` component. Create a `v-col` component with the `cols` attribute defined as `12`. Inside the `v-col` component, create a `v-card` component with the `outlined` attribute and `flat` defined as `true`, and `class` defined as `mx-auto`. As a child element of the `v-card` component, create a `v-card-title` component with the card title; then, as a sibling element, create a `v-divider` component. After that, create a `v-container` element with the `fluid` attribute defined as `true`. Inside the `v-container` element, create a child `v-row` component:

```
<v-col
  cols="12"
>
  <v-card
    outlined
    flat
    class="mx-auto"
  >
    <v-card-title>
      Address
    </v-card-title>
```

```
        <v-divider/>
        <v-container
          fluid
        >
          <v-row>
          </v-row>
        </v-container>
      </v-card>
    </v-col>
```

11. Inside the `v-row` component in the `v-container` component, create a `v-col` component with the `cols` attribute defined as `12`. Then, add `v-text-field` as a child component with the `v-model` directive bound to the `address` variable and the `label` attribute defined as `Address`:

```
<v-col
  cols="12"
>
  <v-text-field
    v-model="address"
    label="Address"
  />
</v-col>
```

12. As a sibling element, create a `v-col` component with the `cols` attribute defined as `6`. Add a `v-text-field` component as a child. Define the `v-model` directive of the `v-text-field` component bound to the `zipcode` variable and the `label` attribute defined as `Zipcode`:

```
<v-col
  cols="6"
>
  <v-text-field
    v-model="zipcode"
    label="Zipcode"
  />
</v-col>
```

13. Then, create a `v-col` component with the `cols` attribute defined as `6`. Add a `v-text-field` component as a child with the `v-model` directive bound to the `country` variable and the `label` attribute defined as `Country`:

```
<v-col
  cols="6"
>
  <v-text-field
    v-model="country"
    label="Country"
  />
</v-col>
```

14. Now we will create the `user Information` card as a sibling of `v-col` in the `v-row` component. Create a `v-col` component with the `cols` attribute defined as `12`. Inside the `v-col` component, create a `v-card` component with the `outlined` attribute and `flat` defined as `true`, and `class` defined as `mx-auto`. As a child element of the `v-card` component, create a `v-card-title` component with the card title; then, as a sibling element, create a `v-divider` component. After that, create a `v-container` element with the `fluid` attribute defined as `true`. Inside the `v-container` element, create a child `v-row` component:

```
<v-col
  cols="12"
>
  <v-card
    outlined
    flat
    class="mx-auto"
  >
    <v-card-title>
      user Information
    </v-card-title>
    <v-divider/>
    <v-container
      fluid
    >
      <v-row>
      </v-row>
    </v-container>
  </v-card>
</v-col>
```

15. Inside the `v-row` component in the `v-container` component, create a `v-col` component with the `cols` attribute defined as `6`. Then, add `v-text-field` as a child component with the `v-model` directive bound to the `username` variable and the `label` attribute defined as `username`:

    ```
    <v-col
      cols="6"
    >
      <v-text-field
        v-model="username"
        label="username"
      />
    </v-col>
    ```

16. As a sibling create a `v-col` component with the `cols` attribute defined as `6`, and add a `v-text-field` component as a child with the `v-model` directive bound to the `password` variable, the `rules` variable attribute defined as the `passwordRules` computed property, and the `label` attribute defined as `Password`:

    ```
    <v-col
      cols="6"
    >
      <v-text-field
        v-model="password"
        :rules="passwordRules"
        label="Password"
        type="password"
        required
      />
    </v-col>
    ```

17. Now we will create the action buttons. As a sibling of the `v-col` on the top `v-row` component, create a `v-col` component with the `cols` attribute defined as `12` and the `class` attribute defined as `text-right`. Inside the `v-col` component, create a `v-btn` component with the `color` attribute defined as `error`, the `class` attribute as `mr-4`, and the `click` event listener attached to the `cancel` method. Finally, create a `v-btn` component as a sibling, with the `disabled` variable attribute as the negation of the `valid` variable, the `color` attribute as `success`, the `class` attribute as `mr-4`, and the `click` event listener attached to the `register` method:

    ```
    <v-col
      cols="12"
      class="text-right"
    ```

```
            >
              <v-btn
                color="error"
                class="mr-4"
                @click="cancel"
              >
                Cancel
              </v-btn>
              <v-btn
                :disabled="!valid"
                color="success"
                class="mr-4"
                @click="register"
              >
                Register
              </v-btn>
            </v-col>
```

How it works...

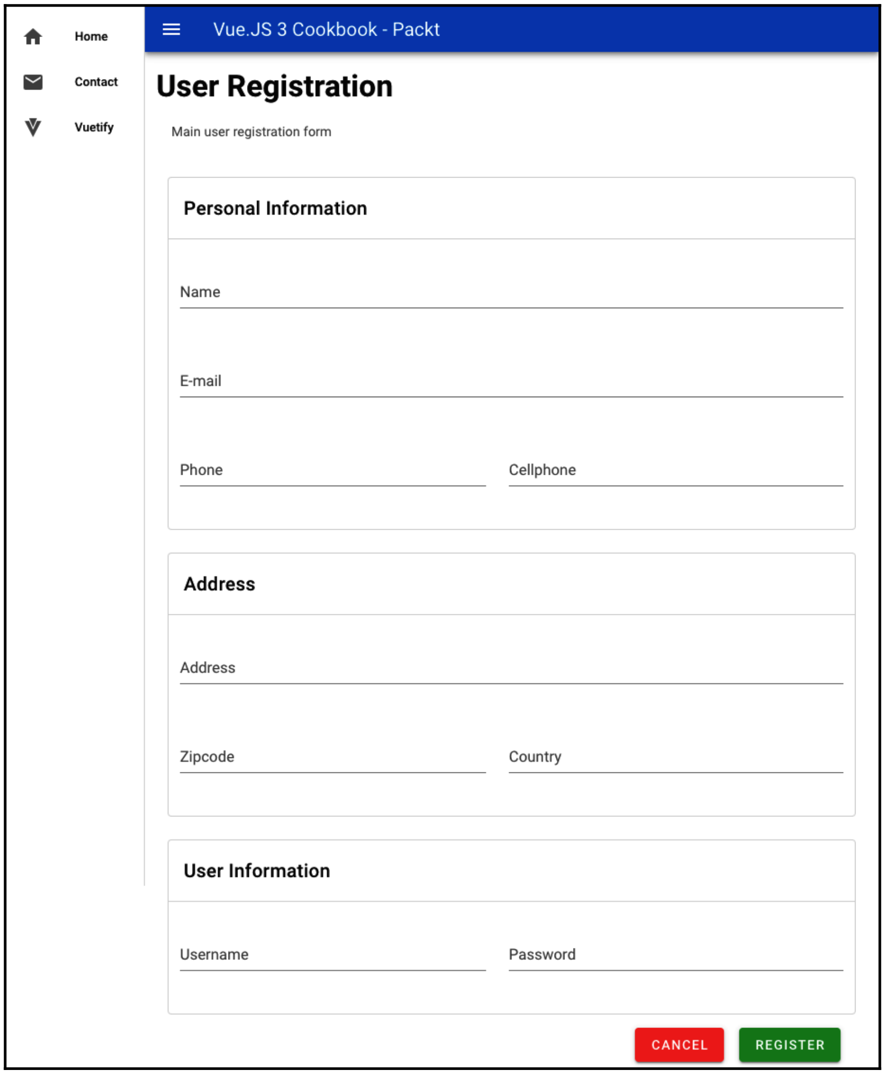
In this recipe, we learned how to create a user registration page with Vuetify and Vue-CLI. To create this page, we first needed to create the project using the Vue-CLI tool and then add the Vuetify plugin to it, so that the framework was available to be used.

Then, we created the `top-bar` component, which holds the application title and the menu button toggle. To use the menu, we created the `drawer-menu` component to hold the navigation items. Finally, we created the `layout` component to hold together the `top-bar` and `drawer-menu` components and added a `<slot>` component to place the page content.

For the user registration form page, we created three cards that hold the input forms, which were bound to the variables on the component. Some of the inputs on the form are attached to a set of validation rules that checks for required fields and email validation.

Finally, the user registration form is checked to see whether it's valid before sending the data to the server.

Here is a screenshot of the final project up and running:



See also

You can find more information about Vuetify at `https://vuetifyjs.com/en/`.

You can find more information about Material Design at `https://material.io/`.

Creating a page, a layout, and a user form with Ant-Design

The Ant-Design framework was created by the AliBaba group, specifically by the tech team behind AliPay and Ant Financial. It's an ecosystem design that is being mainly used by Asian tech giants and it has a large presence in the React and Vue communities. Focused on the back office UI, the main core of the framework is its solutions for custom data inputs and data tables.

Here, we will learn how to create a user registration form using the Ant-Design and Vue, by creating a top bar component, a drawer menu, a layout wrapper, and a user registration page with a form.

Getting ready

The pre-requisites for this recipe are as follows:

- Node.js 12+
- A Vue-CLI project

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

In this recipe, we will create a user registration form using the Ant-Design framework. To do so, we will create a layout wrapper and the components needed for the wrapper, and finally, we will create the page that will hold the user registration form.

Creating the Vue-CLI project

We need to create a Vue-CLI project to be able to install the Ant-Design plugin and start developing the user registration form:

1. We need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create antd-vue
```

2. First, Vue-CLI will ask for you to choose a preset; select `Manually select features` using the space bar:

```
? Please pick a preset: (Use arrow keys)
  default (babel, eslint)
❯ Manually select features
```

3. Now Vue-CLI will ask for the features, and you will need to select `CSS Pre-processors` as an additional feature on top of the default ones:

```
? Check the features needed for your project: (Use arrow keys)
❯ Babel
  TypeScript
  Progressive Web App (PWA) Support
  Router
  Vuex
❯ CSS Pre-processors
❯ Linter / Formatter
  Unit Testing
  E2E Testing
```

4. Here, Vue-CLI will ask which `CSS pre-processor` you want to use; select `Less`:

```
? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules
  are supported by default): (Use arrow keys)
  Sass/SCSS (with dart-sass)
  Sass/SCSS (with node-sass)
❯ Less
  Stylus
```

5. Continue this process by selecting a linter and formatted. In our case, we will select `ESLint + Airbnb` config:

```
? Pick a linter / formatter config: (Use arrow keys)
  ESLint with error prevention only
❯ ESLint + Airbnb config
```

```
    ESLint + Standard config
    ESLint + Prettier
```

6. Choose the additional features of the linter (here, `Lint on save`):

```
  ? Pick additional lint features: (Use arrow keys)
    Lint on save
  ❯ Lint and fix on commit
```

7. Select where you want to place the linter and formatter configuration files (here, `In dedicated config files`):

```
  ? Where do you prefer placing config for Babel, ESLint, etc.? (Use
    arrow keys)
  ❯ In dedicated config files
    In package.json
```

8. Finally, the CLI will ask you whether you want to save the settings for future projects; you should select `N`. After that, Vue-CLI will create a folder and install the dependencies for you:

```
  ? Save this as a preset for future projects? (y/N) n
```

Adding Ant-Design to the Vue-CLI project

To add the Ant-Design framework to a Vue-CLI project, we need to use the Vue-CLI plugin to install the framework as a project dependency and have it available in the global scope of the application:

1. In the folder that you created your Vue-CLI project in, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
  > vue add ant-design
```

2. Vue-CLI will ask you how the import of the Ant-Design component should occur; we will select the `Fully import` option:

```
  ? How do you want to import Ant-Design-Vue?
  ❯ Fully import
    Import on demand
```

3. Vue-CLI will ask you whether you want to overwrite the Ant-Design `LESS` variables; we will choose `N`:

```
? Do you wish to overwrite Ant-Design-Vue's LESS variables? (y/N)
```

4. Finally, Vue-CLI will ask about the main language Ant-Design will use in the project; we will select `en_US`:

```
? Choose the locale you want to load
❯ en_US
  zh_CN
  zh_TW
  en_GB
  es_ES
  ar_EG
  bg_BG
(Move up and down to reveal more choices)
```

Creating the layout with Ant-Design

To be able to create a user registration form, we will create a base layout that will wrap the page content and the form. Here, we will create the `top-bar` component, the `drawer-menu` component, and the `layout` wrapper.

Creating the top-bar component

In the `layout` wrapper, we will have a `top-bar` component that will hold the breadcrumbs for where the user currently is. Now we will create the `top-bar` component and make it available for the layout:

1. In the `src/components` folder, create a new file called `TopBar.vue` and open it.
2. In the `<script>` section of the single file component, we will export a `default` JavaScript object, with a `name` property defined as `TopBar`:

```
<script>
export default {
  name: 'TopBar',
};
</script>
```

3. In the `<style>` section of the single file component, we will make the `<style>` section `scoped` and create a class named `header-bread`. Now, `background-color` will be defined as `#f0f2f5` with a class named `bread-menu` with the margin defined as `16px 0`:

```
<style scoped>
  .header-bread {
    background-color: #f0f2f5;
  }

  .bread-menu {
    margin: 16px 0;
  }
</style>
```

4. In the `<template>` section of the single file component, we will create an `a-layout-component` component with the `class` attribute defined as `header-bread`. Also, add an `a-breadcrumb` component as a child element with the `class` attribute as `bread-menu`:

```
<template>
  <a-layout-header class="header-bread">
    <a-breadcrumb class="bread-menu">
    </a-breadcrumb>
  </a-layout-header>
</template>
```

5. As a child of the `a-breadcrumb` component, create two `a-breadcrumb-item` components and add to each the directions for the user's location. In our case, the first one will be `user` and the second `Registration Form`:

```
<a-breadcrumb-item>user</a-breadcrumb-item>
<a-breadcrumb-item>Registration Form</a-breadcrumb-item>
```

Creating the drawer menu

In the layout design, we will have a drawer menu component as a navigation menu for the user. Here we will create the `Drawer` component:

1. In the `src/components` folder, create a file named `Drawer.vue` and open it.

2. In the `<script>` section of the single file component, we will export a `default` JavaScript object with two properties. The `name` property, defined as `Drawer`, and the `data` property, as a `singleton` function returning a JavaScript object. In the `data` property, create the following:
    - A `drawer` property defined as `false`.
    - A `menu` property, which we will define as an array of the menu items that will be used. The menu array will have three JavaScript objects with the `name` and `icon` properties. This array will have:
        - A JavaScript object with the properties `name` defined as `Home` and `icon` defined as `home`
        - A JavaScript object with the properties `name` defined as `Ant Design Vue` and `icon` defined as `ant-design`
        - A JavaScript object with the properties `name` defined as `Contact` and `icon` defined as `mail`:

```
<script>
export default {
  name: 'Drawer',
  data: () => ({
    drawer: false,
    menu: [
      {
        name: 'Home',
        icon: 'home',
      },
      {
        name: 'Ant Design Vue',
        icon: 'ant-design',
      },
      {
        name: 'Contact',
        icon: 'mail',
      },
    ],
  }),
};
</script>
```

3. In the `<template>` section of the single file component, create an `a-layout-sider` component, with the `v-model` directive bound to the `drawer` variable and the `collapsible` attribute defined as `true`. As a child, create a `a-menu` component with the `default-selected-keys` variable attribute defined as `['1']`, the `theme` attribute defined as `dark`, and the `mode` attribute as `inline`:

```
<template>
  <a-layout-sider
    v-model="drawer"
    collapsible
  >
    <a-menu
      :default-selected-keys="['1']"
      theme="dark"
      mode="inline"
    >
    </a-menu>
  </a-layout-sider>
</template>
```

4. Finally, inside the `a-menu` component, create an `a-menu-item` component, with the `v-for` directive iterating over the `menu` variable, and create the `item` and `index` temporary variables. Then, define the `key` variable attribute as `index`. Create a child `AIcon` component with the `type` variable attribute as `item.icon` with a sibling `span` HTML element and the content as `item.name`:

```
<a-menu-item
  v-for="(item,index) in menu"
  :key="index"
>
  <a-icon
    :type="item.icon"
  />
  <span>{{ item.name }}</span>
</a-menu-item>
```

Creating the layout component

Here, we will create the `layout` component. This component will join together the `top-bar` component and the `Drawer` menu component to make a wrapper for the user registration page:

1. In the `src/components` folder, create a new file named `Layout.vue` and open it.

2. In the `<script>` section of the single file component, import the `top-bar` component and the `drawer-menu` component:

```
import TopBar from './TopBar.vue';
import Drawer from './Drawer.vue';
```

3. Then, we will export a `default` JavaScript object, with a `name` property, defined as `layout`. Then define the `components` property with the imported components.

```
export default {
  name: 'layout',
  components: {
    Drawer,
    TopBar,
  },
};
```

4. In the `<template>` section of the single file component, create an `a-layout` component with the `style` attribute defined as `min-height: 100vh`. Then, add the `Drawer` component as a child. As a sibling of the `drawer` component, create an `a-layout` component:

```
<template>
  <a-layout
    style="min-height: 100vh"
  >
    <drawer />
    <a-layout>
      <top-bar />
      <a-layout-content style="margin: 0 16px">
        <div style="padding: 24px; background: #fff;
            min-height: auto;">
          <slot />
        </div>
      </a-layout-content>
      <a-layout-footer style="text-align: center">
        Vue.js 3 Cookbook | Ant Design ©2020 Created by Ant UED
      </a-layout-footer>
    </a-layout>
  </a-layout>
</template>
```

5. To the `a-layout` component, add the `top-bar` component and a sibling `a-layout-content` component with the `style` attribute defined as `margin: 0 16px`. As a child of that component, create a `div` HTML element with the `style` attribute defined as `padding: 24px; background: #fff; min-height: auto;`, and add a `slot` placeholder. Finally, create an `a-layout-footer` component with the `style` attribute defined as `text-align:center;` with the footer text of the page:

```
<top-bar />
<a-layout-content style="margin: 0 16px">
  <div style="padding: 24px; background: #fff; min-height: auto;">
    <slot />
  </div>
</a-layout-content>
<a-layout-footer style="text-align: center">
  Vue.js 3 Cookbook | Ant Design ©2020 Created by Ant UED
</a-layout-footer>
```

Creating the user registration form with Ant-Design

Now we will create the user registration page and form that will be placed inside the layout that was created in the preceding steps.

Single file component <script> section

Here we will create the `<script>` section of the single file component:

1. In the `src` folder, open the `App.vue` file and clear its contents.
2. Import the `layout` component:

```
import Layout from './components/Layout.vue';
```

3. Then, we will export a `default` JavaScript object, with the `name` property defined as `App`, define the `components` property with the imported component, and finally define the `data` property as a singleton function returning a JavaScript object. In the `data` property, create the following:
    - A `labelCol` property defined as a JavaScript object, with the `span` property and the value `4`.

- A `wrapperCol` property defined as a JavaScript object, with the `span` property and the value `14`.
- A `form` property defined as a JavaScript object, with the `name`, `username`, `password`, `email`, `phone`, `cellphone`, `address`, `zipcode`, and `country` properties all defined as empty strings:

```
export default {
  name: 'App',
  components: {
    Layout,
  },
  data() {
    return {
      labelCol: { span: 4 },
      wrapperCol: { span: 14 },
      form: {
        name: '',
        username: '',
        password: '',
        email: '',
        phone: '',
        cellphone: '',
        address: '',
        zipcode: '',
        country: '',
      },
    };
  },
};
```

Single file component <template> section

It's time to create the `<template>` section of the single file component:

1. In the `src` folder, open the `App.vue` file.
2. In the `<template>` section, create a `layout` component element and add an `a-form-model` component as a child with the `model` variable attribute bound to `form`, the `label-col` variable attribute bound to `labelCol`, and the `wrapper-col` variable attribute bound to `wrapperCol`:

```
<layout>
  <a-form-model
    :model="form"
    :label-col="labelCol"
    :wrapper-col="wrapperCol"
  >
  </a-form-model>
</layout>
```

3. Then, as a sibling of the `layout` component, create an `h1` HTML element with the page title `User Registration`, and a `p` HTML element with the `Main user registration form` page subtitle. Then, create an `a-card` element with the `title` attribute defined as `Personal Information`:

```
<h1>
  User Registration
</h1>
<p>Main user registration form</p>
<a-card title="Personal Information"></a-card>
```

4. In the `a-card` component, create an `a-form-model-item` component as a child element with the `label` attribute defined as `Name`, and add a child `a-input` component with the `v-model` directive bound to the `form.name` variable:

```
<a-form-model-item label="Name">
  <a-input v-model="form.name" />
</a-form-model-item>
```

5. Next, as a sibling, create an `a-form-model-item` component with the `label` attribute defined as `Email` and add a child `a-input` component with the `v-model` directive bound to the `form.email` variable and the `type` attribute defined as `email`:

```
<a-form-model-item label="Email">
  <a-input
    v-model="form.email"
    type="email"
  />
</a-form-model-item>
```

6. Create an element an `a-form-model-item` component with the `label` attribute defined as `Phone`, and add a child `a-input` component with the `v-model` directive bound to the `form.phone` variable:

```
<a-form-model-item label="Phone">
  <a-input v-model="form.phone" />
</a-form-model-item>
```

7. Create an `a-form-model-item` component with the `label` attribute defined as `Cellphone`, and add a child `a-input` component with the `v-model` directive bound to the `form.cellphone` variable:

```
<a-form-model-item label="Cellphone">
  <a-input v-model="form.cellphone" />
</a-form-model-item>
```

8. As a sibling of the `a-card` component, create an `a-card` component with the `title` attribute defined as `Address` and the `style` attribute as `margin-top: 16px;`. Then, add a child `a-form-model-item` component with the `label` attribute defined as `Address`, and add a child `a-input` component with the `v-model` directive bound to the `form.address` variable:

```
<a-card title="Address" style="margin-top: 16px">
  <a-form-model-item label="Address">
    <a-input v-model="form.address" />
  </a-form-model-item>
</a-card>
```

9. Next, as a sibling of the `a-card` component, create an `a-form-model-item` component with the `label` attribute defined as `Zipcode`, and add a child `a-input` component with the `v-model` directive bound to the `form.zipcode` variable:

```
<a-form-model-item label="Zipcode">
  <a-input v-model="form.zipcode" />
</a-form-model-item>
```

10. Create an `a-form-model-item` component with the `label` attribute defined as `Country`, and add a child `a-input` component with the `v-model` directive bound to the `form.country` variable:

```
<a-form-model-item label="Country">
  <a-input v-model="form.country" />
</a-form-model-item>
```

11. As a sibling of the `a-card` component, create an `a-card` component with the `title` attribute defined as `User Information` and the `style` attribute as `margin-top: 16px;`. Then, add a child `a-form-model-item` component with the `label` attribute defined as `username`, and add a child `a-input` component with the `v-model` directive bound to the `form.username` variable:

```
<a-card title="user Information" style="margin-top: 16px">
  <a-form-model-item label="username">
    <a-input v-model="form.username" />
  </a-form-model-item>
</a-card>
```

12. Create an `a-form-model-item` component with the `label` attribute defined as `Password`, and add a child `a-input-password` component with the `v-model` directive bound to the `form.password` variable, the `visibility-toggle` attribute defined as `true`, and the `type` attribute defined as `password`:

```
<a-form-model-item label="Password">
  <a-input-password
    v-model="form.password"
    visibility-toggle
    type="password"
  />
</a-form-model-item>
```

13. Finally, as a sibling of the `a-card` component, create `a-form-model-item` with the `wrapper-col` variable attribute defined as a JavaScript object, `{span: 14, offset: 4}`. Then, add a child `a-button` with `type` defined as `primary` with the text `Create` and another `a-button` with the `style` attribute defined as `margin-left: 10px;` and the text `Cancel`:

```
<a-form-model-item :wrapper-col="{ span: 14, offset: 4 }">
  <a-button type="primary">
    Create
  </a-button>
  <a-button style="margin-left: 10px;">
    Cancel
  </a-button>
</a-form-model-item>
```

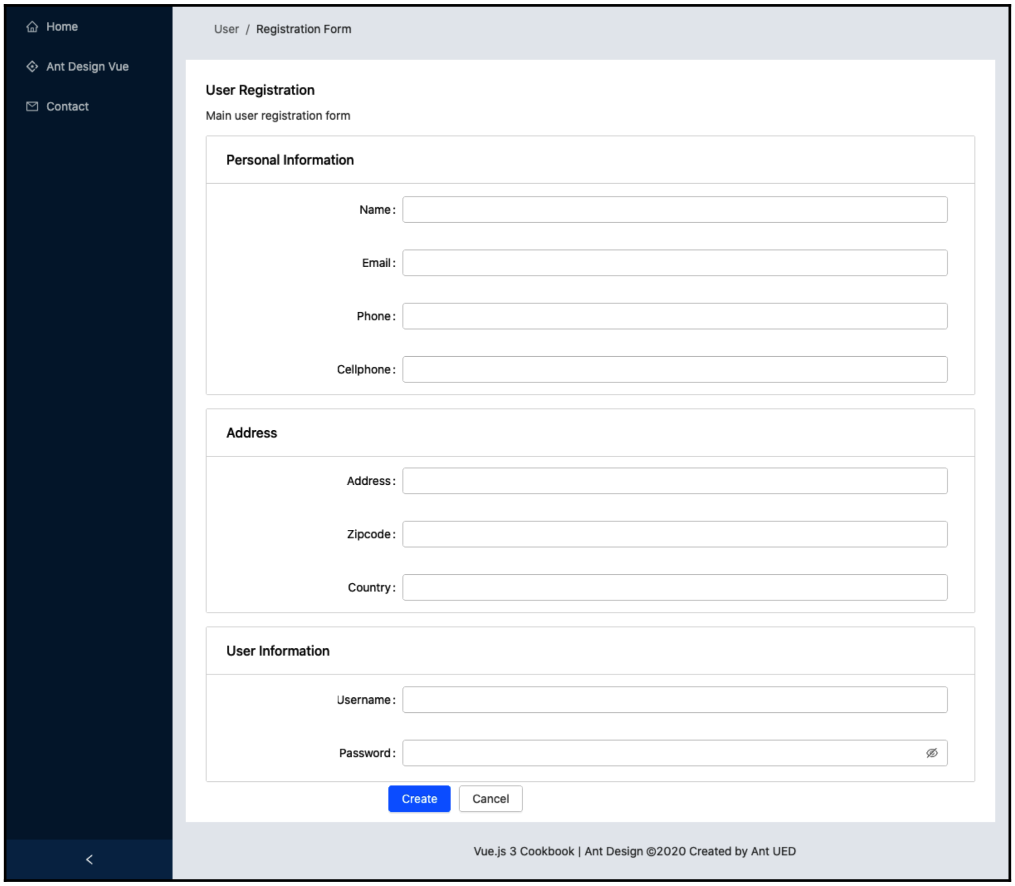
How it works...

In this recipe, we learned how to create a user registration page with Ant-Design and Vue-CLI. To create this page, we first needed to create a project using Vue-CLI and add the Ant-Design of Vue plugin to it, so that the framework was available to be used.

Then, we created the `top-bar` component, which holds the navigation breadcrumbs. For user navigation, we created a custom `Drawer` component that has an inline toggle button at the bottom. Finally, we created the `layout` component to hold together both the components and we added a `<slot>` component to place the page content.

Finally, we created the user registration form page, with three cards that hold the input forms that are bound to the variables on the component.

Here is a screenshot of the final project up and running:



User Registration

Main user registration form

Personal Information

Name :

Email :

Phone :

Cellphone :

Address

Address :

Zipcode :

Country :

User Information

Username :

Password :

Vue.js 3 Cookbook | Ant Design ©2020 Created by Ant UED

See also

You can find more information about Ant-Design and Vue at `https://vue.ant.design/`.

10
Deploying an Application to Cloud Platforms

Now it's time to deploy your application to the World Wide Web and make it available to everyone across the globe.

In this chapter, we will learn how to do it with three different hosting platforms – Netlify, Now, and Firebase. Here, we will learn the process of creating the account on each platform, setting up the environment, configuring the application for deployment, and finally deploying it to the web.

In this chapter, we'll cover the following recipes:

- Creating a Netlify account
- Preparing your application for deployment in Netlify
- Preparing for automatic deployment on Netlify with GitHub
- Creating a Vercel account
- Configuring the Vercel-CLI and deploying your project
- Preparing for automatic deployment on Vercel with GitHub
- Creating a Firebase project
- Configuring the Firebase-CLI and deploying your project

Technical requirements

In this chapter, we will be using **Node.js** and **Vue-CLI**.

Attention Windows Users! You need to install an NPM package called windows-build-tools in order to be able to install the following requisite packages. To do this, open PowerShell as the administrator and execute the following command: `> npm install -g windows-build-tools`

To install Vue-CLI, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install -g @vue/cli @vue/cli-service-global
```

Creating a Vue project

To create a Vue-CLI project, follow these steps:

1. Open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create vue-project
```

2. Vue-CLI will ask for you to choose a preset; select `Manually select features` using the *spacebar*:

```
? Please pick a preset: (Use arrow keys)
  default (babel, eslint)
❯ Manually select features
```

3. Now, Vue-CLI will ask for the features, and you will need to select `Router`, `Vuex`, and `Linter / Formatter` as an additional feature on top of the default ones:

```
? Check the features needed for your project: (Use arrow keys)
❯ Babel
  TypeScript
  Progressive Web App (PWA) Support
❯ Router
❯ Vuex
  CSS Pre-processors
❯ Linter / Formatter
  Unit Testing
  E2E Testing
```

4. Now, Vue-CLI will ask whether you want to use history mode for route management. We will choose `y` (yes):

    ```
    ? Use history mode for router? (Requires proper server setup for
      index fallback in production) (Y/n) y
    ```

5. Continue the process by selecting the linter and formatter. We will select `ESLint + Airbnb config`:

    ```
    ? Pick a linter / formatter config: (Use arrow keys)
      ESLint with error prevention only
    ❯ ESLint + Airbnb config
      ESLint + Standard config
      ESLint + Prettier
    ```

6. Choose the additional features of the linter (here, `Lint and fix on commit`):

    ```
    ? Pick additional lint features: (Use arrow keys)
      Lint on save
    ❯ Lint and fix on commit
    ```

7. Select where you want to place the linter and formatter configuration files (here, `In dedicated config files`):

    ```
    ? Where do you prefer placing config for Babel, ESLint, etc.? (Use
    arrow keys)
    ❯ In dedicated config files
      In package.json
    ```

8. Finally, the CLI will ask you whether you want to save the settings for future projects; you will select `N`. After that, Vue-CLI will create the folder and install the dependencies for you:

    ```
    ? Save this as a preset for future projects? (y/N) n
    ```

Creating a Netlify account

It's time to start the deployment process to the Netlify platform. In this recipe, we will learn how to create our Netlify account so that we can deploy our application to the web.

Getting ready

The prerequisites for this recipe are as follows:

- An email address
- A GitHub account
- A GitLab account
- A BitBucket account

In the process of creating an account on Netlify, you can do this with an **email** address, a **GitHub** account, a **GitLab** account, or a **BitBucket** account.

How to do it...

Here, we will learn how to create a Netlify account with an email address:

1. Go to the Netlify website at `https://www.netlify.com/` and click on **Sign up →** in the header menu. You will be redirected to the initial **Sign up** page.
2. On this page, you can select the method that you want to use to sign up to Netlify. In this process, we will continue with the email address. Click on the **Email** button to be redirected to the **Email Sign up** form.
3. Fill in the form with your email address and a password of your choosing. There is a password rule of 8 characters minimum. After completing the form, click on the **Sign up** button. From there, you will be redirected to the **Success** page.
4. Now, you will receive a verification email in your inbox that you need in order to continue using the Netlify platform. To continue, open your email inbox and check the Netlify email.
5. In your email inbox, open the Netlify email and then click on the **Verify email** button. At this point, a new window will open, and you will be able to log in with the recently registered email and password.
6. Here, you can complete the login form with your email address and the password you chose at step 3. After that, click on the **Log in** button to be redirected to the main window of the Netlify platform.
7. Finally, you will find yourself at the main screen of the Netlify platform, with a blank page to begin deployment on the platform.

How it works...

In this recipe, we learned how to create our account on Netlify. We saw that it is possible to do this with various OAuth methods and the basic email that we used in the recipe.

The email address creation process involves defining the email address that will be used and a password for the account, verifying the account email. Then, you can log in to the platform.

See also

Find out more information about Netlify at `https://docs.netlify.com/`.

Preparing your application for deployment in Netlify

To start the deployment process, we need to configure our project to have a valid Netlify deployment schema. In this recipe, you will learn how to set up the Netlify deployment schema on any Vue-based application.

Getting ready

The prerequisites for this recipe are as follows:

- Node.js 12+
- A Vue project

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

How to do it...

In this recipe, we will learn how to prepare our application to be deployed to Netlify:

1. Open your Vue project and open the `package.json` file. Check whether you have the `build` script defined, as in the following example:

```
"scripts": {
  "serve": "Vue-CLI-service serve",
  "build": "Vue-CLI-service build",
  "lint": "Vue-CLI-service lint"
},
```

2. Open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm run build
```

3. Make sure your application `build` script creates a `dist` folder in the main folder.
4. If your `vue-router` is defined to work with history mode, you have to create a `_redirects` file in the `public` folder. In this file, you need to add the instruction to the Netlify router:

```
# Netlify settings for single-page application
/*    /index.html   200
```

5. Publish your application to a GitHub repository. Don't worry about the build folder, because it's already declared on the `.gitignore` file and it won't be sent to your repository.

How it works...

In this recipe, we learned how to check and prepare our application for the Netlify deployment.

To make the deployment work, we needed to make sure that we have the `build` command in the script section at `package.json`, and verify that the build destination is the `dist` folder.

Finally, we created a `_redirects` file in the public folder to instruct the Netlify router to understand the vue-router history mode.

See also

Find out more information about the official Vue-CLI documentation on Netlify deployment at `https://cli.vuejs.org/guide/deployment.html#netlify`.

Find out more information about Netlify router rewrites at `https://docs.netlify.com/routing/redirects/rewrites-proxies/#history-pushstate-and-single-page-apps`.

Preparing for automatic deployment on Netlify with GitHub

It's time to prepare the ground for deployment. In this recipe, you will learn how to set up the Netlify deployment process to fetch your application automatically on GitHub and deploy it.

Getting ready

The prerequisites for this recipe are as follows:

- A Netlify account
- A Vue project
- A GitHub account

How to do it...

Finally, following the creation of your Netlify account, having published your project on a GitHub repository, and having configured everything, it's time to prepare the Netlify platform to perform automatic deployment on each GitHub push:

1. Go to Netlify (`https://www.netlify.com/`), sign in, and open your initial dashboard. There, you will find a **New site from Git** button. You will be redirected to the **Create new site** page.
2. Now you may click on the **GitHub** button to open a new window for the Netlify authorization on GitHub and continue the process there.
3. Sign in with your GitHub account and then you will be redirected to the **Application install** page.

4. On this page, you can choose to give access to Netlify to all of your repositories or just the selected one, but make sure you make available the repository of your application.
5. When you finish the installation of Netlify on GitHub, the repository that you gave access to in the previous step will be available to be selected on the Netlify platform. Choose the one that contains your application.
6. To finish the creation process, you need to select the branch that will be used for auto-deployment. Then, you need to fill up the build command used on the application, in our case, `npm run build`. Open the folder that will contain the built files, in our case, this is the `dist` folder, and click on the **Deploy site** button.
7. Finally, the Netlify-CLI will start the building process and publish your application when the build is finished without any errors.

How it works...

The Netlify platform connects to your GitHub account and installs as an application, giving access to selected repositories. Then, on the platform, you can select the repository that you want to use to deploy. With the repository selected, we needed to configure the Netlify-CLI with the build instructions and the built destination folder. Finally, the CLI runs, and we have our application up and running on the web.

See also

Find out more information about advanced Netlify deployments at `https://docs.netlify.com/configure-builds/file-based-configuration/`.

Creating a Vercel account

Vercel is a famous platform for deploying your application on the web. With Vercel, you can automate the deployment process with GitHub, GitLab, and BitBucket. In this recipe, we will learn how to create our account on the Vercel platform.

Getting ready

The prerequisite for this recipe is just one of the following options:

- A GitHub account
- A GitLab account
- A BitBucket account

How to do it...

Let's start our journey on the Vercel platform. Here, we will learn how to create our account on the platform to start our project deployment:

1. Open the Vercel website (`https://vercel.com/`) and click on the **Sign Up** button on the top bar. You will be redirected to the **Sign Up** page.
2. Here, you have the option to select one of these repository managers – GitHub, GitLab, or BitBucket. We will continue by clicking on the **GitHub** button. After choosing the sign-up method, you will be redirected to the authorization page.
3. On this page, you are giving access to the Vercel platform to access the information on your account. By clicking on the **Authorize** button, you will be redirected back to the Vercel dashboard.
4. Finally, you have your Vercel account created and ready to be used.

How it works...

In this recipe, we entered the Vercel platform, and signed up to it using a repository manager. We were able to create our account, and can now start the deployment process on the platform through repository integration or the CLI tool.

See also

You can find out more information about Vercel at `https://vercel.com/`.

Configuring the Vercel-CLI and deploying your project

You have created a Vercel account. Now it's time to configure the Vercel-CLI on your project, so it's available on the Vercel platform and on the web.

Getting ready

The prerequisites for this recipe are as follows:

- A Vercel account
- A Vue project
- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`
- `vercel`

To install `vercel`, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm i -g vercel
```

How to do it...

In this recipe, we will learn how to link our project to the Vercel platform through the Vercel-CLI and then deploy the platform with it:

1. Open your Vue project and then open the `package.json` file. Check whether you have the `build` script defined, as in the following example:

```
"scripts": {
  "serve": "Vue-CLI-service serve",
  "build": "Vue-CLI-service build",
  "lint": "Vue-CLI-service lint"
},
```

2. Make sure your application build script creates a `dist` folder in the main folder.

3. In your `project` folder, open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> vercel
```

This will prompt you for a login to the Vercel platform:

```
> No existing credentials found. Please log in:
  Enter your email:
```

4. Enter the email address that is linked to the repository manager that you have used to sign in to Vercel. You will receive an email with a **Verify** button; click on it to verify your email address:
5. Once your email is verified, you are able to deploy applications in your Terminal with the `> vercel` command.
6. To deploy your application to the web, we need to execute the `> vercel` command in the `project` folder, and it will ask some questions about the project settings prior to deployment. The first question will relate to the project path:

```
? Set up and deploy "~/Versionamento/Vue.js-3.0-Cookbook/chapter-
  14/14.5"? [Y/n] y
```

7. Now it will ask for the scope that will deploy the project. This is used when you have multiple account access options defined under the same username. In most of the scenarios, it will only have one, and you can press *Enter*:

```
? Set up and deploy "~/Versionamento/Vue.js-3.0-Cookbook/chapter-
  14/14.5"? y
? Which scope do you want to deploy to?
❯ Heitor Ramon Ribeiro
```

8. Then, it will ask to link to an existing project on Vercel. In our case, this is a brand new project, so we will choose `n`:

```
? Link to existing project? [Y/n] n
```

9. You will be asked to define the project's name (only lowercase alphanumeric characters and hyphens are allowed):

```
? What's your project's name? vuejscookbook-12-5
```

10. You now need to define the location of the source code of the project. This location is where the `package.json` file is located; in our case, this will be the `./` folder, or the main project folder:

```
? In which directory is your code located? ./
```

11. Vercel-CLI will detect that the project is a Vue-CLI project, and will automatically define all the commands and directory settings for the deployment of the application. We will choose `n` in our case:

    ```
    Auto-detected Project Settings (Vue.js):
    - Build Command: `npm run build` or `Vue-CLI-service build`
    - Output Directory: dist
    - Development Command: Vue-CLI-service serve --port $PORT
    ? Want to override the settings? [y/N] n
    ```

12. Once everything is set up, the CLI will deploy the first preview of your application, and you will receive a link to access the preview of your application. To deploy your application as production-ready, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

    ```
    > vercel --prod
    ```

How it works...

In this recipe, we learned how to connect the Vercel-CLI to the online platform with the email address that is linked to the repository manager, and set up project deployment.

In this process, we learned how to configure the advanced options of the CLI by defining the project settings for the build commands, distribution folder, and development command.

Finally, we were able to get the preview URL of our project before deploying it to production.

See also

You can find out more information about the Vercel-CLI at `https://vercel.com/docs/cli?query=CLI#getting-started`.

You can find out more information about Vercel advanced configurations at `https://vercel.com/docs/configuration?query=now.json#introduction/configuration-reference`.

Preparing for automatic deployment on Vercel with GitHub

We learned in the previous recipe how to use the Vercel-CLI to deploy our application to the web using your local terminal, but it is possible to integrate the repository manager with the Vercel platform and deploy automatically through any push or open pull requests. That's what we will do in this recipe.

Getting ready

The prerequisites for this recipe are as follows:

- A Vercel account
- A Vue project on a repository manager

How to do it...

In this recipe, we will learn how to integrate the Vercel platform with the repository manager and make an automatic deployment:

1. Open your Vercel dashboard (`https://vercel.com/dashboard`) and click on the **Import Project** button.
2. On the **Import Project** page, click on the **Continue** button inside the **From Git Repository** card.
3. Now, the Vercel website will ask whether the user who holds the repository of the project you are importing is your personal account. Click **Yes** if it is. If it isn't, Vercel will fork the project into your personal account before starting the process.
4. Then, Vercel will ask which account you want to bind the project to. In our case, this will be our personal account. Select it, and click on the **Continue** button.
5. You will be redirected to the GitHub web page, to give Vercel access to your repositories. You can give access to all your repositories, or just the ones you want to deploy. In our case, we will give access to all the repositories on our account.

6. After installing the Vercel application on your GitHub account, you will be sent back to the Vercel web page. Here, you can define the settings for the project you are creating, including the project name, the preset you are using, build instructions, and environment variables. Vercel will automatically detect that our project is a Vue-CLI project and will configure the build and deployment settings for us. Then, click on the **Deploy** button to continue.
7. Vercel will start the first deployment process. When it's finished, Vercel will give you the link for the application, along with a link for you to open the dashboard.

How it works...

The Vercel platform connects to your GitHub account and installs as an application, giving access to selected repositories. Then, on the platform, you can select the repository that you want to use to deploy.

With the repository selected, you need to configure the Vercel-CLI with the build instructions and the built destination folder.

Finally, the CLI runs, and we have our application up and running on the web.

See also

Find out more information about Vercel integrations with Git repositories at `https://zeit.co/docs/v2/git-integrations`.

Creating a Firebase project

Firebase is an all-in-one solution created by Google to help developers with dedicated tools for analytics, notifications, machine learning, and cloud solutions. One of the cloud solutions they provide is the hosting platform.

With the hosting platform, we are able to host our single-page applications in the Google cloud servers and have them available to everyone, through a global content delivery network.

Getting ready

The prerequisites for this recipe are as follows:

- A Google account
- A Vue project

How to do it...

In this recipe, we will learn how to create our Firebase project so that we can deploy our application to the Firebase hosting:

1. Open the Firebase home page (`https://firebase.google.com/`) and click on the **Sign In** link in the header menu. If you are already logged in to your Google account, click on the **Go to console** link.
2. On the **Console** page, click on the **Create a project** button to create a new Firebase project.
3. Firebase will ask for the project name (you can only use alphanumeric characters and spaces).
4. Then, Firebase will ask whether you want to enable Google Analytics in this project. In our case, we will disable this option.
5. Finally, you will be redirected to the project overview dashboard.

How it works...

In this recipe, we created our first Firebase project. To do it, we started by signing in to our Google account and going to the Firebase console. On the Firebase console, we created a new project, and in the setup wizard steps, we disabled the Google Analytics options because we won't be using attached analytics in this recipe. Finally, when we finished the setup wizard, our project was ready.

See also

Find out more information about Google Firebase at `https://firebase.google.com`.

Configuring the Firebase-CLI and deploying your project

To deploy our application to Firebase Hosting, we need to use the Firebase-CLI. The CLI will help with the process of packing the files and sending them to the Google Cloud server.

In this recipe, we will learn how to configure the Firebase-CLI to deploy your application to the web using your local terminal.

Getting ready

The prerequisites for this recipe are as follows:

- A Google account
- A Vue project
- A Firebase project
- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`
- `firebase-tools`

To install `firebase-tools`, you need to open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command:

```
> npm install -g firebase-tools
```

How to do it...

In this recipe, we will learn how to set up the Firebase-CLI on our project, and how to initialize it with our project created during the previous recipe:

1. Open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command on the root folder of your project:

```
> firebase login
```

2. The Firebase-CLI will open a browser window so you can log in to your Google account, and give access on the part of the Firebase-CLI to your Google Account. (If the browser doesn't open automatically, a link will appear on the Firebase-CLI, copy the link, and then paste it into your browser to continue.)
3. Open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command on the root folder of your project:

```
> firebase init
```

4. Now we are initializing the configuration process of the CLI with our project. For the first question of the CLI, we are going to use just the **Hosting** feature, so we need to select just `Hosting`:

```
? Which Firebase CLI features do you want to set up
    for this folder?
  Press space to select feature, then Enter to confirm
    your choices.
   Database: Deploy Firebase Realtime Database Rules
   Firestore: Deploy rules and create indexes for Firestore
   Functions: Configure and deploy Cloud Functions
❯  Hosting: Configure and deploy Firebase Hosting sites
   Storage: Deploy Cloud Storage security rules
   Emulators: Set up local emulators for Firebase features
```

5. Then, the CLI will ask which Firebase project we want to use. In our case, we created the project earlier in the previous recipe, so we will select `Use an existing project`:

```
? Use an existing project
❯ Use an existing project
  Create a new project
  Add Firebase to an existing Google Cloud Platform project
  Don't set up a default project
```

6. Now a list of available projects on your account will appear. Select the one you want to use to deploy with this application:

   ```
   ? Select a default Firebase project for this directory: (Use arrow
     keys)
   ❯ vue-3-cookbook-firebase-18921 (Vue 3 Cookbook Firebase)
   ```

7. The CLI will ask about the public directory of the application, or in our case, because it's a single-page application, we need to use the build destination folder. Type the name of the destination folder, in our case it's `dist`:

   ```
   ? What do you want to use as your project public directory? dist
   ```

8. Finally, the last step in the process is to select whether we want to use the configuration as a single-page application. Type `y` to enable rewrites of all the URLs to `index.html` so we can use the history mode of `vue-router`:

   ```
   ? Configure as a single-page app (rewrite all urls to /index.html)?
     (y/N) y
   ```

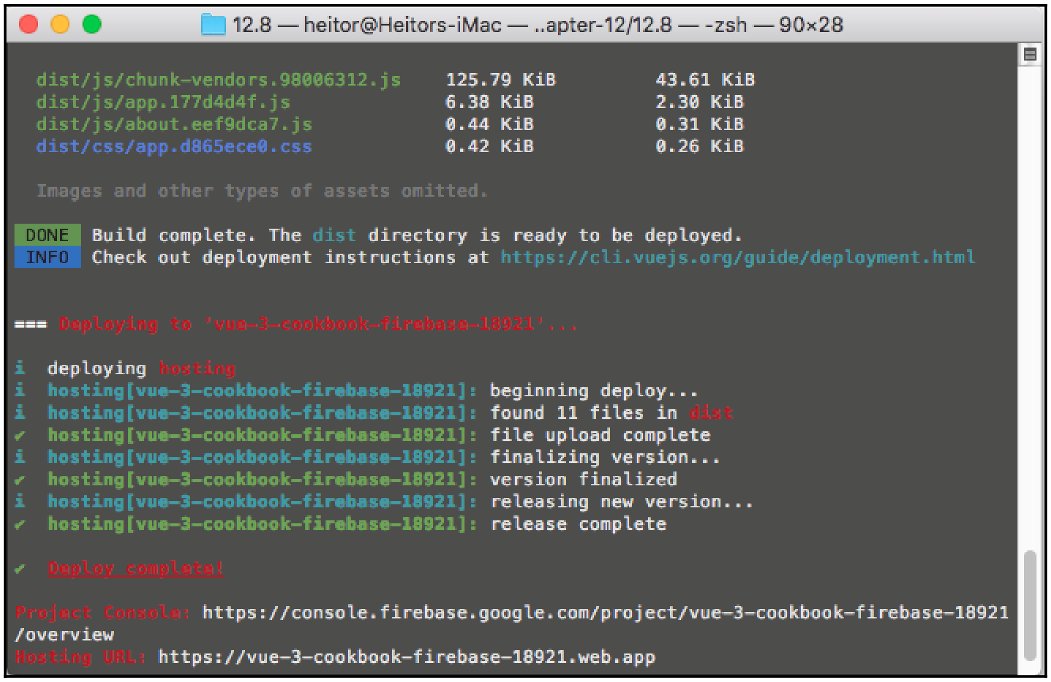
9. Open the `package.json` file on the root directory of your project, and add a new script to automate the build and deployment process:

   ```
   "scripts": {
     "serve": "Vue-CLI-service serve",
     "build": "Vue-CLI-service build",
     "deploy": "npm run build && firebase deploy",
     "lint": "Vue-CLI-service lint"
   },
   ```

10. Open Terminal (macOS or Linux) or Command Prompt/PowerShell (Windows) and execute the following command on the root folder of your project:

    ```
    > npm run deploy
    ```

Now your project is deployed and available on the web, and the CLI will give you the links to access it:


12.8 — heitor@Heitors-iMac — ..apter-12/12.8 — -zsh — 90×28

```
  dist/js/chunk-vendors.98006312.js    125.79 KiB         43.61 KiB
  dist/js/app.177d4d4f.js              6.38 KiB           2.30 KiB
  dist/js/about.eef9dca7.js            0.44 KiB           0.31 KiB
  dist/css/app.d865ece0.css            0.42 KiB           0.26 KiB

  Images and other types of assets omitted.

 DONE  Build complete. The dist directory is ready to be deployed.
 INFO  Check out deployment instructions at https://cli.vuejs.org/guide/deployment.html


=== Deploying to 'vue-3-cookbook-firebase-18921'...

i  deploying hosting
i  hosting[vue-3-cookbook-firebase-18921]: beginning deploy...
i  hosting[vue-3-cookbook-firebase-18921]: found 11 files in dist
✔  hosting[vue-3-cookbook-firebase-18921]: file upload complete
i  hosting[vue-3-cookbook-firebase-18921]: finalizing version...
✔  hosting[vue-3-cookbook-firebase-18921]: version finalized
i  hosting[vue-3-cookbook-firebase-18921]: releasing new version...
✔  hosting[vue-3-cookbook-firebase-18921]: release complete

✔  Deploy complete!

Project Console: https://console.firebase.google.com/project/vue-3-cookbook-firebase-18921
/overview
Hosting URL: https://vue-3-cookbook-firebase-18921.web.app
```

How it works...

In this recipe, we learned how to configure the Firebase CLI and to deploy our application.

First, we installed the Firebase-CLI and signed in to the Google authentication platform. Then, we were able to initialize the CLI in our project folder.

In the process, we selected the project we created in the previous recipe and pointed the building folder to the corrected one on a Vue-CLI project.

Then, we configured that we want to use a single-page application router structure, and added a deployment script to `package.json`. Finally, we were able to deploy our application and make it available to everyone.

See also

Find out more information about Firebase Hosting at `https://firebase.google.com/docs/hosting`.

11
Directives, Plugins, SSR, and More

Now you are in the Pro League! You are an advanced Vue developer. Let's have some fun and check out some great recipes that are custom made for you! Here are some hand-picked optimization solutions that can improve the quality of your Vue application and make your life easier.

In this chapter, we'll cover the following recipes:

- Automatically loading `vue-router` routes
- Automatically loading `vuex` modules
- Creating a custom directive
- Creating a Vue plugin
- Creating an SSR, SPA, PWA, Cordova, and Electron application in Vue with Quasar
- Creating smarter Vue watchers and computed properties
- Creating a `Nuxt.js` SSR with Python `Flask` as the API
- The dos and don'ts of Vue applications

Technical requirements

In this chapter, we will be using Node.js, `Vue-CLI`, `Cordova`, `Electron`, `Quasar`, `Nuxt.js`, and Python.

Attention Windows users: you are required to install an `npm` package called `windows-build-tools` to be able to install the following required packages. To do so, open PowerShell as an Administrator and execute the following command:

```
> npm install -g windows-build-tools
```

To install `Vue-CLI`, you need to execute the following command in Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows):

```
> npm install -g @vue/cli @vue/cli-service-global
```

To install `Cordova`, you need to execute the following command in Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows):

```
> npm install -g cordova
```

If you are running on a macOS and you want to run an iOS simulator, you need to execute the following command in Terminal (macOS):

```
> npm install -g ios-sim ios-deploy
```

To install `Electron`, you need to execute the following command in Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows):

```
> npm install -g electron
```

To install `Quasar`, you need to execute the following command in Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows):

```
> npm install -g @quasar/cli
```

To install `Nuxt.js`, you need to execute the following command in Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows):

```
> npm install -g create-nuxt-app
```

Automatically loading Vue routes

In order to create maintainable code, we can use the strategy of auto-importing files that have the same structure in our project. Like the routes in `vue-router`, when the application gets larger, we find a huge amount of files being imported and handled manually. In this recipe, we will learn a trick to use the webpack `require.context` function to automatically inject files for us.

This function will read the file content and add the routes to an array that will be exported into our file by default. You can improve this recipe by adding a more controlled route import or even environment-based route rules.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

We will need to create a new Vue project with `Vue-CLI`, or use the project created in previous recipes:

1. We need to open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > vue create router-import
   ```

2. The CLI will ask some questions that will help with the creation of the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *spacebar* to select an option.

3. There are two methods for starting a new project. The default method is a basic `babel` and `eslint` project without any plugins or configuration, and the `Manually` mode, where you can select more modes, plugins, linters, and options. We will go for `Manually`:

   ```
   ? Please pick a preset: (Use arrow keys)
     default (babel, eslint)
   ❯ Manually select features
   ```

4. Now we are asked about the features that we want on the project. Those features are some Vue plugins such as `Vuex` or `Router` (`vue-router`), testers, linters, and more. Select `Babel`, `Router`, and `Linter / Formatter`:

   ```
   ? Check the features needed for your project: (Use arrow keys)
   ❯ Babel
     TypeScript
     Progressive Web App (PWA) Support
   ❯ Router
     Vuex
     CSS Pre-processors
   ❯ Linter / Formatter
     Unit Testing
     E2E Testing
   ```

5. Continue this process by selecting a linter and formatter. In our case, we will select the `ESLint + Airbnb` config:

   ```
   ? Pick a linter / formatter config: (Use arrow keys)
     ESLint with error prevention only
   ❯ ESLint + Airbnb config
     ESLint + Standard config
     ESLint + Prettier
   ```

6. After the linting rules are set, we need to define when they are applied to your code. They can be either applied `on save` or fixed `on commit`:

   ```
   ? Pick additional lint features: (Use arrow keys)
     Lint on save
   ❯ Lint and fix on commit
   ```

7. After all those plugins, linters, and processors are defined, we need to choose where the settings and configs are stored. The best place to store them is in a dedicated file, but it is also possible to store them in `package.json`:

   ```
   ? Where do you prefer placing config for Babel, ESLint, etc.? (Use
     arrow keys)
   ❯ In dedicated config files
     In package.json
   ```

8. Now you can choose whether you want to make this selection a preset for future projects so that you don't need to reselect everything:

   ```
   ? Save this as a preset for future projects? (y/N) n
   ```

`Vue-CLI` will create the project, and automatically install the packages for us.

If you want to check the project on `vue-ui` when the installation has finished, open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue ui
```

Or you can run the built-in `npm` commands by opening Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute one of the following commands:

- `npm run serve` – To run a development server locally
- `npm run build` – To build and minify the application for deployment
- `npm run lint` – To execute the lint on the code

How to do it...

Follow these instructions to create an auto-import of the router files in your project that will handle the router files inside a specific folder:

1. With our route files created and placed inside the `routes` folder, we need to make sure that every route file has a default `export` object in it. In the `index.js` file, inside the `src/router` folder, remove the default array of `routes` that is present in the file:

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

Vue.use(VueRouter);

export default new VueRouter({});
```

2. Now create an empty array of `routes` that will be populated by the imported ones from the folder, and start the import. With that, `requireRoutes` will be an object with the keys being the filename and the values being the ID of the file:

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

Vue.use(VueRouter);

const routes = [];
const requireRoutes = require.context(
  './routes',
  true,
  /^(?!.*test).*\.js$/is,
);

const router = new VueRouter({
  routes,
});

export default router;
```

3. To push those files inside the `routes` array, we need to add the following code and create a folder named `routes` inside the `router` folder:

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

Vue.use(VueRouter);

const routes = [];
const requireRoutes = require.context(
  './routes',
  true,
  /^(?!.*test).*\.js$/is,
);

requireRoutes.keys().forEach((fileName) => {
  routes.push({
    ...requireRoutes(fileName).default,
  });
});

const router = new VueRouter({
  routes,
});

export default router;
```

Now you have your routes loaded on your application automatically as you create a new `.js` file inside the `routes` folder.

How it works...

`require.context` is a webpack built-in function that allows you to pass in a directory to search, a flag indicating whether subdirectories should be examined too, and a regular expression to match files.

When the building process starts, webpack will search for all the `require.context` functions and will pre-execute them, so the files needed on the import will be there for the final build.

We pass three arguments to the function: the first is the folder where it will start the search, the second asks whether the search will go to descending folders, and finally, the third is a regular expression for filename matching.

In this recipe, to automatically load the routes as the first argument of the function, we define `./routes` for the folder. As the second argument of the function, we define `false` to not search in subdirectories. Finally, as the third argument, we define `/^(?!.*test).*\.js$/is` as the Regex to search for `.js` files and ignore the files that have `.test` in their names.

There's more...

With this recipe, it's possible to take your application to the next level by using the subdirectories for router modules and environments for router control.

With those increments, the function may be extracted to another file, but in `router.js`, it still needs to be imported into the `main.js` file. Or, you can obtain the `import` function, and pass the array of `routes` to `router.js`.

See also

Read more about webpack dependency management and `require.context` in the webpack documentation at `https://webpack.js.org/guides/dependency-management/`.

Automatically loading Vuex modules

Sometimes, when we are working on a big project, we need to manage a lot of imported `Vuex` modules and stores. To handle those modules, we always need to import them by creating a file that will have all the files imported and then export those to the Vuex store creation.

In this recipe, we will learn about a function that uses the webpack `require.context` function to automatically load and inject those files into the Vuex store creation.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

We will need to create a new Vue project with `Vue-CLI`, or use the project created in previous recipes:

1. We need to open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue create vuex-import
```

2. The CLI will ask some questions that will help with the creation of the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *spacebar* to select an option.

3. There are two methods for starting a new project. The default method is a basic `babel` and `eslint` project without any plugins or configuration, and the `Manually` mode, where you can select more modes, plugins, linters, and options. We will go for `Manually`:

```
? Please pick a preset: (Use arrow keys)
  default (babel, eslint)
❯ Manually select features
```

4. Now we are asked about the features that we will want on the project. Those features are some Vue plugins such as `Vuex` or `Router` (`vue-router`), testers, linters, and more. Select `Babel`, `Vuex`, and `Linter / Formatter`:

```
? Check the features needed for your project: (Use arrow keys)
❯ Babel
  TypeScript
  Progressive Web App (PWA) Support
  Router
❯ Vuex
  CSS Pre-processors
❯ Linter / Formatter
  Unit Testing
  E2E Testing
```

5. Continue this process by selecting a linter and formatter. In our case, we will select the `ESLint + Airbnb` config:

```
? Pick a linter / formatter config: (Use arrow keys)
  ESLint with error prevention only
❯ ESLint + Airbnb config
  ESLint + Standard config
  ESLint + Prettier
```

6. After the linting rules are set, we need to define when they are applied to your code. They can be either applied `on save` or fixed `on commit`:

```
? Pick additional lint features: (Use arrow keys)
  Lint on save
❯ Lint and fix on commit
```

7. After all those plugins, linters, and processors are defined, we need to choose where the settings and configs are stored. The best place to store them is in a dedicated file, but it is also possible to store them in `package.json`:

```
? Where do you prefer placing config for Babel, ESLint, etc.? (Use
  arrow keys)
❯ In dedicated config files
  In package.json
```

8. Now you can choose whether you want to make this selection a preset for future projects, so you don't need to reselect everything:

```
? Save this as a preset for future projects? (y/N) n
```

`Vue-CLI` will create the project, and automatically install the packages for us.

If you want to check the project on `vue-ui` when the installation has finished, open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue ui
```

Or you can run the built-in `npm` commands by opening Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and executing one of the following commands:

- `npm run serve` – To run a development server locally
- `npm run build` – To build and minify the application for deployment
- `npm run lint` – To execute the lint on the code

How to do it...

Follow these instructions to create an auto-import of the `vuex` modules in your project that will handle the router files inside a specific folder:

1. With our route files created and placed inside the `store` folder, we need to make sure that every `store` file has a default `export` object in it. In the `index.js` file, inside the `src/store` folder, we will need to extract the array of `stores` or `modules`:

```
import Vue from 'vue';
import Vuex from 'vuex';

Vue.use(Vuex);

export default new Vuex.Store({});
```

2. Create another file named `loader.js` in the `src/store` folder (which will be our `module` loader). It's important to remember that when using this recipe, you will use `vuex` namespaced because all the `stores` need to be used as a module and need to be exported in a single JavaScript object. Each filename will be used as a reference to a namespace, and it will be parsed to a camelCase text style:

```
const toCamel = (s) => s.replace(/([-_][a-z])/ig, (c) =>
    c.toUpperCase()
   .replace(/[-_]/g, ''));
const requireModule = require.context('./modules/', false,
   /^(?!.*test).*\.js$/is);
const modules = {};

requireModule.keys().forEach((fileName) => {
  const moduleName = toCamel(fileName.replace(/(\.\/|\.js)/g, ''));

  modules[moduleName] = {
    namespaced: true,
    ...requireModule(fileName).default,
  };
});

export default modules;
```

3. As we will be importing by default each file inside the `modules` folder, a good practice is to create a file for each module. For example, as you will be creating a module named `user`, you need to create a file named `user.js` that imports all the `stores` actions, mutations, getters, and state. Those can be placed inside a folder that has the same name as the module. The `modules` folder will have a structure similar to this:

```
modules
├── user.js
├── user
│  └── types.js
│    └── state.js
│    └── actions.js
│    └── mutations.js
│    └── getters.js
└────────
```

The `user.js` file inside the `src/store/modules` folder will look like this:

```
import state from './user/state';
import actions from './user/actions';
import mutations from './user/mutations';
import getters from './user/getters';

export default {
  state,
  actions,
  mutations,
  getters,
};
```

4. In the `index.js` file in the `src/store` folder, we need to add the imported modules that were automatically loaded:

```
import Vue from 'vue';
import Vuex from 'vuex';
import modules from './loader';

Vue.use(Vuex);

export default new Vuex.Store({
  modules,
});
```

Now you have your `vuex` modules loaded on your application automatically as you create a new `.js` file inside the `src/store/modules` folder.

How it works...

`require.context` is a webpack built-in function that receives a directory to execute a search, a Boolean flag indicating whether subdirectories are included in this search, and a regular expression for the pattern matching for the filename (all as arguments).

When the building process starts, webpack will search for all the `require.context` functions, and will pre-execute them, so the files needed on the import will be there for the final build.

In our case, we passed `./modules` for the folder, `false` to not search in subdirectories, and `/^(?!.*test).*\.js$/is` as the Regex to search for `.js` files and ignore the files that have `.test` in their names.

Then, the function will search for the files and will pass the result through a `for` loop to add the content of the files in the array of `vuex` modules.

See also

Read more about webpack dependency management and `require.context` in the webpack documentation at `https://webpack.js.org/guides/dependency-management/`.

Creating a custom directive

Talking about visual frameworks such as Vue, we always think about components, rendering, and visual elements, and we forget that there are a lot of things besides the components themselves.

There are the directives that make the components work with the template engine, which are the binding agents between the data and the visual result. And there are built-in directives in the core of Vue, such as `v-if`, `v-else`, and `v-for`.

In this recipe, we will learn how to make our directive.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

We will need to create a new Vue project with `Vue-CLI`, or use the project created in previous recipes:

1. We need to open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > vue create vue-directive
   ```

2. The CLI will ask some questions that will help with the creation of the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the s*pacebar* to select an option.

3. There are two methods for starting a new project. The default method is a basic `babel` and `eslint` project without any plugins or configuration, and the `Manually` mode, where you can select more modes, plugins, linters, and options. We will go for `Manually`:

   ```
   ? Please pick a preset: (Use arrow keys)
     default (babel, eslint)
   ❯ Manually select features
   ```

4. Now we are asked about the features that we want on the project. Those features are some Vue plugins such as `Vuex` or `Router` (`vue-router`), testers, linters, and more. Select `Babel` and `Linter / Formatter`:

   ```
   ? Check the features needed for your project: (Use arrow keys)
   ❯  Babel
   ```

```
    TypeScript
    Progressive Web App (PWA) Support
    Router
    Vuex
    CSS Pre-processors
 ❯  Linter / Formatter
    Unit Testing
    E2E Testing
```

5. Continue this process by selecting a linter and formatter. In our case, we will select the `ESLint + Airbnb` config:

```
? Pick a linter / formatter config: (Use arrow keys)
  ESLint with error prevention only
❯ ESLint + Airbnb config
  ESLint + Standard config
  ESLint + Prettier
```

6. After the linting rules are set, we need to define when they are applied to your code. They can be either applied `on save` or fixed `on commit`:

```
? Pick additional lint features: (Use arrow keys)
  Lint on save
❯ Lint and fix on commit
```

7. After all those plugins, linters, and processors are defined, we need to choose where the settings and configs are stored. The best place to store them is in a dedicated file, but it is also possible to store them in `package.json`:

```
? Where do you prefer placing config for Babel, ESLint, etc.? (Use
arrow keys)
❯ In dedicated config files
  In package.json
```

8. Now you can choose whether you want to make this selection a preset for future projects so you don't need to reselect everything:

```
? Save this as a preset for future projects? (y/N) n
```

`Vue-CLI` will create the project, and automatically install the packages for us.

If you want to check the project on `vue-ui` when the installation has finished, open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue ui
```

Or, you can run the built-in npm commands by opening Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and executing one of the following commands:

- `npm run serve` – To run a development server locally
- `npm run build` – To build and minify the application for deployment
- `npm run lint` – To execute the lint on the code

How to do it...

Follow these instructions to create a directive for a masked input field:

1. Create a file named `formMaskInputDirective.js` in the `src/directives` folder, and a file named `tokens.js` in the same folder.
2. In the `tokens.js` file, we will add our mask base tokens. Those tokens will be used to identify the kind of value our input will accept:

```
export default {
  "#": { pattern: /[\x2A\d]/ },
  0: { pattern: /\d/ },
  9: { pattern: /\d/ },
  X: { pattern: /[0-9a-zA-Z]/ },
  S: { pattern: /[a-zA-Z]/ },
  A: { pattern: /[a-zA-Z]/, transform: v => v.toLocaleUpperCase()
},
  a: { pattern: /[a-zA-Z]/, transform: v => v.toLocaleLowerCase()
},
  "!": { escape: true }
};
```

3. We import the token from `token.js` and create our functions:

```
import tokens from './tokens';

function maskerValue() {
  // Code will be developed in this recipe
}

function eventDispatcher() {
  // Code will be developed in this recipe
}

function maskDirective() {
 // Code will be developed in this recipe
}

export default maskDirective;
```

4. In the `maskDirective` function, we will need to check for the binding value on the directive that is passed by the callee of the directive and check whether it's a valid binding. To do so, we will first check whether the `value` property is present on the `binding` argument, and then add it to the `config` variable with the `tokens` that were imported:

```
function maskDirective(el, binding) {
  let config;

  if (!binding.value) return false;

  if (typeof config === 'string') {
    config = {
      mask: binding.value,
      tokens,
    };
  } else {
    throw new Error('Invalid input entered');
  }
```

5. Now we need to check for the element and validate whether it's an `input` HTML element. To do so, we will check whether the element that was passed down by the directive has a `tagName` of `input`, and if it doesn't, we will try to find an `input` HTML element in the element that was passed down:

```
let element = el;

  if (element.tagName.toLocaleUpperCase() !== 'INPUT') {
    const els = element.getElementsByTagName('input');

    if (els.length !== 1) {
      throw new Error(`v-input-mask directive requires 1 input,
         found ${els.length}`);
    } else {
      [element] = els;
    }
  }
```

6. Now we need to add an event listener to the input on the element. The listener will call two external functions, one for dispatching the events and another to return the masked value to the input:

```
element.oninput = (evt) => {
    if (!evt.isTrusted) return;
    let position = element.selectionEnd;

    const digit = element.value[position - 1];
    element.value = maskerValue(element.value, config.mask,
       config.tokens);
    while (
      position < element.value.length
      && element.value.charAt(position - 1) !== digit
    ) {
      position += 1;
    }
    if (element === document.activeElement) {
      element.setSelectionRange(position, position);
      setTimeout(() => {
        element.setSelectionRange(position, position);
      }, 0);
    }
    element.dispatchEvent(eventDispatcher('input'));
  };

  const newDisplay = maskerValue(element.value, config.mask,
     config.tokens);
  if (newDisplay !== element.value) {
    element.value = newDisplay;
    element.dispatchEvent(eventDispatcher('input'));
  }

  return true;
}
// end of maskDirective function
```

7. Let's create the `eventDispatcher` function; this function will emit the events that will be listened to by the `v-on` directive:

```
function eventDispatcher(name) {
  const evt = document.createEvent('Event');

  evt.initEvent(name, true, true);

  return evt;
}
```

8. And now the complicated part: returning the masked input value to the input. To do so, we will need to create the `maskerValue` function. This function receives the value, mask, and token as parameters. The function checks for the current value against the mask, to see whether the mask is complete or the value is of a valid token. If everything's okay, it will pass the value to the input:

```
function maskerValue(v, m, tkn) {
  const value = v || '';

  const mask = m || '';

  let maskIndex = 0;

  let valueIndex = 0;

  let output = '';

  while (maskIndex < mask.length && valueIndex < value.length) {
    let maskCharacter = mask[maskIndex];
    const masker = tkn[maskCharacter];
    const valueCharacter = value[valueIndex];

    if (masker && !masker.escape) {
      if (masker.pattern.test(valueCharacter)) {
        output += masker.transform ?
           masker.transform(valueCharacter) : valueCharacter;
        maskIndex += 1;
      }

      valueIndex += 1;
    } else {
      if (masker && masker.escape) {
        maskIndex += 1;
        maskCharacter = mask[maskIndex];
      }

      output += maskCharacter;

      if (valueCharacter === maskCharacter) valueIndex += 1;

      maskIndex += 1;
    }
  }

  let outputRest = '';
  while (maskIndex < mask.length) {
    const maskCharacter = mask[maskIndex];
```

```
        if (tkn[maskCharacter]) {
          outputRest = '';
          break;
        }

        outputRest += maskCharacter;

        maskIndex += 1;
      }

      return output + outputRest;
    }
    //end of maskerValue function
```

9. With our file ready, we need to import the mask directive in the `main.js` file and add the directive to Vue, giving the directive the name `'input-mask'`:

```
import Vue from 'vue';
import App from './App.vue';
import InputMaskDirective from
'./directives/formMaskInputDirective';

Vue.config.productionTip = false;

Vue.directive('input-mask', InputMaskDirective);

new Vue({
  render: (h) => h(App),
}).$mount('#app');
```

10. To use the directive on our application, we need to call the directive on an `input` HTML element inside a single file component `<template>` section, passing the `token` template `'###-###-###'` in the `v-input-mask` directive like this:

```
<template>
  <div id="app">
    <input
      type="text"
      v-input-mask="'###-###-###'"
      v-model="inputMask"
    />
  </div>
</template>

<script>
export default {
  name: 'app',
  data: () => ({
    inputMask: '',
  }),
};
</script>
```

How it works...

A Vue directive has five possible hooks. We used just one, `bind`. It's bound directly to the element and component. It gets three arguments: `element`, `binding`, and `vnode`.

When we add the directive in the `main.js` file to Vue, we make it available everywhere in our application, so the directive is already at `App.vue` to be used by the input.

At the same time we call `v-input-mask` on the input element, we pass the first argument, `element`, to the directive, and the second argument, `binding`, is the value of the attribute.

Our directive works by checking each new character value on the input. A Regex test is executed and validates the character to see whether it is a valid character on the token list that was given on the directive instantiation. Then, it returns the character if it passes the test, or returns nothing if it's an invalid character.

Creating a Vue plugin

Sometimes a new addition to your application is needed, and this addition needs to be shared. The best way to share it is by using a plugin. In Vue, a plugin is an addition to the Vue global prototype by extending the initialized application with new features such as directives, mixings, filters, prototype injection, or totally new functions.

Now we will learn how to make our plugin, and how we can use it to interact with Vue as a whole (without messing with the prototype and breaking it).

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global objects that are required are as follows:

- `@vue/cli`
- `@vue/cli-service-global`

We will need to create a new Vue project with the `Vue-CLI`, or use the project created in previous recipes:

1. We need to open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

   ```
   > vue create vue-plugin
   ```

2. The CLI will ask some questions that will help with the creation of the project. You can use the arrow keys to navigate, the *Enter* key to continue, and the *spacebar* to select an option.
3. There are two methods for starting a new project. The default method is a basic `babel` and `eslint` project without any plugins or configuration, and the `Manually` mode, where you can select more modes, plugins, linters, and options. We will go for `Manually`:

   ```
   ? Please pick a preset: (Use arrow keys)
     default (babel, eslint)
   ❯ Manually select features
   ```

4. Now we are asked about the features that we want on the project. Those features are some Vue plugins such as `Vuex` or `Router` (`vue-router`), testers, linters, and more. Select `Babel`, and `Linter / Formatter`:

   ```
   ? Check the features needed for your project: (Use arrow keys)
   ❯  Babel
      TypeScript
      Progressive Web App (PWA) Support
      Router
      Vuex
      CSS Pre-processors
   ❯  Linter / Formatter
      Unit Testing
      E2E Testing
   ```

5. Continue this process by selecting a linter and formatter. In our case, we will select the `ESLint + Airbnb` config:

   ```
   ? Pick a linter / formatter config: (Use arrow keys)
     ESLint with error prevention only
   ```

```
❯ ESLint + Airbnb config
  ESLint + Standard config
  ESLint + Prettier
```

6. After the linting rules are set, we need to define when they are applied to your code. They can be either applied `on save` or fixed `on commit`:

```
? Pick additional lint features: (Use arrow keys)
   Lint on save
❯  Lint and fix on commit
```

7. After all those plugins, linters, and processors are defined, we need to choose where the settings and configs are stored. The best place to store them is in a dedicated file, but it is also possible to store them in `package.json`:

```
? Where do you prefer placing config for Babel, ESLint, etc.? (Use
  arrow keys)
❯ In dedicated config files
  In package.json
```

8. Now you can choose whether you want to make this selection a preset for future projects, so you don't need to reselect everything:

```
? Save this as a preset for future projects? (y/N) n
```

`Vue-CLI` will create the project, and automatically install the packages for us.

If you want to check the project on `vue-ui` when the installation has finished, open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> vue ui
```

Or, you can run the built-in npm commands by opening Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and executing one of the following commands:

- `npm run serve` – To run a development server locally
- `npm run build` – To build and minify the application for deployment
- `npm run lint` – To execute the lint on the code

How to do it...

Writing a Vue plugin is simple, and there is no need to learn more about Vue itself. The basic concept of a plugin is an object that needs to have an `install` function, which will be executed when called by the `Vue.use()` method. The `install` function will receive two arguments: Vue, and the options that will be used to instantiate the plugin.

Follow these instructions to write a plugin that adds two new functions to the Vue global prototype, `$localStorage` and `$sessionStorage`:

1. In our project, we need to create a file inside the `src/plugin` folder named `storageManipulator.js`.
2. In this file, we will create the plugin installation object – we'll add the default plugin options and the base prototype for the functions:

```
/* eslint no-param-reassign: 0 */

const defaultOption = {
  useSaveFunction: true,
  useRetrieveFunction: true,
  onSave: value => JSON.stringify(value),
  onRetrieve: value => JSON.parse(value),
};

export default {
  install(Vue, option) {
    const baseOptions = {
      ...defaultOption,
      ...option,
    };

    Vue.prototype.$localStorage = generateStorageObject(
       window.localStorage,
        baseOptions,
      ); // We will add later this code

    Vue.prototype.$sessionStorage = generateStorageObject(
        window.localStorage,
        baseOptions,
      ); // We will add later this code
  },
};
```

3. Now we need to create the `generateStorageObject` function. This function will receive two arguments: the first will be the window storage object, and the second will be the plugin options. With this, it will be possible to generate the object that will be used on the prototype that will be injected into Vue:

```
const generateStorageObject = (windowStorage, options) => ({
  set(key, value) {
    windowStorage.setItem(
      key,
      options.useSaveFunction
        ? options.onSave(value)
        : value,
    );
  },

  get(key) {
    const item = windowStorage.getItem(key);
    return options.useRetrieveFunction ? options.onRetrieve(item) :
      item;
  },

  remove(key) {
    windowStorage.removeItem(key);
  },

  clear() {
    windowStorage.clear();
  },
});
```

4. We need to import the plugin into the `main.js`, and then with the `Vue.use` function, install the plugin in our Vue application:

```
import Vue from 'vue';
import App from './App.vue';
import StorageManipulatorPlugin from './plugin/storageManipulator';

Vue.config.productionTip = false;

Vue.use(StorageManipulatorPlugin);

new Vue({
  render: h => h(App),
}).$mount('#app');
```

Now you can use the plugin anywhere in your Vue application, calling the `this.$localStorage` method or `this.$sessionStorage`.

How it works...

The Vue plugin works by adding all the code that was instructed to be used to the Vue application layer (like a mixin).

When we used `Vue.use()` to import our plugin, we told Vue to call the `install()` function on the object of the imported file and executed it. Vue will automatically pass the current Vue as the first argument, and the options (if you declare them) as the second argument.

In our plugin, when the `install()` function is called, we first create `baseOptions`, merging the default options with the passed parameter, then we inject two new properties into the Vue prototype. Those properties are now available everywhere because the `Vue` parameter that was passed is the `Vue global` being used in the application.

Our `generateStorageObject` is a pure abstraction of the Storage API of the browser. We use it as a generator for our prototypes inside the plugin.

See also

You can find more information about Vue plugins at `https://vuejs.org/v2/guide/plugins.html.`

You can find a curated list of awesome Vue plugins at `https://github.com/vuejs/awesome-vue.`

Creating an SSR, SPA, PWA, Cordova, and Electron application in Vue with Quasar

Quasar is a framework based on Vue and Material Design that takes advantage of "write once, use everywhere."

The CLI can deploy the same code base to different flavors, such as **Single-Page Application** (**SPA**), **Server-Side Rendering** (**SSR**), **Progressive Web Application** (**PWA**), **Mobile Application** (Cordova), and **Desktop Application** (Electron).

This takes some of the problems away from the developer, such as configuring webpack, Cordova, and Electron with **HMR** (**Hot Module Reload**) for development, or adding an SSR configuration in the SPA project. The framework helps the developer start production as soon as possible.

In this recipe, we will learn how to use Quasar and the CLI to create a basic project, and how to use the CLI to add the development targets for SPA, PWA, SSR, Mobile Application, and Desktop Application.

Getting ready

The pre-requisite for this recipe is as follows:

- Node.js 12+

The Node.js global object that is required is as follows:

- `@quasar/cli`

We will need to create a new Quasar project with the Quasar CLI, or use the project created in previous recipes.

To do it, open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> quasar create quasar-project
```

Now, when asked, we need to choose to manually select features:

1. `Quasar-CLI` will ask you for a project name. Define your project name. In our case, we choose `quasar_project`:

   ```
   > Project name: quasar_project
   ```

2. Then `Quasar-CLI` will ask for a project product name. This will be used by mobile apps to defined their title name. In our case, we stayed with the default name provided:

   ```
   > Project product name (must start with letter if building mobile
       apps) (Quasar App)
   ```

3. Now `Quasar-CLI` will ask for a project description. This is used for a meta tag in search engines when the page is shared. In our case, we used the default description provided:

   ```
   > Project description: (A Quasar Framework app)
   ```

4. Then `Quasar-CLI` will ask for the project author. Fill this with a `package.json` valid name (for example, `Heitor Ribeiro<heitor@example.com>`):

   ```
   > Author: <You>
   ```

5. Now it's time to choose the CSS preprocessor. In our case, we will go with `Sass with indented syntax`:

   ```
   Pick your favorite CSS preprocessor: (can be changed later) (Use
   arrow keys)
   ❯ Sass with indented syntax (recommended)
     Sass with SCSS syntax (recommended)
     Stylus
     None (the others will still be available)
   ```

6. Then `Quasar-CLI` will ask about the import strategy for the components and directives. We will use the default `auto-import` strategy:

   ```
   Pick a Quasar components & directives import strategy: (can be
     changed later) (Use arrow keys )
   ❯ * Auto-import in-use Quasar components & directives - also
       treeshakes Quasar; minimum bundle size
     * Import everything from Quasar - not treeshaking Quasar;
       biggest bundle size
   ```

7. Now we need to choose the extra features for the project. We will select `EsLint`:

   ```
   Check the features needed for your project: EsLint
   ```

8. After that, `Quasar-CLI` will ask for a preset for ESLint. Choose the `Airbnb` preset:

   ```
   Pick an ESLint preset: Airbnb
   ```

9. Finally, `Quasar-CLI` will ask for the application you want to use to install the dependencies of the project. In our case, we used `yarn` because we have installed it already (but you can choose the one you prefer):

   ```
   Continue to install project dependencies after the project has been
     created? (recommended) (Use arrow keys)
   ❯ Yes, use Yarn (recommended)
     Yes, use npm
     No, I will handle that myself
   ```

Now open the created folder in your IDE or code editor.

How to do it...

When using Quasar to create an application, you always need to choose a flavor to start, but the main code will be an SPA. Therefore, the other flavors will have their special treats and delicacies based on their needs, but you can personalize and make your build execute some code based on the build environment.

Developing an SPA (Single-Page Application)

Starting the development of an SPA is an out-of-the-box solution; there is no need to add any new configuration.

So let's start adding a new page to our application. Open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> quasar new page About
```

`Quasar-CLI` will automatically create the Vue page for us. We need to add the reference to the page in the router file, and the page will be available on the application:

1. To do it, we need to open the `routes.js` file in the `src/router` folder, and add the `About` page:

```
const routes = [
  {
    path: '/',
    component: () => import('layouts/MainLayout.vue'),
    children: [
      { path: '', name: 'home', component: () =>
         import('pages/Index.vue') },
      { path: 'about', name: 'about', component: () =>
         import('pages/About.vue') },
    ],
  },
  {
    path: '*',
    component: () => import('pages/Error404.vue'),
  }
];

export default routes;
```

2. Then open the `About.vue` file in the `src/pages` folder. You will find that the file is a single file component that has an empty `QPage` component in it, so we need to add a basic title and page indication in the `<template>` section:

    ```
    <template>
    <q-page
        padding
        class="flex flex-start"
      >
        <h1 class="full-width">About</h1>
        <h2>This is an About Us Page</h2>
      </q-page>
    </template>

    <script>
    export default {
      name: 'PageAbout',
    };
    </script>
    ```

3. Now, in the `MainLayout.vue` file, in the `src/layouts` folder, to the `q-drawer` component, we need to add the links to the `Home` and `About` page:

    ```
    <template>
      <q-layout view="lHh Lpr lFf">
        <q-header elevated>
          <q-toolbar>
            <q-btn flat dense round
            @click="leftDrawerOpen = !leftDrawerOpen"
            aria-label="Menu">
              <q-icon name="menu" />
            </q-btn>

            <q-toolbar-title>
              Quasar App
              </q-toolbar-title>

            <div>Quasar v{{ $q.version }}</div>
          </q-toolbar>
        </q-header>

        <q-drawer v-model="leftDrawerOpen"
        bordered content-class="bg-grey-2">
          <q-list>
            <q-item-label header>Menu</q-item-label>
            <q-item clickable tag="a" :to="{name: 'home'}">
              <q-item-section avatar>
    ```

```
            <q-icon name="home" />
          </q-item-section>
          <q-item-section>
            <q-item-label>Home</q-item-label>
          </q-item-section>
        </q-item>
        <q-item clickable tag="a" :to="{name: 'about'}">
          <q-item-section avatar>
            <q-icon name="school" />
          </q-item-section>
          <q-item-section>
            <q-item-label>About</q-item-label>
          </q-item-section>
        </q-item>
      </q-list>
    </q-drawer>

    <q-page-container>
      <router-view />
    </q-page-container>
  </q-layout>
</template>

<script>
export default {
  name: "MyLayout",
  data() {
    return {
      leftDrawerOpen: this.$q.platform.is.desktop
    };
  }
};
</script>
```

And we are finished with a simple example of an SPA running inside a Quasar framework.

Commands

You can run the `Quasar-CLI` commands by opening Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and executing one of the following:

- `quasar dev` – To start development mode
- `quasar build` – To build the SPA

Developing a PWA (Progressive Web Application)

To develop a PWA, we first need to inform Quasar that we want to add a new mode of development. Open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> quasar mode add pwa
```

`Quasar-CLI` will create a folder called `src-pwa` that will have our `service-workers` files, separated from our main code.

To clean the newly added files, and to lint it into the Airbnb format, we need to open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> eslint --fix --ext .js ./src-pwa
```

The code that we added to the SPA will still be used as our base so that we can add new pages, components, and other functions to it as well, which will be used on the PWA.

So, are you wondering why `service-worker` is not in the main `src` folder? This is because those files are exclusively for PWAs, and are not needed in any other case than this one. The same will happen in different build types, such as Electron, Cordova, and SSR.

Configuring quasar.conf on a PWA

For PWA development, you can set some special flags on the `quasar.conf.js` file in the `root` folder:

```
pwa: {
  // workboxPluginMode: 'InjectManifest',
  // workboxOptions: {},
  manifest: {
    // ...
  },

  // variables used to inject specific PWA
  // meta tags (below are default values)
  metaVariables: {
    appleMobileWebAppCapable: 'yes',
    appleMobileWebAppStatusBarStyle: 'default',
    appleTouchIcon120: 'statics/icons/apple-icon-120x120.png',
    appleTouchIcon180: 'statics/icons/apple-icon-180x180.png',
    appleTouchIcon152: 'statics/icons/apple-icon-152x152.png',
    appleTouchIcon167: 'statics/icons/apple-icon-167x167.png',
    appleSafariPinnedTab: 'statics/icons/safari-pinned-tab.svg',
    msapplicationTileImage: 'statics/icons/ms-icon-144x144.png',
    msapplicationTileColor: '#000000'
  }
}
```

Commands

You can run the `Quasar-CLI` commands by opening Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and executing one of the following:

- `quasar dev -m pwa` – To start development mode as a PWA
- `quasar build -m pwa` – To build the code as a PWA

Developing SSR (Server-Side Rendering)

To develop SSR, we first need to inform Quasar that we want to add a new mode of development. Open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> quasar mode add ssr
```

`Quasar-CLI` will create a folder called `src-ssr` that will have our `extension` and `server` starter files, separated from our main code.

The `extension` file is not transpiled by `babel` and runs on the Node.js context, so it is the same environment as an Express or `Nuxt.js` application. You can use server plugins, such as `database`, `fileread`, and `filewrites`.

The `server` starter files will be our `index.js` file in the `src-ssr` folder. As the extension, it is not transpiled by `babel` and runs on the Node.js context. For the HTTP server, it uses Express, and if you configure `quasar.conf.js` to pass the client a PWA, you can have an SSR with PWA at the same time.

Configuring quasar.conf on SSR

For SSR development, you can configure some special flags on the `quasar.conf.js` file in the `root` folder:

```
ssr: {
  pwa: true/false, // should a PWA take over (default: false), or just
                                                        // a SPA?
},
```

Commands

You can run the `Quasar-CLI` commands by opening Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and executing one of the following:

- `quasar dev -m ssr` – To start development mode as SSR
- `quasar build -m ssr` – To build the code as SSR
- `quasar serve` – To run an HTTP server (can be used in production)

Developing a mobile application (Cordova)

To develop SSR, we first need to inform Quasar that we want to add a new mode of development. Open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> quasar mode add cordova
```

Now the `Quasar-CLI` will ask you some configuration questions:

1. **What is the Cordova app ID?** `(org.cordova.quasar.app)`
2. **May Cordova anonymously report usage statistics to improve the tool over time? (Y/N)** `N`

`Quasar-CLI` will create a folder called `src-cordova`, which will have a Cordova project inside.

The folder structure of a Cordova project looks like this:

```
src-cordova/
├── config.xml
├── packages.json
├── cordova-flag.d.ts
├── hooks/
├── www/
├── platforms/
├── plugins/
```

TIP

As a separate project inside Quasar, to add Cordova plugins, you need to call `plugman` or `cordova plugin add command` inside the `src-cordova` folder.

Configuring quasar.conf on Cordova

For Cordova development, you can set some special flags on the `quasar.conf.js` file in the `root` folder:

```
cordova: {
  iosStatusBarPadding: true/false, // add the dynamic top padding on
     // iOS mobile devices
  backButtonExit: true/false // Quasar handles app exit on mobile phone
      // back button
},
```

Commands

You can run the `Quasar-CLI` commands by opening Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and executing one of the following:

If you don't have a Cordova environment already configured on your desktop, you can find more information on how to set it up here: `https://quasar.dev/quasar-cli/developing-cordova-apps/preparation#Android-setup`.

- `quasar dev -m cordova -T android` – To start development mode as an Android Device Emulator
- `quasar build -m cordova -T android` – To build the code as Android
- `quasar dev -m cordova -T ios` – To start development mode as an iOS device emulator (macOS only)
- `quasar build -m cordova -T ios` – To start build mode as an iOS device emulator (macOS only)

Developing a desktop application (Electron)

To develop an SSR, we first need to inform Quasar that we want to add a new mode of development. Open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> quasar mode add electron
```

`Quasar-CLI` will create a folder called `src-electron`, which will have an Electron project inside.

The folder structure for Electron projects looks like this:

```
src-electron/
├── icons/
├── main-process/
├── electron-flag.d.ts
```

Inside the `icons` folder, you will find the icons that `electron-packager` will use when building your project. In the `main-process` folder will be your main Electron files, spliced into two files: one that will only be called on development and another that will be called on development and production.

Configuring quasar.conf on Electron

For Electron development, you can set some special flags on the `quasar.conf.js` file in the root folder:

```
electron: {
  // optional; webpack config Object for
  // the Main Process ONLY (/src-electron/main-process/)
  extendWebpack (cfg) {
    // directly change props of cfg;
    // no need to return anything
  },

  // optional; EQUIVALENT to extendWebpack() but uses webpack-chain;
  // for the Main Process ONLY (/src-electron/main-process/)
  chainWebpack (chain) {
    // chain is a webpack-chain instance
    // of the Webpack configuration
  },

  bundler: 'packager', // or 'builder'

  // electron-packager options
  packager: {
    //...
  },

  // electron-builder options
  builder: {
    //...
  }
},
```

The `packager` key uses the API options for the `electron-packager` module, and the `builder` key uses the API options for the `electron-builder` module.

Commands

You can run the `Quasar-CLI` commands by opening Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and executing one of the following:

- `quasar dev -m electron` – To start development mode as Electron
- `quasar build -m electron` – To build the code as Electron

How it works...

This is all possible because Quasar framework encapsulates the building, parsing, and bundling for you on the CLI. You don't need to worry about webpack and configurations with Electron, Cordova, or even Babel.

A simple CLI command can generate an entirely new page, layout, component, store, route, or even a new build for you. As the CLI is just a wrapper around Vue, webpack, Babel, and other tools, you are not tied to using only Quasar visual components. If you don't want to use them, it's possible to not import them and use the power of the CLI for building your application.

See also

You can check out more about Quasar framework in the documentation at `https://quasar.dev/introduction-to-quasar`.

Read more about SPA development with Quasar at `https://quasar.dev/quasar-cli/developing-spa/introduction`.

Read more about PWA development with Quasar at `https://quasar.dev/quasar-cli/developing-pwa/introduction`.

Read more about SSR development with Quasar at `https://quasar.dev/quasar-cli/developing-ssr/introduction`.

Read more about mobile application development with Quasar at `https://quasar.dev/quasar-cli/developing-cordova-apps/introduction`.

Read more about the Cordova project at `https://cordova.apache.org`.

Read more about desktop application development with Quasar at `https://quasar.dev/quasar-cli/developing-electron-apps/introduction`.

Read more about the Electron project at `https://electronjs.org/`.

Read more about `electron-packager` at `https://github.com/electron/electron-packager`.

Find the `electron-packager` options API at `https://electron.github.io/electron-packager/master/interfaces/electronpackager.options.html`.

Read more about `electron-build` at `https://www.electron.build/`.

Find the electron-build options API at `https://www.electron.build/configuration/configuration`.

Creating smarter Vue watchers and computed properties

In Vue, using watchers and computed properties is always an excellent solution to check and cache your data, but sometimes that data needs some special treatment or needs to be manipulated differently than expected. There are some ways to give these Vue APIs a new life, helping your development and productivity.

How to do it...

We will divide this recipe into two categories: one for the watchers and another for the computed properties. Some methods are commonly used together, such as the `non-cached` computed and `deep-watched` values.

Watchers

These three watcher recipes were selected to improve productivity and the final code quality. The usage of these methods can reduce code duplication and improve code reuse.

Using method names

All watchers can receive a method name instead of functions, preventing you from writing duplicated code. This will help you avoid re-writing the same code, or checking for values and passing them to the functions:

```
<script>
export default {
  watch: {
    myField: 'myFunction',
  },
  data: () => ({
    myField: '',
  }),
  methods: {
    myFunction() {
      console.log('Watcher using method name.');
    },
 },
};
</script>
```

Immediate calls and deep listening

You can set your watcher to execute as soon as it is created by passing a property immediately and make it execute no matter the value's depth of mutation by calling the `deep` property:

```
<script>
export default {
  watch: {
    myDeepField: {
      handler(newVal, oldVal) {
        console.log('Using Immediate Call, and Deep Watch');
        console.log('New Value', newVal);
        console.log('Old Value', oldVal);
      },
      deep: true,
      immediate: true,
    },
  },
  data: () => ({
    myDeepField: '',
  }),
};
</script>
```

Multiple handlers

You can make your watcher execute various handlers at the same time, without needing to set the watch handler to bind to a unique function:

```
<script>
export default {
  watch: {
    myMultiField: [
      'myFunction',
      {
        handler(newVal, oldVal) {
          console.log('Using Immediate Call, and Deep Watch');
          console.log('New Value', newVal);
          console.log('Old Value', oldVal);
        },
        immediate: true,
      },
    ],
  },
  data: () => ({
    myMultiField: '',
  }),
  methods: {
    myFunction() {
      console.log('Watcher Using Method Name');
    },
  },
};
</script>
```

Computed

Sometimes computed properties are just used as simple cache-based values, but there is more power to them. Here are two methods that show how to extract this power.

No cached value

You can make your computed property an always updated value, rather than a cached value, by setting the `cache` property to `false`:

```
<script>
export default {
  computed: {
    field: {
      get() {
        return Date.now();
      },
      cache: false,
    },
  },
};
</script>
```

Getter and setter

You can add a setter function to your computed property and make it a fully complete data attribute, but not bound to the data.

It's not recommended to do this, but it's possible, and in some cases, you may need to do it. An example is when you have to save a date in milliseconds, but you need to display it in an ISO format. Using this method, you can have the `dateIso` property `get` and `set` the value:

```
<script>
export default {
  data: () => ({
    dateMs: '',
  }),
  computed: {
    dateIso: {
      get() {
        return new Date(this.dateMs).toISOString();
      },
      set(v) {
        this.dateMs = new Date(v).getTime();
      },
    },
  },
};
</script>
```

See also

You can find more information about the Vue `watch` API at `https://vuejs.org/v2/api/#watch`.

You can find more information about the Vue `computed` API at `https://vuejs.org/v2/api/#computed`.

Creating a Nuxt.js SSR with Python Flask as the API

`Nuxt.js` is a server-side rendering framework that renders everything at the server and delivers it loaded. With this process, the page gets the power of SEO and fast API fetching before rendering.

Using it correctly, you can achieve a powerful SPA or PWA with other functions that weren't possible before.

In the backend, Python is an interpreted dynamic language that is fast and stable. With an active user base and quick learning curve, this is perfect for server APIs.

Joining both together, it is possible to get a powerful application deployed as fast as possible.

Getting ready

The pre-requisites for this recipe are as follows:

- Node.js 12+
- Python

The Node.js global object that is required is as follows:

- `create-nuxt-app`

To install `create-nuxt-app`, you need to execute the following command in Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows):

```
> npm install -g create-nuxt-app
```

For the backend of this recipe, we will use **Python**. The Python global objects required for this recipe are as follows:

- `flask`
- `flask-restful`
- `flask-cors`

To install `flask`, `flask-restful`, and `flask-cors`, you need to execute the following command in Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows):

```
> pip install flask
> pip install flask-restful
> pip install flask-cors
```

How to do it...

We will need to split our recipe into two parts. The first part is the backend part (or API if you prefer), which will be done with Python and Flask. The second part will be the frontend part, and it will run on `Nuxt.js` in SSR mode.

Creating your Flask API

Our API server will be based on the Python Flask framework. We will need to create a server folder to store our server files and start the development of the server.

You will need to install the following Python packages. To do so, open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following commands:

- To install the Flask framework, use the following command:

  ```
  > pip install flask
  ```

- To install the Flask RESTful extension, use the following command:

  ```
  > pip install flask-restful
  ```

- To install the Flask CORS extension, use the following command:

  ```
  > pip install flask-cors
  ```

Initializing the application

To create our simple RESTful API, we will create a single file and use SQLite3 as a database:

1. Create a folder named `server` and create a file named `app.py` in it:

   ```
   import sqlite3 as sql
   from flask import Flask
   from flask_restful import Resource, Api, reqparse
   from flask_cors import CORS
   ```

```
app = Flask(__name__)
api = Api(app)
CORS(app)

parser = reqparse.RequestParser()

conn = sql.connect('tasks.db')
conn.execute('CREATE TABLE IF NOT EXISTS tasks (id INTEGER PRIMARY
   KEY AUTOINCREMENT, task TEXT)')
conn.close()
```

2. Then, we will create our `ToDo` class, and on the constructor of the class, we will connect to the database and select all `tasks`:

```
class ToDo(Resource):
    def get(self):
        con = sql.connect('tasks.db')
        cur = con.cursor()
        cur.execute('SELECT * from tasks')
        tasks = cur.fetchall()
        con.close()

        return {
            'tasks': tasks
        }
```

3. To implement the RESTful POST method, create a function that receives `task` as an argument, and will add an object with the `task` that was added, the `status` of the addition to the tasks list, and then return the list to the user:

```
 def post(self):
        parser.add_argument('task', type=str)
        args = parser.parse_args()
        con = sql.connect('tasks.db')
        cur = con.cursor()
        cur.execute('INSERT INTO tasks(task) values ("
                    {}")'.format(args['task']))
        con.commit()
        con.close()

        return {
            'status': True,
            'task': '{} added.'.format(args['task'])
        }
```

4. Next, we will create the RESTful PUT method by creating a function that will receive the `task` and `id` as arguments of the function. Then, this function will update `task` with the current `id`, and return to the user the updated `task` and the `status` of the update:

```
def put(self, id):
        parser.add_argument('task', type=str)
        args = parser.parse_args()

        con = sql.connect('tasks.db')
        cur = con.cursor()
        cur.execute('UPDATE tasks set task = "{}" WHERE id =
           {}'.format(args['task'], id))
        con.commit()
        con.close()

        return {
            'id': id,
            'status': True,
            'task': 'The task {} was updated.'.format(id)
        }
```

5. Then, create a RESTful DELETE method by creating a function that will receive the `ID` of the `task`, which will be removed, and then will return to the user the `ID`, `status`, and the `task` that was removed:

```
 def delete(self, id):
        con = sql.connect('tasks.db')
        cur = con.cursor()
        cur.execute('DELETE FROM tasks WHERE id = {}'.format(id))
        con.commit()
        con.close()

        return {
            'id': id,
            'status': True,
            'task': 'The task {} was deleted.'.format(id)
        }
```

6. Finally, we will add the `ToDo` class as a resource to the API on the `'/'` route, and initialize the application:

```
api.add_resource(ToDo, '/', '/<int:id>')

if __name__ == '__main__':
    app.run(debug=True)
```

Starting the server

To start your server, you need to open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> python server/app.py
```

Your server will be running and listening on `http://localhost:5000`.

Creating your Nuxt.js server

To render your application, you will need to create your `Nuxt.js` application. Using the `Nuxt.js` `create-nuxt-app` CLI, we will create it and choose some options for it. Open Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and execute the following command:

```
> create-nuxt-app client
```

Then, you will be asked some questions about the installation process. We will use the following:

1. When you start creating your project with `Nuxt-CLI`, it will first ask for the project name. In our case, we will choose `client` as the name:

   ```
   Project Name: client
   ```

2. Then you need to choose the programming language that will be used in the project. We will select `JavaScript`:

   ```
   > Programming language: (Use arrow keys)
      ❯ JavaScript
        TypeScript
   ```

3. Next, `Nuxt-CLI` will ask for the package manager that will be used to install the dependencies. In our case, we choose `Yarn`, but you can choose the one you prefer:

   ```
   > Package manager: (Use arrow keys)
      ❯ Yarn
        npm
   ```

4. Now, `Nuxt-CLI` will ask for a UI framework to be used in the project. From the available list, choose `Bulma`:

   ```
   > UI Framework: Bulma
   ```

5. Then, `Nuxt-CLI` will ask whether you want to select extra modules for the project. We will select `Axios` from the current list of modules:

   ```
   > Nuxt.JS modules: Axios
   ```

6. `Nuxt-CLI` will ask for the linting tools we want to use on our project; we will choose `None`:

   ```
   > Choose Linting tools: None
   ```

7. Then, `Nuxt-CLI` will ask for the test framework we want to implement on our project; we will choose `None`:

   ```
   > Choose Test Framework: None
   ```

8. Next, `Nuxt-CLI` will ask for the rendering mode that will be used by the project; we will select `Universal (SSR)`:

   ```
   > Choose Rendering Mode: Universal (SSR)
   ```

9. `Nuxt-CLI` will ask for the deployment target that will be used on the building structure; we will choose `Server (Node.js hosting)`:

   ```
   > Deployment target: Server (Node.js hosting)
   ```

10. Finally, `Nuxt-CLI` will ask for the development tool configuration that we want to use; we will select `jsconfig.json`:

    ```
    > Development tools: jsconfig.json
    ```

After the CLI finishes the installation process, we can open the `client` folder on our editor or IDE.

Adding Bulma to the global CSS

To add Bulma to the application, we need to declare it in the `nuxt` configuration file by doing the following:

1. Open `nuxt.config.js,` in the `client` folder.
2. Then, update the CSS property and add the Bulma import, to make it available in the global scope of the application:

   ```
   export default {
     /* We need to change only the css property for now, */
     /* the rest we will maitain the same */
   ```

```
    /*
     ** Global CSS
     */
    css: ['bulma/css/bulma.css'],
  }
```

Configuring the axios plugin

To start creating our API calls, we need to add the `axios` plugin in our application:

1. To do so, we will need to open the `nuxt.config.js,` file in the root folder, and add the `axios` property:

   ```
   export default {
     /* We need to change only the axios property for now, */
     /* the rest we will maitain the same */
     axios: {},
   }
   ```

2. On the `axios` property, add the following configuration properties:
   - `HOST` and define it as `'127.0.0.1'`
   - `PORT` and define it as `'5000'`
   - `https` and define it as `false`
   - `debug` and define it as `true`:

     ```
     axios: {
       HOST: '127.0.0.1',
       PORT: '5000',
       https: false,
       debug: true, // Only on development
     },
     ```

Running the Nuxt.js server

Now that you have everything set, you want to run the server and start to see what is going on. `Nuxt.js` comes with some pre-programmed `npm` scripts out of the box. You can run one of the following commands by opening Terminal (macOS or Linux) or the Command Prompt/PowerShell (Windows) and executing the following:

- `npm run dev` – To run the server in development mode
- `npm run build` – To build the files with webpack and minify the CSS and JS for production

- `npm run generate` – To generate static HTML pages for each route
- `npm start` – To start the server in production, after running the build command

Creating the TodoList component

For the TodoList app, we will need a component that will fetch the tasks and delete the tasks.

Single file component <script> section

Here, we will create the `<script>` section of the single file component:

1. In the `client/components` folder, create a file named `TodoList.vue` and open it.
2. Then, we will export a `default` JavaScript object, with a `name` property defined as `TodoList`, then define the `beforeMount` life cycle hook as an asynchronous function. Define the `computed` and `methods` properties as an empty JavaScript object. Then, create a `data` property defined as a singleton function returning a JavaScript object. In the `data` property, create a `taskList` property as an empty array:

   ```
   export default {
     name: 'TodoList',
     data: () => ({
       taskList: [],
     }),
     computed: {},
     async beforeMount() {},
     methods: {},
   };
   ```

3. In the `computed` property, create a new property called `taskObject`. This `computed` property will return the result of `Object.fromEntries(new Map(this.taskList))`:

   ```
   taskObject() {
         return Object.fromEntries(new Map(this.taskList));
       },
   ```

4. In the `methods` property, create a new method called `getTask` – it will be an asynchronous function. This method will fetch the tasks from the server, then will use the response to define the `taskList` property:

   ```
   async getTasks() {
         try {
           const { tasks } = await
              this.$axios.$get('http://localhost:5000');
           this.taskList = tasks;
         } catch (err) {
           console.error(err);
         }
       },
   ```

5. Then, create a `deleteTask` method. This method will be an asynchronous function and will receive an `id` as a parameter. Using this parameter, it will execute an API execution to delete the task and then execute the `getTask` method:

   ```
   async deleteTask(i) {
         try {
           const { status } = await
              this.$axios.$delete(`http://localhost:5000/${i}`);
           if (status) {
             await this.getTasks();
           }
         } catch (err) {
           console.error(err);
         }
       },
   ```

6. Finally, in the `beforeMount` life cycle hook, we will execute the `getTask` method:

   ```
   async beforeMount() {
       await this.getTasks();
     },
   ```

Single file component <template> section

It's time to create the `<template>` section of the single file component:

1. In the `client/components` folder, open the `TodoList.vue` file.
2. In the `<template>` section, create a `div` HTML element, and add the `class` attribute with the value `box`:

   ```
   <div class="box"></div>
   ```

3. As a child of the `div.box` HTML element, create a `div` HTML element, with the `class` attribute defined as `content`, with a child element defined as an `ol` HTML element and the attribute `type` defined as `1`:

   ```
   <div class="content">
     <ol type="1"></ol>
   </div>
   ```

4. As a child of the `ol` HTML element, create a `li` HTML element, with the `v-for` directive defined as `(task, i) in taskObject`, and the `key` attribute defined as a variable, `i`:

   ```
   <li
     v-for="(task, i) in taskObject"
     :key="i">
   </li>
   ```

5. Finally, as a child of the `ol` HTML element, add `{{ task }}` as the inner text, and as a sibling of the text, create a `button` HTML element, the `class` attribute defined as `delete is-small`, and the `@click` event listener defined as the `deleteTask` method, passing the `i` variable as an argument:

   ```
   {{ task }}
   <button
     class="delete is-small"
     @click="deleteTask(i)"
   />
   ```

Creating the Todo form component

To send the task to the server, we will need a form. That means we need to make a form component that will handle this for us.

Single file component <script> section

Here, we will create the `<script>` section of the single file component:

1. In the `client/components` folder, create a file named `TodoForm.vue` and open it.
2. Then, we will export a `default` JavaScript object, with a `name` property defined as `TodoForm`, then define the `methods` property as an empty JavaScript object. Then, create a `data` property defined as a singleton function returning a JavaScript object. In the `data` property, create a `task` property as an empty array:

```
export default {
  name: 'TodoForm',
  data: () => ({
    task: '',
  }),
  methods: {},
};
```

3. In the `methods` property, create a method named `save`, which will be an asynchronous function. This method will send the `task` to the API, and if the API receives `Ok Status`, it will emit a `'new-task'` event with the `task` and clean `task` property:

```
async save() {
      try {
        const { status } = await
          this.$axios.$post('http://localhost:5000/', {
          task: this.task,
        });
        if (status) {
          this.$emit('new-task', this.task);
          this.task = '';
        }
      } catch (err) {
        console.error(err);
      }
    },
```

Single file component <template> section

It's time to create the `<template>` section of the single file component:

1. In the `client/components` folder, open the `TodoForm.vue` file.

2. In the `<template>` section, create a `div` HTML element, and add the `class` attribute with the value `box`:

   ```
   <div class="box"></div>
   ```

3. Inside the `div.box` HTML element, create a `div` HTML element with the `class` attribute defined as `field has-addons`:

   ```
   <div class="field has-addons"></div>
   ```

4. Then, inside the `div.field.has-addons` HTML element, create a child `div` HTML element, with the `class` attribute defined as `control is-expanded`, and add a child input HTML element with the `v-model` directive defined as the `task` property. Then, define the `class` attribute as `input`, the `type` attribute as `text`, and `placeholder` as `ToDo Task`. Finally, in the `@keypress.enter` event listener, define the `save` method:

   ```
   <div class="control is-expanded">
     <input
       v-model="task"
       class="input"
       type="text"
       placeholder="ToDo Task"
       @keypress.enter="save"
     >
   </div>
   ```

5. Finally, as a sibling of the `div.control.is-expanded` HTML element, create a `div` HTML element, with the `class` attribute defined as `control`, and add a child `a` HTML element, with the `class` attribute defined as `button is-info`, and on the `@click` event listener, define it as the `save` method. As inner text of the `a` HTML element, add the `Save Task` text:

   ```
   <div class="control">
     <a
       class="button is-info"
       @click="save"
     >
       Save Task
     </a>
   </div>
   ```

Creating the layout

Now we need to create a new layout to wrap the application as a simple high-order component. In the `client/layouts` folder, open the file named `default.vue`, remove the `<style>` section of the file, and change the `<template>` section to the following:

```
<template>
    <nuxt />
</template>
```

Creating the page

Now we will create the main page of our application, where the user will be able to view their `TodoList` and add a new `TodoItem`.

Single file component <script> section

Here, we will create the `<script>` section of the single file component:

1. Open the `index.vue` file in the `client/pages` folder.
2. Import the `todo-form` and the `todo-list` component that we created, then we will export a `default` JavaScript object, with a `components` property with the imported components:

```
<script>
import TodoForm from '../components/TodoForm.vue';
import TodoList from '../components/TodoList.vue';

export default {
  components: { TodoForm, TodoList },
};
</script>
```

Single file component <template> section

It's time to create the `<template>` section of the single file component:

1. In the `client/pages` folder, open the `index.vue` file.
2. In the `<template>` section, create a `div` HTML element, add as a child a `section` HTML element, with the `class` property defined as `hero is-primary`. Then, as a child of the `section` HTML element, create a `div` HTML element, with the `class` attribute defined as `hero-body`. As a child of the `div.hero-body` HTML element, create a `div` HTML element with the `class` attribute defined as `container` and add as a child an `h1` HTML element with `class` defined as `title`, with the inner text as `Todo App`:

```
<section class="hero is-primary">
  <div class="hero-body">
    <div class="container">
      <h1 class="title">
        Todo App
      </h1>
    </div>
  </div>
</section>
```

3. As a sibling of the `section.hero.is-primary` HTML element, create a `section` HTML element, with the `class` attribute defined as `section` and the `style` attribute defined as `padding: 1rem`. Add as a child a `div` HTML element with the `class` attribute defined as `container` with a child `todo-list` component with the `ref` attribute defined as `list`:

```
<section
  class="section"
  style="padding: 1rem"
>
  <div class="container">
    <todo-list
      ref="list"
    />
  </div>
</section>
```

4. Finally, as a sibling of the `section.section` HTML element, create a `section` HTML element, with the `class` attribute defined as `section` and the `style` attribute defined as `padding: 1rem`. Add as a child a `div` HTML element with the `class` attribute defined as `container` with a child `todo-form` component, with the `@new-task` event listener defined as `$refs.list.getTasks()`:

```
<section
  class="section"
  style="padding: 1rem"
>
  <div class="container">
    <todo-form
      @new-task="$refs.list.getTasks()"
    />
  </div>
</section>
```

How it works...

This recipe shows the integration between a local API server via Python and an SSR platform served via `Nuxt.js`.

When you start the Python server first, you are opening the ports to receive data from clients as a passive client, just waiting for something to happen to start your code. With the same process, the `Nuxt.js` SSR can do a lot of stuff behind the scenes, but when it finishes, it goes idle, waiting for user action.

When the user interacts with the frontend, the application can send some requests to the server that will be handed back to the user with data, to be shown on the screen.

See also

You can learn more about Flask and the HTTP project inside Python at `https://palletsprojects.com/p/flask/`.

If you want to learn more about `Nuxt.js`, you can read the documentation at `https://nuxtjs.org/guide/`.

If you want to learn more about the `Nuxt.js` implementation of Axios and how to configure it and use the plugin, you can read the documentation at `https://axios.nuxtjs.org/options`.

If you want to learn more about Bulma, the CSS framework used in this recipe, you can find more information at `https://bulma.io`.

The dos and don'ts of Vue applications

Security is always something everyone is worried about, and this is no different for technology. You need to be aware and alert all the time. In this section, we'll look at how you can prevent attacks with some techniques and simple solutions.

Linters

When using ESLint, make sure you have enabled the Vue plugin, and you are following the strongly recommended rules. Those rules will help you with the development, checking for some common mistakes that can open doors to attacks such as the `v-html` directive.

In a `Vue-CLI` project, with the options for linters selected, a file named `.eslintrc.js` will be created along with the project files. In this file, a set of basic rules will be pre-determined. The following is an example of a set of good practice rules for an `ESLint + AirBnb` project:

```
module.exports = {
  root: true,
  env: {
    node: true,
  },
  extends: [
    'plugin:vue/essential',
    'plugin:vue/recommended',
    'plugin:vue/strongly-recommended',
    '@vue/airbnb',
  ],
parserOptions: {
    parser: 'babel-eslint',
  },
rules: {
 'no-console': process.env.NODE_ENV === 'production' ? 'error' : 'off',
 'no-debugger': process.env.NODE_ENV === 'production' ? 'error' : 'off',
 },
};
```

Now, if you have any code that breaks the lint rules, it won't be parsed on development or build.

JavaScript

JavaScript has some vulnerabilities that can be prevented by following some simple checklists and simple implementations. Those implementations can be in client-server communications or DOM manipulation, but you always need to be careful not to forget them.

Here are some tips for using JavaScript:

- Always use an authenticated and encrypted API when possible. Remember that JWT isn't encrypted by itself; you need to add the layer of encryption *(JWE)* to have the whole JSON.
- Always use `SessionStorage` if you want to store an API token.
- Always sanitize the HTML input from the user before sending it to the server.
- Always sanitize the HTML before rendering it to the DOM.
- Always escape any `RegeExp` from the user; it will be executed, to prevent any CPU thread attack.
- Always catch errors and don't show any stack trace to the user, to prevent any code manipulation.

Here are some tips on what not to do when using JavaScript:

- Never use `eval()`; it makes your code run slowly and opens a door for malicious code to execute inside your code.
- Never render any input from the user without any sanitization or escaping the data.
- Never render any HTML on the DOM without any sanitization.
- Never store an API token on `LocalStorage`.
- Never store sensitive data in the JWT object.

Vue

When developing a Vue application, you need to check for some basic rules that can help the development and won't open any doors for the external manipulation of your application.

Here are some tips for using Vue:

- Always add type validation to your props, and if possible, a validator check.
- Avoid the global registration of components; use local components.
- Always use lazy-loaded components, when possible.
- Use `$refs` instead of direct DOM manipulation.

Here are some tips on what not to do when using Vue:

- Never store `Vue`, `$vm`, `$store`, or any application variable on the window or any global scope.
- Never modify the Vue prototype; if you need to add a new variable to the prototype, make a new Vue plugin.
- It's not recommended to use a direct connection between components, as it will make the component bound to the parent or child.

See also

You can find more information about XSS (cross-site scripting) on OWASP CheatCheat at `https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.md` and about HTML XSS at `https://html5sec.org/`.

Find more information about `eslint-vue-plugin` at `https://eslint.vuejs.org/`.

You can read more about Node.js security best practices at `https://github.com/i0natan/nodebestpractices#6-security-best-practices`.

Find more information about the dos and don'ts of a Vue application at `https://quasar.dev/security/dos-and-donts`.

Other Books You May Enjoy

Leave a review - let other readers know what you think

Index