

# Vue CLI 3 Quick Start Guide

Build and maintain Vue.js applications quickly with the standard CLI

**Packt**>

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

Ajdin Imsirovic

# Vue CLI 3 Quick Start Guide

Build and maintain Vue.js applications quickly with the standard CLI

**Ajdin Imsirovic**



**BIRMINGHAM - MUMBAI**

# Vue CLI 3 Quick Start Guide

Copyright © 2019 Packt Publishing

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

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

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

**Commissioning Editor:** Pavan Ramchandani

**Acquisition Editor:** Siddharth Mandal

**Content Development Editor:** Roshan Kumar

**Technical Editor:** Snehal Dalmet

**Copy Editor:** Safis Editing

**Project Coordinator:** Namrata Swetta

**Proofreader:** Safis Editing

**Indexer:** Priyanka Dhadke

**Graphics:** Jisha Chirayil

**Production Coordinator:** Aparna Bhagat

First published: April 2019

Production reference: 1300419

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78995-034-2

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

Thanks to my family and friends for their support during the writing of this book.

– *Ajdin Imsirovic*





[mapt.io](http://mapt.io)

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

## Packt.com

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

At [www.packt.com](http://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

# Contributors

## About the author

**Ajdin Imsirovic** is a full-stack web developer who has published several courses on the subject of web design and web development. He has also authored three books, *Bootstrap 4 Cookbook*, *Elm Web Development*, and *Vue.js Quick Start Guide*. In his fourth book, *Vue CLI 3 Quick Start Guide*, he introduces readers to the rich ecosystem of frontend tooling and best practices.

## About the reviewer

**Bogdan-Alin Bâlc** is a software architect/tech lead with a passion for UI technologies. He has worked with JavaScript for the past 13 years, from the emergence of jQuery and Ajax to state-of-the-art solutions such as React, Angular, Ember.js, and Vue.js. When he is not fiddling with some new coding challenge, he spends his time with friends and family, watching sports or movies. Nowadays, he channels most of his efforts into making Vulcan Digital Reforging, one of the most successful and creative start-ups in Cluj. His passion for Vue.js pushed him to collaborate on other great books, such as *Learning Vue.js*, *Vue.js 2 Cookbook*, and *Vue.js by Example*.

## Packt is searching for authors like you

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

# Table of Contents

<b>Preface</b>	1
<b>Chapter 1: Introducing Vue CLI 3</b>	7
<b>Technical requirements</b>	8
<b>Setting up Vue CLI 3 on our system</b>	8
Is Vue CLI 3 already available?	8
Installing Node.js using Node Version Manager	9
Why install NVM?	9
Installing NVM on Windows	9
<b>Installing Vue CLI 3</b>	10
<b>Installing VS Code</b>	12
<b>Using Vue CLI with no configuration</b>	12
<b>The benefits of using Vue CLI 3</b>	16
<b>Default toolchain, fatigue level zero</b>	16
Vue CLI 3 default app via the command line	16
Vue CLI 3 default app via the UI	18
<b>Summary</b>	23
<b>Chapter 2: Webpack in Vue CLI 3</b>	24
<b>The evolution of the JS language</b>	25
The script tag	25
Immediately Invoked Function Expressions	26
The problem with IIFEs	26
Sharing third-party libraries in a team environment with NPM	27
JavaScript task runners and NPM scripts	30
Modules in JavaScript	32
Working with modules in Node.js	32
Module bundlers, a way to use modules in the browser	34
<b>What is webpack?</b>	35
How webpack works	37
Building a new project and running webpack on it	37
Running webpack on a project	40
<b>Adding a Vue project via NPM and using webpack with it</b>	44
Adding our Vue component as a JavaScript module	45
Compiling JavaScript modules with webpack	46
Adding options with the help of the webpack configuration file	47
Adding an HTML file so that we can render our Vue component	47
Giving webpack the ability to output an HTML file	48
Adding a .vue file as a JavaScript module	49
Adding a webpack loader to work with .vue files	50
Fixing the issue with our index.html file	52

Delivering HTML files via webpack with the help of html-webpack-plugin	53
<b>Understanding Hot Module Replacement in Vue</b>	54
Observing HMR updates	55
<b>Summary</b>	57
<b>Chapter 3: Babel in Vue CLI 3</b>	58
<b>Understanding Babel</b>	58
The building blocks of @vue/cli-plugin-babel	61
Babel 7 in @vue/cli-plugin-babel	61
The role of babel-loader in @vue/cli-plugin-babel	62
The role of @vue/babel-preset-app	62
Other features of @vue/cli-plugin-babel	62
<b>Using Babel and webpack in Vue 2 without Vue CLI</b>	63
Installing the necessary NPM packages	63
Running webpack with ES5 code	66
Adding webpack-dev-server	67
Updating the methods option to ES6 syntax	68
Adding babel-loader to our webpack configuration	68
Updating our webpack configuration to work with babel	70
<b>The problem with the this keyword in arrow functions in Vue</b>	71
Recommended solutions to keyword problems	71
Adding one counter app	71
<b>Summary</b>	74
<b>Chapter 4: Testing in Vue CLI 3</b>	75
<b>Understanding Vue plugins</b>	75
<b>Beginning testing on a brand new Vue app</b>	76
Adding a new project with Vue CLI UI	76
Adding the Jest plugin to our Vue app	79
Examining the updates	83
Adding the code for our Vue application	84
Writing our first unit test inside a Vue app with Jest	86
Running our first unit test inside the Vue CLI UI	87
Importing mount from test-utils and writing another unit test	90
Mounting the component to be tested	91
Writing a failing assertion	92
Writing a passing assertion	93
Triggering a button click in our test	93
<b>Test-driven development in Vue CLI 3</b>	95
<b>Improving our tests in Vue CLI 3</b>	95
Watching our tests in Vue CLI 3	96
<b>Writing end-to-end tests with Cypress</b>	97
Introducing Cypress	97
Adding a Cypress plugin to our project	98
Verifying updates to package.json made by the Cypress plugin installation	98
Updating Cypress tests in our Vue app	100

<b>Summary</b>	102
<b>Chapter 5: Vue CLI 3 and Routing</b>	103
<b>Adding a new Vue project with vue-router and vuex</b>	103
Configuring preset options through VS Code's command line	105
Adding vue-router and vuex	106
Adding ESLint and Prettier	107
Finishing the configuration	107
Installing all the plugins for our new project	108
Serving our project from the Vue CLI UI	108
Running the serve task from the UI	109
<b>Working with routes in vue-router</b>	110
Inspecting the two default routes	110
Inspecting project files	111
The contents of main.js	112
The App.vue file and router-link navigation	112
Rendering a Vue instance and mounting it into #app	113
Running the build task from the UI	114
Serving the app in production mode from the UI	114
Some basics of working with routes	115
The router-view tag needs to be nested	116
Using named routes	116
Navigating to a route by triggering a method of a Vue instance	117
Working with nested routes	119
Lazy- loading routes	122
Working with dynamic routes	122
<b>Summary</b>	123
<b>Chapter 6: Using ESLint and Prettier in Vue CLI 3</b>	124
<b>What ESLint is, and how to configure it on its own</b>	124
Installing npm packages globally versus installing npm packages locally	125
What is Prettier?	126
Running locally-installed npm modules on the command line using npx	126
Installing Prettier globally and running it on index.html	128
Running Prettier with npx	129
Working with ESLint	129
Installing ESLint globally and unsuccessfully running it from the command line	130
Running eslint --init command	131
Understanding the structure of .eslintrc.js config file	133
Running ESLint on a JavaScript file	134
Updating error level rules in ESLint	135
<b>Configuring ESLint in Vue CLI 3</b>	136
Setting up ESLint configurations in Vue CLI 3 GUI	137
<b>Using ESLint and Prettier together in a Vue CLI 3 UI project</b>	139
<b>Summary</b>	141
<b>Chapter 7: Improving CSS with SCSS</b>	142

<b>Installing Bootstrap + Vue</b>	142
Adding BootstrapVue as a plugin	143
<b>Using the bootstrap-vue plugin to add styles to our project</b>	144
<b>Using SCSS in our project</b>	146
Dealing with errors	146
Writing some SCSS in our project	147
Overriding bootstrap-vue styles	148
Using variables, mixins, and interpolation	149
Using Sassmeister to practice SCSS	149
Using SCSS mixins and interpolation in our Vue project	152
<b>Summary</b>	154
<b>Chapter 8: Deploying Vue CLI 3 Apps on GitHub Pages</b>	155
<b>Installing GitHub Desktop app</b>	156
<b>Understanding how Git works</b>	157
Starting a new Vue app and tracking changes with Git	158
Committing changes to our app	160
The three trees concept	161
Branching in Git	161
Merging branches	162
<b>Registering a new GitHub account</b>	163
<b>Adding origin/master using GitHub Desktop</b>	164
Publishing your local repository	164
<b>Publishing our project on GitHub Pages</b>	167
Getting our project ready to be published	167
Updating the Vue configuration file	171
Saving changes and pushing them to the remote repository with GitHub Desktop	172
Debugging your GitHub pages deployment	174
<b>Summary</b>	175
<b>Other Books You May Enjoy</b>	176
<b>Index</b>	179

---

# Preface

Vue began as a project of a single person, Evan You. It is amazing that it has gotten to the point where it is today: a contender for the spot of the favorite frontend framework, competing with the corporate-backed React and Angular.

Of course, these are not the only frontend frameworks, but the trio of Vue, React, and Angular seem to be the most popular, with the internet buzzing with comparisons of these frameworks and experiences with them. It's not uncommon to come across an article that compares Vue and React, for example, or a blog post on ten ways that Vue is better than Angular. Regardless of whether these articles are someone's opinion, or clickbait, or statement of the facts, there must be *some truth* to these claims.

What was the reason for Vue's success? Dedication, hard work, or luck? Probably a bit of each. But another key to Vue's success was Evan's obvious priority of making things easy for the developer. Vue's no longer developed by a single person, but it is still very approachable. The community has kept the gist of what Vue was all about from the beginning: an easy-to-use framework that gets out of your way and just lets you code.

Vue CLI is another example of this. Besides a command-line interface that's up to par with other modern frontend frameworks, Vue CLI 3 sets a new standard in frontend JavaScript frameworks, and comes with a **graphical user interface (GUI)**. This interface makes it a breeze to set up, extend, run, and serve a Vue project.

When you pair this addition of a GUI with the successful attempt to ease the pain of toolchain fatigue by offering a really well thought-out setup process, you get a really powerful combination, and happy developers as a result.

## Who this book is for

This book is for web developers and JavaScript developers who want to learn more about Vue CLI 3. The readers must have basic knowledge of HTML/CSS and JavaScript. Essentially, the reader should also be familiar with basic operational system workflow, such as using a UNIX-like command-line interface, including Git Bash, Windows PowerShell, or any related command-line tool.



This book dives into the technological building blocks of Vue CLI 3. This is not a book about writing apps in Vue. It is more of a foundational book that will help you understand how things work inside Vue CLI. If you were never completely sure how NPM works and how to use it properly, this book will explain that, through the lens of Vue CLI 3. Similarly, we'll look at webpack, HMR, using single-file `.vue` components, SCSS, ECMAScript, unit testing with Jest, and E2E testing with Cypress.

## What this book covers

Chapter 1, *Introducing Vue CLI 3*, explains how to use Vue CLI 3, and why you should use it. It covers best practices and what you'll get out of using Vue CLI 3. We'll set up Node version manager and NPM, install Vue CLI 3, and show how to start a new app through the command line or with the help of the GUI.

Chapter 2, *Webpack in Vue CLI 3*, takes the reader through an overview of the evolution of JavaScript over the last several years, which lead to the appearance of webpack. It explains some background concepts: NPM and NPM scripts, CommonJS, modules in JS and Node.js, along with module bundlers and how they can be used in the browser. Further on, we cover webpack, how it works, and how to run it. We close off the chapter with a step-by-step explanation of how to add a Vue project via NPM and use webpack with it. Essentially, we're setting up a Vue toolchain by hand, so that we can appreciate what Vue CLI 3 does for us automatically.

Chapter 3, *Babel in Vue CLI 3*, examines how to work with Babel and what are the benefits of using it. We inspect the building blocks of Vue's core Babel plugin, including `Babel 7`, `babel-loader`, and `@vue/babel-preset-app`. We also look at the differences between running webpack with ES5 and ES6, and updating our webpack configuration so that it understands Babel.

Chapter 4, *Testing in Vue CLI 3*, covers using Vue plugins, with a focus on plugins that are used for testing. We add the Jest plugin to our Vue app, run unit tests with Jest, and show some additional techniques and workflows inside Vue CLI 3 GUI, including running tasks from the **Project tasks** page and running unit tests inside the GUI. We discuss **test-driven development (TDD)** and working with assertions, and close the chapter with an overview of Cypress.

Chapter 5, *Vue CLI 3 and Routing*, discusses adding Vue projects with `vue-router` and `vuex`, configuring preset options, and understanding `vue-router` topics. These include named routes, dynamic routes, navigating to routes using methods in Vue instances, working with child routes, and lazy-loading routes.

Chapter 6, *Using ESLint and Prettier in Vue CLI 3*, shows us what ESLint is and why it's useful. We also look at Prettier, a handy code formatter that formats your code on every save. We discuss linters in general and what they're useful for.

Chapter 7, *Improving CSS with SCSS*, describes the basics of SCSS and shows how it's different from CSS and what additional features are available. We use a simple app we built in Chapter 5, *Vue CLI 3 and Routing*, and we see how to improve its styling by adding the bootstrap-vue plugin to the app. Working with SCSS in VDOM libraries can sometimes get confusing, and in this chapter, we see one possible workflow option in practice.

Chapter 8, *Deploying Vue CLI 3 Apps on GitHub Pages*, explains what Git is and how to set it up. We discuss some basics, including tracking changes with Git and committing changes in our app. We continue by discussing the three trees concept, branching, and merging branches. We register a GitHub account, add the origin/master using GitHub Desktop, and look at how to publish a local repository. We finally discuss how to deploy a Vue app on GitHub pages using the *subtree* functionality.

## To get the most out of this book

To get the most of this book, you should be somewhat familiar with using Windows, HTML, CSS, the very basics of JavaScript, and using a command-line tool such as Git Bash for Windows. It would be beneficial to have some familiarity with Node, NPM, and some basic command line utilities, but this is not compulsory.

## Download the example code files

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

You can download the code files by following these steps:

1. Log in or register at [www.packt.com](http://www.packt.com).
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

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

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

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Vue-CLI-3-Quick-Start-Guide>. In case there's an update to the code, it will be updated on the existing GitHub repository.

## Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: [http://www.packtpub.com/sites/default/files/downloads/9781789950342\\_ColorImages.pdf](http://www.packtpub.com/sites/default/files/downloads/9781789950342_ColorImages.pdf).

## Conventions used

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

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

A block of code is set as follows:

```
{
  "name": "vue-from-npm",
  "version": "1.0.0",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
}
```

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

```
let CustomArticle = Vue.component('custom-article', {
  template: `
    <article>
      Our own custom article component!
    </article>`
})
```

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

```
mkdir new-project-with-webpack && cd new-project-with-webpack
```

**Bold:** Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."



Warnings or important notes appear like this.



Tips and tricks appear like this.

## Get in touch

Feedback from our readers is always welcome.

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

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

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

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

## **Reviews**

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

For more information about Packt, please visit [packt.com](http://packt.com).

# 1 Introducing Vue CLI 3

This book introduces you to **Vue CLI 3** and answers questions such as how to use it, why to use it, the best practices, and what you'll get out of it.

In this chapter, we'll look at how to set up Vue CLI 3 on our system. We'll begin by checking to see whether Vue CLI 3 is already available, and we'll see the exact steps to take if we either need a fresh installation or an update from a previous version.

We'll then look at how to install the **Node Version Manager (NVM)**, and why that is a better alternative than simply installing Node. We'll see how easy it is to get started with Vue CLI 3 in VS Code, and how to integrate all these tools by using our command line.

We'll also discuss why Vue CLI 3 is great to use and work with, and we'll see this in practice, by running a default Vue CLI 3 app, both from the command line and by using the built-in Vue CLI 3 UI functionality.

The topics we will cover in this chapter include the following:

- Setting up Vue CLI 3 on your system
- Installing Vue CLI 3
- Installing VS Code
- Using Vue CLI with no configuration
- The benefits of using Vue CLI 3
- Avoiding JavaScript fatigue with the default toolchain

We'll begin this chapter by setting up Vue CLI 3.

## Technical requirements

There are only a few technical requirements that we need; they are as follows:

- A Windows installation (Windows 7 or greater)
- An installation of NVM for Windows (the specific steps for the installation are described in the chapter)
- An installation of VS Code (code editor)

Let's begin setting up Vue CLI 3 on our system.

## Setting up Vue CLI 3 on our system

A common way to work with Vue CLI 3 is via a command-line app called the **Command-Line Interface (CLI)**, where we run our Vue CLI 3 commands. Another prerequisite is to have Node.js installed on our machine.

If you are working on a shared computer, say, within a team of your fellow developers, the chances are that you already have all the prerequisites. In that case, you could just verify that you can start using Vue CLI 3 right away by running a couple of checks.

## Is Vue CLI 3 already available?

To quickly check whether you can run Vue CLI 3 right now and skip all the installation steps, run the following command in your command-line app:

```
node --version
```

Also, check for Vue CLI 3 with this command:

```
vue -V
```

If you get back any version of Node above 8.9 (ideally, above 8.11.0), you're good to go. Obviously, for Vue CLI, you'd want any version above 3.0.0.

Additionally, if you have a version of Vue CLI lower than V3, or you'd like to update to the newest Vue CLI, such as 3.3.0, just run this command:

```
npm install @vue/cli
```

What if you don't have Node.js or Vue CLI installed?

We'll use `nvm` or `nvm-windows` for Node, and after that, we'll install Vue CLI 3.

## Installing Node.js using Node Version Manager

What is the recommended version of Node.js we should be using? This information is available at the following link: <https://cli.vuejs.org/guide/installation.html>.

Currently, as of early 2019, to get the best results with Vue CLI, the minimum version of Node required is 8.11.0+, but you can kind of get by with 8.9 if you really have to.

This brings us to another important decision: the installation of NVM.

## Why install NVM?

While it is not absolutely necessary to install NVM in order to run Vue CLI 3 on your system, installing NVM is desirable for a couple of reasons.

First, you never know when Node will get a recommended update with security fixes, which usually means you'd be better off installing the update on your machine.

Second, if you need to run a different technology, other than Vue, this other technology might also require a different version of Node. To easily switch between these required Node installations on your system, you can simply install NVM.

## Installing NVM on Windows

You can download the NVM for Windows from this address:

```
https://github.com/coreybutler/nvm-windows/releases
```



Locate the `nvm-setup.zip` file, download and extract `nvm-setup.exe` from it, and then install it by following these installation steps:

1. Open the **Run** prompt by pressing Windows + R. Type `cmd` into the prompt.
2. While inside the prompt, press `Ctrl + Shift + Enter`. This will run Command Prompt with administrator privileges, which is required for the next step.
3. Visit <https://nodejs.org>, and see the current **Long-term Support (LTS)** version number. For example, currently, on 64-bit Windows, the LTS version is 10.15.1.
4. To install it, run the following command in Command Prompt with administrator privileges:

```
nvm install 10.15.1
```

5. Command Prompt will log out the following message:

```
Downloading node.js version 10.15.1 (64-bit) ...
```

6. Once the download is finished, we can use the downloaded version of Node. We do it with the following command:

```
nvm use 10.15.1
```

7. Finally, you can verify whether the installation was successful by running the following command:

```
node --version
```

8. If you're curious about the version of `npm` that came with your Node installation, simply run the following:

```
npm --version
```

Next, we'll install Vue CLI 3.

## Installing Vue CLI 3

We can install Vue CLI 3 with either `npm` or `yarn`. Since `npm` comes bundled with the Node.js installation, we'll use `npm`:

```
npm install -g @vue/cli --loglevel verbose
```

The preceding command installs Vue CLI 3 globally. That's what the `-g` flag is for. The `@vue/cli` syntax is what we use in Vue CLI 3, and `--loglevel verbose` will log out the details of our installation, which is very useful, especially on slower connections and slower machines, where sometimes we could start wondering whether our console froze. With `--loglevel verbose`, there's more clarity, which is always good.

Once done, let's double-check the Vue CLI version installed by running this command:

```
vue --version
```

Here are a few other useful commands that you should try in your console:

```
vue -h
```

Note that `vue -h` is an alias for `vue --help`. I'm using the former as it's easier to type.

Also, note that you can run the `-h` flag on each individual `vue` command, for example:

```
vue create -h  
vue add -h  
vue invoke -h  
vue inspect -h  
vue serve -h  
vue build -h  
vue ui -h  
vue init -h  
vue config -h  
vue upgrade -h  
vue info -h
```

Running any of the preceding lines will return the usage instructions for the specific command, a description of what it does, and the options (the flags) to append to each individual command. Obviously, the `-h` flag is a great way to explore the capabilities of Vue CLI, and to refresh your memory on the fly, if need be.

Next, we'll install our code editor of choice, VS Code.

## Installing VS Code

To install VS Code, simply navigate to <https://code.visualstudio.com>, and then download the appropriate version for your operating system.



If you're not sure whether you're using a 32-bit or a 64-bit computer, you can quickly check that on Windows by running the following command in Command Prompt (with admin rights):

```
wmic os get osarchitecture
```

The output will be `OSArchitecture`, and on the next line, either `32-bit` or `64-bit`.

Once VS Code is downloaded, simply install it by running the downloaded installation file and follow the installation instructions.

After you've installed VS Code, you'll get an additional command available via the command line, `code`.

The `code` command is very useful, as we'll see in the next section.

## Using Vue CLI with no configuration

In this section, we'll see the quickest and the easiest way to start using Vue CLI. It includes no configuration whatsoever! The reason to use Vue CLI without configuration is to run some quick experiments without having to answer prompts about our project's configuration that Vue CLI normally asks when we run a project with configuration steps included (which is the default approach to building apps using Vue CLI).

To begin, hold down the *Shift* key and right-click on an empty area on your desktop. From the contextual menu that pops up, click the **Open Command window here** command.

Once open, type the following command:

```
mkdir noConfig
```

This will make a new directory called `noConfig`. Next, let's change into that directory with the `cd` command:

```
cd noConfig
```

Finally, start VS Code from Command Prompt with this command:

```
code .
```

The dot in the preceding command means open VS Code in the current folder. Feel free to close the **Welcome** tab.

Next, use the `Alt + F` keyboard shortcut to open the **File** menu, and press the `N` key to open a brand new file.

In the new file, which opens in the tab that reads `Untitled-1`, type the following code:

```
<template>
  <h1>What's up, Vue CLI 3?</h1>
  <hr>
</template>
```

Next, press the `Ctrl + S` keyboard shortcut, and save the file as `App.vue`.

VS Code will save the file. It will give a new icon, the Vue logo icon, a visual cue that the file you just saved is indeed a Vue file.

VS Code might also prompt you to install an extension, `Vetur`, with the following prompt:

```
The 'Vetur' extension is recommended for this file type.
```

Go ahead and install the extension by clicking the **Install** button at the bottom of the popup.



Note that installing the `Vetur` extension has nothing to do with using the Vue CLI 3 with no configuration, but it has to do with us being more productive in VS Code when working with Vue.

Now we can serve our Vue app by running `vue serve`. But, before we actually run the command, let's use the `-h` flag to see what we have available:

```
vue serve -h
```

This is what we'll get back:

```
Usage: serve [options] [entry]

serve a .js or .vue file in development mode with zero config

Options:
  -o, --open Open browser
  -c, --copy Copy local url to clipboard
  -h, --help Output usage information
```

Now that we know what to expect, let's serve our Vue app with the following command:

```
vue serve -o -c
```

So, as we can see as mentioned previously, this command will serve our Vue app and open it in the browser. It will also copy the served URL to the clipboard. This allows us to, for example, open a different, non-default browser, and easily paste in the URL in the browser's address bar, so that we can preview our app there as well.

However, we'll come across a minor stumbling block.

Instead of our Vue app being served, we'll get this notification in our command:

```
Command vue serve requires a global addon to be installed.
Please run npm install -g @vue/cli-service-global and try again.
```

This is an easy fix. Better yet, we'll extend the preceding command with `--loglevel verbose`:

```
npm install -g @vue/cli-service-global --loglevel verbose
```

After some time, depending on your download speed, you'll get the `npm info ok` message.

This means that you can now run the `vue serve` command again:

```
vue serve -o -c
```

This time it works! Kind of..

Now we get an error that reads `Failed to compile with 1 errors`. And then, further down, we see the root cause:

```
Component template should contain exactly one root element.
```

There are a few ways to solve this issue, but what it basically says is that we can wrap our `h1` and `hr` tags inside a `div` tag, and we'll be good to go. So, let's update the `App.vue` file in VS Code to this:

```
<template>
  <div>
    <h1>What's up, Vue CLI 3?</h1>
    <hr>
  </div>
</template>
```

Make sure to save your changes, and now, finally, let's serve it again:

```
vue serve -o -c
```

You might get a bit of a surprise, because a new tab will open with the app loaded automatically, in your default browser.

Let's say that your default browser is Chrome. Let's open another browser (for example, Firefox), click inside the admin bar, and press the `Ctrl + V` shortcut to paste in the contents of the clipboard. Of course, it's going to be `http://localhost:8080/`.

The ease with which we've performed these repetitive tasks of opening our app in the browser and copying its URL with the help of `-o` and `-c` flags is just the tip of the iceberg. Vue CLI 3 has a lot more in store to help us code our apps faster and easier.

For example, let's get back to our code and erase the line with the `hr` tag, and then save the file. Look at your browser tab, the one with our Vue app open. It will auto-refresh, reflecting the changes in code. That's webpack in action, running under the hood of Vue CLI 3, watching for changes to our Vue files, and hot-reloading the app in the browser accordingly.

If you've been coding for more than a couple of years, you'll appreciate the ease of this workflow. In the past, we either had to set up our tools so that they performed the auto-refresh of our apps in the browser whenever we save our files, or we had to set up our IDE or code editor, or both. Even up until recently, we still had to tinker with webpack to automate this kind of a workflow, and this as with anything coding-related, sometimes did not go as smoothly as we'd hope.

With Vue CLI 3, this is all automated and made very simple.

Let's see some other ways in which Vue CLI 3 helps us code better and be more productive.

## The benefits of using Vue CLI 3

When Vue CLI 3 came out, Evan You, the maker of Vue, listed these objectives for it:

- Avoid toolchain fatigue for frontend development by simplifying the setup
- Follow best practices in tooling
- Make these best practices the default in Vue apps

Besides these great objectives, Vue CLI came with a number of updates, such as the following:

- Preset webpack configuration
- ES2017 and Babel 7 support out of the box
- Excellent CSS support, including **Sassy CSS (SCSS)** and **PostCSS** support
- A number of integration options (TypeScript, PWA, web components, end-to-end testing, Jest, and so on)

That is a lot of functionality. The aim of this book is to go through all of the options and get you familiar with their inner workings.

Now, to wrap this chapter up, let's see how easy it is to set up a default app using the default options.

## Default toolchain, fatigue level zero

In this section, we'll create a default boilerplate Vue app. Contrary to the previous section, in this section, we will actually build an entire app, properly. We'll use two approaches: Vue CLI 3 on the command line, and Vue CLI 3 with a GUI.

You might ask why we ran Vue CLI 3 with no configuration in the first place? The answer is, it might be useful for quick experiments, and to get started with some basic commands.

## Vue CLI 3 default app via the command line

We create Vue CLI 3 apps with the `vue create` command. Let's see what options we have available:

```
vue create -h
```

This is what will get returned:

```
Usage: create [options] <app-name>

create a new project powered by vue-cli-service

Options:
  -p, --preset <presetName> Skip prompts and use saved or remote preset
  -d, --default Skip prompts and use default preset
  -i, --inlinePreset <json> Skip prompts and use inline JSON string as preset
  -m, --packageManager <command> Use specified npm client when installing
dependencies
  -r, --registry <rul> Use specified npm registry when installing
dependencies (only for npm)
  -g, --git [message] Force git initialization with initial commit message
  -n, --no-git Skip git initialization
  -f, --force Overwrite target directory if it exists
  -c, --clone Use git clone when fetching remote preset
  -x, --proxy Use specified proxy when creating project
  -b, --bare Scaffold project without beginner instructions
  -h, --help output usage information
```

Let's begin by skipping all the prompts and using the default option:

```
vue create -d first-default-app
```

Your console will display the following output:

```
Vue CLI v3.3.0
? Creating project in C:\...
? Initializing git repository...
? Installing CLI plugins. This might take a while...
```

And indeed it does take a while. Luckily, there's a visual cue, a progress bar that lets us know how far along we are in setting up our project.

Once ready, we'll simply run the following:

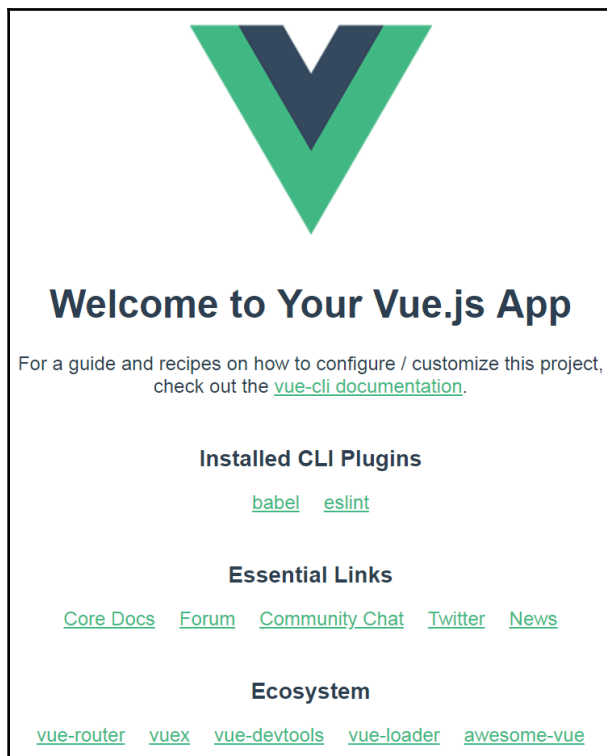
```
cd first-default-app
```

Once our console is pointing to the correct directory, we can run the app with the following:

```
npm run serve
```



And now we can view the default app in the browser:



As we can see, we have a welcome message, and then the page lists the installed CLI plugins. Obviously, the **babel** and **eslint** plugins come as defaults. Each of these links points to their respective sections inside the `vue-cli` repository on GitHub.

Next, we see some essential links and some links to get acquainted with the larger ecosystem of `Vue.js` (namely, the links to `vue-router`, `vuex`, `vue-devtools`, `vue-loader`, and `awesome-vue`).

## Vue CLI 3 default app via the UI

To get started with the Vue CLI GUI, let's first stop the server that was running in the previous section, using the `Ctrl + C` shortcut keys. The console will respond with the following message:

```
Terminate batch job (Y/N)?
```

Type `Y` (capitalization is irrelevant) and press the *Enter* key.

This will give us back the control of our current console window and allow us to type new commands.

Let's first go one level up from the current directory in our console:

```
cd ..
```

Next, let's run this command:

```
vue ui -h
```

And we'll get the following output:

```
Usage: ui [options]

start and open the vue-cli ui

Options:
  -H, --host <host> Host used for the UI server (default: localhost)
  -p, --port <port> Port used for the UI server (by default search for
available port)
  -D, --dev Run in dev mode
  --quiet Don't output starting messages
  --headless Don't open browser on start and output port
  -h, --help output usage information
```

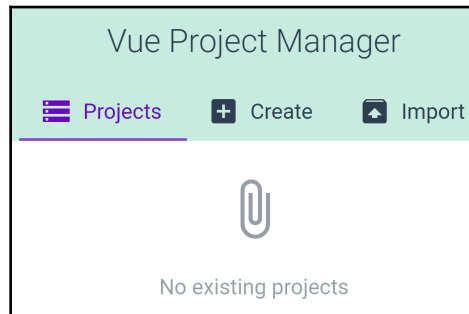
This time, we'll run the command without any flags:

```
vue ui
```

We'll be greeted with the following output in the console:

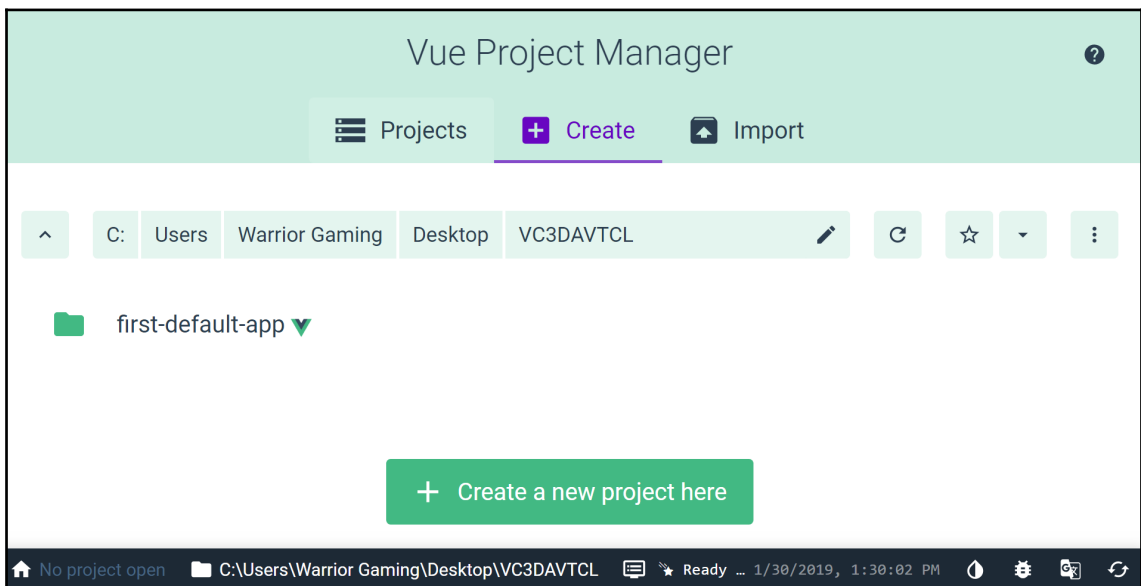
```
? Starting GUI...
? Ready on http://localhost:8000
```

This time, we can create a project, visually. Initially, we see that there are no Vue projects in the current folder:



Let's create one by clicking the **Create** tab.

A new window will open, with a big button that reads **Create a new project here**:



As we can see in the preceding screenshot, there are also a number of other buttons and options that we can use. We will examine these in the chapters that follow; at this point, we are just getting familiar with the tool we're using:

Create a new project

Details Presets Features Configuration

Project folder

Type a name

C:/Users/Warrior Gaming/Desktop/VC3DAVTCL/

Package manager

Default

Additional options

Overwrite target folder if it exists

Git repository

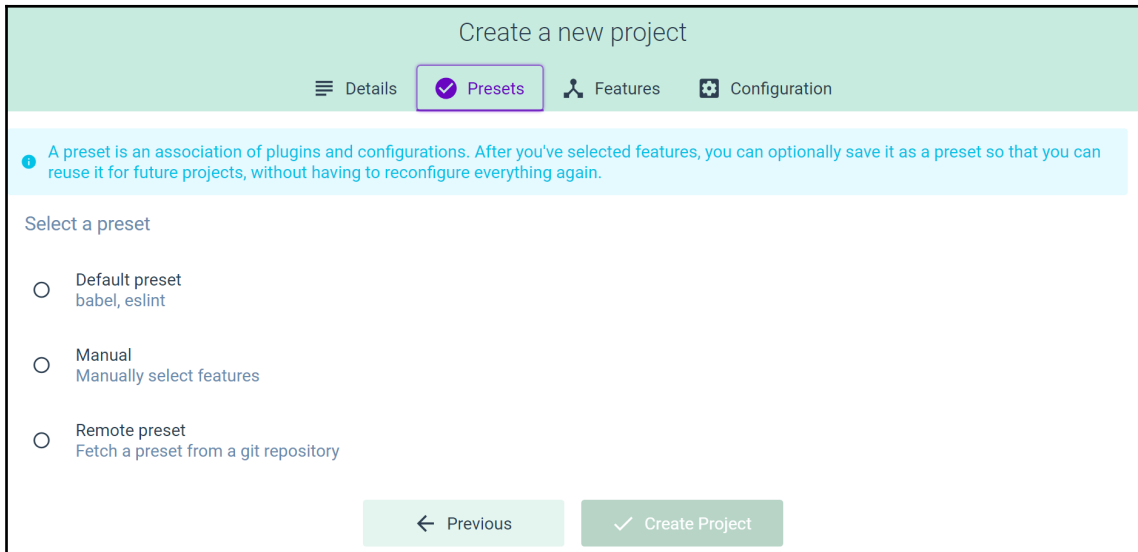
Initialize git repository (recommended)

Initial commit message (optional)

Cancel Next

As we can see from the previous screenshot, the **Next** button, at the bottom of the page, is currently disabled. To enable it, simply type the project folder name inside the top-most input. We'll call our folder `second-vue-project`. Now click **Next**.

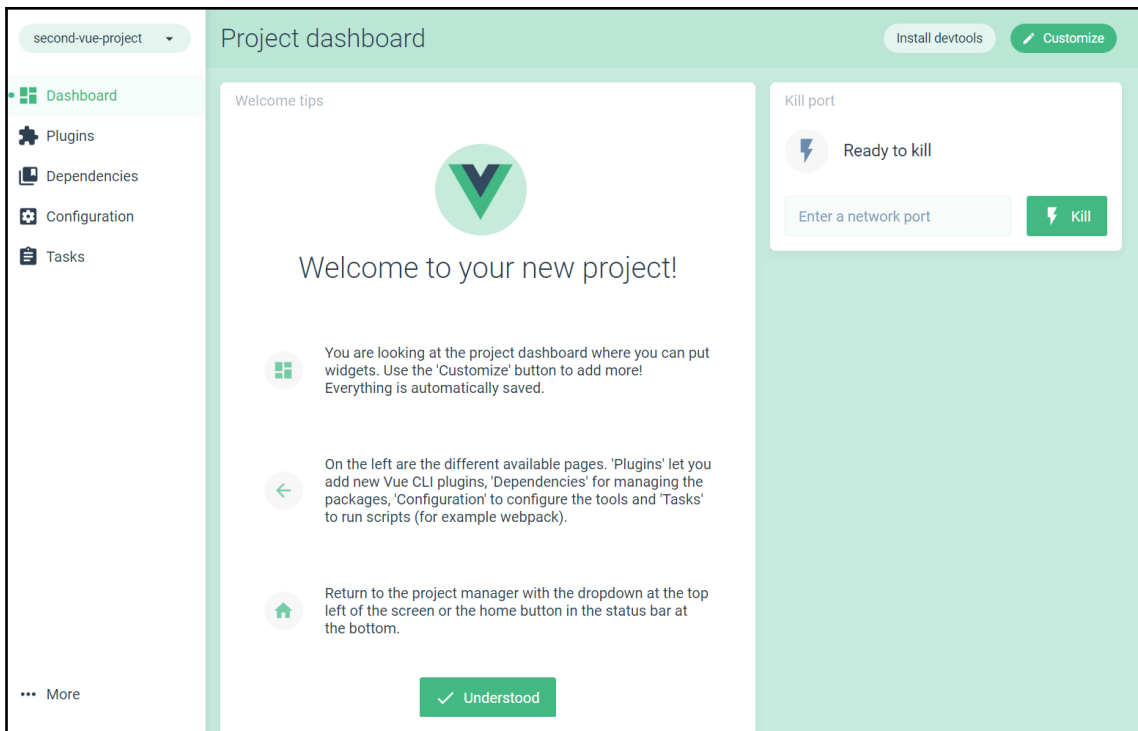
Inside the next window, you can choose a preset. Let's set it to **Default preset**:



Choosing a preset will make the **Create Project** button clickable. You'll see a loader icon in the center of the screen, with the following message:

```
Installing Vue CLI plugins. This might take a while...
```

You'll see a few other messages as the installation progresses. Finally, when done, you'll be greeted with the following window:



Our project is now ready to be worked on, which we'll do in the next chapter.

## Summary

In this chapter, we looked at how to get started with the Vue CLI 3, both by using the command line and by using the Vue CLI UI.

We've seen how to install all the prerequisites, and we saw the suggested workflow and some basic pointers to get started easily. As a result of understanding how to use Vue CLI both on the command line and in the UI, we can now initialize Vue apps without any effort. We know what options are available to us when starting a new project. However, there are many more things that we need to understand the inner workings of Vue CLI.

In the next chapter, we'll look at improving our workflow further by focusing on webpack, the module bundler that sits at the core of Vue CLI 3.

# 2 Webpack in Vue CLI 3

In the previous chapter, we saw how to start using Vue CLI via both the command line and the UI. In this chapter, we will cover webpack basics from the viewpoint of Vue CLI 3. We will begin with an overview of what webpack is. We'll look at concepts behind module bundling, tree shaking, webpack loaders and output, webpack plugins, **Hot Module Replacement (HMR)**, code coverage and code splitting, and then we'll look at how these concepts fit in with Vue CLI 3 as follows:

- The evolution of the **JavaScript (JS)** language from the script tag to module bundlers
- The script tag
- **Immediately Invoked Function Expressions (IIFEs)**, what problems they solve, and what problems they don't
- How **Node Package Manager (NPM)** helps teams share third-party libraries in their code
- The role of JS task runners and NPM scripts
- What the CommonJS specification is and how modules work in JavaScript and Node.js
- What module bundlers are and how they bridge the gap between Node.js and the browser
- What webpack, and how it works
- How to run webpack on a project
- Bundling assets with webpack using production and development modes
- Adding a Vue project via NPM and using webpack with it

Understanding exactly how webpack works is crucial to understanding the magic that Vue CLI 3 performs. If you are familiar with webpack, you still might find some sections of this chapter useful. If you feel you are a webpack pro, you can probably just skip this chapter.

Before getting into what is webpack and to properly understand the issues that webpack solves, we need to look back at a bit of history of changes that happened to the JS language in the last decade.

## The evolution of the JS language

Looking from the vantage point of webpack, here is the chronological list of approaches, techniques, best practices, and patterns that were added to the JS ecosystem, which led to the current state of things:

- The `script` tag as the answer to adding interactivity to web pages
- Immediately invoked function expressions as the answer to modularizing libraries and avoiding code collisions
- The problem with IIFEs
- Sharing third-party libraries in a team environment with NPM
- JS task runners and NPM scripts
- Modules in JS

Let's look at each one of these solutions in more detail.

### The script tag

Initially, adding JS to your web page meant that you would need to add some `script` tags directly in your HTML. For quick prototypes, this is still a valid way of doing things even to this day. Very often, third-party libraries get added via the `src` attribute inside a `script` tag (which usually gets placed right above the closing `body` tag in our HTML).

Unfortunately, you usually need more than one `script` tag inside your HTML. And regardless of whether you add your JS code directly to your page, you add it from another file in your project, or you add it from a remote location (such as from a **content delivery network (CDN)** using the `src` attribute), ultimately, all these scripts are added to the global JS scope. This means one thing, collision.

To avoid collisions, a clever approach was taken, the use of IIFEs.



## Immediately Invoked Function Expressions

What are IIFEs all about? IIFEs simply exploit the fact that in JS, *parentheses can't contain statements*. This fact alone allowed JS developers to put in anonymous functions that they could immediately invoke, without getting any errors from the parser, by simply wrapping them in parentheses.

An IIFE is essentially a quirk of the JS language, but a very useful one; with an IIFE, all the code is scoped to the function, and thus your code is safe from anything else that is outside of it. In other words, using IIFEs is a simple way to avoid collisions, that is, the accidental overwriting of variables or functions. Thus, at one point, many popular libraries started wrapping their own code into IIFEs. For example, if you open the code for the jQuery library (<https://code.jquery.com>), or for the Chart.js library (<https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.7.3/Chart.bundle.js>), or for many other popular JS libraries, you'd find that they use the IIFE pattern.

Thus, with IIFEs, we could add different scripts to our pages, without the worry that code collisions might happen.

## The problem with IIFEs

Unfortunately, simply using IIFEs does not fix all our problems. To illustrate the issue at hand, let's quote Joe Armstrong, creator of Erlang:

*"You wanted a banana, but what you got is the gorilla holding the banana, and the entire jungle."*

Bear in mind that in this quote, Mr. Armstrong was discussing a problem with object-oriented languages, but the underlying issue is applicable in JS code modularization.

Basically, our problem with IIFEs is that we can't cherry-pick specific functionality that we'd like to use from a JS library. With the IIFE pattern, we *must* use everything that is wrapped up in an IIFE, even if we are only using just a small piece of a specific library's codebase. Of course, to be honest, IIFEs are not the ones to blame for this. JS, the language, for a long time simply did not have the ability to cherry-pick any kind of code functionality, because in JS, it was simply impossible to split your code into modules.

Another major pain point of JS was the issue of reusing third-party code across teams.

## Sharing third-party libraries in a team environment with NPM

IIFEs solved the problem of code collisions, but they didn't solve the problem of code reuse. What if a developer on my team has a different, updated version of a library, with breaking changes? What if I decide to update the dependencies on my computer? How will my other team members deal with that? Besides using source version control, are there any other options for faster collaboration?

**Node Package Manager (NPM)** was the answer to these problems. Node is just a Google V8 JS engine that can run on a server. NPM simply allows a developer to install new libraries into a project, regardless of whether it is to be used on the frontend or the backend of an app. So, effectively, NPM is the JS package manager, similar to what we have in Ruby (gems (<https://rubygems.org/>)), C# (NuGet (<https://www.nuget.org/>)), or in Linux (apt-get, yum).

For example, let's say we wanted to install Vue via NPM. If we have Node installed on our machine, we'll have NPM too, since NPM comes bundled with a Node installation.

Next, we need to create a new directory. Let's change this directory's name to `vue-from-npm`, and point our command-line console to it. We can then follow it up with this command:

```
npm init -y
```

Running the preceding command will create a `package.json` file. The `-y` flag accepts all the default answers that you'd be prompted with in the console.

If we looked at the newly created `package.json` file inside our project directory, we'd see the following contents:

```
{
  "name": "vue-from-npm",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Note that the `npm init` command only adds a `package.json` file to an empty directory. That's all that it does!

Adding Vue is then as simple as running this command:

```
npm install vue --save --verbose
```

The preceding command will do a few things, namely:

- It will add the `node_modules` directory.
- It will put the entire Vue library inside the `node_modules` directory.
- It will create the `package-lock.json` file in the root of our project.
- It will update the `package.json` file in the root of our project.

The updated `package.json` file now looks like this:

```
{
  "name": "vue-from-npm",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "vue": "^2.6.7"
  }
}
```

If it is not obvious, the file has been updated with a new entry: `dependencies`. This entry lists all the dependencies included in the project. Specifically, we have added Vue (version 2.6.7 or above) to our project.

A great thing about NPM is that we can add any other library to our project, just as easily as we did with Vue. For example, to update our project with `accounting.js`, we just run this command:

```
npm install accounting-js --save --verbose
```

Once the installation is complete, let's again inspect the `node_modules` directory as follows:

```
vue-npm/node_modules/
├── accounting-js/
│   ├── dist/
│   ├── lib/
│   ├── CHANGELOG.md
│   ├── package.json
│   └── README.md
├── is-string/
├── object-assign/
└── vue/
```

Note that for the sake of brevity, we are only showing the second level of folders and files inside the `accounting-js` folder. The `is-string`, `object-assign`, and `vue` folders are shown collapsed.

What this shows us is that sometimes other NPM modules come bundled with the actual libraries that we installed. In the case of `accounting-js`, we also got the `is-string` and `object-assign` NPM modules. Let's also inspect the updated `package.json` file in the root of our directory as follows:

```
{
  "name": "vue-from-npm",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "accounting-js": "^1.1.1",
    "vue": "^2.6.7"
  }
}
```

As we can see, the root `package.json` file has been updated with the proper version of `accounting-js`. Let's locate another `package.json` file, this time inside the `node_modules/accounting-js` folder. If you opened that file, it holds a lot more information, in just a bit over 100 lines of code. This information is specific to the actual NPM module, `accounting-js`.

OK, so now our project is ready for collaboration. The way to do it? Let's see how a colleague of ours, let's call him `John`, would add the project we just created, and all the project dependencies, on his own computer.

To do this, we'll create a new folder, let's call it `johns-computer`, and let's just copy the root level `package.json` from our `vue-from-npm` folder into our `johns-computer` folder.

Next, let's simply run this command:

```
npm install --verbose
```

Running the preceding command will install all the projects and dependencies that we had in the `vue-from-npm` folder.

## JavaScript task runners and NPM scripts

Around the same time during which NPM was gaining popularity, another frontend technology was also on the rise: task runners. Task runners are simple tools; they run repetitive tasks. Sometimes, task runners are referred to as build tools because they act as intermediaries between updates that a developer makes to a codebase, and what the production ready code ends up being. This is what's known as a *build step*, a part of the software development process in which something happens to your code *after* you've written it.

For example, new features that get added to CSS3 usually start off as *vendor prefixes* (also known as *browser prefixes*). In other words, before a new CSS functionality is available in all browsers, it gets implemented in an experimental stage in individual browsers, using browser-specific prefixes as follows:

```
-ms-  
-moz-  
-o-  
-webkit-
```

In this alphabetically ordered list of browser prefixes, we can see browser prefixes for Microsoft browsers, Mozilla, old versions of Opera, and finally, all the webkit-based browsers (Chrome, Safari, newer Opera, and so on).

Keeping track of updates to browser prefixes is kind of difficult. It is probably not the best use of a developer's time to monitor for changes to CSS implementations, and then update their code accordingly. For example, at a certain point in the past, it was necessary to use the following browser prefixes on the CSS `transition` property:

```
-webkit-transition: background-color 1s;  
-moz-transition: background-color 1s;  
-o-transition: background-color 1s;  
-ms-transition: background-color 1s;
```

Obviously, today we simply use the `transition` property in our CSS declarations, without any browser prefixes, because the `transition` property is widely supported in all modern browsers.

Having to deal with the constantly shifting landscape of changes to the CSS specification and its implementations in various browsers led to a solution in the form of a task runner. Instead of having to manually add vendor prefixes to their CSS code, frontend developers could now simply add a plugin to their task runner and it would do the dirty work for them: add vendor prefixes where needed.

Of course, what we saw previously was just one example of what task runners are used for. Some other examples include: minifying CSS and JS files, transpiling from ES6 to ES5, compiling CSS from SASS, removing unused CSS, reloading the browser whenever a file is saved in your project, and so on.

Today, there are a number of different tools that help us to effectively automate some tasks in our development process. Three tools stand out: Grunt, Gulp, and NPM scripts.

While Grunt and Gulp are standalone task runners that you can install via NPM, NPM scripts are an interesting alternative for the following few reasons:

- You are using NPM anyway, so why not get more familiar with a tool you are already using?
- Using NPM scripts instead of the aforementioned task runners will streamline your development further.
- By using NPM, you avoid the complexity of having to use a task runner plugin for tasks that can be automated straight in NPM.

Up until this point, we've looked back at the history and the evolution of the JS ecosystem. We've seen how IIFEs were used to take care of accidental scope leaks. We've also seen how NPM takes care of code sharing. We've further looked at how some repetitive tasks can be automated with the help of task runners, and how NPM can be used to remove an unnecessary layer of abstraction by keeping our tasks inside NPM scripts.

However, we still haven't looked at ways to solve the problem of code modularization in JS. So, let's look at that next.

## Modules in JavaScript

In any programming language, a module is a self-contained piece of functionality. You can think of them as different episodes of a TV show. They can be looked at independently. They can stand on their own, although they are a part of a greater whole.

And just like an episode in a TV show has a season and a number, so that we know *where it fits* in the bigger plot, a module also holds the information that tells us what other modules it relies on (the *module dependencies*), and what functionality it adds to the entire app; this is what's known as a *module's interface*, the API that's exposed to other modules.

We've seen how in the beginning, JS did not have modules at all. This changed with the introduction of Node.js. Node.js is actually an implementation of CommonJS, a project that was started in 2009 by Mozilla's Kevin Dangoor.

CommonJS project's purpose was to define a standard library that would provide JS APIs to be used outside of the browser. This included a module specification, which resulted in developers being able to use code like this in Node.js:

```
var bootstrap = require('bootstrap');
```

## Working with modules in Node.js

Let's require and use some modules in Node.js:

1. To begin, we'll create a new directory. Let's call it `module-practice`. Let's point our Git Bash to this folder.
2. Once inside it, let's create two new files. Let's call these files `main.js` and `whatever.js` as follows:

```
touch main.js whatever.js
```

3. Next, let's open this entire folder in VS Code as follows:

```
code .
```

4. Now, let's add some code to `whatever.js` as follows:

```
console.log('whatever');
```

This is as simple as a piece of code in JS can get.

5. Let's now see how can we make it available in our `main.js` file. We'll just need to `require` `whatever.js` as follows:

```
let whatever = require('./whatever');
```

6. Now that it is required, we can use it, so let's update `main.js` to this:

```
let whatever = require('./whatever');  
  
whatever.returnWhatever();
```

7. Let's now run this code with the following:

```
node main.js
```

What will happen now is, we will see the word `whatever` printed inside Git Bash.

Let's take our experiment a step further. Here's our updated `whatever.js`:

```
module.exports = {  
  returnWhatever: function() {  
    console.log('whatever');  
  }  
}
```

Consequently, we need to update `main.js` too as follows:

```
whatever.returnWhatever();
```

As we have already seen, the `require` keyword imports a module's code and makes it available in another file; in our example, the `main.js` file.



The `exports` keyword lets us make the code available to other files, with one caveat. It also allows us to choose what parts of our module we want to make available to other files. As we see, the `module.exports` is an object. The contents of this object is what will be returned when our `whatever` module is required by `main.js`. This allows us to expose only certain sections of our code, and enables the setup of a module's interface. In other words, `module.exports` is what enables us to keep parts of our code private. Consider this update to `whatever.js`:

```
module.exports = {
  returnWhatever: function() {
    returnSomething();
  }
}

let returnSomething = () => {
  console.log('whatever');
}
```

We don't need to make any changes to `main.js`. If we ran it from Git Bash, we'd still get the word `whatever` output to console. But we have made parts of `whatever.js` not directly accessible.

As a side note, notice that in the preceding code, the function syntax of ES3 and ES5 are used alongside one another. The section of code that defines the `returnSomething` function is using newer syntax, which lets us write function definitions without having to use the `function` keyword.

## Module bundlers, a way to use modules in the browser

Unfortunately, you can't just use the `require` keyword, as we just saw, straight in the browser. The `require` keyword is not a part of JS's browser API. It is important to note here that Node.js has the ability to read from and to the filesystem of your computer. Thus, if you installed any NPM package in a project, using Node.js, you would be able to require such a module as explained previously.

However, JS in the browser does not have access to the file system of your OS, and thus this leaves us with a conundrum: how do we use the JS module syntax inside the browser?

The answer: we have a tool for that, and it's called a **module bundler**.

Today, in 2019, there are a number of different module bundlers available, such as webpack (<http://webpack.github.io/>), FuseBox (<https://fuse-box.org/>), Parcel (<https://parceljs.org/>), rollup.js (<https://rollupjs.org/guide/en>), or Browserify (<http://browserify.org/>).

What is a module bundler? Here is a quote from the Browserify home page, which puts it succinctly:

*"Browserify lets you require ('modules') in the browser by bundling up all of your dependencies."*

Besides bundling all the dependencies that get required through modules in your project, module bundlers also solve issues such as cyclical dependencies; that is, they use algorithms that resolve the order in which all the project's dependencies should be bundled in your project.

We have almost finished our overview of the JS ecosystem. Next, we'll look at one particular flavor of a module bundler, and that is webpack.

Once we know just what exactly webpack is and how it works under the hood, we'll be able to fully comprehend where it fits within Vue CLI.

## What is webpack?

Webpack is a module bundler for the web. Some people also refer to it as an asset compiler for web applications.

According to webpack's GitHub page:

*"It packs many modules into a few bundled assets and so on. Modules can be CommonJS, AMD, ES6 modules, CSS, images, JSON, CoffeeScript, LESS, and so on, and your custom stuff."*

Earlier in this chapter, in the section titled *Working with modules in Node.js*, we barely scratched the surface of how modules get exported and required in Node apps. What we did not mention is that there are all kinds of different module syntaxes that we can use. As already mentioned, Node.js works with the CommonJS module syntax. Besides CommonJS, there is also **Asynchronous Module Definition (AMD)**. Alongside AMD, you can use ESM modules. With ESM modules, the syntax is a bit different from what we saw earlier.

Let's rewrite the `whatever` module using ESM syntax by taking the following steps, and use it in `main.js`:

1. To keep things easy, let's also create a new folder as follows:

```
mkdir es6-module-practice;
```

2. Let's point our Git Bash by using the `cd` command (the *change directory* command) as follows:

```
cd es6-module-practice
```

3. Let's add our two files as follows:

```
touch whatever2.mjs main2.mjs
```

4. Now, let's open our folder with VS Code as follows:

```
code .
```

5. Next, let's add the code for `main2.mjs` as follows:

```
import returnWhatever from './whatever2';  
  
returnWhatever();
```

6. Finally, let's follow it up with code for `whatever2.mjs` as follows:

```
let returnWhatever = () => {  
  returnSomething();  
}  
  
let returnSomething = () => {  
  console.log('whatever');  
}  
  
export default returnWhatever;
```

7. As we can see, we needed to save the files as ESM modules, using the `mjs` file extension. Node.js supports ESM modules experimentally, so you need to run the following command in Git Bash:

```
node --experimental-modules main2.mjs
```

8. Upon running the preceding command, this is the output you'll see in the console:

```
(node:12528) ExperimentalWarning: The ESM module loader is experimental.
whatever
```

As we can see, besides receiving the expected output in the console, we also got the `ExperimentalWarning` message. Hopefully, this demonstration of two different kinds of module syntaxes helps us understand what webpack will do for us. Among other things, it will *level the playing field*, so that we can use various standard and non-standard ways to work with modules in our projects.

Basically, what webpack does is, it takes our modules with dependencies (including our project's assets such as `.png`, `.jpeg`, and `.scss` files), and it outputs static assets (`.js`, `.css`, and `.image` files).

## How webpack works

We've seen how to use the CommonJS and ESM module syntax. Again, CommonJS is the syntax used for Node.js modules. This means that all the dependencies in a Node.js module are described using the `require` command. Contrary to this, a webpack module's dependency can be described in a wide variety of syntaxes. For example, if your module's dependency was an SCSS partial, you'd use an `@import` statement. If you were importing an AMD module's dependency, you'd be using its own `require` and `define` syntax.

What this means is, that basically, *webpack modules accept all the different syntaxes for importing various dependencies* of a module. Even the `src` attribute (for the `img` HTML element) is considered a webpack module's dependency.

## Building a new project and running webpack on it

Let's now build a project and integrate webpack bundling into our workflow by taking the following steps:

1. Let's add a new directory. Let's run a command that doesn't exist as follows:

```
new-project-with-webpack
```

The console will return the following:

```
bash: new-project-with-webpack: command not found
```

2. Great! Now, let's use the *repeat last command* shortcut, the double exclamation mark, as follows:

```
mkdir !! && cd !!
```

Running the preceding command will create our `new-project-with-webpack` folder, and will `cd` into this new directory. The double ampersand command (`&&`) is just a way to run multiple commands one after another, rather than having to type them out one by one. The double exclamation mark command (`!!`) means *repeat the previous line*, so the preceding command actually means the following:

```
mkdir new-project-with-webpack && cd new-project-with-webpack
```

3. Next, let's add our `package.json`, and accept all the defaults (using the `-y` flag) as follows:

```
npm init -y
```

4. Let's inspect the contents of our folder in VS Code as follows:

```
code .
```

5. Once VS Code is running on our screen, we can double-click the `package.json` file and verify its contents as follows:

```
{
  "name": "new-project-with-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

6. Now, let's add webpack to our project as follows:

```
npm install --save-dev webpack webpack-cli --verbose
```

7. Once done, let's go back to VS Code and review our `package.json` again as follows:

```
{
  "name": "new-project-with-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "webpack": "^4.29.5",
    "webpack-cli": "^3.2.3"
  }
}
```

As we can see, a new key has been added: `devDependencies`. Inside it, we have the `webpack` and the `webpack-cli` development dependencies. These `devDependencies` are the dependencies that you will be using only while building your project, and `webpack` is a perfect example of such a dependency: you will not need `webpack` in production. That's why we've used the `--save-dev` flag when we installed `webpack` via NPM.

Looking at the file structure of our project, we can now see the following:

```
node_modules/
package.json
package-lock.json
```

If you open the `node_modules` folder, you see over 300 folders inside. This huge list of dependencies begins with a `.bin` folder. Contrast this with one of our previous examples, `vue-from-npm`, where we had only four subfolders inside the `node_modules` folder, even though we installed both the `vue` and `accounting-js` NPM packages. Note also that inside the `vue-from-npm` folder, there is no `.bin` folder. This is the case regardless of whether you install it with the `--save` or the `--save-dev` flag when running `npm install`. While this might be obvious to more experienced developers, it might be important for better understanding to those developers who do not have enough experience with Node.js and NPM ecosystem.

So, what is this `.bin` folder? It just stores the compiled native binaries (that is, executable files) of a Node module that you installed using `npm install`. Not all NPM modules have these compiled native binaries and this is why you won't always see the `.bin` folder inside the `node_modules` folder. Inside this `.bin` folder, there are many different Node modules. These are all necessary for webpack to work properly.

Back to our project, let's now add two files to it: `index.js`, and `whatever.js` as follows:

```
touch index.js whatever.js
```

Currently, we won't be adding any code to either of those files. Right now, we'll focus on running webpack in our project.

## Running webpack on a project

Go back to our `new-project-with-webpack` folder, and examine the contents of `package.json` again, with the focus on the `scripts` key as follows:

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

Use the following command in Git Bash to run `test` script:

```
npm run test
```

This will throw an error, with `exit code 1`.

Let's make some changes to it as follows:

```
"scripts": {
  "test": "echo \"You haven't specified any tests\""
},
```

Let's run the test again with `npm run test`. This time the output in the console will not be as ghastly, because we removed the `exit 1` command, and we altered the wording of what will be echoed when running the `test` command.

Let's try something else entirely as follows:

```
"scripts": {
  "test": "node index.js"
},
```

Now, we get no errors, because our `index.js` is empty. Let's add something to it as follows:

```
// add up 2 numbers
console.log(2+2)
```

Save changes to `index.js`, run `npm run test` again, and this time the output in Git Bash will be number 4 printed to the screen.

What does this tell us? It tells us that we are in complete control of what our scripts are going to do! So, initially we had a script that was named `test`. This script would echo out a message and throw an error with `exit code 1`.

And just as we can give our scripts any arbitrary key name, such as `test`, we can also give them any arbitrary value. Of course, `console.log(2+2)` is a silly value to give to a script key. We can give our scripts keys much better values, for example:

```
"scripts": {
  "webpack": "webpack"
},
```

Now, when we run an NPM script with the value of `webpack`, this script will then run the `webpack` executable. Let's try it out as follows:

```
npm run webpack
```

This returns an error, but out of all the information that gets logged out, the following two lines are the most important:

```
Insufficient number of arguments or no entry found.
Alternatively, run 'webpack(-cli) --help' for usage info.
```

The reason we got this error is because `webpack` looks for an entry point to get started on. By default, this entry point is set to `./src/index.js`. So, let's add this `src` folder, and move our `index.js` into it as follows:

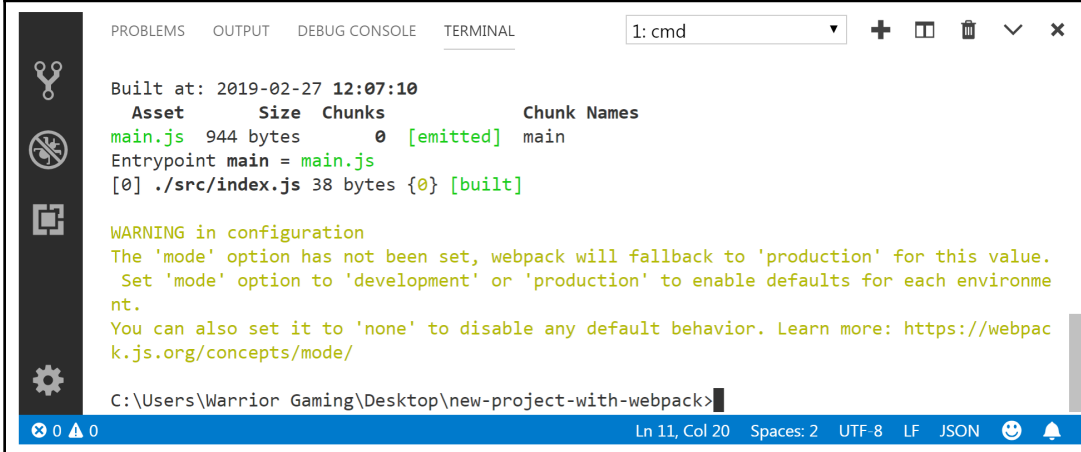
```
mkdir src && mv index.js $_
```

Now, let's run `webpack` from the command line again as follows:

```
npm run webpack
```



This time we'll get a much better output. However, there is no syntax highlighting in Git Bash by default. This can be fixed really fast. Since we are already using VS Code, simply type the keyboard shortcut of `Ctrl + ~`. If you're not familiar with this symbol, it's called a *tilde*, and it's located right under the *Esc* key, and above the *Tab* key. Pressing this keyboard shortcut will open up a terminal window inside VS Code, and if you execute the `npm run webpack` command again, you'd get nicely formatted and color highlighted output, like this:

A screenshot of the VS Code terminal window. The terminal title is "1: cmd". The output shows the following: "Built at: 2019-02-27 12:07:10", a table with columns "Asset", "Size", "Chunks", and "Chunk Names" containing "main.js 944 bytes 0 [emitted] main", "Entrypoint main = main.js", and "[0] ./src/index.js 38 bytes {0} [built]". Below this is a "WARNING in configuration" section with text explaining that the 'mode' option is not set and will default to 'production'. The terminal prompt is "C:\Users\Warrior Gaming\Desktop\new-project-with-webpack>".

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: cmd
Built at: 2019-02-27 12:07:10
  Asset      Size  Chunks  Chunk Names
main.js    944 bytes    0 [emitted]  main
Entrypoint main = main.js
[0] ./src/index.js 38 bytes {0} [built]

WARNING in configuration
The 'mode' option has not been set, webpack will fallback to 'production' for this value.
Set 'mode' option to 'development' or 'production' to enable defaults for each environment.
You can also set it to 'none' to disable any default behavior. Learn more: https://webpack.js.org/concepts/mode/

C:\Users\Warrior Gaming\Desktop\new-project-with-webpack>
```

Figure 2.1: Webpack logging out information to console in VS Code



As a side note, your screen's colors might look different, depending on the color scheme you are using in your VS Code. To access the color themes, use the following keyboard shortcut: `Ctrl + K Ctrl + T`.

Looking at the message that was output to the console, we can see that it can be grouped in two sections: the one with the actual information (hash, version, time, built at, entrypoint, and so on) and the warning. The warning reads that we haven't set the `mode` option.

If unset, the `mode` option defaults to `production`. However, we can also set it to `development`, which is optimized for faster builds. This means that we can add another script in our `scripts` section of `package.json`, and this script can then be used for the development build of our project. This is the updated `scripts` section:

```
"scripts": {
  "webpack": "webpack",
  "dev": "webpack --mode=development"
},
```

Now, we can run the development mode in webpack, with the following command:

```
npm run dev
```

Here is the complete output in the console:

```
Hash: 86c0da41f48381d9bd70
Version: webpack 4.29.5
Time: 108ms
Built at: 2019-02-27 12:23:30
    Asset Size Chunks Chunk Names
main.js 3.81 KiB main [emitted] main
Entrypoint main = main.js
[./src/index.js] 38 bytes {main} [built]
```

As we can see, it took webpack 108ms to bundle my project in the development mode. When I ran it in production mode (the default `npm run webpack` command in my setup), it took 447ms.

What does actually happen when we run this command? What does webpack do in the background? It builds a dependency graph of all the module dependencies. To go back to our analogy from earlier in the chapter, it's as if we gave it a bunch of episodes of a TV show, recorded on a bunch of Blu-ray discs, and it takes all of them and lines them up properly. Webpack figures out the correct place for each of these modules, and then bundles them up and serves them to the `dist` folder. If you look at your project's file structure again, you'll see that now there's a new addition: the `dist` folder. If we inspect the contents of the `dist` folder, this is what we'll see:

```
./dist/
|- main.js
```

If we inspect the `main.js` file, we'll see that there is a lot of stuff that webpack adds. Even on a tiny project like ours, the output gets to be about 100 lines long.

The first few lines of our `main.js` file look like this:

```
/***/ (function(modules) { // webpackBootstrap
/***/ // The module cache
/***/ var installedModules = {};
...

```

Let's run our `npm run webpack` command again to see how it will affect the output in `main.js`.

If we inspect `main.js`, we will see that now we have only a single line of code, beginning with the following:

```
!function(e){var t={};function n(r){if(t[r])return t[r].exports;var ...
```

What this means is that webpack uglifies and minifies our code when run in production mode.

Obviously, this affects the file size too. While the bundled `main.js` file has the file size of 3.81 KB in development mode, when bundled in production mode, it has just 944 bytes.

Finally, to avoid seeing the warning message, we can update our scripts entry in `package.json` to this:

```
"scripts": {
  "webpack": "webpack --mode=production",
  "dev": "webpack --mode=development"
},
```

At this point, we can start using webpack with Vue. However, we won't be using the Vue CLI. Instead, we'll see how to set everything up manually. This is not the best way to do things, but it will help us better understand why things are done the way they are in the Vue ecosystem.

## Adding a Vue project via NPM and using webpack with it

In this section, we'll build a new project with NPM, then add webpack to it, and finally add a Vue single file component to it.

First, let's make a new directory by taking the following steps. We'll call our project `npm-vue-webpack`:

1. Open Git Bash and add a new folder as follows:

```
mkdir npm-vue-webpack && cd $_
```

2. Initialize `npm` as follows:

```
npm init -y
```

3. Next, install `Vue` and `webpack` into our new project as follows:

```
npm install vue webpack webpack-cli --save-dev --verbose
```

Once the NPM installation is complete, we can verify the folders and the contents of `package.json` as we did earlier in the chapter.

4. Next, add the source and output folders our project will use as follows:

```
mkdir dist src
```

5. Open our new project in VS Code as follows:

```
code .
```

Now, we can add two new files right from the VS Code editor. We'll call the first file `source.js`, and we'll call the second file `output.js`. Make sure you have added and saved these two empty files in your project at this stage: `source.js` in the `src` folder, and `output.js` in the `dist` folder.

## Adding our Vue component as a JavaScript module

Now let's add our `Vue` component:

1. Next, let's add this code to `source.js` as follows:

```
import CustomArticle from './CustomArticle.js';

new Vue({
  el: '#app',
  render: h => h(CustomArticle),
})
```

2. On the first line, we're importing a file called `CustomArticle.js`.

3. Let's make a new file inside the `src` folder. We'll call this file `CustomArticle.js`.
4. And add the following code to it:

```
let CustomArticle = Vue.component('custom-article', {
  template: `
    <article>
      Our own custom article component!
    </article>`
})

export default CustomArticle;
```

As we can see, we're using the ESM syntax to export and import a JS module.

## Compiling JavaScript modules with webpack

We are almost ready now to compile our `source.js` to `output.js` with the help of webpack. However, before we do it, we still need to update the `scripts` section of our `package.json` as follows:

```
"scripts": {
  "webpack": "webpack"
},
```

Now, we can run the following command inside Git Bash:

```
npm run webpack ./src/source.js ./dist/output.js
```

As expected, we see the output in the console, together with the warning about setting the mode option. We now know what this means, so it's not important to deal with it at this time.

If we inspect the contents of `output.js`, we'll see that it's empty, and that, by default, webpack minified and obfuscated our output code to the default `main.js` file as follows:

```
!function(e){var t={};function n(r){if(t[r])return t[r].exports;var o=t[r]={i ...
```

So, how do we make webpack output to a different file, other than the default `main.js`? We use the webpack configuration file!

## Adding options with the help of the webpack configuration file

Using the webpack configuration file, we can add various options to the way our apps get bundled with webpack. Here is how:

1. Add a new file to the root of our project. We'll call this `webpack.config.js` file. Here is the code:

```
module.exports = {
  output: {
    filename: 'output.js',
  }
};
```

2. Now, run our command again as follows:

```
npm run webpack ./src/source.js ./dist/output.js
```

This time it outputs to the correct file.

3. Just as we can specify the output file, we can also specify the input file as follows:

```
module.exports = {
  entry: './src/source.js',
  output: {
    filename: 'output.js',
  }
};
```

We still need to render our Vue component somewhere on the screen. We need an HTML file for that.

## Adding an HTML file so that we can render our Vue component

Let's add a new HTML file, we'll call it `index.html`, to our `dist` folder as follows:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Compiled HTML file</title>
</head>
<body>
```

```
<div id="entryPoint"></div>

<script
src="https://cdnjs.cloudflare.com/ajax/libs/vue/2.5.13/vue.min.js"></script
>
  <script src="output.js"></script>
</body>
</html>
```

Just as we discussed at the very beginning of this chapter, we are adding scripts directly to our HTML in the *old-school* way, by just stacking `script` tags at the bottom of our HTML file. The first `script` tag we're using is fetching Vue from a CDN, and the second `script` tag fetches our `output.js` file from the `dist` folder.

If you're using VS Code, you can now right-click on your new `dist/index.html` file, and click the **Open in default browser** command.

You will see the following sentence on the web page that opens:

**Our own custom article component!**

Now, we need to give webpack the ability to output an HTML file. For that, we'll need to use the `html-webpack-plugin`.

## Giving webpack the ability to output an HTML file

In this section, we'll see how to output an HTML file using a webpack plugin by taking the following steps:

1. Install `html-webpack-plugin` via NPM as follows:

```
npm install html-webpack-plugin --save-dev --verbose
```

2. Our `devDependencies` of `package.json` have updated accordingly as follows:

```
"devDependencies": {
  "html-webpack-plugin": "^3.2.0",
  "vue": "^2.6.7",
  "webpack": "^4.29.5",
  "webpack-cli": "^3.2.3"
}
```

3. Now, update our `webpack.config.js`, as follows:

```
let HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: './src/source.js',
  output: {
    filename: 'output.js',
  },
  plugins: [new HtmlWebpackPlugin()]
};
```

4. Before we continue, delete the `index.html` file that we had in our `dist` folder. Don't worry about deleting it, though, since webpack will soon recreate it.
5. Next, let's run the webpack script again as follows:

```
npm run webpack
```

Webpack just created a new `index.html` file for us! Here's the contents of the file:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Webpack App</title>
  </head>
  <body>
    <script type="text/javascript" src="output.js"></script></body>
</html>
```

This is all great, but obviously, our file is not the same. We have lost our entry point for the Vue component we had. Also, we need to update our Vue code so that it works as a single file component.

## Adding a `.vue` file as a JavaScript module

First, let's update the `source.js` file, to the following:

```
import Vue from 'vue';
import CustomArticle from './CustomArticle.vue';

new Vue({
  el: '#entryPoint',
  render: h => h(CustomArticle),
})
```



We can now also rename our `CustomArticle.js` to `CustomArticle.vue`, and add the following code to it:

```
<template>
  <div id="entryPoint">
    <article>
      Our own custom article component!
    </article>
  </div>
</template>
```

Unfortunately, webpack doesn't work with `.vue` files out of the box. To fix the issue at hand, we need to use a **webpack loader**. A webpack loader helps webpack understand the files it's working with. There are many loaders out there, but for now, we need to work with Vue.

## Adding a webpack loader to work with `.vue` files

To work with `.vue` files, take the following steps:

1. Install a webpack loader called `vue-loader` via NPM as follows:

```
npm install vue-loader --save-dev --verbose
```

2. Now that we've saved it, we need to use it, and we'll do that by updating the webpack configuration as follows:

```
let HtmlWebpackPlugin = require('html-webpack-plugin');
let VueLoaderPlugin = require('vue-loader/lib/plugin');

module.exports = {
  entry: './src/source.js',
  output: {
    filename: 'output.js',
  },
  plugins: [
    new HtmlWebpackPlugin(),
    new VueLoaderPlugin(),
  ]
};
```

3. Try running webpack now as follows. Spoiler alert: it will fail:

```
npm run webpack
```

The error message we get reads as follows:

```
Error: [VueLoaderPlugin Error] No matching rule for .vue files found.
```

4. To fix this error, we need to add a rule for our Vue loader, by updating our `webpack.config.js` file to the following:

```
let HtmlWebpackPlugin = require('html-webpack-plugin');
let VueLoaderPlugin = require('vue-loader/lib/plugin');

module.exports = {
  entry: './src/source.js',
  output: {
    filename: 'output.js',
  },
  module: {
    rules: [
      { test: /\.vue$/, use: 'vue-loader' }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin(),
    new VueLoaderPlugin(),
  ]
};
```

The `test` key inside the array in the `rules` option receives a regex as a value. This regex is checking for the presence of a file with a `vue` file extension. If it matches, that is, if it finds a `vue` file, it will use the `vue-loader` module on it.

5. Let's run our webpack script again as follows:

```
npm run webpack
```

This will throw another error as follows:

```
ERROR in ./src/CustomArticle.vue  
Module Error (from ./node_modules/vue-loader/lib/index.js):  
[vue-loader] vue-template-compiler must be installed as a peer  
dependency, or a compatible compiler implementation must be passed  
via options
```

6. There are more errors logged out to the console, but this is the one that we need to fix by adding another NPM package, as follows:

```
npm install vue-template-compiler --save-dev --verbose
```

The `devDependencies` entry inside `package.json` just received another update, as follows:

```
"devDependencies": {
  "html-webpack-plugin": "^3.2.0",
  "vue": "^2.6.7",
  "vue-loader": "^15.6.4",
  "vue-template-compiler": "^2.6.7",
  "webpack": "^4.29.5",
  "webpack-cli": "^3.2.3"
}
```

So, now we can run `webpack` again as follows:

```
npm run webpack
```

After `webpack` ran, if you opened `output.js` at this time, you'll see that it has the full Vue library inside of it, along with our `CustomArticle` at the very end. This all compiled without any errors.

## Fixing the issue with our `index.html` file

We still have the issue of the `index.html` file in the `dist` folder. It's an easy fix! We'll just add our own `index.html` file to the `src` folder, with the following contents:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Compiled HTML file</title>
</head>
<body>
  <div id="entryPoint"></div>
</body>
</html>
```

Note that we have now removed our own `script` tags, since webpack will be adding them. Also, make sure to delete the `index.html` file inside the `dist` folder. Now, run the `npm run webpack` command again, and you'll get this output in `dist/index.html` as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Webpack App</title>
  </head>
  <body>
    <script type="text/javascript" src="output.js"></script></body>
</html>
```

Why is this not working?

It is not working because we need to update both the JS files to be output, as well as the HTML files. Currently, we are only updating the JS files, but we still need to do it for our `index.html` file. Luckily, we already have `html-webpack-plugin` to help out.

## Delivering HTML files via webpack with the help of `html-webpack-plugin`

We'll begin by updating the `html-webpack-plugin` in the `webpack.config.js` file as follows:

```
plugins: [
  new HtmlWebpackPlugin({
    template: './src/index.html',
  }),
  new VueLoaderPlugin(),
]
```

What we did in the `plugins` section was, we passed an `options` object to our `HtmlWebpackPlugin()` call. Inside this `options` object, we specified our `template`: `./src/index.html`.

Before we run the webpack script again, we need to make sure to add the `meta` tag with `charset` set to `utf-8`. Otherwise, we'll get an error in our console when we open `dist/index.html` in the browser.

Let's now run `npm run webpack` again. This time, everything works! We get our humble looking sentence on the screen:

### Our own custom article component!

Congratulations! Although it is humble looking, you have successfully added a Vue app to an NPM-powered project, running on webpack.

Next, we'll learn about HMR, and how it helps us in our Vue development.

## Understanding Hot Module Replacement in Vue

HMR has become a sort of a buzzword in the past couple of years. What's the big deal? In this section, we'll discuss how HMR works.

To do that, we'll build another default simple app, just like we did in [Chapter 1, Introducing Vue CLI 3](#), as follows:

```
vue create -d second-default-app
```

After a while, once it's done, we'll go into our app's directory as follows:

```
cd second-default-app
```

Let's open the project's folder in VS Code as follows:

```
code .
```

Now, we can see the entire contents of our `second-default-project`.

Now, we can serve the app as follows:

```
npm run serve
```

Of course, our app is now serving in our browser.



To view your app, visit `localhost:8080` in your browser.

Let's look at HMR updates happening in real time.

## Observing HMR updates

While the browser window is active, let's press the *F12* key to turn on the developer tools. Let's also make sure that our the **Elements** panel is the active tab inside the dev tools, so that we can see the **Document Object Model (DOM)** structure, like in the following zoomed-in screenshot from the browser:

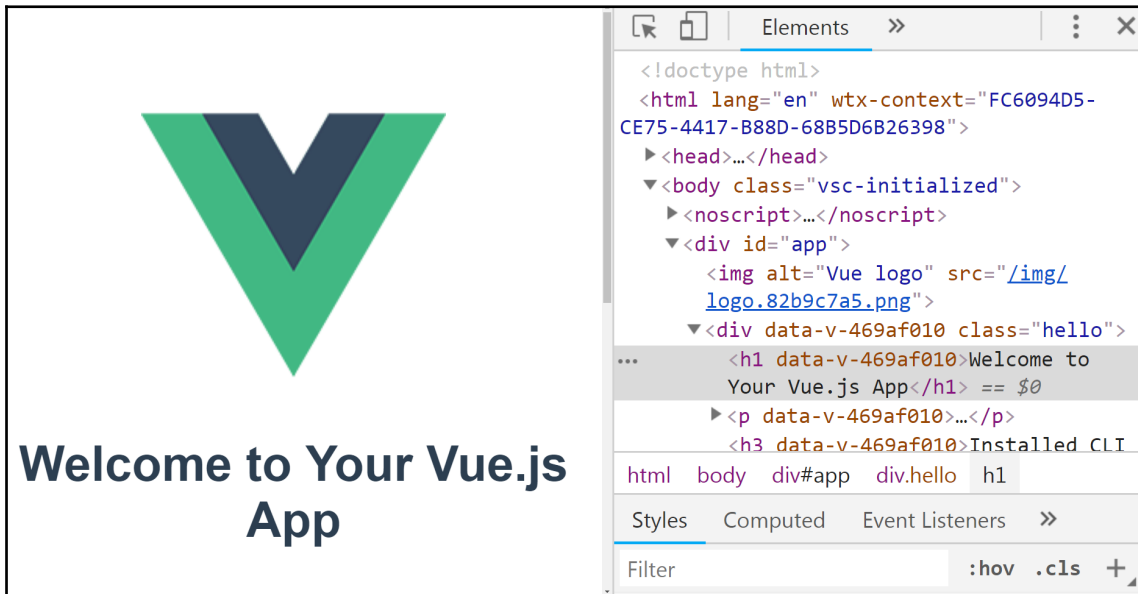


Figure 2.2: The welcome screen of second-default-app with the **Elements** panel open in dev tools

Now, let's see HMR in action. Ideally, to see this in action, you'd need to use two monitors. Thus, you could, for example, move your VS Code window to the left monitor, and your browser with the app served to the right monitor. Alternatively, you can use a single monitor and view both apps side by side (with each one taking up half the screen width). The point of this exercise is to be able to see at the same time your VS Code window, your Vue app browser window, and your **Elements** panel inside your browser's dev tools.

Next, inside VS Code, open the `App.vue` file (inside the `src` folder of your project). Look at line 4, which currently reads as follows:

```
<HelloWorld msg="Welcome to Your Vue.js App"/>
```

We'll soon change that line to something else. Before you change the line, pay attention to how these changes reflect in the browser serving your app. Will the browser refresh?

Now that you're focused on tracking changes in browser, let's update line 4 in `App.vue` as follows:

```
<HelloWorld msg="HMR is cool"/>
```

Pay attention to the browser while you're saving changes to `App.vue` in VS Code. The best way to do this is to have VS Code in focus, but watch the browser window, specifically the **Elements** panel. With VS Code in focus, you can save changes with the shortcut key `Ctrl + S`.

If you pay close attention, and if you're using Chrome, you'll notice a flash of purple color inside the **Elements** panel. That's the Chrome browser notifying us of the changes that happened to the DOM of our Vue app. If you pay close attention, you'll notice that there is a flash on the `head` element, as well as the `h1` element and its child text node, `HMR is cool`.

Also, you might notice that there was no refresh on the browser. Neither webpack, our app's code, or we ourselves forced the browser to refresh. What's the conclusion here? Webpack is actually not forcing a page refresh with HMR! Instead, it just injects an HMR.

While the change on the `h1` element is obvious (as it is the immediately visible consequence of us changing the text inside the `App.vue` file), the update that happens in the `head` element is both more obscure and more helpful. To see what is happening, let's twirl open the `head` tag by clicking on the little black triangle to the left of the `head` tag inside the **Elements** panel, as shown in the following screenshot:

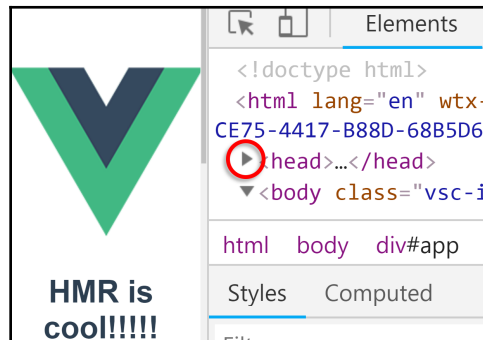


Figure 2.3: Twirl-open the head tag inside **Elements** panel in dev tools

Next, we'll need to scroll all the way down to the closing `</head>` tag. Right above it, there will be a `script` tag that reads something similar to this:

```
<script charset="utf-8" src="/app.c7e7b2f6599f49948328.hot-  
update.js"></script>
```

Let's keep close attention on this section of the DOM tree in the **Elements** panel, while we make another change to `App.vue`. Let's update our `msg` attribute on line 4 to this:

```
<HelloWorld msg="HMR is cool indeed"/>
```

If you observed the `script` tag, you'll notice it change as follows:

```
<script charset="utf-8" src="/app.417e697f270d544a21b3.hot-  
update.js"></script>
```

Did you notice that Vue injected another script above the closing `</head>` tag? That injected script is HMR in action, right there.

Let's inspect the very first line of the appended script file as follows:

```
webpackHotUpdate("app", {
```

How does this whole process work? Webpack simply runs our changes, bundles an update to our app, and, since it is already running, injects the new code using Vue loader.

So, as we've just seen, HMR is just a feature of webpack that helps us do things more smoothly, without the need to worry about refreshing our apps.

## Summary

In this chapter we discussed the evolution of the JS language and its ecosystem, and how this evolution lead to the appearance of module bundlers. We also looked at webpack, the module bundler of choice of Vue CLI 3.

We looked at how to plug in a very basic Vue-powered app, running on single file Vue templates. Alongside this mini-project, we also looked at a number of important webpack concepts. We finished the chapter by observing HMR on a Vue project.

Now that we know the very basics of how webpack works, in the chapters that follow, we'll discuss some other related technologies and we'll build on our knowledge of webpack and Vue CLI 3. The next topic we'll be closely examining is Babel.



# 3

## Babel in Vue CLI 3

In this chapter, we will work with Babel to bring the new features of **JavaScript (JS)** to browsers before they can understand them by transpiling it to older versions of JS. We'll discuss the following:

- Understanding Babel
- Running webpack with ES5 and ES6
- Updating our webpack configuration to work with Babel
- Vue, Babel, and JSX
- Adding Babel plugins manually

Let's begin by looking at what problems Babel solves.

### Understanding Babel

As we've seen already in the previous chapters, once you build a default Vue application using the Vue CLI, you can then serve it with `npm run serve`.

Your app will usually be served at `localhost:8080`. Looking at the served page with its default contents, you'll notice that there are two plugins listed under the **Installed CLI Plugins** heading: `babel` and `eslint`.

Why would these two plugins come pre-installed with the default application? Obviously, the Vue framework's team is trying hard to follow best practices and be up-to-date with the modern approach to building web applications. Using Babel is one of these best practices.

If you visit the Babel website, you'll see the following definition of what it is:

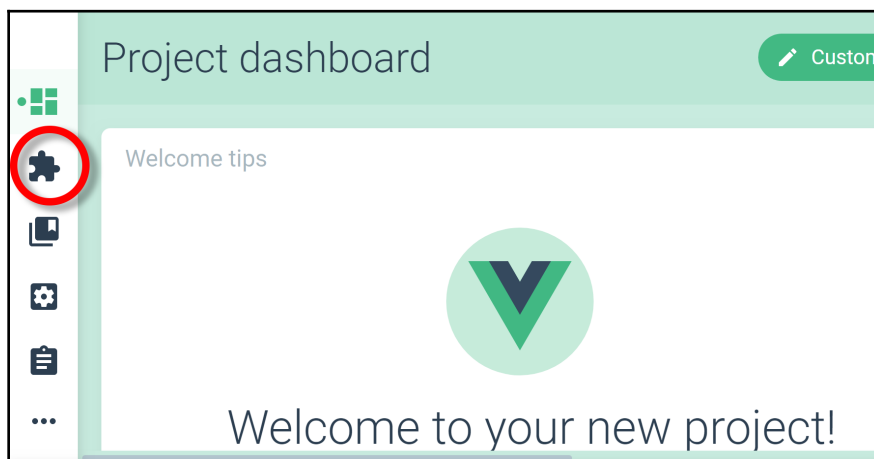
*"Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JS in current and older browsers or environments."*

So, how can we use the Vue CLI Babel plugin? And what would be the easiest way to get more information about it?

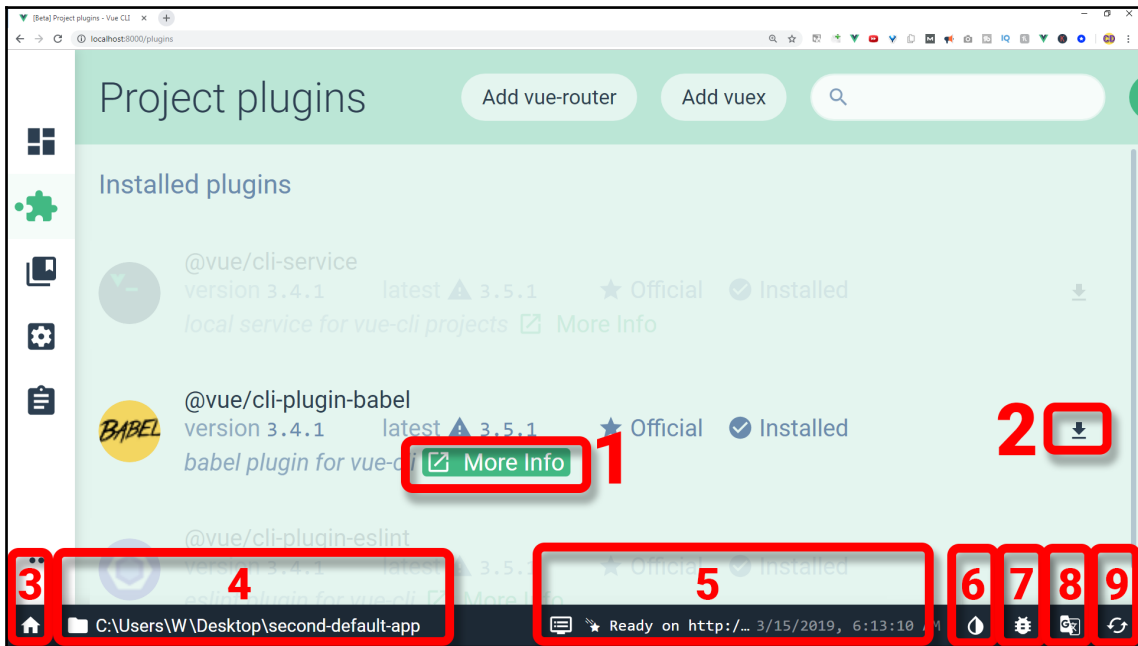
Since we have already used Vue CLI to create default Vue apps, and since we have also learned about Vue CLI's UI, we can reach the official documentation easily, by just opening our Git Bash and starting the Vue CLI UI:

```
vue ui
```

As we've seen in [Chapter 1, \*Introducing Vue CLI 3\*](#), this command will have webpack serve our most recent project's dashboard in the browser. There, we can click the **Plugins** icon, as can be seen in the following screenshot:



Once you've clicked on the **Installed plugins** link, you'll see the following screen:



This app lists three default plugins installed: `@vue/cli-service`, `@vue/cli-plugin-babel`, and `@vue/cli-plugin-eslint`. To make things easier to understand, the other plugins have been grayed out, and numbered boxes have been added to the screenshot:

1. **More info** links to the GitHub repository of `cli-plugin-babel`
2. Update `@vue/cli-plugin-babel`
3. The button with the home icon is the link to Vue UI's project manager that lists all the available projects
4. This section of the UI shows the location of the current Vue project in your OS
5. Clicking here lets you toggle Vue UI's logs on and off
6. As we've seen earlier, this allows you to switch between the two color variations of the Vue UI
7. The report bug icon takes you to the Vue-CLI bug-reporting web page
8. If you're interested in translating the UI, this button links to the page for UI localization
9. This icon simply refreshes the plugin's API

If you need to use the popular **vuex** or **vue-router** plugins, you can simply click on the appropriate buttons at the very top of the plugins page to install them.



The search input to the right of the **Add vuex** and **Add vue-router** buttons lets you filter the installed plugins, and the **Add plugin** button takes you to the `localhost:8000/plugins/add` screen, where you can choose from a number of plugins, such as `@vue/cli-plugin-unit-jest`, `@vue/cli-plugin-typescript`, `@vue/cli-plugin-pwa`, and so on. There is a huge selection of plugins here, and we'll look into it in more detail in later chapters.

In the next section, we'll discuss all the features of `cli-plugin-babel`.

## The building blocks of `@vue/cli-plugin-babel`

There are several parts that `@vue/cli-plugin-babel` comes with out-of-the-box. These are Babel 7, `babel-loader`, and `@vue/cli-plugin-babel`.

### Babel 7 in `@vue/cli-plugin-babel`

Here is the problem that Babel solves.

Let's say that you are developing your web app's frontend, and you are using the more modern, ES6+ syntax of the JS language. Once your app is complete, and released into the wild, some of your users run your web app on, say, Internet Explorer. Contrary to other users of your web app who get a smooth experience running your app, the Internet Explorer users would get a syntax error instead.

Babel was the answer to problems such as this. It *leveled the playing field*: it allowed developers to outsource their JS browser compatibility problems to Babel. Instead of having to worry about, and cater to older browsers, they could simply write their JS code using the newest features of the language, even before they were fully available in any browser. Babel was then responsible for converting this code to the older JS dialect, the one that older browsers could speak.

The `@vue/cli-plugin-babel` runs on Babel 7, which was released on August 27, 2018. There is a three-year difference between Babel 6 and Babel 7, and this iteration came with a slew of improvements. Vue CLI supporting such a recent update is another testament to its team being dedicated to keeping up with the times as much as possible.

## The role of babel-loader in @vue/cli-plugin-babel

As we've seen in the previous chapter, Vue CLI runs on webpack 4.

To be able to use Babel 7, `@vue/cli-plugin-babel` employs `babel-loader`, available here: <https://github.com/babel/babel-loader>.

As mentioned in the previous chapter, with webpack loaders, we can preprocess and bundle a bunch of different resources, not just regular JS, but virtually any other static resource.

Specifically, `babel-loader` takes ES6+ JS, and converts it to ES5 JS. This process is commonly referred to as **transpiling**. Thus, the role of `babel-loader` in `@vue/cli-plugin-babel` is to **transpile** our ES6+ code to ES5.

## The role of @vue/babel-preset-app

There is more to `@vue/cli-plugin-babel`. It includes the `@vue/babel-preset-app`, whose sole purpose is to be used in projects generated via Vue CLI. Without going into too many details on how `@vue/babel-preset-app` works, we can list its main features:

- It looks at your browser targets using `browserslist`
- It automatically applies transforms and polyfills needed (achieved with the help of `@babel/preset-env`)
- It adds support for Vue JSX
- It prevents inlining helpers in all the files during builds

Other than the previously listed features, there are also other things that `@vue/cli-plugin-babel` is capable of, and we'll discuss them in the next section.

## Other features of @vue/cli-plugin-babel

Besides the default settings listed in the previous section, `@vue/cli-plugin-babel` is also extendable. We can add other Babel presets and plugins using `babel.config.js`.

It uses a couple of webpack loaders to perform two other major tasks: caching (with the help of `cache-loader`), and utilizing multi-core processors (with the help of `thread-loader`). This is referred to as **parallelization**.

In the next section, similar to what we did in [Chapter 2, Webpack in Vue CLI 3](#), we will go through setting up Babel in Vue without CLI. After that, we will look at how CLI makes things a lot easier and how it can further be extended.

## Using Babel and webpack in Vue 2 without Vue CLI

Let's call our new project `npm-vue-babel-webpack`. We'll open Git Bash, add the folder for this project, and `cd` into it:

```
mkdir npm-vue-babel-webpack && cd $_
```

We'll initialize NPM and accept all the defaults:

```
npm init -y
```

In [Chapter 2, Webpack in Vue CLI 3](#), we installed NPM packages one by one, explained what each one does, and fixed any errors along the way. That gave us a thorough understanding of the building blocks of webpack and of how Vue and webpack fit together. To avoid unnecessary repetition, this time we'll just install everything at once.

## Installing the necessary NPM packages

Install the necessary NPM packages:

```
npm install vue webpack webpack-cli html-webpack-plugin vue-loader vue-template-compiler --save-dev --verbose
```

Now add the `src` and `dist` folders to our project, and open our project in VS Code:

```
mkdir dist src && code .
```

Feel free to inspect the contents of `package.json` in VS Code, to confirm all the NPM packages were indeed installed.

Let's create three new files inside the `src` folder, specifically `main.js`, `App.vue`, and `index.html`, almost the same as we did in [Chapter 2, Webpack in Vue CLI 3](#).

Here's the code to be added to `index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Compiled HTML file</title>
</head>
<body>
  <div id="entryPoint"></div>
</body>
</html>
```

Here is the contents of `main.js`:

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

new Vue({
  el: '#entryPoint',
  render: h => h(App),
})
```

Finally, here is the contents of `App.vue`:

```
<template>
  <div id="entryPoint">
    <article>
      Our own custom article component!
    </article>
    <AnotherComponent />
  </div>
</template>

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

export default {
  name: 'entryPoint',
  components: {
    AnotherComponent
  }
}
</script>
```

Notice that inside the preceding `script` tag, we are importing `AnotherComponent` from a `components` folder.

So, let's add a `components` folder inside our project's `src` folder. Inside the `components` folder, we'll add a new file and call it `AnotherComponent.vue`.

Next, add this code to `AnotherComponent.vue`:

```
<template>
  <p>
    This is another component.
    <button v-on:click="alertTime">What's the time?</button>
  </p>
</template>

<script>
export default {
  name: "AnotherComponent",
  data() {
    return {
    }
  },
  methods: {
    alertTime: function() {
      alert(new Date());
    }
  }
};
</script>

<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>
</style>
```

In the preceding code, we are finally seeing an example in which our component has some basic JS-powered Vue functionality. We are using Vue's built-in `data` and `methods` options. Inside the `methods` option, we define the `alertTime` function, which, whenever it gets called, shows the current time in an alert box.



Discussing the details of how all these moving parts work is outside of the scope of this book. The focus of this chapter is understanding Babel. If you need more information on the basic concepts of Vue such as the ones mentioned in the previous paragraphs, refer to one of many useful resources in the Packt library. A great companion to this book would be a quick introduction to the Vue 2 framework: *Vue.js Quick Start Guide* by *Ajdin Imsirovic* (<https://prod.packtpub.com/in/application-development/vuejs-quick-start-guide>).



The thing that we need to focus on at this point is working with ES6+ features in our `methods` option. Currently, the `methods` option's code is written in ES5 JS, and, thus, it will be easy to run webpack on this code, as we'll soon see.

## Running webpack with ES5 code

To run webpack, do the following:

1. Add another file to the root of our project, `webpack.config.js`, so that we can set up our webpack configuration as follows:

```
let HtmlWebpackPlugin = require('html-webpack-plugin');
let VueLoaderPlugin = require('vue-loader/lib/plugin');

module.exports = {
  entry: './src/main.js',
  output: {
    filename: 'main.js',
  },
  module: {
    rules: [
      { test: /\.vue$/, use: 'vue-loader' }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',
    }),
    new VueLoaderPlugin(),
  ]
};
```

Note that the entry and output files are both `main.js`, so we don't have to specify them, but we're doing this anyway in the preceding code, to make things more obvious.

2. Next, in `package.json`, update the `scripts` key:

```
"scripts": {
  "webpack": "webpack"
},
```

3. Now run our project from Git Bash, with the following command:

```
npm run webpack
```

4. Now, in VS Code, navigate to the `dist` folder in our project.
5. Right-click on `index.html`, and click on the **Open in Default Browser** command.

Our browser will now show the following output (zoomed-in for easier viewing):

```
Our own custom article component!  
This is another component. 
```

If a user clicks on the **What's the time?** button, an alert box will appear on the web page, showing the current time. Now let's update our `methods` option to ES6 syntax, and see what happens.

## Adding webpack-dev-server

Before we begin updating our code to ES6 syntax, there is one more thing that we can do to make things quicker and more convenient:

1. Add the `webpack-dev-server`. With the help of this NPM package, our code will be continually served and watched for changes. Let's install it with the following:

```
npm install webpack-dev-server --save-dev --verbose
```

2. To get the webpack dev server running and serving our code, we also need to update `scripts` of `package.json` entry to this:

```
"scripts": {  
  "webpack": "webpack",  
  "webpack-dev": "webpack-dev-server --mode development"  
},
```

Now we can play around with adding various features to our components and watch them being hot reloaded in the browser whenever we save our code in VS Code.

3. Let's test it out right now by running the following command:

```
npm run webpack-dev
```

You can test out the web page served at `http://localhost:8080/`, and you'll notice that it still works, just as before.

Next, we'll add some ES6 syntax to our `methods` option.

## Updating the `methods` option to ES6 syntax

Let's update the `methods` option in `AnotherComponent.vue` file. Here is the updated code:

```
methods: {
  alertTime: () => {
    alert(new Date());
    alert('something else');
  }
}
```

Once you save changes you made in VS Code, you can click the **What's the time?** button, and the expected alert will appear, followed by another alert that reads **something else**. This way, we can be sure that we're looking at the updated app.

Now let's find our ES6 code in the compiled `main.js` file inside the `dist` folder.

If we inspect the `/dist/index.html` file inside the dev tools, we can see a reference to `main.js`, the webpack-compiled JS code. If you right-click `main.js` and click the **Open in new tab** command on the contextual right-click menu, you'll get the full code showing in a new tab. To find our ES6 code, let's press the `Ctrl + F` shortcut key so that we can enter our search term: `alertTime`.

Toward the very bottom of the file, we see our ES6 arrow function:

```
alertTime: () => {\r\n    alert('something else');
```

In the next section, we'll update our webpack configuration with `babel-loader`, and see how webpack will transpile the preceding code to ES5.

## Adding `babel-loader` to our webpack configuration

Before we begin, we'll need to stop our `webpack-dev-server`, by using the `Ctrl + C` keyboard combination inside Git Bash.

Next, to be able to transpile ES6+ syntax in our project, we need to update our webpack configuration with Babel. Let's first install the babel-loader package with NPM:

```
npm install babel-loader --save-dev --verbose
```

Next, let's run webpack on our project again:

```
npm run webpack
```

Unfortunately, this still doesn't work. If we inspected our transpiled `main.js`, we'd still see `alertTime` key, and it's accompanying ES6 anonymous function. This means we still need another package: `babel core`.

```
npm install @babel/core --save-dev --verbose
```

If we ran webpack at this point, we'd see that our issue was still not fixed.

This means that we still need to add `babel-preset-env`:

```
npm install @babel/preset-env --save-dev --verbose
```

At this point, it doesn't hurt to verify that our `package.json` has all the expected updates in the `devDependencies`:

```
"devDependencies": {
  "@babel/core": "^7.3.4",
  "@babel/preset-env": "^7.3.4",
  "babel-loader": "^8.0.5",
  "html-webpack-plugin": "^3.2.0",
  "vue": "^2.6.9",
  "vue-loader": "^15.7.0",
  "vue-template-compiler": "^2.6.9",
  "webpack": "^4.29.6",
  "webpack-cli": "^3.2.3",
  "webpack-dev-server": "^3.2.1"
}
```

Finally, before we can re-run webpack, we need to set up a `babel.config.js` file, Babel's own configuration file (similar to webpack's `webpack.config.js`).

Let's create a new file, `babel.config.js`, in the root of our project, and let's add it the following code:

```
module.exports = {
  presets: ['@babel/preset-env']
}
```

All that we need to do now is update our webpack configuration so that it can work with Babel.

## Updating our webpack configuration to work with babel

To make our webpack work with babel, we need to explain to it when to use babel-loader. We do that by adding a rule to test for, inside the `module` option of `webpack.config.js`, as follows:

```
module: {
  rules: [
    { test: /\.js$/, use: 'babel-loader' },
    { test: /\.vue$/, use: 'vue-loader' }
  ]
},
```

Now that we've set everything up, we can run our `npm run webpack-dev` command in Git Bash again.

Here's a quick way to see whether webpack bundled our JS file differently than before: just look at the webpack logging information inside Git Bash. During our previous attempts at hooking up Babel to work with webpack, the bundle size was exactly 70.2 KB. However, after this last run of the webpack NPM script, the bundle size for `main.js` is 70.6 KB. We can inspect the `./dist/main.js` file in our dev tools again. Alternatively, you can search for an `alertTime` string inside `./dist/main.js` right inside your VS Code.

Regardless of how we locate it, our bundled `main.js` file's `methods` entry looks like this:

```
methods:{alertTime:function(){alert(new Date),alert("something else")}}...
```

Just glancing at the preceding code and seeing the `function` keyword, it should be obvious that this code is ES5, which means Babel was successfully run by webpack, and our ES6 `alertTime` Vue method in our input file inside the `src` folder was successfully transpiled to the output file inside the `dist` folder.

To verify that our setup works, we can run our `webpack-dev-server` again, and while it's running, make another minor change to our `methods` option inside

`AnotherComponent.vue`:

```
methods: {
  alertTime: () => {
    alert(new Date());
  }
}
```

```
    alert('whatever');
  }
}
```

If you take a look at the project served at `localhost:8080`, you'll see it works as expected, and if you open `main.js` from the dev tools, you'll see the transpiled syntax as well.

In the next section, we'll briefly mention a common source of confusion and how it is dealt with in Vue. It has to do with the arrow function syntax, and the `this` keyword.

## The problem with the `this` keyword in arrow functions in Vue

Unfortunately, when Babel transpiles the `this` keyword inside an arrow function, it transpiles it to `_this`. That means that any of our methods will be broken and our apps won't work. The underlying cause for this is due to the fact that arrow functions are scoped differently than ES5 functions.

In the next section, we'll look at the recommended approach in defining functions inside the `methods` option of a Vue instance.

## Recommended solutions to keyword problems

The recommended solution to the `this` keyword inside Vue components is to not use the arrow function syntax since it doesn't produce the expected results due to scoping issues. Specifically, arrow functions are scoped to the parent context.

Let's see a simple app as an example of the problem.

## Adding one counter app

Before we begin, make sure you're back inside the `VUE-CLI-3-QSG/Chapter03` folder, the root folder for all our projects in this chapter.

Before we even begin building our app, we'll need to remind ourselves of some options we have when using `vue create`, so let's run this:

```
vue create --help
```

Inside the list of options, we can see that `-d` is for the `--default` flag, which skips prompts and uses default presets, and that the `-b` option is shorthand for `--bare` flag, which scaffolds our project without beginner instructions.

Interestingly, we can combine these individual flags, which we'll do right now. Let's begin our app by running the following command:

```
vue create add-one-counter -db
```

As we can see, we can chain the letter aliases for flags appended to the `vue create` command, which is a nice little productivity boost.

In the app that gets built, we'll alter the contents of `main.js` inside the `src` folder. This file will be exactly the same as it was in the previous example app (the `npm-vue-b-w-es6-syntax` app from the previous section), so you can just copy and paste this file from that previous Vue app into our new example, the `add-one-counter` app.

If you open our new `add-one-counter` app inside VS Code, you'll also notice another folder: the `public` folder, with `index.html` inside it. We will leave this file as it is.

Back inside the `src` folder, we need to change the contents of `App.vue`, to this:

```
<template>
  <div id="app">
    <article>
      Our own custom article component!
    </article>
    <AnotherComponent />
  </div>
</template>

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

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

Finally, we'll need to add a `components` folder to the root of our project, and inside it, the `AnotherComponent.vue` file. Here is the contents of `AnotherComponent.vue`:

```
<template>
  <p>
    This is another component.
    <button v-on:click="incrementUp">Add One</button>
    <br>
    <span>Current value of the counter: {{ counter }}</span>
  </p>
</template>

<script>
export default {
  name: "AnotherComponent",
  data() {
    return {
      counter: 0
    }
  },
  methods: {
    incrementUp: function() {
      this.counter++;
    }
  }
};
</script>

<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>
</style>
```

In the `methods` option, we can see the `incrementUp` function defined using the ES6 syntax.

If you tried running this app, it would not work. This is the case, because the arrow function's scoping and the Babel setup make it difficult to properly set up Vue apps with `methods` in the arrow functions.

The only improvement, and the usual way to write functions inside the method options, is to avoid both the arrow function syntax, and the use of the `function` keyword altogether, as follows:

```
methods: {
  incrementUp() {
    this.counter++;
  }
}
```



The `incrementUp` function is called a **shorthand** function. You can read more about it at following URL: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object\\_initializer#Method\\_definitions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#Method_definitions).

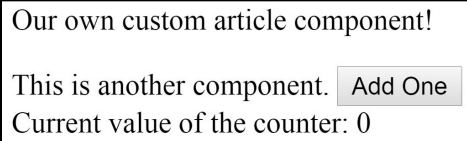
Let's test-drive the app by serving it from the UI:

```
vue ui
```

Once the Vue UI is serving in the browser, let's point our browser's address bar to `http://localhost:8000/project/select`. Next, click on the `add-one-counter` folder, and click the **Import this folder** button.

Next, click on the **Tasks** button on the main menu. Finally, click the **Serve** icon. Click the **Output** button to see the app building and serving.

Finally, open the website at `http://localhost:8080/`. You'll see a working app being served:



The previous example showed us the best practice of how to name and organize methods inside our Vue instances. Also, we saw how to use Vue UI to automate webpack building and serving our apps in a few clicks—much better than what we were doing for most of this chapter! This leads us to a conclusion: a lot of the piping and functionality is abstracted away and hence, this made a lot easier for using and accessing with the help of Vue UI, including Babel settings.

## Summary

In this chapter, we had a brief overview of what Babel is, what it does, and what is needed to make it work with Vue. All of this is abstracted away with the help of `vue-cli-service`, which is powered by webpack under the hood. Now that we have an understanding of how all these different parts work together, we will start using only the Vue CLI and its UI, and learn how to work with it better in the following chapters.

We'll begin by understanding testing in Vue CLI with the help of Jest. We'll also learn about **Test-Driven Development (TDD)** and running tests with the help of Vue CLI UI.

# 4

## Testing in Vue CLI 3

In the previous chapter, we examined the role of Babel in modern JavaScript development. We've also seen some practical examples of using it in Vue. In this chapter, we will introduce testing in JS. We will get acquainted with testing in general and see it in practice using Jest and Cypress. We'll discuss assertions and **test-driven development (TDD)**. We'll then move on to understanding how Jest and Cypress work with Vue CLI 3. We'll discuss test utils and test cases. Specifically, we'll look at the following:

- Understanding Vue plugins
- Adding the Jest plugin to our Vue app
- Writing unit tests inside a Vue app with Jest
- Running tasks from the project tasks page
- Running unit tests inside the Vue CLI UI
- Working with assertions
- Implementing TDD
- Working with Cypress

We'll begin this chapter with a brief overview of Vue plugins.

### Understanding Vue plugins

When making a new Vue app with the help of the Vue CLI from the command line, we use the `vue create` command. We are then taken through a number of steps and prompts that we need to choose, so that our app is properly configured. What we are actually doing, among other things, is choosing which Vue plugins will end up being used in our app.

Plugins are a way to add functionality to our Vue projects. Some plugins are more complex than others; they sometimes have their own prompts that appear during installation. Our Vue app's configurations, that is, the underlying code, will reflect our choices. The way our app is set up will be based on the answers we provide to these installation prompts.

All the official `npm` packages of a project are scoped with the `@` symbol, followed by the project name. Thus, official Vue plugins, built by the Vue maintainers, begin with `@vue`.



To find out more about scoped `npm` packages, visit: <https://docs.npmjs.com/about-scopes>.

To add a plugin from the command line, we use the `vue add` command, but we can use the Vue UI as well, as we'll see in this chapter. The Vue UI is also a great way to search for Vue plugins, which we'll also examine in this chapter.

## Beginning testing on a brand new Vue app

We've seen many different options available in the Vue CLI and UI in previous chapters. We'll begin this chapter by utilizing the best approach to start a new app, the Vue CLI UI. This will help us learn about some other features of the UI. Along the way, we'll also slowly introduce testing to our project.

## Adding a new project with Vue CLI UI

Let us now add a new project with Vue CLI UI:

1. To begin, let's open Git Bash and navigate to the root folder for all of our projects, `vue-cli-3-qsg`.
2. Now we'll run the Vue CLI UI command as follows:

```
vue ui
```

This will result in a new page being served in the browser. The default address is `http://localhost:8000/dashboard`.

3. Next, click on the home icon (or simply type out this URL in the address bar: `http://localhost:8000/project/select`), which will take you to the **Vue Project Manager** screen.

Note from the following that the home icon is the leftmost icon in the footer of the Vue CLI UI:

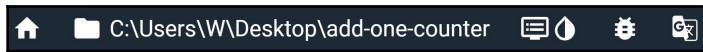


Figure 4.1: The home icon in the Vue CLI UI

4. Regardless of how you access the **Vue Project Manager** screen, it will show a list of available apps, together with three tabs at the top: **Projects**, **Create**, and **Import**. Click on the **Create** tab to create a new project.
5. Once you've clicked on the **Create** tab, you need to get back to the root of your project, and then click on the **Create a new project here** button as follows:

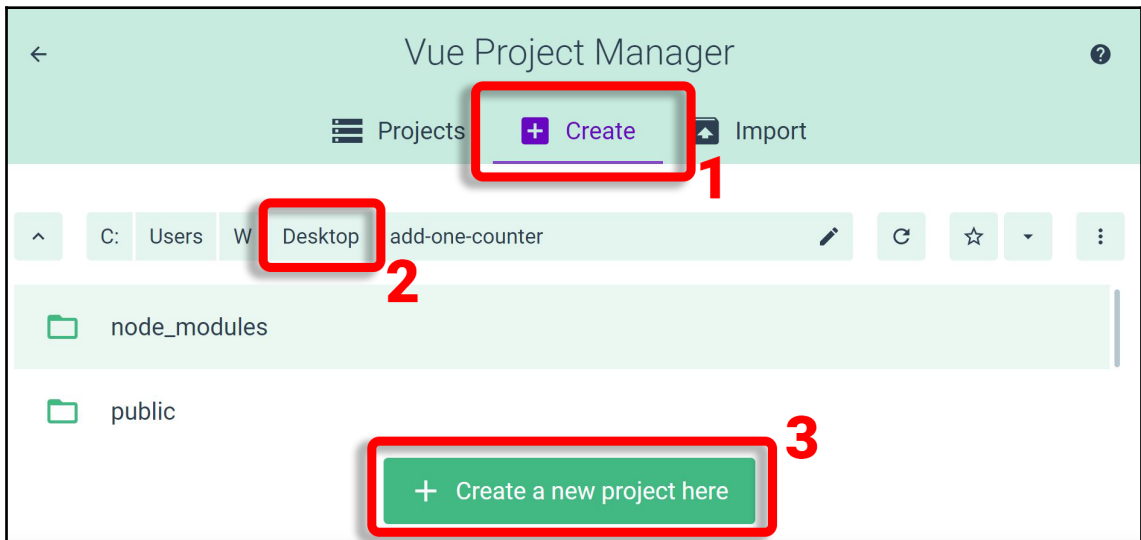


Figure 4.2: Create, get to root of all projects, Create a new project here button

Once you click the **Create a new project here** button, you'll see the **Create a new project** screen. We will only type our new app's folder name. We'll call it `testing-debugging-vuecli3`. We will not change anything else: we'll accept the default package manager and other default options as follows:

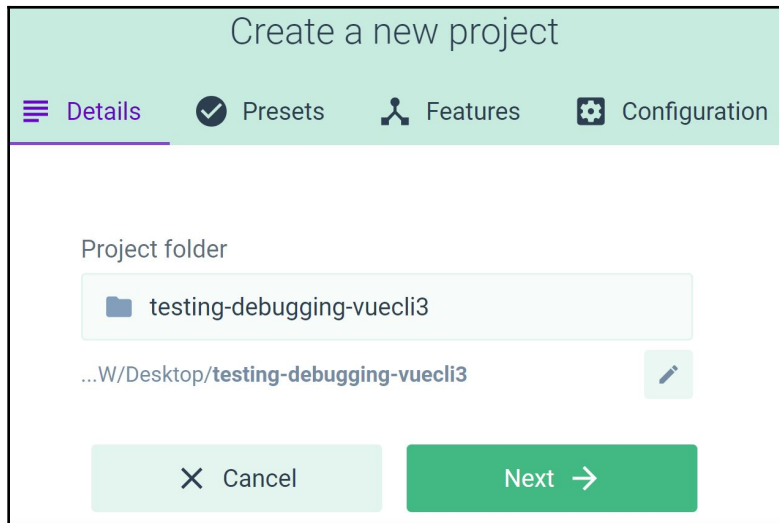


Figure 4.3: Adding the name for the folder that will hold our new app

We've completed all the necessary changes in the **Details** tab.

- By clicking the **Next** button, we'll be taken to the **Presets** screen. There, we can just accept the **Default preset (babel, eslint)** as follows:

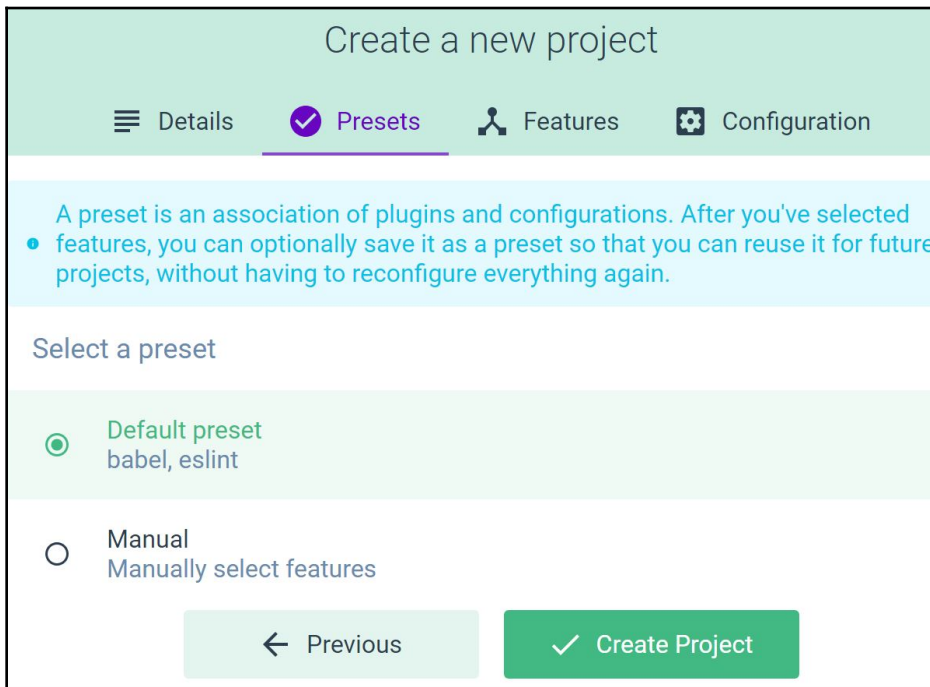


Figure 4.4: Accepting the default presets

- Next, we'll click **Create Project** to scaffold our project. It will take some time for the Vue CLI UI to complete the project scaffolding. Once done, we'll be greeted with the **Welcome to your new project!** screen.

## Adding the Jest plugin to our Vue app

Now let us add our Jest plugin:

- Click on the Plugins icon (marked with number 1 in the following screenshot).

2. Once the **Project plugins** screen appears, click on the **Add Plugin** button (2 in the following screenshot):

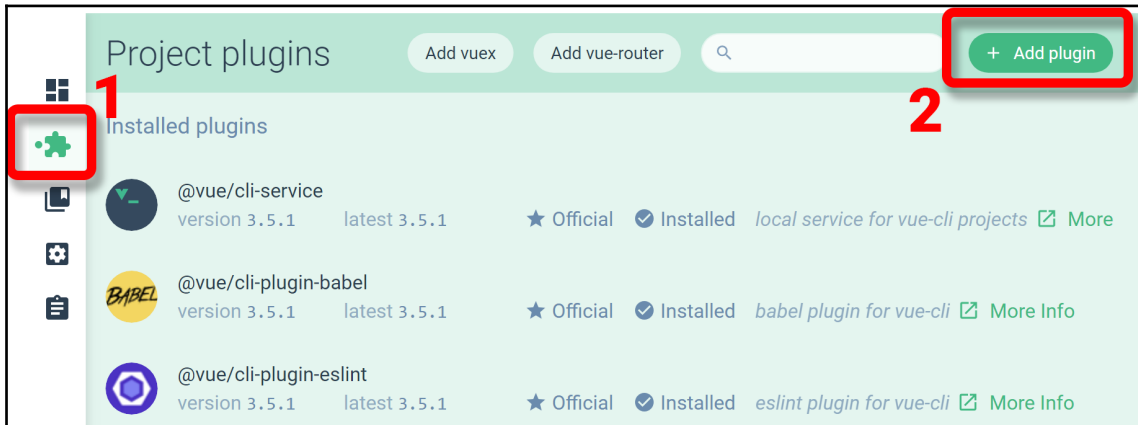


Figure 4.5: Adding a new plugin to our installation

3. Doing this will take us to the **Add a plugin** screen, where we have an input field to search for plugins. We need to locate a unit testing plugin, so we can just type out `cli-plugin-unit` as follows:

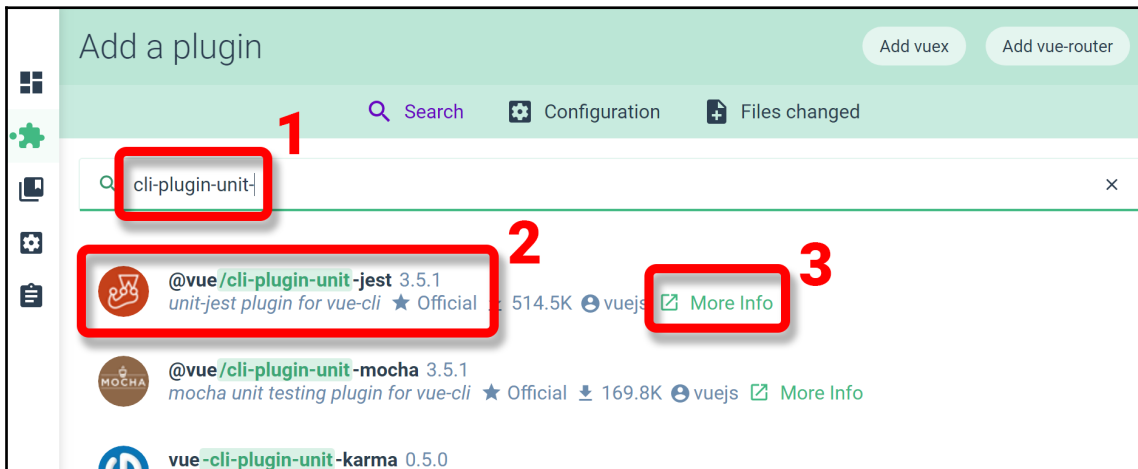


Figure 4.6: Finding unit testing plugins for Vue

- Typing out this search term will show all the available unit testing plugins. Jest should be right there on top. You can see it in the previous screenshot, flagged as **2**. In the description under the plugin name, you can see that it's an **Official** plugin. Similar to what we've already seen with the `@vue/cli-plugin-babel`, you can click the **More info** link to see the corresponding GitHub repository for the plugin in question (flagged as **3** in the preceding screenshot). Doing this will take you to the **vue-cli** GitHub page.



You can find out more about `@vue/cli-plugin-unit-jest` at the npm package page, at the following URL: <https://www.npmjs.com/package/@vue/cli-plugin-unit-jest>.

- To install the Jest plugin, simply click on it in the plugins list. Doing this will add a checkbox in a green circle next to the Jest plugin (flagged as **1** in the following screenshot). A new install button will also appear (box number **2** in the following screenshot):

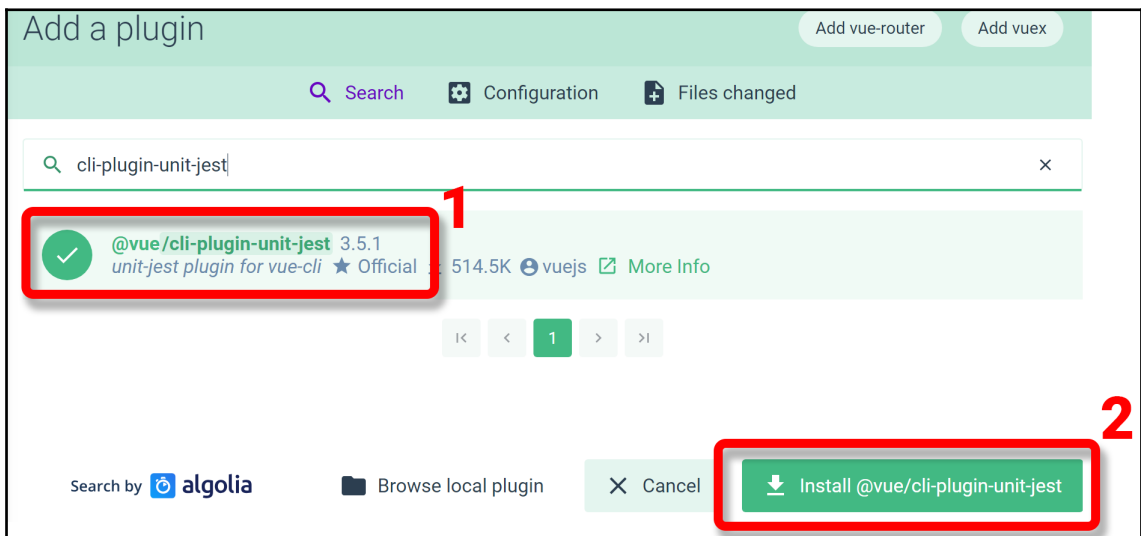


Figure 4.7: Adding the Jest plugin



6. Clicking the **Install @vue/cli-plugin-unit-jest** button will result in a loader appearing on the page, with the following message being shown: **Installing @vue/cli-plugin-unit-jest...**
7. Once done, just click the **Finish installation** button as follows:

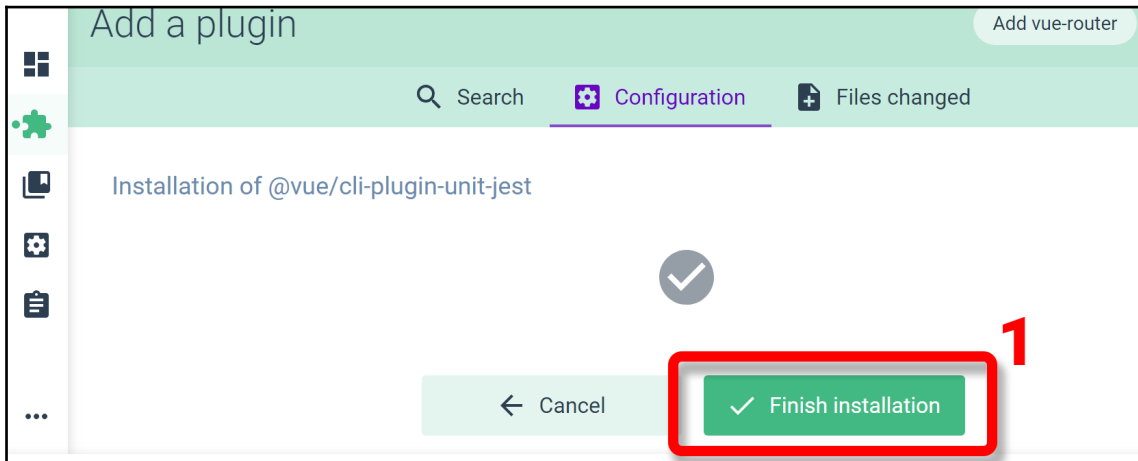


Figure 4.8: Finishing the installation of the Jest plugin

8. Clicking the **Finish installation** button will show the following message on the screen: **Invoking @vue/cli-plugin-unit-jest...**
9. Once this update is finished, we'll be greeted with another screen as follows, showing the changes to files, and asking us to commit the updates made:

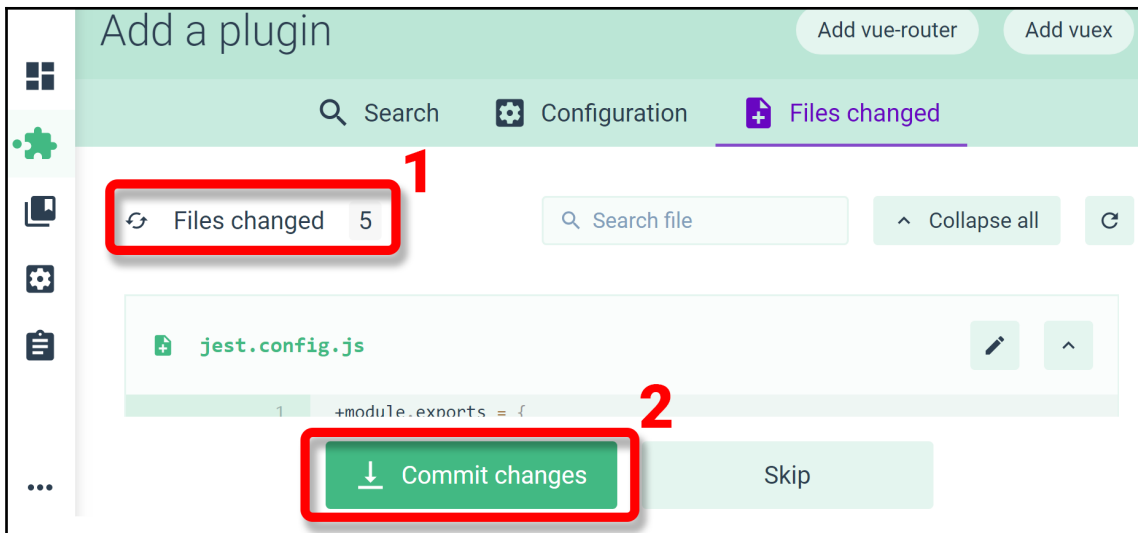


Figure 4.9: Vue CLI UI showing changes after the installation of the Jest plugin

## Examining the updates

In the preceding screenshot, we can see that the **Files changed** tab is active. Inside the **Files changed** tab, we can see the number of files changed (box 1).

When we were building our project with the Vue CLI UI, we were offered the default option to track changes to our project with Git, and we accepted this default setting; that's why we now see the **Commit changes** button (flagged as 2 in the preceding screenshot).

We can also see all the changes and updates made to the two existing files: `package-lock.json` and `package.json`, as well as the contents of the three new files added when we installed the plugin: `jest.config.js`, `tests/unit/.eslintrc.js`, and `tests/unit/example.spec.js`.

It would be beneficial to inspect the contents of each of these files to get more familiar with how they are set up and what changes have been made to them. One of the most important changes that we need to be aware of is in the `package.json` file, inside the `scripts` key, as follows:

```
"test:unit": "vue-cli-service test:unit"
```

The previous line shows that our `vue-cli-service` just got a new command, `test:unit`, specifically set up for unit testing with Jest.

Once we click the **Commit changes** button, we'll see a dialog that invites us to **Enter a commit message**. We can type a simple message, such as `Add Jest plugin`.

After we've added the commit, we'll be taken back to the **Installed plugins** screen. Now we can see that `@vue/cli-plugin-unit-jest` is also added:

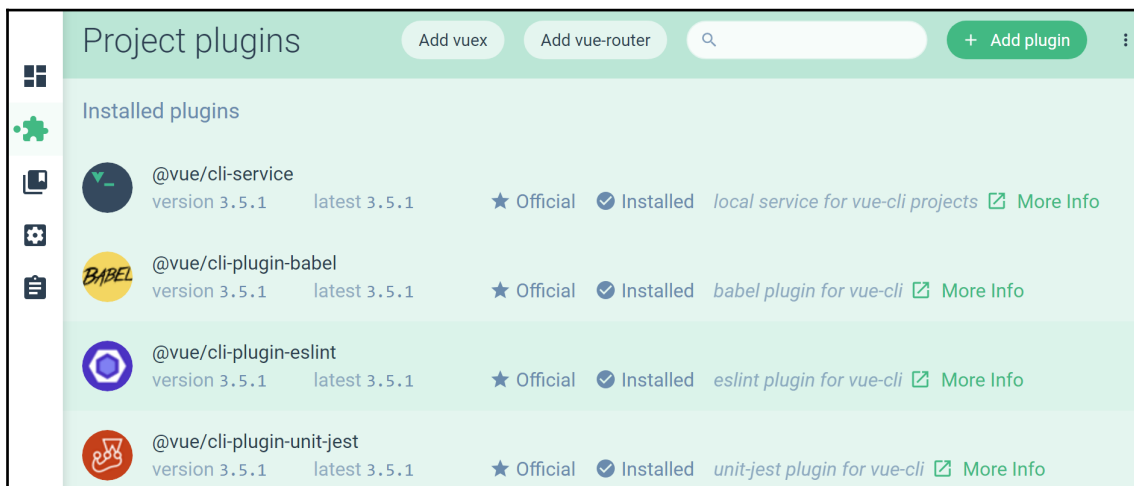


Figure 4.10: Vue CLI UI showing changes after the installation of the Jest plugin

In the next section, we'll be adding our app so that we can begin testing it with Jest.

## Adding the code for our Vue application

To add our app, we need to do the following:

1. Navigate in Windows Explorer to the `testing-debugging-vuecli3` folder.
2. Next, right-click on an empty space inside the folder, and click the **Git Bash here** command.
3. Once Git Bash is open, type `code .` and press the *Enter* key. This will open our `testing-debugging-vuecli3` project in VS Code.

Our focus at this point is on Jest. To avoid unnecessary complications, we'll simply copy and paste the entire `add-one-counter` app from the previous chapter. The easiest way to do that is to point Windows Explorer to the `add-one-counter` app and start another instance of Git Bash with a right-click inside the folder, as already explained. We'll again type out the `code .` command inside Git Bash, and another instance of VS Code will open, this time showing the files inside the `add-one-counter` app. Now it's just a matter of copying and pasting all the files and folders from `add-one-counter` to `testing-debugging-vuecli3`.

Alternatively, you could use Linux commands inside Git Bash to copy the relevant files.

Regardless of how you do it, the updated project structure will now look as follows:

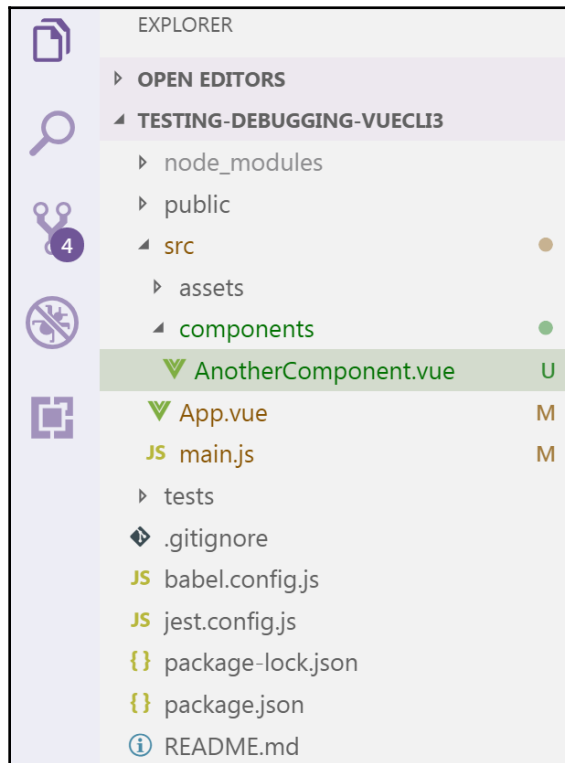


Figure 4.11: The updated project structure of `testing-debugging-vuecli3`

Now, we are ready to start our first unit test with Jest.

## Writing our first unit test inside a Vue app with Jest

Inside the `tests` folder, there is another folder called `unit`. Let's add a new file to the `unit` folder. We'll call this new file `AnotherComponent.spec.js`.



Any file that has a `spec.js` extension will be recognized by Jest.

To describe a test, we use the `describe` function, so let's add it to `AnotherComponent.spec.js` as follows:

```
describe()
```

The `describe` function takes two arguments. The first argument is the name of the Vue component we are testing, and the second argument is an anonymous function as follows:

```
describe('AnotherComponent.vue', function() {})
```

We can rewrite the preceding code using an arrow function as our second argument as follows:

```
describe('AnotherComponent.vue', () => {})
```

In the body of the anonymous function, we call another function, the `test` function as follows:

```
describe('AnotherComponent.vue', () => {  
  test()  
})
```

The `test` function accepts two arguments: the first argument is the name of our test, and the second argument is another anonymous arrow function as follows:

```
describe('AnotherComponent.vue', () => {  
  test('Jest is setup correctly and working', () => {})  
})
```

We are specifying a test we've called `setup working`, and we need to give it an assertion inside the second argument, that is, inside the body of the anonymous function, as follows:

```
describe('AnotherComponent.vue', () => {  
  test('Jest is setup correctly and working', () => {
```

```
    expect(true).toBeTrue();  
  })  
})
```

This assertion will always be true, and so we give it the following code:

```
expect(true).toBeTrue();
```



We have just seen an example of a Jest matcher in action. A matcher is a way to test values in Jest. One way to check if something is true is to use the `toBeTrue` matcher. There are many other Jest matchers. For more information, check out the official Jest documentation at the following URL: <https://jestjs.io/docs/en/using-matchers>.

Now, let's go to the Vue CLI UI, and run our unit test.

## Running our first unit test inside the Vue CLI UI

To run our unit test inside the Vue CLI UI, we need to navigate to the `localhost:8080/tests` page, either by visiting the URL right from the address bar, or by clicking the lowest icon on the main navigation (the Tests icon in the leftmost column in the Vue CLI UI dashboard). Once we do, we'll see the following list of tests:

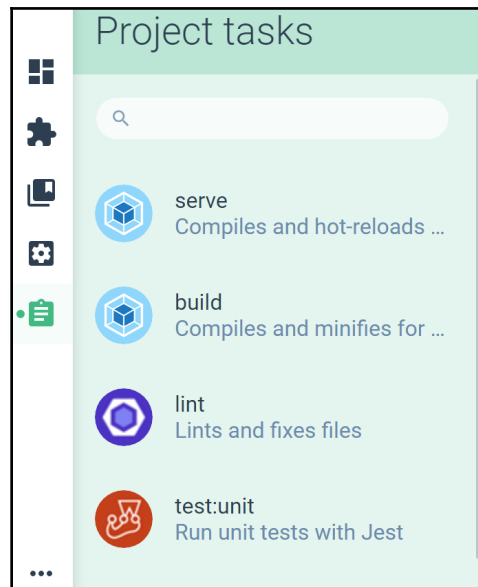


Figure 4.12: Viewing the available tasks inside Vue CLI UI

Next, let's get ready to run the `test:unit` task by clicking on it. Doing so will result in a panel appearing on the right-hand side of the **Project tasks** page as follows:

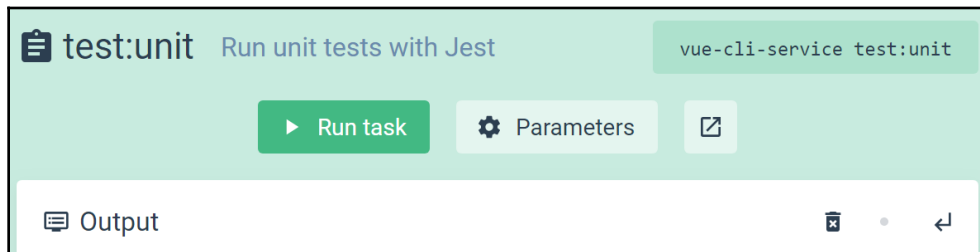


Figure 4.13: The run tasks panel for running the `test:unit` command

This **Run task** panel gives us a nice overview of everything that is happening. Let's now run our test by clicking the **Run task** button.

The following information will be shown in the **Output** section:

```
...
...
PASS tests/unit/AnotherComponent.spec.js

Test Suites: 1 failed, 1 passed, 2 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 2.702s
Ran all test suites.

Total task duration: 3.93s
```

As we can see, our `AnotherComponent.spec.js` test passed successfully. The test that failed and whose output is logged before the `AnotherComponent.spec.js` test, is the test for the `HelloWorld.vue` component. We have removed this `HelloWorld.vue` file from our default app after the app was built by Vue UI.

However, the `example.spec.js` file inside the `./tests/unit` folder is the test file that calls the non-existing `HelloWorld.vue` file. Looking at `example.spec.js`, we can see that it imports the `HelloWorld.vue` component, on line number 2 as follows:

```
import HelloWorld from '@components/HelloWorld.vue'
```

Why is this the case? Haven't we already specified `AnotherComponent.vue` inside the `describe` function?

As it turns out, we could have specified any name as the first argument of the `describe` function inside our unit test. For example, we could update our `AnotherComponent.spec.js` file to this code:

```
describe('whatever', () => {
  test('Jest is setup correctly and working', () => {
    expect(true).toBeTrue();
  })
})
```

If we ran our test again, it would still run.

What this means is that the string `AnotherComponent.vue` is what we developers pass in as the first argument to our `describe` function so that it is easier for us to work in our project. Jest doesn't care what name it has.

What it does care about, however, is importing the file to be tested. As we saw in the `HelloWorld.vue` import, we need to add a similar import to our `AnotherComponent.spec.js` file, so that now it looks as follows:

```
import AnotherComponent from '@components/AnotherComponent.vue';

describe('AnotherComponent.vue', () => {
  test('Jest is setup correctly and working', () => {
    expect(true).toBeTrue();
  })
})
```

Interestingly, we are importing a Vue file, but our tests are still passing, even though the `.vue` extension is not JS. How is this achieved?

We can easily see what is going on if we open the `jest.config.js` file, located at the root of our project. Looking at the first 12 lines of this file, we see the following code:

```
module.exports = {
  moduleFileExtensions: [
    'js',
    'jsx',
    'json',
    'vue'
  ],
  transform: {
    '^.+\\.vue$': 'vue-jest',
    '^.+\\.css|styl|less|sass|scss|svg|png|jpg|ttf|woff|woff2)$': 'jest-transform-stub',
    '^.+\\.jsx?$': 'babel-jest'
  },
}
```



As we can see, the `vue` extension is listed on line 6, and the `.vue` file extension will be transformed using the `vue-jest` plugin, as specified on line 9.

Before we continue, let's rename our `example.spec.js` file to `example.js`, so that Jest doesn't catch it. We still need the contents of the file, so let's not delete it, but rather just rename it.

## Importing `mount` from `test-utils` and writing another unit test

We'll begin by adding a `mount` import from `@vue/test-utils` in the very first line our `AnotherComponent.spec.js` file as follows:

```
import { mount } from '@vue/test-utils';
```

Before we continue, we need to look at what this syntax does. Why the curly brackets around `mount`?

To answer this question, it's good to know that this is accepted JS syntax. To explain what is happening, we need to begin with the `package.json` file.

This file was created by our Vue CLI, when we built our project. If we look at the contents of the `package.json` file, we'll see `@vue/test-utils` listed as one of our project's `devDependencies`.

In the preceding code, we are importing a single function, `mount`, from this `@vue/test-utils` JS module. By doing this, we are inserting the `mount` function into our `AnotherComponent.spec.js` file's scope.

In simpler terms, we are importing the `mount` functionality from `@vue/test-utils`, so that we can use it in the `AnotherComponent.spec.js` file, and only test this component alone.

With the Vue CLI UI running in our browser, let's look at a list of our project's dependencies by visiting the following URL: <http://localhost:8000/dependencies>.

You should see a screen similar to the following screenshot:

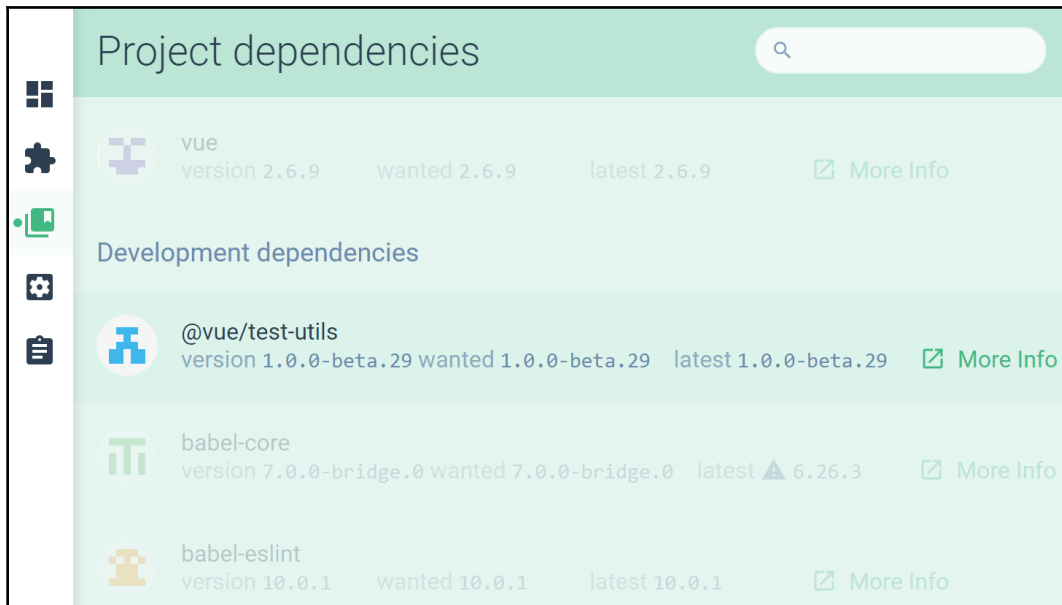


Figure 4.14: Our project's devDependencies listed in the Vue CLI UI dashboard

As usual, clicking the **More info** link for our `@vue/test-utils` project dependency will take us to the corresponding GitHub repository for the project: <https://github.com/vuejs/vue-test-utils#readme>.

## Mounting the component to be tested

We begin by importing the `mount` method and the component to be tested as follows:

```
import { mount } from '@vue/test-utils';
import AnotherComponent from '@components/AnotherComponent.vue';
```

The `mount` function receives a component as its argument. The result of the call to the `mount` function is a wrapper with an instance of the component we gave it. This wrapper also comes with additional functions that help our testing process. Let's first assign the returned value of the call to `mount` (`AnotherComponent`) to a variable as follows:

```
import { mount } from '@vue/test-utils';
import AnotherComponent from '@components/AnotherComponent.vue';
```

```
describe('AnotherComponent.vue'), () => {
  test('Adds one when a user clicks the button', () => {
    const wrapped = mount(AnotherComponent);
  })
}
```

## Writing a failing assertion

When our app is initially loaded into a browser, we are expecting the current value of the counter to be 0. However, since we want our assertion to initially fail, let's assert that the value of the counter is going to be 1 instead of 0 as follows:

```
import { mount } from '@vue/test-utils';
import AnotherComponent from '@components/AnotherComponent.vue';

describe('AnotherComponent.vue'), () => {
  test('Adds one when a user clicks the button', () => {
    const wrapped = mount(AnotherComponent);
    expect(wrapped.text()).toContain('Current value of the counter: 1');
  })
}
```

We have written a failing assertion in the preceding code. We claim that we are expecting that our wrapped component will contain the following text:

```
Current value of the counter: 1
```

The initial value of our counter is not going to be 1; it will actually be 0, thus the preceding assertion should fail.

So, let's save and run our unit test by going to the **Project tasks** screen and running the test as described earlier in the chapter.

The output will be shown in the **Output** panel of **Run task** inside **Project tasks** as follows:

```
Test Suites: 1 failed, 1 total
Tests: 0 total
Snapshots: 0 total
Time: 1.947s
Ran all test suites.
```

```
Total task duration: 3.14s
```

Next, we'll fix the preceding test by writing a passing assertion.

## Writing a passing assertion

To write a passing assertion, all we need to do is revert our 1 to a 0, as follows:

```
expect(wrapped.text()).toContain('Current value of the counter: 0');
```

Next, let's run our task again inside the Vue UI, and we'll get back the following output:

```
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 2.418s
Ran all test suites.
```

```
Total task duration: 3.55s
```

Next, we'll trigger a button click in our test.

## Triggering a button click in our test

How can we test for a button click in our unit test? Perform the following steps:

1. We need to find the button to be clicked. This is easy, since there is only one button in our app. We will use the find method to do that.
2. We are going to trigger a click on a button, using the trigger method.
3. We need to check for the value of the counter to change from 0 to 1. However, since we first need to write a failing test, we'll write out the following code:

```
import { mount } from '@vue/test-utils';
import AnotherComponent from '@components/AnotherComponent.vue';

describe('AnotherComponent.vue', () => {
  test('Adds one when a user clicks the button', () => {
    const wrapped = mount(AnotherComponent);
    expect(wrapped.text()).toContain('Current value of the counter:
0');
    const button = wrapped.find('button');
    button.trigger('click');
    expect(wrapped.text()).toContain('Current value of the counter:
0');
  })
})
```

As expected, the output of running our test in the Vue CLI UI is the following:

```
Test Suites: 1 failed, 1 total
Tests: 1 failed, 1 total
Snapshots: 0 total
Time: 2.383s
Ran all test suites.

Total task duration: 3.55s
```

Let's make things a bit more interesting by writing two tests, and asserting them in such a way that they both pass as follows:

```
import { mount } from '@vue/test-utils';
import AnotherComponent from '@/components/AnotherComponent.vue';

describe('AnotherComponent.vue', () => {
  const wrapped = mount(AnotherComponent);
  const button = wrapped.find('button');
  test('Checks that the initial counter value is 0', () => {
    expect(wrapped.text()).toContain('Current value of the counter:
0');
  });
  test('Adds one when a user clicks the button', () => {
    button.trigger('click');
    expect(wrapped.text()).toContain('Current value of the counter:
1');
  });
});
```

Let's save our test again and run it as a task inside the Vue CLI UI one more time.

The following is the output:

```
AnotherComponent.vue
  ✓ Checks that the initial counter value is 0 (3ms)
  ✓ Adds one when a user clicks the button (4ms)

Test Suites: 1 passed, 1 total
Tests: 2 passed, 2 total
Snapshots: 0 total
Time: 2.307s
Ran all test suites.

Total task duration: 3.64s
```

We have successfully written two separate unit tests in one test suite for a Vue component, and both our tests have passed.

## Test-driven development in Vue CLI 3

TDD is development based on the idea of the **red-green-refactor** cycle. Similar to what we've seen in the preceding code, we first write our code so that our test fails. Next, we write our code so that our test passes, and finally we refactor our code.

For each new feature in our apps, we repeat the same process. This is essentially what TDD is.

TDD is just a streamlined way of writing any application in any language or framework. It streamlines our work by allowing us to split our entire project into testable, clearly separated chunks of functionality.

The red-green-refactor approach is clearly visible in our output on the **Project tasks** page too. If we write a failing test, we'll see a red background on the word **fail**. If we write a passing test, we'll see a green background on the word **pass**.

Through the rest of this chapter, we'll look at some other concepts related to testing with the help of Vue CLI 3.

## Improving our tests in Vue CLI 3

There are several ways in which we can utilize Jest and other testing platforms in Vue CLI 3 for a better testing experience. In the next sections, we'll look at the following:

- Watching our unit tests in Vue CLI 3
- Setting parameters for our tasks
- Writing end-to-end tests with Cypress

Let's begin by working with the `--watch` flag.

## Watching our tests in Vue CLI 3

The `test:unit` command comes with a `--watch` flag. To see it in action, all we need to do is go back to the **Project tasks** page in Vue UI and, with the `test:unit` task selected, click on the **Parameters** button as follows:

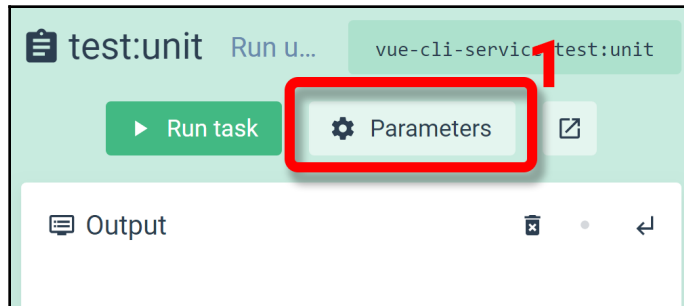


Figure 4.15: The **Parameters** button inside the `test:unit` task

Clicking the **Parameters** button will trigger a dialog with the following two options:

- **Watch files for changes and re-run tests related to changed files**
- **Re-record every snapshot that fails during this test run**

Click on the first option to turn on watch mode. The following option will appear right under it:

- **Display a notification after each run**

This option is available only when watch mode is enabled. Let's enable the **Display a notification after each run** option too, and click **Save**.

You can set these options inside scripts in `package.json`. The first option is `--watch`, and the display a notification option is the `--notify` flag.

To do this, simply update the `scripts` key inside your project's `package.json` to following code:

```
"scripts": {
  "serve": "vue-cli-service serve",
  "build": "vue-cli-service build",
  "lint": "vue-cli-service lint",
  "test:unit": "vue-cli-service test:unit",
  "test:unit-watch": "vue-cli-service test:unit --watch --notify"
},
```

You'll see that your task list has now expanded to include another testing task:

```
test:unit-watch.
```

However, even though you can, it's better not to do it like this. It's not a best practice and it kind of defeats the purpose of using Vue UI in the first place. Still, understanding that this is how it can be done gives us a better understanding of what is happening under the hood of the Vue CLI UI.

Now, let's see if watch mode is working properly by adding a change to the `AnotherComponent.spec.js` file. It's enough to just add a space somewhere and save the updates.

## Writing end-to-end tests with Cypress

End-to-end testing is a testing practice in which we test the application flow from beginning to end. With end-to-end testing, we simulate a scenario in which a user flows through our application from some kind of an entry point to a certain outcome.

For example, an end-to-end test of a web app could include the following flow:

- User opening the URL of the web app in their browser
- User clicking the login link and logging in
- User checking notifications in the web app
- User logging out

## Introducing Cypress

In this section, we'll be using Cypress for our end-to-end tests. Cypress works great with the Chrome browser. Alternatively, if you'd like to work with a Selenium-based tool, you could look at Nightwatch.js at this website: <http://nightwatchjs.org/>.

To find out more about Cypress, visit the official web site at the following URL: <https://www.cypress.io/>.

If you visit the Cypress website, you'll see it described as:

*Fast, easy, and reliable testing for anything that runs in a browser.*

Let's dive right in.



## Adding a Cypress plugin to our project

Now let's add a Cypress plugin to our project with your Vue CLI UI running:

1. Open the following address in the browser:  
`http://localhost:8000/plugins/add`
2. Next, type `cypress` into the search field, and locate the `@vue/cli-plugin-e2e-cypress` plugin.
3. Go through the plugin installation steps just like we did before with the Jest plugin.
4. Once our Cypress plugin is added, we need to commit the changes. Similar to how we did it with Jest, we can just commit with a simple message such as `Add Cypress plugin`.

Notice that installing Cypress added a new folder called `e2e` to our `tests` folder. Inside the `e2e` folder, we can find the following subfolders: `plugins`, `specs`, and `support`.

Let's inspect the contents of the `package.json` file next.

## Verifying updates to `package.json` made by the Cypress plugin installation

Let's inspect our project's `package.json` inside VS Code. We'll notice a new entry inside the `scripts` option as follows:

```
"test:e2e": "vue-cli-service test:e2e",
```

Also, our `devDependencies` have been updated with the Cypress plugin, which we can also see by visiting our Vue UI dashboard and inspecting the installed plugins.

Finally, if we click on the Tasks icon, we'll see that the `test:e2e` task has been added to the bottom of the tasks list for our project, exactly the same as what we saw inside the `package.json` file.

If we clicked on the `test:e2e` task, the right-hand pane will be updated accordingly, with the **Run task** button ready to be clicked. Clicking the **Run task** button would produce the following output:

```
$ vue-cli-service test:e2e --mode development
INFO Starting e2e tests...
INFO Starting development server...

DONE Compiled successfully in 1691ms18:06:27

App running at:
- Local: http://localhost:8082/
- Network: http://192.168.1.70:8082/

Note that the development build is not optimized.
To create a production build, run npm run build.

It looks like this is your first time using Cypress: 3.2.0

[18:06:28] Verifying Cypress can run
C:\Users\W\AppData\Local\Cypress\Cache\3.2.0\Cypress [started]

[18:06:30] Verified Cypress!
C:\Users\W\AppData\Local\Cypress\Cache\3.2.0\Cypress [title changed]

[18:06:30] Verified Cypress!
C:\Users\W\AppData\Local\Cypress\Cache\3.2.0\Cypress [completed]

Opening Cypress...
```

A new Electron-powered window will open on our computer. Working with Cypress is easy. As the **To help you get started...** window tells us, you can run tests in the `examples` folder or add your own test files to `cypress/integration`.

If you look at the top-right of the Cypress window, you'll see the **Run all specs** button. By default it will run in Chrome (with the version number specified). If you click on the dropdown that has **Chrome** selected, you can switch to **Electron**. Regardless of what option you choose, your tests will be run in a new window, either a new Chrome window or a new Electron window.

At this point, our end-to-end Cypress tests will fail, because Cypress is trying to run tests on a default Vue scaffold project, as follows:

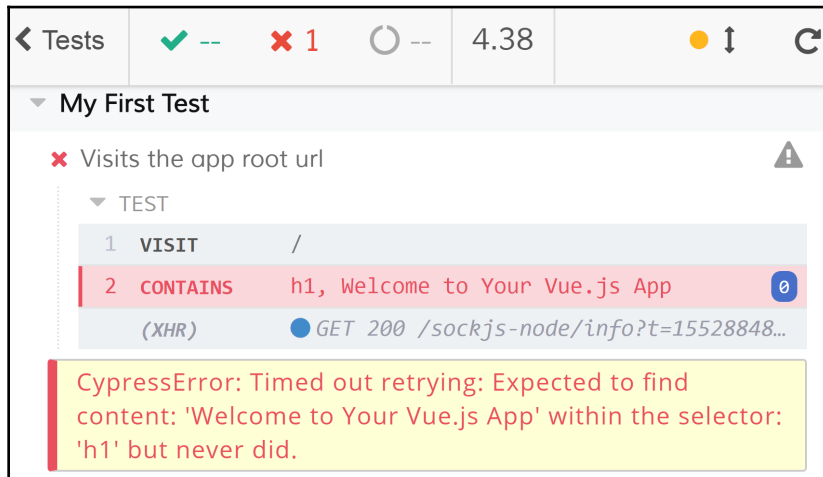


Figure 4.16: A failing test in Cypress

If you wanted to see these tests pass from the get go, you'd need to create a brand-new project and set up Cypress e2e testing in the configuration. We'll look at these different options later in the book. For now, let's update our tests so that they pass.

## Updating Cypress tests in our Vue app

Back in VS Code, open the `test.js` file inside the `./tests/e2e/specs/` folder. You'll see a `describe` function with two arguments.



To understand Cypress terminology better, refer to the following URLs: <https://docs.cypress.io/guides/core-concepts/writing-and-organizing-tests.html#Support-file>, and <https://docs.cypress.io/guides/references/bundled-tools.html#Mocha>.

While in Jest, we saw the word `test` as the name of a single unit test's function, whereas in Cypress, we see the word `it` being used. The `cy` object is the actual Cypress test runner. Let's look at the following updated code for the `test.js` file, and explain how and why it works:

```
describe('My First Test', () => {
  it('Visits the app root url', () => {
    cy.visit('/')
    cy.contains('article', 'Our own custom article component!')
  })
})
```

This time our tests are pass. Notice that we only had to save the updated test file in VS Code for the tests to be run automatically. You can navigate to <http://localhost:8000/tasks/> and click the `test:e2e` task for more information on the running task as follows:

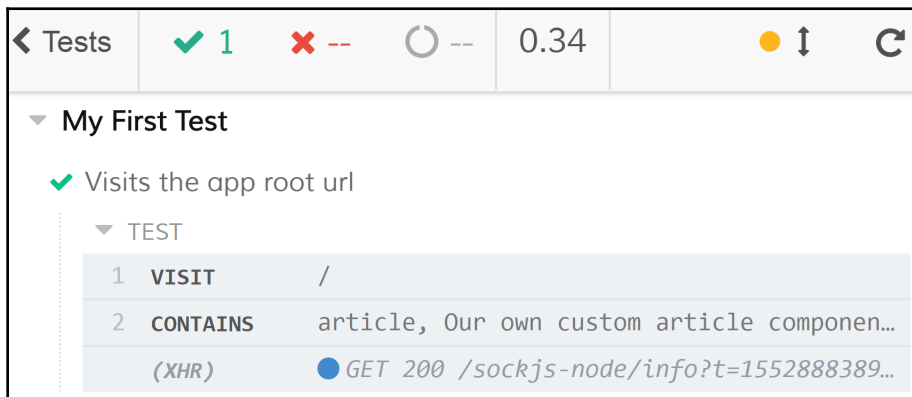


Figure 4.17: A screenshot of a passing Cypress test for our Vue app's homepage

Also notice that, if you hover over the **CONTAINS** section in your test results, the appropriate part of the served Vue app will be highlighted, a wonderful visual cue as to the exact section of our Vue app that is being tested.

## Summary

In this chapter, we discussed many testing-related topics through the lens of the Vue CLI 3. TDD is a great technique that should be utilized by teams and individuals alike, and Vue CLI 3, its UI, and Jest make a wonderful match for streamlining this process. Like any other aspect of development with Vue CLI 3, a lot of the piping is hidden away from us, which makes for a great coding experience.

In the next chapter, we will look at routing in Vue CLI 3.

# 5

## Vue CLI 3 and Routing

In the previous chapter, we saw how to work with some testing suites such as Jest and Cypress in Vue. In this chapter, we'll look at working with vue-router and use it with Vue CLI 3. We'll look at some practical tasks such as lazy-loading components. We'll look at using the Vue add command to add vue-router and why that's not the best strategy, and steps to take to mitigate it. Understanding routing in Vue is beneficial if you want to build larger and more complex apps. The following is a list of subjects covered in this chapter:

- Adding a new Vue project with vue-router and vuex
- Configuring preset options through VS Code's command line
- Understanding vue-router routes
- Using named routes
- Adding a dynamic route
- Navigating to a route from the methods option in a Vue instance
- Working with nested (child) routes
- Lazy-loading routes

We'll begin by adding a new project.

### Adding a new Vue project with vue-router and vuex

Let's begin by creating a new folder in our file system. Let's call this folder `vueclichapter5`.

1. Open the `vueclichapter5` folder from Windows Explorer, right-click anywhere on the whitespace inside the `vueclichapter5` folder, and click on the **Git Bash here** command.

2. Once Git Bash opens, type `code .` and press the *Enter* key. This will open a brand-new instance of VS Code, with no files or tabs inside other than the welcome tab.
3. Next, we'll be adding a new Vue project right from VS Code's integrated Terminal. To access this Terminal, click on the VS Code window (to get it in focus), and press the following shortcut key: *Ctrl + `*.



We've already mentioned the **Ctrl backtick shortcut** earlier in the book. As a reminder, the ``` character can be found when you press the key above the *Tab* key on your keyboard.

Using the *Ctrl + `* keyboard shortcut will open the Terminal inside VS Code.

4. Next, we'll run the `vue create .` command with a dot, as follows:

```
vue create .
```

Doing so will generate a Vue project in the existing folder, that is, it will not make a subfolder for our new Vue project, as follows:

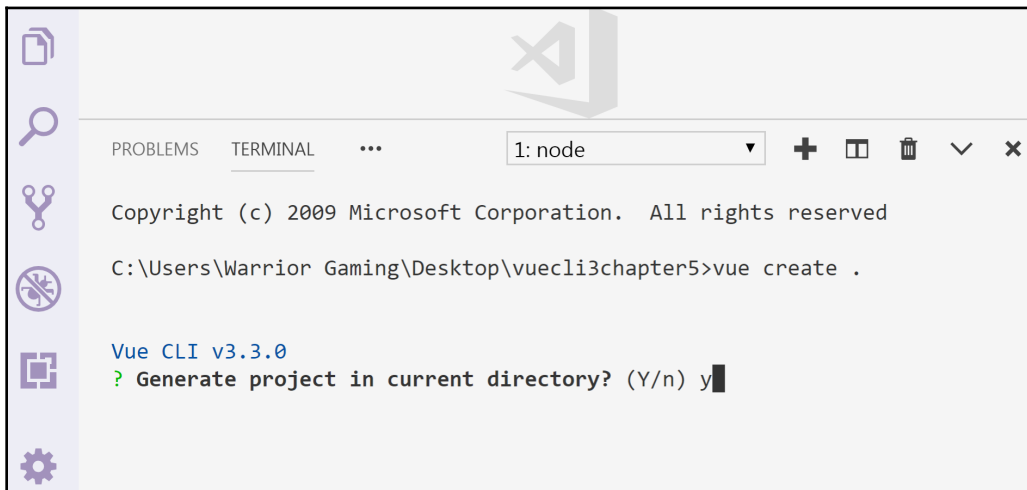


Figure 5.1: Generating a project in the current directory from VS Code's Terminal

5. Press the *Y* key to confirm that we'll be generating our project in the current directory.

## Configuring preset options through VS Code's command line

Next, we'll pick the **Manually select features** option by pressing the down arrow key once, followed by the *Enter* key, as shown in the following screenshot:

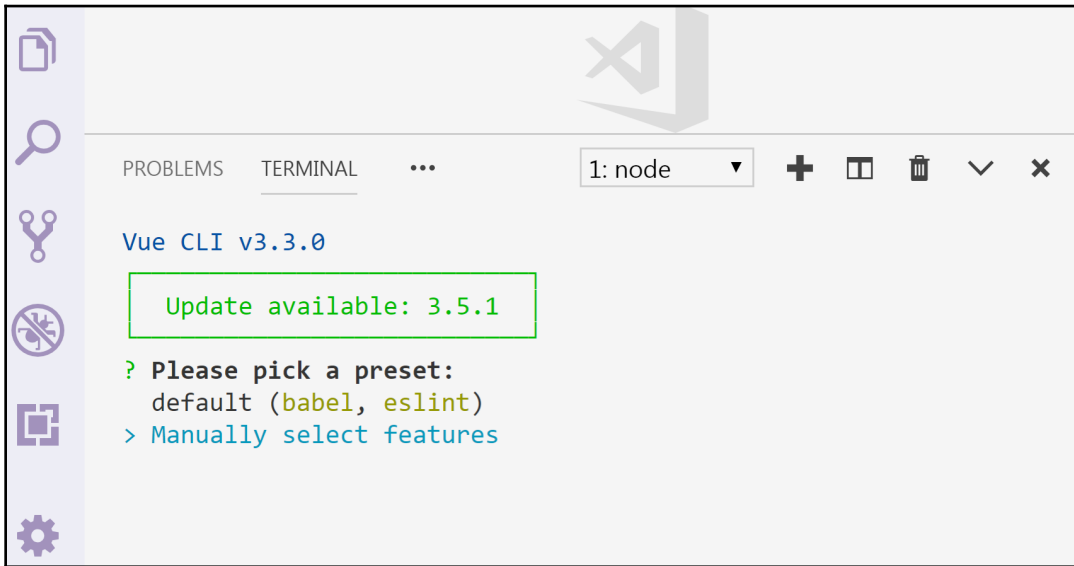


Figure 5.2: Generating a project in the current directory from VS Code's Terminal



## Adding vue-router and vuex

Next, we'll use the down arrow key and the spacebar key to select **Router** and **Vuex** as additional features for use in our project, as shown in the following screenshot:

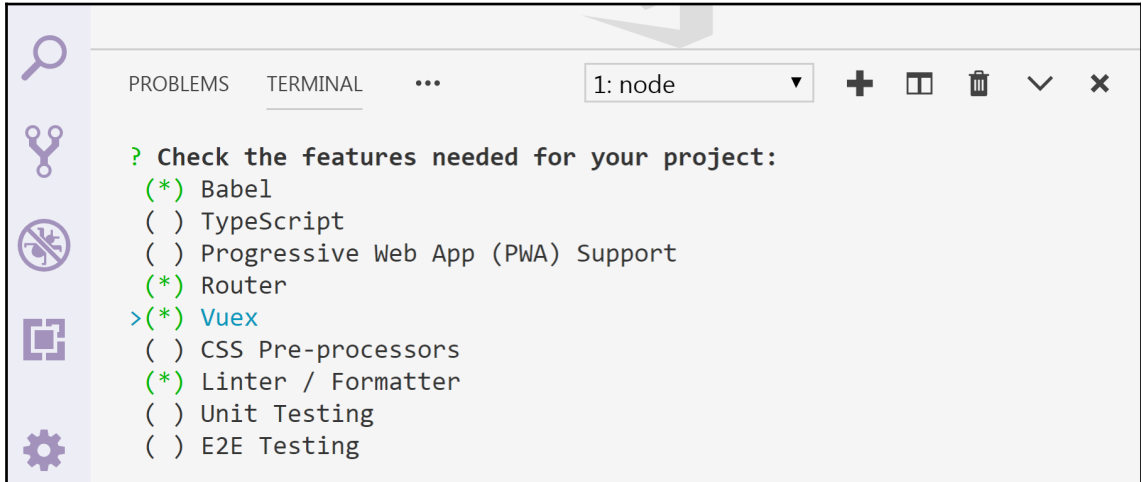


Figure 5.3: Adding the vue-router and vuex plugins to our project

A prompt will ask us if we want to use the History mode for the router.

We don't want to use it now, so we'll just type `n` and press *Enter*.

Later on in this chapter, we'll discuss what the History mode does and how it works.

## Adding ESLint and Prettier

Another prompt asks us about our linter or formatter preferences. We'll be using ESLint and Prettier as follows:

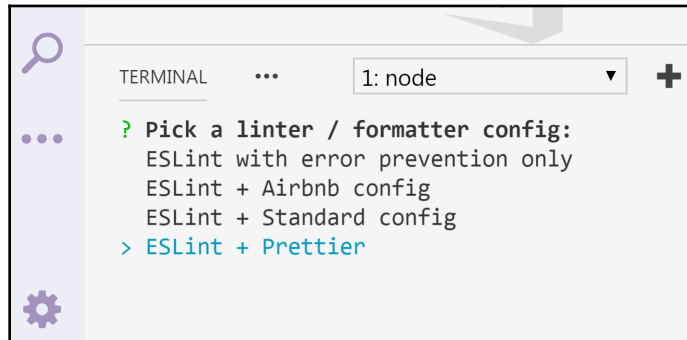


Figure 5.4: Choosing ESLint and Prettier as or linter / formatter configuration

## Finishing the configuration

Finally, we'll accept the default **Lint on save** feature, and opt to place the config for Babel, PostCSS, ESLint, and so on in dedicated config files as follows:

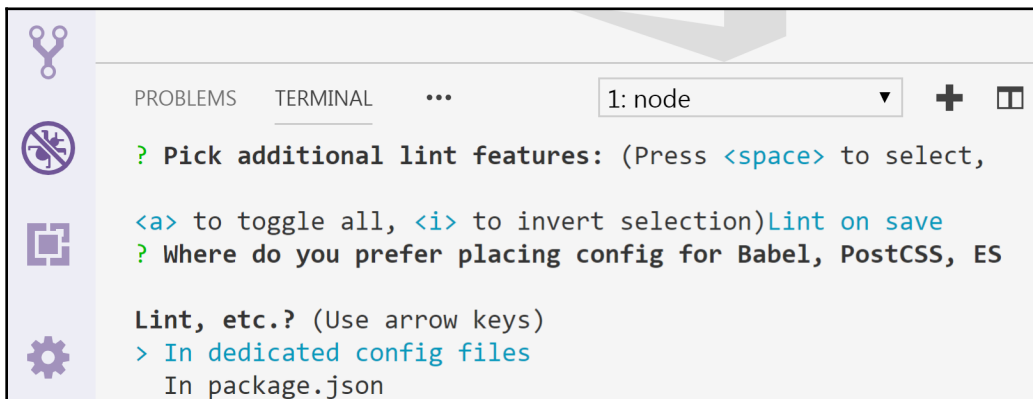


Figure 5.5: Choosing to save configs in dedicated config files

Finally the Vue CLI will ask us if we'd like to save this as a preset for future projects. We'll say no to this for the time being.

## Installing all the plugins for our new project

Finally, the Vue CLI installs all the plugins as follows:

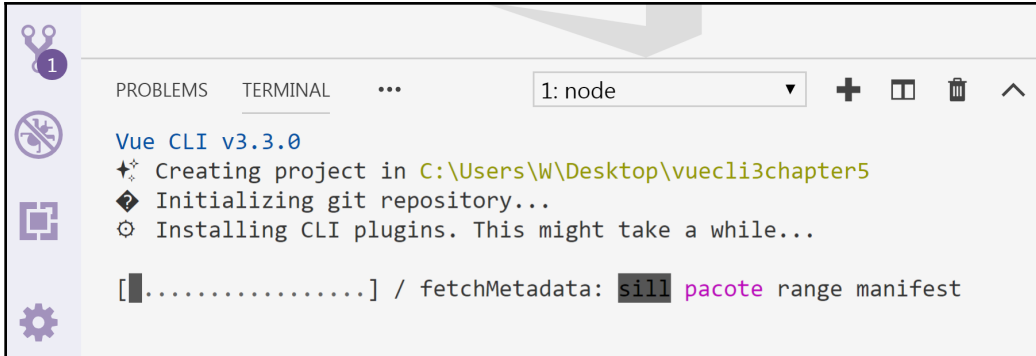


Figure 5.6: Vue CLI installing our project's plugins

Once this installation is complete, we'll have a working Vue project installed with vue-router and vuex, ESLint, and Prettier set up and ready to use.

## Serving our project from the Vue CLI UI

Once our project is installed, we could use the `npm run serve` command to run it. However, we'll use the Vue CLI UI for this, so let's run it with `vue ui` as follows:

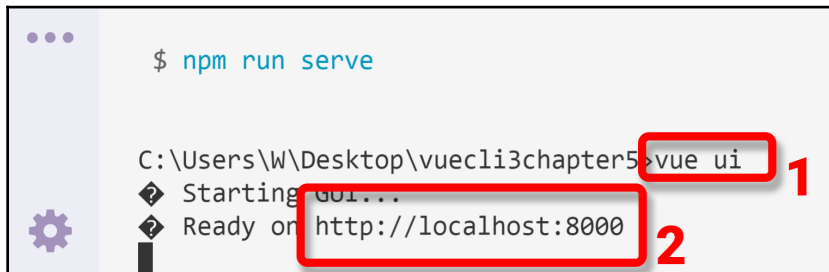


Figure 5.7: Serving our project via vue UI command from VS Code command line

As expected, our Vue CLI UI will automatically open in a browser window at `http://localhost:8000/dashboard`. At this point, the old project might be loaded in the dashboard, so we need to click on the Home icon and import our new `vueclichapter5` project.



Note that we could have installed a new app using the Vue CLI UI from the very beginning, but it is interesting to know that you can switch between the command line and the UI without issues.

Once the project is loaded, we can click on the **Plugins** link to see the installed plugins. Notice the absence of the **Add vue-router** and **Add vuex** buttons. They are not present since we've already installed them.

## Running the serve task from the UI

Finally, we'll click on the **Tasks** icon to open a list of available tasks, then click the **serve** task to compile and serve our project. Just like before, a **Run task** panel will appear on the right-hand side of the dashboard as follows:

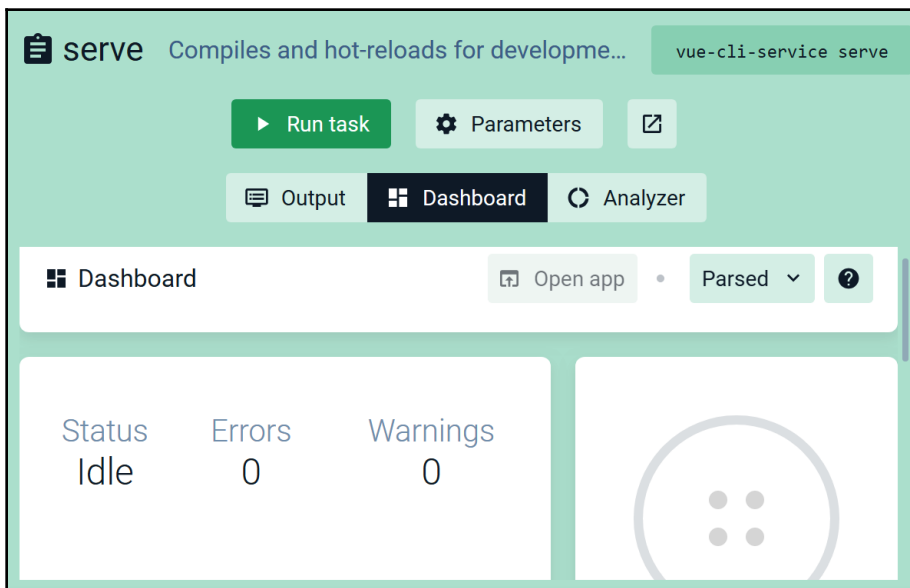


Figure 5.8: Run task for the npm run serve script

On the preceding screenshot, we can see a zoomed-in portion of the **serve** task screen. Clicking the **Run task** button will start building the app, and we'll get notified on the Windows Start bar that our app was successfully built. Looking at the available tabs, we can see that the current view shows our **serve** task's dashboard. Clicking on the **Output** button to the left of the currently active **serve** task's dashboard button will show us logging information, as follows:

```
App running at:  
- Local: http://localhost:8082/  
- Network: http://192.168.1.70:8082/
```

```
Note that the development build is not optimized.  
To create a production build, run npm run build.
```

Of course, the specific port that the `vue-cli-service` will be serving our Vue app from will depend on what other ports are already in use. To open our running Vue app in the browser, all that is needed now is to click on either the `Local` or the `Network` URLs listed in the output log.

## Working with routes in vue-router

In the section that follows, we'll be looking into how routes in `vue-router` work and how to add pages and routes using them. We'll begin by inspecting existing routes, and after that we'll be adding additional routes and their corresponding components.

## Inspecting the two default routes

This will open a familiar starting project with the Vue logo, with a slight addition: a link on top to the **About** page as shown in the following screenshot:

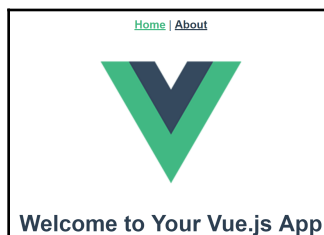


Figure 5.9: Default navigation of the project with `vue-router` installed

If you click on the **About** link, you'll be greeted with another page that has an `h1` tag and inside it, the following text:

```
This is an about page
```

Notice that the route to the **About** page has a pound sign in front of the page name. In our example app, it looks like this: `http://localhost:8082/#/about`. How do we get rid of the pound sign? In other words, how to we set the address to `http://localhost:8082/about`?



The answer is simple, we just need to set up our server to always return the `index.html` page, and then we're going to add another setting, the `mode` setting, for the `vue-router`, such as the following: `mode: 'history'`

You need to add the preceding code right under the `export default new Router({` line in `router.js`. Doing this will get rid of the pound symbol.

Next, we'll inspect the contents of the project that comes preinstalled with `vue-router` and `vuex`.

## Inspecting project files

Go back to VS Code and look at the project structure. You'll see some differences from what we had before. The following is the contents of our `src` folder:

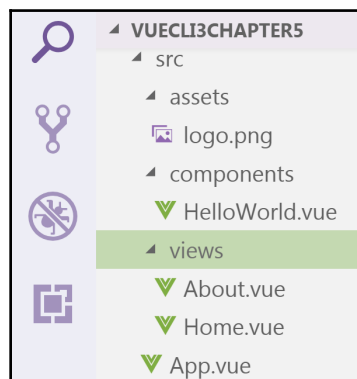


Figure 5.10: The contents of the `src` folder with `vue-router` and `vuex` pre-installed

Inside the `src` folder, we can see a folder that we haven't seen before: the `views` folder, which holds two view files: `Home.vue`, and `About.vue`. In the root of our project, we can see some additional files as well: `router.js`, `store.js`, `.browserslistrc`, `eslint.config.js`, `postcss.config.js`, and `README.md`.

The `router.js` file is what `vue-router` uses to set up the paths in our app. The `routes` array is an array of objects: one object for a single route. Since we have two routes in the default installation, we have two objects inside the `routes` array. The `store.js` file is what `vuex` uses to keep track of our app's state, mutations, and actions; these are all referred to as the **vuex store**. The `README.md` file lists common `npm` commands for use with our app, and the other files at the root of our project are configuration files; this convention, at this point in our journey through Vue CLI 3, should be expected.

## The contents of `main.js`

Just like we've seen before, the `main.js` file inside our `src` folder imports all the following dependencies:

- The Vue library from `node_modules`
- The root component, `App.vue`
- The router file to set up `vue-router` routes
- The store file to set up the `vuex` store

## The `App.vue` file and `router-link` navigation

By inspecting the contents of the root component, `App.vue`, we can see that the file looks different from before: there is no `script` section!

Also, the `template` tag inside `App.vue` holds the so-called *navigation components*. Each navigation item is enclosed inside the `router-link` tag.

Why not use just an `anchor` tag instead of the `router-link` tag? Because an `anchor` tag would go on and send a server request. With the `router-link` tag, this behavior is avoided.

If we inspect the compiled HTML code in the `dist` folder, we'll see that the bundled HTML indeed ends up being an `anchor` tag, as can be seen in the following compiled production code:

```
<div id="nav">
  <a href="#" class="router-link-active">Home</a> |
```

```
    <a href="#/about" class="router-link-exact-active router-link-  
    active">About</a>  
</div>
```

Going back to the `Home.vue` and `About.vue` files inside the `view` folder, we can also see the `router-view` tag (below the `div` with the `id` of `nav`). This is what actually renders either the `Home` view or the `About` view. The component that will be rendered will be determined by whatever is mapped with the `to` prop.

## Rendering a Vue instance and mounting it into #app

Finally, a new instance of `Vue` is called and passed an argument in the form of an options object. The object receives the following key-value pairs: `router: router`, `store: store`, and `render: h => h(App)`.

Written in ES5 code, that code would look like the following:

```
new Vue({  
  router: router,  
  store: store,  
  render: function render(h) {  
    return h(App);  
  }  
}).$mount("#app");
```

Our `main.js` file, is luckily, written in a more modern syntax, so when all these things are put together, it looks as follows:

```
import Vue from "vue";  
import App from "./App.vue";  
import router from "./router";  
import store from "./store";  
  
Vue.config.productionTip = false;  
  
new Vue({  
  router,  
  store,  
  render: h => h(App)  
}).$mount("#app");
```

The `render` method will take all the different parts of our app, combine them, and get them ready to be mounted. Finally, our `Vue` app will be mounted into the `div` with the `id` of the `app` inside our `./public/index.html`.



You might notice that our current app's directory doesn't contain a `dist` folder. As we've learned in [Chapter 2, Webpack in Vue CLI 3](#), the `dist` folder is the product of webpack building our site. So let's run the build task in our Vue CLI 3 UI next, to see the `dist` folder being created and served.

## Running the build task from the UI

To run the build task from the Vue CLI 3 UI, all we need to do is go to `http://localhost:8000/tasks/`, and from there click on **Build task**, and then click the **Run task** button.

The output tab of the build task we ran will log out the following information:

```
File Size Gzipped

dist\js\chunk-vendors.1030118d.js 116.48 KiB 40.51 KiB
dist\js\app.51b1d496.js 5.97 KiB 2.23 KiB
dist\js\about.d288b4f1.js 0.44 KiB 0.31 KiB
dist\css\app.08e7a232.css 0.42 KiB 0.26 KiB

Images and other types of assets omitted.

DONE Build complete. The dist directory is ready to be deployed.
INFO Check out deployment instructions at
https://cli.vuejs.org/guide/deployment.html
```

What are these different JavaScript files inside the `./dist/js/` folder? They are the result of webpack bundling our assets from our Vue app's single-file components, the router, and the store, into deployment-ready bundles. These bundles are now added to the compiled and minified `index.html` page inside our `dist` folder, so this is how they end up on our production-ready web page.

Finally, let's see our updated app. For that, we'll click on the **serve** task, and stop the task by clicking the **Stop task** button.

## Serving the app in production mode from the UI

To serve the app in production mode, we need to click on the **Parameters** button inside the **Run task** panel with the **serve** task selected.

Once we click the **Parameters** button, we'll see the following dialog:

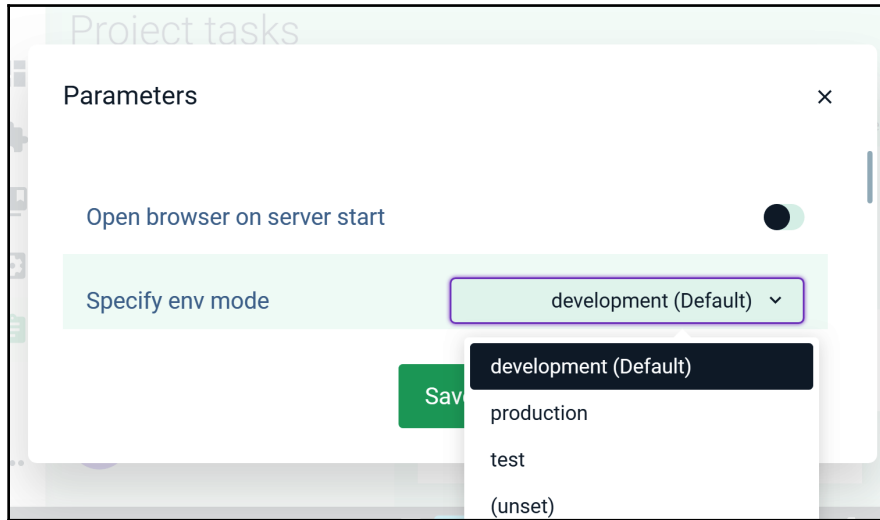


Figure 5.11: Specifying the env mode in serve task's parameters dialog inside the Vue CLI 3 UI

In the **Specify env mode** setting, click on the drop-down and select **production**. Save the changes, and you'll see the **serve** task dashboard again. Now, click on the **Run task** button. Finally, to make sure everything still works, point your browser to whatever port your Vue app is being served on. In this example, the proper address is `http://localhost:8082/`. Next, click on the **About** link.

Back in VS Code, change the `About.vue` page's `h1` tag to the following:

```
<h1>This is an about page. HMR rocks!</h1>
```

Save the file and see your **About** page get an HMR update.

## Some basics of working with routes

Although this book is about Vue CLI 3, we will take this opportunity to quickly list some features of `vue-router`. This is by no means an extensive list: it is just a quick overview of certain features you need to be aware:

- Using named routes
- Adding a dynamic route

- Navigating to a route from the methods option in a Vue instance
- Working with nested (child) routes
- Lazy-loading routes

We'll begin by understanding that the `router-view` tag needs to be nested.

## The router-view tag needs to be nested

Before we begin, let's look at the `App.vue` template tag as follows:

```
<template>
  <div id="app">
    <div id="nav">
      <router-link to="/">Home</router-link> |
      <router-link :to="{ name: 'about' }">About</router-link> |
      <router-link :to="{ name: 'cars' }">Cars</router-link>
    </div>
    <router-view />
  </div>
</template>
```

Notice that the `router-view` element (the third line from the bottom), is wrapped inside a parent `div` tag, which is then wrapped inside a `template` tag. This is important, because you can't have the `router-view` tag as the direct child of the `template` tag. In other words, the following is not possible:

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

Now that we know that this approach will not work, we can go on discussing named routes.

## Using named routes

To use a named route, you just pass an object to the `to` prop inside the `router-link` tag, as follows:

```
<router-link :to="{ name: 'about' }">About</router-link>
```

Notice the `:` before the `to` prop. This `:` is added whenever we want to add a named route.

With the `name` key inside the object we passed in to the `to` prop, `vue-router` will check the `routes` array inside `router.js`. It will look for an object that has a `name` key with the value specified, and if it finds it, it will re-route it to the specific view component.

## Navigating to a route by triggering a method of a Vue instance

Every component in Vue is a stand-alone Vue instance. We'll be working in the `HelloWorld.vue` component. To navigate to a route from the `methods` option of `HelloWorld.vue`, you can use the following code inside the `scripts` section of the component:

```
methods: {
  goToAboutPage () {
    this.$router.push({path: '/About'})
  },
}
```

To make this work, we need to add a `v-*` directive to determine when a method that has the preceding code will be triggered. Thus, still inside `HelloWorld.vue`, update the `template` tag to the following:

```
<template>
  <!-- code skipped for brevity -->
  <p v-on:mouseover="goToAboutPage"
    style="color: red; font-size: 50px; background: gray; max-width: 500px;
margin: 0 auto">
    Hover to see the About Page
  </p>
  <!-- code skipped for brevity -->
</template>
```

Obviously, the preceding `template` tag is incomplete. We're focusing on the important part: the `p` tag with the text colored red and the font size of 50 pixels. You can't miss it on the screen! You can find this line of code inside `vuecli3chapter5/HelloWorld.vue`, starting on line 4.

When you hover over the preceding paragraph, Vue will immediately take you to the **About** page.

Let's now add the same functionality we have on the home page to the about page. Thus, once you are on the about page, you can hover over the prominent **Hover to see the Home Page** link, which will take you back to the home page.

To make things less jumpy, we can also use the good-old plain, browser facade function: `setTimeout`. Here's the updated code for the `About.vue` file:

```
<template>
  <div class="about">
    <h1>This is an about page. HMR rocks!</h1>
    <p v-on:mouseover="goToHomePageSlowly" class="go">Hover to see the Home
Page</p>
  </div>
</template>
<script>
export default {
  name: "About",
  props: {
    msg: String
  },
  methods: {
    goToHomePage() {
      this.$router.push({path: '/'})
    },
    goToHomePageSlowly() {
      setTimeout(() => this.goToHomePage(), 1000);
    }
  }
};
</script>
<style>
.go {
  color: purple;
  font-size: 50px;
  cursor: pointer;
  max-width: 500px;
  margin: 0 auto;
  background: #fabdab;
}
</style>
```

What are we doing in the preceding code? We're simply adding another method to our `methods` option. We're naming this new method: `goToHomePageSlowly`. Then, we're calling this method inside our `about` component's template tag.

The `goToHomePageSlowly` uses the `setTimeout` facade function for the browser's `setTimeout` functionality. The `setTimeout` function calls our `goToHomePage` function with a delay of 1000 milliseconds, and once it's run, it will use the `$router` to go back to the home page.

Next, we'll discuss nested routes.

## Working with nested routes

Before we start working with nested routes, we will add the `Cars` component, inside the `src/views` folder as follows:

```
<template>
  <div class="home">
    <HelloCars msg="Welcome to Your Cars" />
    <router-view></router-view>
  </div>
</template>

<script>
// @ is an alias to /src
import HelloCars from "@/components/HelloCars.vue";

export default {
  name: "cars",
  components: {
    HelloCars
  }
};
</script>
```

To work with nested routes, you need to add a `children` array to the `path` object inside `router.js`, as in the following example:

```
routes: [
  {...},
  {...},
  {
    path: "/cars",
    name: "cars",
    component: Cars,
```

```
    children: [
      { path: '/cars', component: CarsHome },
      { path: '/cars/cars-new', component: CarsNew },
      { path: '/cars/cars-used', component: CarsOld }
    ]
  }
]
```

Each of these child components also needs routing. Since they are nested routes, they need to be called from their respective parent component. In our case, this is the `HelloCars` component. We will add the `HelloCars.vue` component inside the `components` folder, using the following code:

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
  </div>
</template>

<script>
export default {
  name: "HelloCars",
  props: {
    msg: String
  }
};
</script>

<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>
* {
  margin: 0 auto;
}
</style>
```

Now all we have to do is add additional child components that our parent `HelloCars.vue` component will call.

The default child component, available from the `/cars` route, is the `CarsHome.vue` component as follows:

```
<template>
  <div class="home">
    <div>
      This is Cars home
    <ul>
      <li><router-link to="/cars/cars-new">See new cars</router-
```

```
link></li>
      <li><router-link to="/cars/cars-used">See old cars</router-
link></li>
    </ul>
  </div>
</div>
</template>

<style>
ul li { list-style-type: none }
</style>
```

As mentioned earlier, one of the `HelloCars.vue` component's children is the `CarsNew.vue` component as follows:

```
<template>
  <div>
    <div>
      This is Cars New
      <ul>
        <li><router-link to="/cars">Back to cars home</router-
link></li>
      </ul>
    </div>
  </div>
</template>
```

Another child of the `HelloCars.vue` component is `CarsOld.vue` as follows:

```
<template>
  <div>
    <div>
      This is Cars Old
      <ul>
        <li><router-link to="/cars">Back to cars home</router-
link></li>
      </ul>
    </div>
  </div>
</template>
```

Now that we understand how nested routing works, we'll shift our focus and briefly discuss lazy-loading routes.



## Lazy- loading routes

Sometimes, webpack produces JavaScript bundles that are too large. This will slow down the time it takes to load our web apps, which is, of course, undesirable.

To avoid this scenario, we can think back to how vue-router works: each route is a separate Vue component.

We've already seen earlier that webpack produces **chunks** when bundling our Vue apps. This behavior can be used to our advantage, so that each component gets bundled into a separate chunk. This is achieved with Vue's **async components** and **code-splitting** in webpack.

## Working with dynamic routes

What is a dynamic route? Let's imagine a shopping platform, in which the URL structure of any item for sale is listed in a link that looks like this:

```
https://example.com/items/:id
```

The `:id` section is what's called a **dynamic segment**. To work with dynamic segments, you first need to list them, just like any other route, inside `router.js` in the route's array of objects as follows:

```
routes: [
  {...},
  {...},
  {
    path: "/item/:id",
    name: "item",
    component: Item
  }
]
```

Obviously, in the preceding code, we're using the three dots to save space.

Back inside the `Item.vue` view component's template tag, we'll need to add the following syntax:

```
<template>
  <article>
    <h1>Shopping item: {{ $route.params.id }}</h1>
  </article>
</template>
```

The state of our active route is stored inside `$route`.

At least an entire chapter could be written about dynamic routes alone, so at this time, we'll need to continue our discussion by looking at triggering routes from the `methods` option in Vue instances.

With this, we conclude this chapter.

## Summary

In this chapter, we looked at how to add a new Vue project with `vue-router` and `vuex`. We also configured preset options through VS Code's command line. We discussed `vue-router` routes, and we saw how to use named routes, add a dynamic route, navigate to a route from the `methods` option in a Vue instance, and work with nested (child) routes.

In the next chapter, we'll see how to work with ESLint and Prettier in Vue CLI 3.

# 6

## Using ESLint and Prettier in Vue CLI 3

In this chapter, we'll learn why ESLint is useful and we'll see how to lint our code with it. We'll also discuss different style guides: standard, Airbnb, and Google. Specifically, we'll look at the following subjects:

- What ESLint is, and how to configure it on its own
- The difference between installing npm packages locally and globally
- Installing ESLint locally and running it with npx
- Using ESLint in Vue CLI 3
- Configuring ESLint using Vue CLI 3 UI

We'll start by looking at what ESLint is and how to configure it as a standalone npm package.

### **What ESLint is, and how to configure it on its own**

In this chapter, we will discuss the role of ESLint. ESLint has to do with code quality. When you work on a team, every team member will bring their own little quirks and ideas on the best way to write JavaScript. Even if your team has some clear-cut rules on specific coding styles and how to write your code, how do you guarantee that each team member will follow the rules? How do you make sure that they deliver consistent code quality? We are human after all, which means that we all have those days when we do our best, but simply forget about one little thing that we had to pay attention to.

Given a long enough period of time, our team will begin producing stylistically inconsistent code. The next thing that happens is, everyone in the team starts perceiving the JavaScript style guide as *a suggestion*, not something that you should *really adhere to*.

Luckily, there are tools that allow us to automatically improve the look of our code and to format it when it doesn't follow the prescribed way of doing things. ESLint is an example of such a tool.

Basically, ESLint is a way for you to outsource the *tabs versus spaces* and other similar debates to a piece of software that will take care of these issues in a streamlined, consistent manner.

In this chapter, we'll also briefly mention the difference between global and local node modules, that is, the difference between installing `npm` packages globally and locally.

## Installing npm packages globally versus installing npm packages locally

Here's an example of a global installation:

```
npm install eslint -g --verbose
```

Here's an example of a local installation:

```
npm install eslint --save-dev --verbose
```

So, what's the difference?

Global packages are installed in one specific place on your computer. It does not matter what folder you were in when you installed them; *they are not saved inside that folder*, and they are made available globally in your command line. Thus, if we want to run a package from the command line, global installation is one way to do it.

You run the `npm install` command from a command-line program. A command-line program will always run from inside a directory. Using commands such as `cd ..` to go up a directory in you OS file system, or `cd <directory-name>` to drill down into a directory, you can choose what folder you want your command-line program to be in, when installing an `npm` package.

To install an `npm` package locally, means that the package is saved inside the current directory your command-line program points to. When the `npm install` command is run, the new package you're installing will be saved inside the `node_modules` directory of the current directory you're in. If the `node_modules` directory doesn't exist, it will be added when the `npm install` command is run, *inside the current directory*.

The downside of using global modules is the versioning problem. With local modules, each different directory and each different application on your machine has its own version of a module. However, if you have a global package and you decide to update it, the updated version of your globally installed module might break your application because of compatibility issues between that new version and the rest of the code in your specific app in a specific directory.

As a rule of thumb, the `npm` packages should be installed locally, since you avoid the versioning problem.



To learn more about installing `npm` packages locally, visit this web page: <https://docs.npmjs.com/downloading-and-installing-packages-locally>.

One downside of using locally installed packages is that the specific command, such as `prettier`, `webpack`, or `eslint`, doesn't become available on the command line. Instead, you need to run a locally-installed node module as an `npm` script.

Alternatively, this locally-installed module can actually be run on the command line directly, using the `npx` command, which we'll discuss in the next section.

## What is Prettier?

Prettier is a code formatter. It formats your code whenever you save a file. The project website can be found at <https://prettier.io/>. In the example that follows, we'll be running a locally-installed Prettier `npm` module.

## Running locally-installed `npm` modules on the command line using `npx`

To begin working with Prettier, we'll start a new project as follows:

```
mkdir Chapter6 && cd $_
```

Next, inside the new Chapter6 folder, we'll add another folder, prettier-practice, as follows:

```
mkdir prettier-practice && cd $_
```

Next, we'll initialize npm with defaults as follows:

```
npm init -y
```

Next, let's add one single file, index.html, as follows:

```
touch index.html && code .
```

The preceding command created and opened a new index.html file inside VS Code.

We'll be adding a small Vue app without any build process. Let's add the following code to our index.html file:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <title>101 - using watchers in Vue</title>
  <style>
    body,
    input {
      font-family: Arial, sans-serif;
      font-size: 20px;
    }
  </style>
</head>

<body>
  <div id="example">
    <p>Enter owner name and the thing that is owned:
      <input v-model="ownerName" placeholder="enter owner">
      <input v-model="thing" placeholder="enter thing">
    </p>
    <span>{{ ownerName }}</span>
    <span> has a </span>
    <span>{{ thing }}</span>
  </div>
  <script
src='https://cdnjs.cloudflare.com/ajax/libs/vue/2.6.10/vue.min.js'></script
>

  <script>
    var example = new Vue({
```

```
    el: '#example',
    data: {
      ownerName: 'e.g Old McDonald',
      thing: 'e.g cow'
    },
    watch: {
      ownerName(currentValue, previousValue) {
        console.log(`The value in the first input has changed from:
        ${previousValue} to: ${currentValue}`);
      }
    },
    computed: {
      // a computed getter
      ownerHasThing: function () {
        // `this` points to the Vue instance's data option
        return this.ownerName + " " + this.thing
      }
    }
  })
</script>
</body>

</html>
```

You can see this simple Vue app running on CodePen, at this URL: <https://codepen.io/AjdinImsirovic/pen/jprwKe>.

Next, let's add Prettier to see how it will format our code.

## Installing Prettier globally and running it on index.html

To install Prettier globally, we can use the following command:

```
npm install prettier -g --loglevel verbose
```

Then, to start using it, you can just run the `--help` flag to see all the options that are available, such as the following:

```
prettier --help
```

Now, you can run the `prettier index.html` command to see the formatted code right in your console. Alternatively, you can run the command `prettier index.html --write`, so that the formatted output actually gets saved in the file you specified.

Next, we'll see how to install Prettier locally and run it with `npx`.

## Running Prettier with npx

The `npx` utility is an **NPM package runner**. It was added to NPM in version 5.2.0 (in the first half of 2017). It makes it easier to run packages from the `node_modules` directory, since we don't have to rely on `npm run <someScriptName>` as we did in `package.json` before.

In this example, we'll see how to install Prettier as a development dependency, and how to run it with `npx`. First, we need to install it with the `--save-dev` flag inside our `prettier-practice` folder as follows:

```
npm install prettier --save-dev --verbose
```

Once it's completed, our `package.json` will be updated, and we can start using Prettier.

The easiest way to do so, is to simply run just the Prettier command inside VS Code's Terminal.

To see the available options, we can run the following command:

```
npx prettier
```

To actually run a command with some options passed, we can run this code:

```
npx prettier --single-quote --trailing-coma es5 --write "index.html" --verbose
```

There are many other ways in which `npx` can be used, and it's a great tool to familiarize yourself with.

Another interesting exercise to try is to compare Prettier's formatting with the readily available VS Code's formatter. The VS Code's formatter is triggered using the `Shift + Alt + F` keyboard combination. We can quickly re-run the `npx prettier` command we just ran, by pressing the up arrow while inside VS Code's console.

Next, we'll make another little project to demonstrate working with ESLint.

## Working with ESLint

ESLint is a JavaScript linter. You can find it at <https://eslint.org/>. A linter performs code analysis. It compares your code against a certain standard. You could say it's a tool to check code quality. Linters are great for team environments, because they make our code base follow a certain code style, and because they ensure that, there's one less thing to talk about during code reviews.



An example of a code style can be found here: <https://github.com/airbnb/javascript>. This is Airbnb's JavaScript style guide. If you scroll down the page, you'll see that there is a lot of information to read through: how to access primitive types, how to reassign references, how to group shorthand properties in object declarations, and so on.

It's important to understand that ESLint only analyzes your code; it doesn't run it. However, it does locate errors, and can also fix them. Effectively, it can change your code if you set it up to do so.

There are other JavaScript linting tools out there, such as JSLint, or JSHint. ESLint is touted as more customizable.

## Installing ESLint globally and unsuccessfully running it from the command line

To install ESLint node module globally, run the following command:

```
npm install eslint -g --loglevel verbose
```

Now, our `eslint` command is globally available.

Now, we'll create a new folder, `eslint-practice`. Inside of it, we'll make a new file, `eslint-test.js`.

Open the file in VS Code and add this code to it:

```
function (a)
{
  console.log(a, b);
}
```

Save the file.

Next, open the command line inside VS Code with `Ctrl + ``, and run the following command:

```
eslint eslint-test.js
```

The output you'll receive is this:

```
Oops! Something went wrong! :(

ESLint: 5.15.3.
ESLint couldn't find a configuration file. To set up a configuration file
for this project, please run:

    eslint --init

ESLint looked for configuration files in C:\Users\Warrior
Gaming\Desktop\Chapter6 and its ancestors. If it found none, it then looked
in your home directory.
```

Why did this happen? There is no ESLint config file, but why does ESLint need a config file to run?

This is because the rules that ESLint will use to check our code are specified in the **config** file. If there's no config file, ESLint doesn't know which rules to apply when linting our code.

The solution for this problem is interesting. Without a `package.json`, we cannot initialize ESLint. Since we have to install a new NPM project, we might as well just install the ESLint node module locally. It's not a must, but it's a preferred option for a streamlined workflow.

## Running `eslint --init` command

Obviously, we can't run the `eslint` command without an ESLint config file. However, we can't initialize an ESLint config file without a `package.json` file. This means we first need to run `npm init`, so let's do that now as follows:

```
npm init -y
```

Now, we can initialize `eslint`. When we run the `eslint --init` command, we'll be asked several questions in command-line prompts. Our ESLint configuration will depend on the answers we select. Here are the selections we need to pick for this example:

- Pick `To check syntax, find problems, and enforce code style`
- Pick `None of these`
- Pick `None of these` (for this simple demonstration we won't use `Vue.js` because that would require installing additional dependencies, and at this point we are only test-driving ESLint)

- Pick Browser
- Pick Answer questions about your style
- Pick JSON
- Pick Spaces
- Pick Single
- Pick Windows
- Type Y (yes, we do require semicolons)
- Pick "JavaScript" (as the answer to the **What format do you want your config file to be in?**)

Once you've answered all the questions, you'll be greeted with the following output:

```
Local ESLint installation not found.
The config that you've selected requires the following dependencies:
eslint@latest
Successfully created .eslintrc.js file in
C:\Users\W\Desktop\Chapter6\eslint-practice
ESLint was installed locally. We recommend using this local copy instead of
your globally-installed copy.
```

Reading this message, we can now appreciate that the maintainers of ESLint are nudging us toward the best practice of limiting the number of globally installed npm packages on our computers, and, instead choosing to install our node modules locally in our projects. The preceding message also reads that we need the `eslint@latest` dependency to be able to run our ESLint configuration, so let's add it right now, as follows:

```
npm install eslint@latest --save-dev --verbose
```



It has been mentioned earlier in the book, but this is a great time for a refresher: the `--save-dev` flag means that the package we are installing is only to be used in development, not in production. The `--save-dev` option does not set the local module. You can install ESLint locally by simply running: `npm install eslint@latest`. However, both the `--save-dev` and `--verbose` flags have their place, but it needs to be clear that they don't have anything to do with locally installing an npm package.

Now, we are ready to run ESLint on our project's JavaScript file. Before we do, let's inspect the contents of the `.eslintrc.js`.

## Understanding the structure of .eslintrc.js config file

Based on the rules we picked, our generated `.eslintrc.js` file looks like this:

```
module.exports = {
  'env': {
    'browser': true,
    'es6': true
  },
  'extends': 'eslint:recommended',
  'globals': {
    'Atomics': 'readonly',
    'SharedArrayBuffer': 'readonly'
  },
  'parserOptions': {
    'ecmaVersion': 2018
  },
  'rules': {
    'indent': [
      'error',
      4
    ],
    'linebreak-style': [
      'error',
      'windows'
    ],
    'quotes': [
      'error',
      'single'
    ],
    'semi': [
      'error',
      'always'
    ]
  }
};
```

Inside the exported object in the config file, we can see that the environment is set to `browser`, and that the `es6` property is set to `true`, which means the ES6 syntax is enabled automatically.

With `'extends': 'eslint:recommended'`, we turn on the recommended core rules that report common problems. These rules are listed on the following page: <https://eslint.org/docs/rules/>.

The `'globals'` entry lists additional global variables that will be accessed when ESLint is run using the specified configuration.

With the `parserOptions` property, we set the ECMA Script version to use as 2018, which is, of course, ECMA Script 9. Finally, the `'rules'` property specifies the rule to be applied, and at what **error level**.

## Running ESLint on a JavaScript file

Now, we can run the `eslint` command on our `eslint-test.js` file. Since it is installed globally, we could simply run `eslint eslint-test.js`.

However, since this is discouraged by the very maintainers of the module, let's instead run it locally, with the following command:

```
npx eslint eslint-test.js
```

Running the preceding command will produce the following output:

```
1:10 error Parsing error: Unexpected token (  
  
✖ 1 problem (1 error, 0 warnings)
```

We can also see the error right inside VS Code, as a single character, the `(` character, underlined with a wavy red color on line one of our `eslint-test.js` file. Of course, this error means that we are missing our function's name. So let's update the code to this:

```
function aFunction(a)  
{  
  console.log(a, b)  
}
```

Notice that we've also removed the semi-colon after the call to `console.log` method. Let's run `eslint` again.

This time we get five new errors showing as follows:

```
1:10 error 'aFunction' is defined but never used no-unused-vars  
3:1 error Expected indentation of 4 spaces but found 0 indent  
3:1 error Unexpected console statement no-console  
3:16 error 'b' is not defined no-undef  
3:18 error Missing semicolon semi  
  
✖ 5 problems (5 errors, 0 warnings)  
  2 errors and 0 warnings potentially fixable with the `--fix` option.
```

Looking at the end of each line in the preceding output, we can see the specific **ESLint rules** that our code violates. At the beginning of each line, we can see the line number, followed by a colon, followed by the exact place of the first character that breaks an ESLint rule. Thus, 1:10 can be read as *line 1, character in the 10th position*.

Let's run `eslint` again, with the suggested `--fix` flag as follows:

```
npx eslint eslint-test.js --fix
```

Now our code gets formatted, as follows:

```
function aFunction(a)
{
  console.log(a, b);
}
```

However, we still get some errors, as follows:

```
1:10 error 'aFunction' is defined but never used no-unused-vars
3:5 error Unexpected console statement no-console
3:20 error 'b' is not defined no-undef
```

```
✖ 3 problems (3 errors, 0 warnings)
```

What we can conclude from this little exercise, is that ESLint will do the following:

- Point out rule violations in our code.
- Allow us to pass the `--fix` flag to remedy those errors that are fixable by the tool itself.

Next, we'll update the error level rules.

## Updating error level rules in ESLint

By default, all rules have their error levels set to `'error'`. To manually set a different error level, such as `'warn'`, we might add an `errorLevel` constant and update our rules to this code:

```
const errorLevel1 = 'warn';
const errorLevel2 = 'error';
module.exports = {
  // ...
  // omitted this section to save space
  // ...
}
```

```
'rules': {
  'indent': [
    'error',
    4
  ],
  'linebreak-style': [
    'error',
    'windows'
  ],
  'quotes': [
    'warn',
    'single'
  ],
  'semi': [
    'warn',
    'always'
  ]
}
```

Now, with this update, our error level for the `linebreak-style` and `indent` rules will be `error`, and the error level for the `quotes` and `semi` rules will be `warn`.

Next, we'll work with ESLint in Vue CLI 3.

## Configuring ESLint in Vue CLI 3

Now that we're familiar with Prettier and ESLint, we'll install them into a new project using the Vue CLI 3. Let's point our Git Bash to the root of `Chapter6` folder and run this command:

```
code .
```

Once VS Code is open, we'll toggle on the command line, and run the following command:

```
vue create vc3-prettier-eslint
```

Then, we'll just accept the default installation with Babel and ESLint, and wait for the app to install.

Next, similar to what we did before, we will run `vue ui` and then import the `vc3-eslint` app to our Vue CLI 3 GUI Dashboard.

As we can see, ESLint comes as a default installation in our Vue CLI 3 apps. But how do we configure ESLint, as we did in the previous section of this chapter?

## Setting up ESLint configurations in Vue CLI 3 GUI

With our `vc3-eslint` app loaded inside the Vue CLI 3 UI, let's click the configuration icon.

We'll click the ESLint configuration, which will update the right-hand pane in the **Project configuration** window as follows:

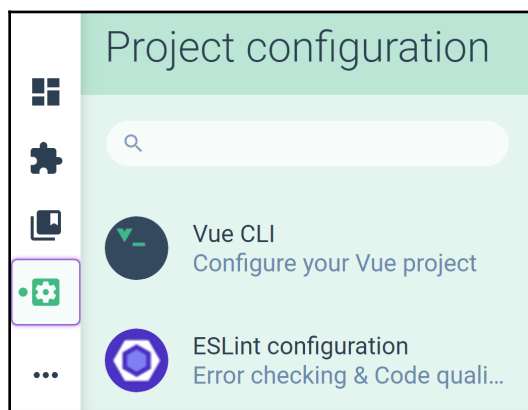


Figure 6.1: Clicking the configuration icon shows possible configurations for your app

There we can see two main entries in this view: **Lint on save**, and **Select config**.

The **Lint on save** option is currently toggled on, but we can toggle it off by clicking on the green switch on the right-hand side of the **Lint on save** option.

We can also choose a different configuration in the **Select config** entry. The default setting can be changed to either the **Strongly recommended** or to the **Recommended** setting. We saw earlier in this chapter how this setting can be configured inside `eslintrc.js` file.

You also have the option to click on the **Open eslintrc** button in the top-right corner of the page to open our project's ESLint configuration file, so that you can edit it directly.



Finally, we can see a list of all the ESLint rules in our project by clicking the **Rules** tab at the top of the configuration panel as follows:

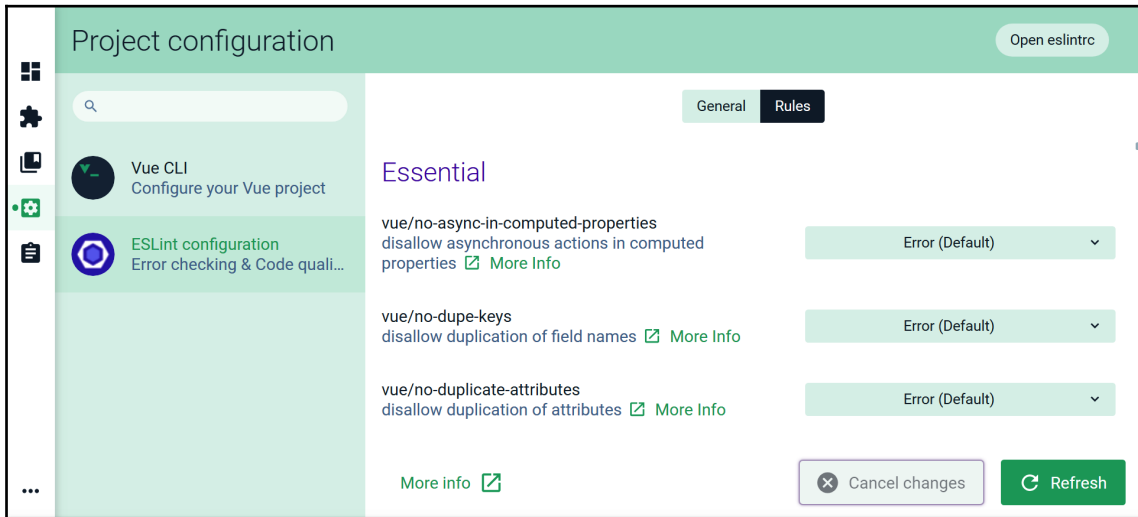


Figure 6.2: Selecting individual rules in the ESLint configuration screen in Vue CLI 3 UI

The screen will show the **Essential** rules that are applied by default in our ESLint configuration, whereas, on the **General** tab we can change the entire ruleset by clicking the drop down in the **Select config** entry; inside the rules tab, we can mix and match individual rules, and then save our updates. This makes it possible to update the rules and make custom changes any way we like.

# Using ESLint and Prettier together in a Vue CLI 3 UI project

If you are starting a project from scratch in Vue CLI 3 UI, you can also make ESLint and Prettier work together. Here's how:

1. We begin by pointing our Vue UI to a root folder as follows. It's important to verify you are not inside an existing Vue app:

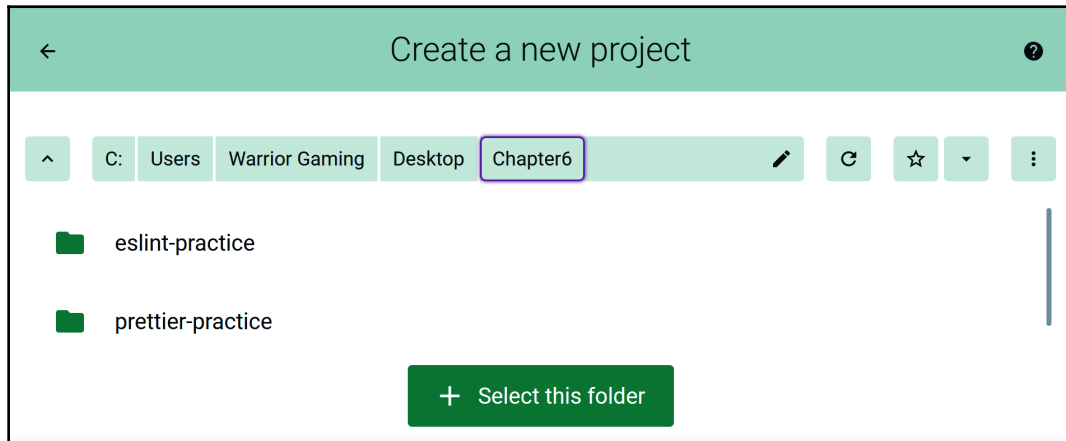


Figure 6.3: Creating a new project in Vue CLI 3 UI

2. Next, we'll click the **Select this folder** button to proceed, and then choose our project's name as follows:

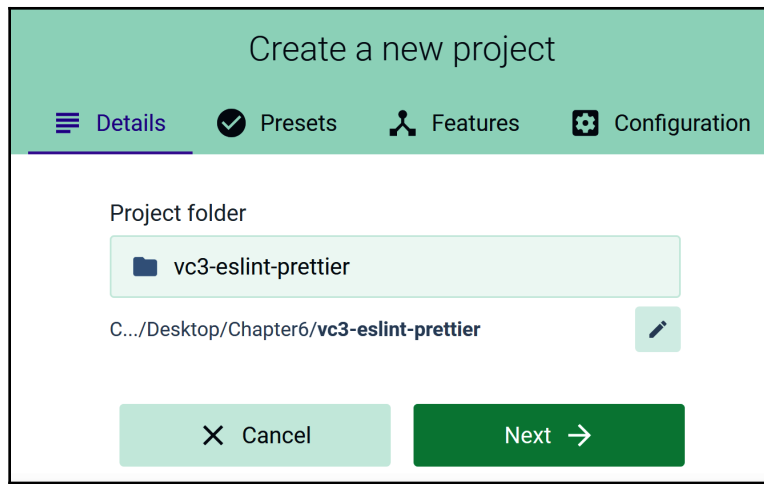


Figure 6.4: Adding project name

3. Next, we'll click the **Select this folder** button to proceed, and then choose our project's name. Click **Next** to proceed and you'll be taken to the **Presets** tab.
4. Once you're on the **Presets** tab, select the **Manual** preset to manually select features, then click **Next** again. On the **Features** tab, the Babel and linter / formatter features should already be preselected, so just click **Next** again to go to the **Configuration** screen, as follows:

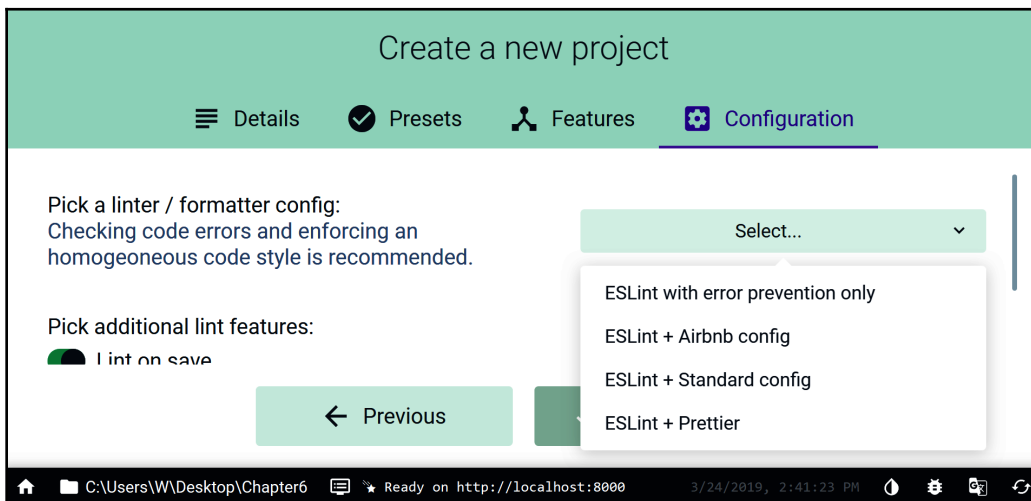


Figure 6.5: Choosing ESLint + Prettier from the Configuration screen

5. The **Pick a linter / formatter config** gives us several options. Choose the **ESLint + Prettier** at the bottom. Once we click it, we'll be greeted with an option to save the preset, so let's save it as in the following image:

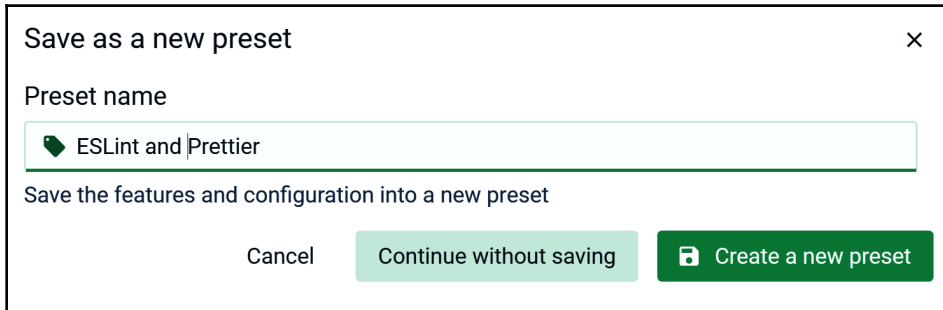


Figure 6.6: Saving a new preset in Vue CLI 3 UI

6. Once we click the **Create a new preset** button, we've finished customizing our project, and we can wait for all the dependencies to be installed.
7. Once it's done, you'll see the **Welcome to your new project** message. Now, you can click on the **Project dependencies** link on the left-hand menu. Besides the expected dependencies (`babel-eslint`, `eslint`, `eslint-plugin-vue`, and `vue-template-compiler`), you'll also see the `@vue/eslint-config-prettier` dev dependency. As we can see, this is an official vue npm module, as it has the `@vue` namespace.

## Summary

In this chapter, we've looked at differences between `npm` and `npx`, and global and local installations of node modules. We've further discussed ESLint and Prettier. We've seen how to install each of them on a stand-alone project, and we've seen how to configure ESLint in a default Vue app using the Vue CLI 3 GUI. We've also seen how to set up ESLint with Prettier, inside Vue CLI 3.

In the next chapter, we will explore using CSS, SCSS, and PostCSS in Vue CLI 3.

# 7

## Improving CSS with SCSS

In this chapter, we'll discuss the basics of SCSS and what problems it solves. We'll also discuss the improvements that are introduced to CSS that bring it closer to SCSS.

We will also work with the app we have made in [Chapter 5, \*Vue CLI 3 and Routing\*](#). The goal of this chapter is to work with CSS, SCSS, and we'll do that by using the Bootstrap + Vue plugin.

The following topics will be covered in the chapter:

- Installing Bootstrap + Vue
- Using the bootstrap-vue plugin to add styles to our project
- Using SCSS in our project

### Installing Bootstrap + Vue

Why not use Bootstrap from the official website, when working with Vue? Well, we definitely could do that, but Bootstrap relies on jQuery, and since we want to keep everything JavaScript-related running through Vue, we'll have to use a project that's built for this purpose exactly: BootstrapVue.

Let's begin by visiting the project website at <https://bootstrap-vue.js.org/>. Here, we can get acquainted with the project in general. Specifically, it would be useful to read the documentation at <https://bootstrap-vue.js.org/docs#vue-cli-3>, which discusses the use of BootstrapVue inside a Vue CLI 3 project.

## Adding BootstrapVue as a plugin

You can create a new Vue project easily with the help of Vue CLI 3 UI. This has been covered a few times in this book, and you should be able to do this on your own.

Name your new project `Chapter7`, and run it through the exact same installation steps as in Chapter 5, *Vue CLI 3 and Routing*.

In your console, go to the root of your project and run the following command:

```
vue add bootstrap-vue
```

You will see the following prompt inside your console:

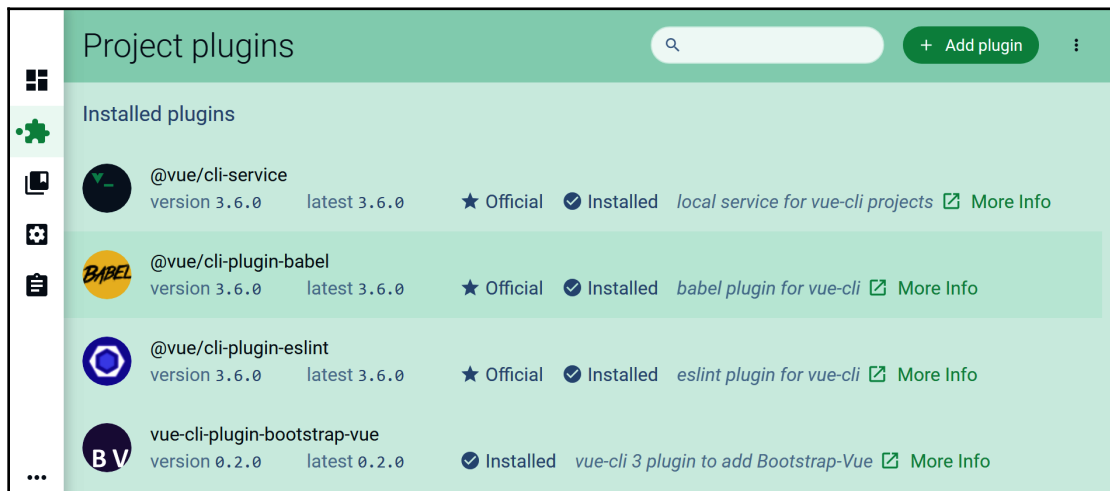
```
Successfully installed plugin: vue-cli-plugin-bootstrap-vue
```

This will be followed up with the following question:

```
? Use babel/polyfill? (Y/n)
```

Just press the *Enter* key to accept the default answer (yes).

Regardless of how you've installed the plugin, your plugins list should now look like this:



Now that we've installed it, we can start using it.

## Using the bootstrap-vue plugin to add styles to our project

Once the bootstrap-vue plugin is installed, let's use it in our project! We'll begin by adding a navbar from the official documentation, which is available at <https://bootstrap-vue.js.org/docs/components/navbar#b-nav-item-dropdown>, as follows:

```
<b-navbar type="dark" variant="dark">
  <b-navbar-nav>
    <b-nav-item href="#">Home</b-nav-item>

    <!-- Navbar dropdowns -->
    <b-nav-item-dropdown text="Lang" right>
      <b-dropdown-item href="#">EN</b-dropdown-item>
      <b-dropdown-item href="#">ES</b-dropdown-item>
      <b-dropdown-item href="#">RU</b-dropdown-item>
      <b-dropdown-item href="#">FA</b-dropdown-item>
    </b-nav-item-dropdown>

    <b-nav-item-dropdown text="User" right>
      <b-dropdown-item href="#">Account</b-dropdown-item>
      <b-dropdown-item href="#">Settings</b-dropdown-item>
    </b-nav-item-dropdown>
  </b-navbar-nav>
</b-navbar>
```

We'll add this navbar to our `App.vue`, in the template section. The updated template element will now look like this:

```
<template>
  <div id="app">
    <div id="nav">

      <b-navbar type="dark" variant="secondary">
        <b-navbar-nav>
          <b-nav-item href="#">Home</b-nav-item>
          <!-- Navbar dropdowns -->
          <b-nav-item-dropdown text="Lang" right>
            <b-dropdown-item href="#">EN</b-dropdown-item>
            <b-dropdown-item href="#">ES</b-dropdown-item>
            <b-dropdown-item href="#">RU</b-dropdown-item>
            <b-dropdown-item href="#">FA</b-dropdown-item>
          </b-nav-item-dropdown>

          <b-nav-item-dropdown text="User" right>
            <b-dropdown-item href="#">Account</b-dropdown-item>
```

```
        <b-dropdown-item href="#">Settings</b-dropdown-item>
      </b-nav-item-dropdown>
    </b-navbar-nav>
  </b-navbar>
  <!--
  <router-link to="/">Home</router-link> |
  <router-link to="/about">About</router-link>
  -->
</div>
<router-view />
</div>
</template>
```

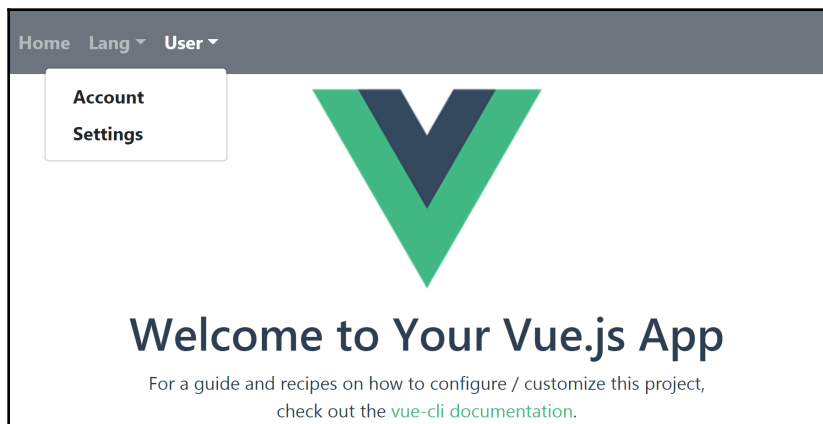
We also need to update the styles so they look like this:

```
<style>
#app {
  text-align: center;
  color: #2c3e50;
}

#nav a {
  font-weight: bold;
  color: #2c3e50;
}

#nav a.router-link-exact-active {
  color: #42b983;
}
</style>
```

After saving the changes, our home page will update to the following screenshot:





To quickly change the color of the background on the `bootstrap-vue` navbar, simply add a contextual Bootstrap color to the `variant` custom HTML attribute. The accepted values are `primary`, `secondary`, `success`, `danger`, `info`, `warning`, `light`, and `dark`.

In the preceding example, we're using `variant` with the value of `secondary`, which gives us a grayish background with white `nav` items on hover.

## Using SCSS in our project

Similar to how TypeScript is referred to as a *superset* of JavaScript, SCSS is often referred to as a *superset* of CSS. But what exactly is this *superset*?

A **superset** of a language is simply a layer around it, with additional functionality that doesn't exist in the *regular* implementation. There could also be additional best practices, methods, and so on.

At the very core, it is possible for SCSS files to contain only plain old CSS syntax. That's because all SCSS ultimately compiles to CSS, and, as such, it is served to our browsers. This makes it very approachable. It is reassuring for an SCSS newbie to **know** that they can keep on writing CSS and it will just work in their SCSS!

Let's make this our starting point too. First off, let's update our `bootstrap-vue` plugin settings, so that we use SCSS rather than CSS.

Navigate to our app's folder structure and locate the following file: `src/plugins/bootstrap-vue.js`. Next, update the code so that it looks like the following:

```
import Vue from 'vue'

import BootstrapVue from 'bootstrap-vue'
import 'bootstrap/scss/bootstrap.scss'
import 'bootstrap-vue/dist/bootstrap-vue.css'

Vue.use(BootstrapVue)
```

## Dealing with errors

It is possible that you might be greeted with the `cannot find sass-loader` error. You can resolve it by running the following command:

```
npm install -D sass-loader sass
```

At this point, if you're unsure about what a loader is, it might be worthwhile revisiting Chapter 2, *Webpack in Vue CLI 3*, to get a refresher on webpack loaders.

Another error that is likely to come up is this one:

```
Module build failed (from ./node_modules/sass-loader/lib/loader.js): Error:
Cannot find module 'node-sass'
```

You can resolve this error by running the following command:

```
npm install node-sass
```

Finally, after the installation of a new package, you'll see a message that reads `found 1 high severity vulnerability` (or something similar to this). You can, and should, fix such a vulnerability, following the cue from the console, and running `npm audit fix`.

## Writing some SCSS in our project

Now that we have imported `bootstrap.scss`, we can start using it.

Let's open `App.vue`, and update the `style` tag to the following:

```
<style lang="scss">
#app {
  text-align: center;
  color: #2c3e50;
}

#nav {
  a {
    font-weight: bold;
  }
  a.router-link-exact-active {
    color: #42b983;
  }
}
</style>
```

What we're seeing in the preceding code is an example of nesting, a feature of SCSS that allows us to mimic the HTML structure of our apps inside our SCSS rules.

If you looked at your project being served in the web browser at this stage, you'd notice that it's still running, and that it is unchanged. That is good! It means everything is working and that everything is getting served just as it should.

Let's now add some variables, mixins, and interpolation to our SCSS.

## Overriding bootstrap-vue styles

We'll begin by adding a custom SCSS file. We'll call it `custom.scss`, and we'll place it inside the `assets` folder of our project.

Now let's add this code to `custom.scss`:

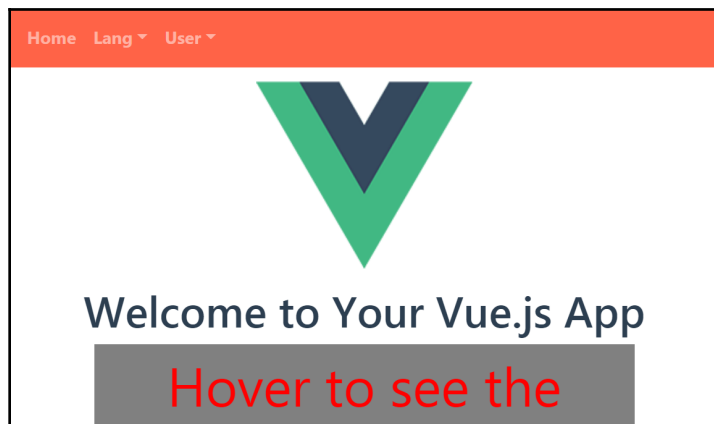
```
$secondary: tomato;
```

We've just overridden the Bootstrap variable. Now let's use it inside our `App.vue` file, by importing the `custom.scss` file. We'll also override the `.bg-secondary` class inside our `#nav` element. The updated `script` tag will look like this:

```
<style lang="scss">
@import './assets/custom.scss';
#app {
  text-align: center;
  color: #2c3e50;
}

#nav {
  a {
    font-weight: bold;
  }
  a.router-link-exact-active {
    color: #42b983;
  }
  .bg-secondary {
    background-color: $secondary;
  }
}
</style>
```

After we've saved all the changes, our project will be updated to this:



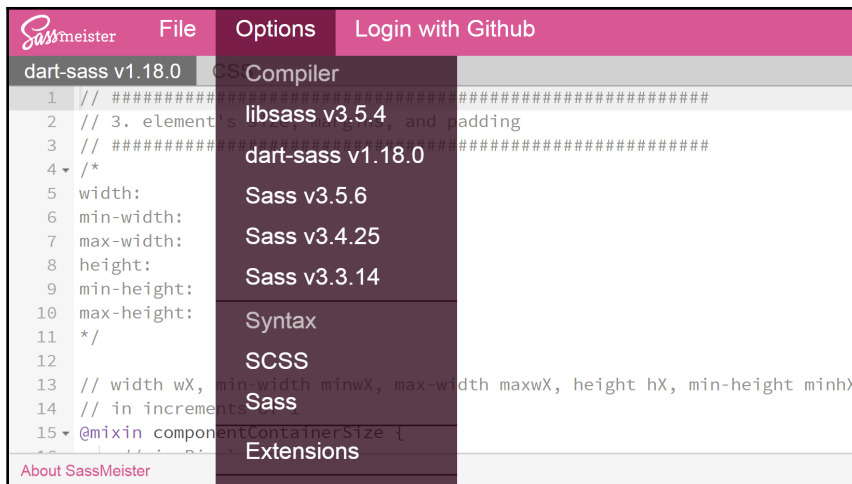
Next, we'll use some of the more advanced SCSS features in our `App.vue` file.

## Using variables, mixins, and interpolation

In this section, we'll use some more variables, mixins, and interpolation syntax in SCSS. Before we even begin this chapter, it is beneficial to mention a website where you can practice writing SCSS, and see the CSS output it will produce. The website is called Sassmeister, and it's available at: <https://www.sassmeister.com/>.

## Using Sassmeister to practice SCSS

Once you visit the website, you can pick a flavor of **Sass** or **SCSS** that you'd like to use, by hovering over the **Options** item as follows:



The Sassmeister website is a great place to practice working with various features of the language. For example, we can navigate to our project's `node_modules` folder, and locate the `bootstrap` folder.



If you can't find the `bootstrap` folder inside `node_modules`, you can install it with `npm i bootstrap`.

Next, open the `scss/mixins` folder, and locate `_alert.scss`, with the following syntax:

```
@mixin alert-variant($background, $border, $color) {
  color: $color;
  @include gradient-bg($background);
  border-color: $border;

  hr {
    border-top-color: darken($border, 5%);
  }

  .alert-link {
    color: darken($color, 10%);
  }
}
```

As we can see, this is a mixin. Let's copy and paste it into Sassmeister and see what output we get.

Actually, we won't get any output. Why?

The reason is simple: a mixin in Sass or SCSS, is actually just like a function in a different language, for example, JavaScript. So, to use a mixin in SCSS, we need to call it, and pass it a few parameters. Here is an example:

```
@include alert-variant (tomato, purple, white);
```



Note that the `@include` syntax is necessary to run a mixin.

The preceding code is supposed to work, but inside our mixin definition, we're calling another mixin as follows:

```
@include gradient-bg($background);
```

This means that in order for our code to compile in Sassmeister, we need to replace the preceding line with something else, for example, with the following:

```
background: $background;
```

Finally, our `@include` for `alert-custom`, needs to be applied on a CSS rule. This means that we need to wrap our call to `alert-custom` with a CSS declaration, as in the following example:

```
.alert-custom {  
  @include alert-variant(tomato, purple, white);  
}
```

Putting all of this together, this is our SCSS code:

```
@mixin alert-variant($background, $border, $color) {  
  color: $color;  
  background: $background;  
  border-color: $border;  
  
  hr {  
    border-top-color: darken($border, 5%);  
  }  
  
  .alert-link {  
    color: darken($color, 10%);  
  }  
}  
.alert-custom {  
  @include alert-variant(tomato, purple, white);  
}
```

Sassmeister will compile the preceding code to the following:

```
.alert-custom {  
  color: white;  
  background: tomato;  
  border-color: purple;  
}  
  
.alert-custom hr {  
  border-top-color: #670067;  
}  
  
.alert-custom .alert-link {  
  color: #e6e6e6;  
}
```

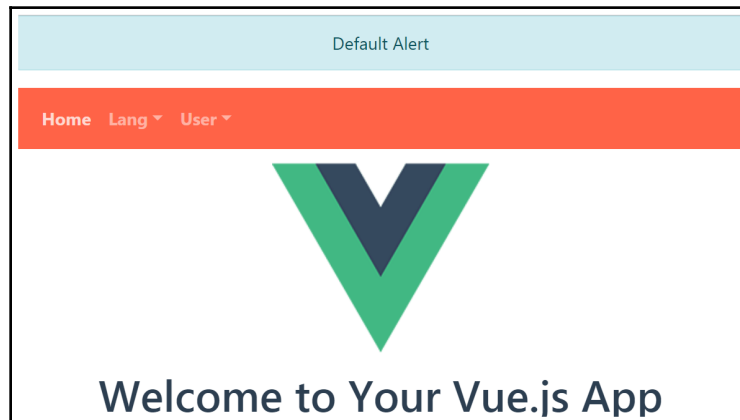
Now that we know how to work with SCSS mixins, we can use them in our project.

## Using SCSS mixins and interpolation in our Vue project

We'll begin with adding an alert to our `App.vue` as follows:

```
<b-alert show>Default Alert</b-alert>
```

This simple addition will update our app look to this:



Now we'll add some mixins to update the styling.

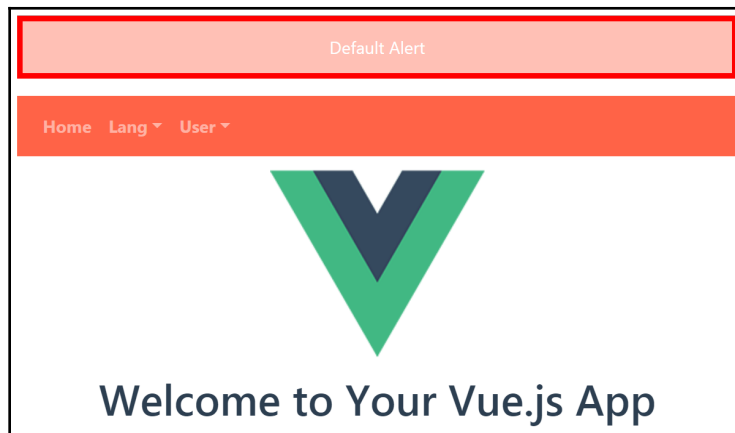
We'll begin with adding our custom mixin to color the alert we've added. Here's the code for `custom.scss`:

```
$secondary: tomato!important;
$border: darken(#ff0000, 10%);

@mixin alert-variant($background, $border, $color) {
  color: $color;
  background: $background;
  border-color: $border;
  border-radius: 0;
  border-width: 5px;
  .alert-link {
    color: darken($color, 10%);
  }
}

.alert-info {
  @include alert-variant(rgb(255, 192, 181), #ff0000, white);
}
```

After we save the preceding file, our alert will look like this:



With this, we conclude the chapter.



## Summary

SCSS is a large topic. Working with SCSS in VDOM frameworks such as Vue can get complex, due to the number of different concepts that are involved. This chapter gave you a sneak peek of what is available and how to get started.

In this chapter, we've touched on using SCSS in our project. We've learned about how to work with CSS and SCSS in our Vue apps. We've also learned about some of the main features of SCSS, such as variables, mixins, and interpolation, and we've also learned how to implement Bootstrap 4 in our Vue apps. Next, we've learned how to edit the built-in components of our bootstrap-vue plugin in Vue, and how to update them with custom SCSS code.

In the next chapter, we'll build a simple project, using everything we've learned so far, and we'll publish it on GitHub Pages.

# 8

## Deploying Vue CLI 3 Apps on GitHub Pages

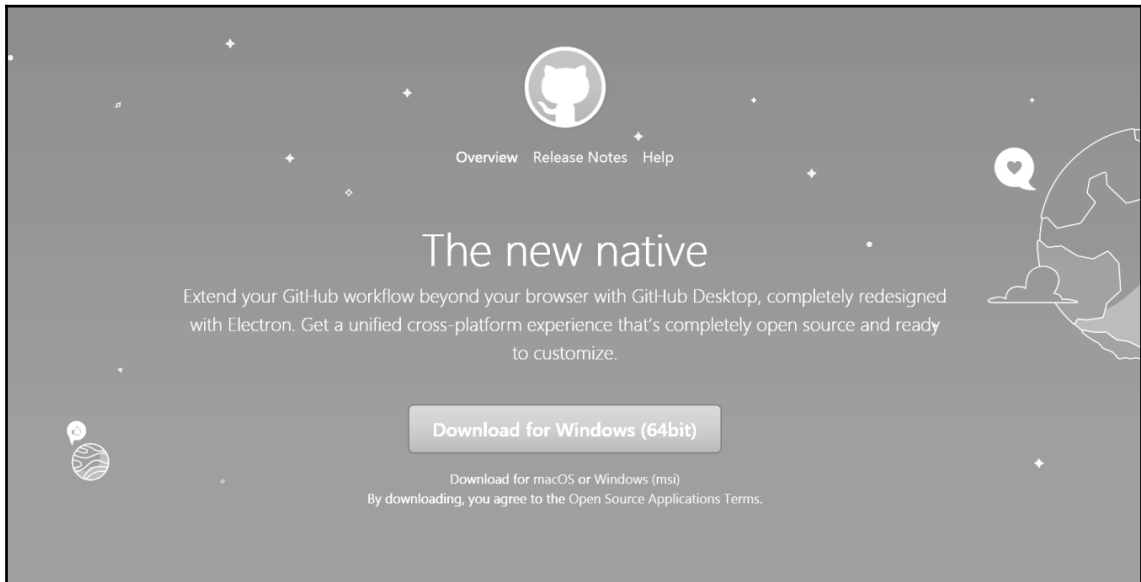
In this chapter, we will set up a new repository on a new GitHub account, and quickly prototype a Vue-powered project. Then, we'll deploy it using GitHub Pages. In this chapter, we'll get acquainted with the following:

- GitHub Desktop app
- Basics of Git (what it is and how it works)
- Some basic Git commands
- Understanding branching in Git
- How to properly set up a Vue app to work with GitHub Pages

By the end of this chapter, you'll have a fully working Vue app being served on GitHub Pages. Let's begin by understanding how to work with the GitHub Desktop app.

## Installing GitHub Desktop app

To begin, navigate to the following URL: <https://desktop.github.com>. Next, simply download the app with the prominent download button in the center of the screen as shown in the following screenshot:



Once you have downloaded the GitHub Desktop app installation file (`GithubDesktopSetup.exe`), you need to run the installation.

Once installed, you'll be greeted with a minimal interface. If you're dealing with Git and GitHub for the first time, this minimal interface might seem like it's almost *too minimal*. What's going on in this app?

In order to understand how this app works, you need to know some very basic concepts of Git in general.

## Understanding how Git works

When you work on a project, if you are not using source version control such as Git, you might think that there is only one folder for your project. For example, any Vue app that you've installed using Vue CLI 3 will end up as one separate folder with a number of subfolders inside of it.

We can simplify the tree structure of such a Vue app, and describe it like this:

```
aVueApp
|- aSubfolder
|- anotherSubfolder
|- yetAnotherSubfolder
```

Note that our imaginary Vue app has one root folder called `aVueApp`. Inside of this root folder are three subfolders that contain the complete functionality of our Vue app.

As long as we're dealing with only a few files and folders, that is, as long as our Vue app is very small, we don't have to worry about versioning our files and folders, since there's almost nothing to be versioned. If we make an error in a file or a folder, we simply undo the changes and we get back to an earlier version of our file. This is a reasonable approach.

However, what would happen if our project grew to, say, 30 files spread across seven folders? Let's also imagine updating only a couple of lines of code in each of these 30 files, during the development of our Vue project. Let's further imagine that we make so many changes to these files and that we make a few major mistakes during development that we simply cannot continue working on our project. Normally, on a smaller project, we could simply undo the changes we made to a few files to get back to the *last working version* of our app.

However, with many changes spread across many different files, not only is it unlikely that we'd be able to remember to undo every single change back to the last working version of our app, undoing all these changes would also be time-consuming and, thus, ultimately impractical.

What is a developer to do?

Luckily, we have version control. Git is what's known as a **DVCS**: a **distributed version control system**. Without getting too technical, you can think of Git as a big undo/redo button, with a slight caveat. Instead of undoing changes on a single file, Git tracks versions of a developer's app through time, sort of like taking a snapshot of the entire app at a certain point in time.

A developer is responsible for deciding just when this should take place. For example, when you've written a new feature in your app, and you are ready to save your app in its present state so you can get back to it, you simply save it by *committing changes*.

Rather than describing this in theory, let's start a new Vue app and start tracking changes made to it with the help of Git.

## Starting a new Vue app and tracking changes with Git

Let's open Git Bash and create a new Vue project in a folder of our choice as follows:

```
vue create chapter8
```

Let's keep things simple and press the *Enter* key whenever Vue CLI 3 prompts us for an answer. As we've already learned, this is a quick and easy way to accept default settings when installing a new Vue app with Vue CLI 3.

To start tracking a project with Git, we run the following command:

```
git init
```

This command initializes a new local Git repository. We're using the word *local* here because, currently, this repository is local to our computer only. If we wiped our computer's disk tomorrow, the local Git repository that we just made would be lost as well.

However, there are cases when a Git repository has already been initialized. The following is a simple command we can run to see whether there is a Git repository in our project or not:

```
git status
```

The preceding command gives us information about the current state of our repository. If the folder we've run this command in is not empty, it will report that the current folder is not a Git repository.

Let's run the `git status` command now in the root of the `chapter8` folder. This is what we get back:

```
On branch master
nothing to commit, working tree clean
```

This information tells us that we are currently on the master branch of our project, and that there is nothing to commit. The working tree is clean. What is this *working tree*?

The **working tree** represents all the files and folders that have been saved and whose changes have been committed in our project.

To inspect the history of changes made to our project's files, we can run this command:

```
git log
```

This command will list all the commits that we've added to the project so far. Here's the output of the preceding command:

```
commit ee07b4828fb217902f2f7c1047f00f936c3a461f (HEAD -> master)
Author: UserWithEmail <user@email.com>
Date: Mon Apr 22 22:18:48 2019 +0200

    init
```

Here, we can see several pieces of information:

- Commit with a 40-character SHA unique identifier
- Commit author (including author's username and email address)
- The date and time when the commit was made

Let's now make a minor change to our `App.vue` file. We'll update the `msg` attribute from **Welcome to Your Vue.js App** to just **Welcome**. Once we've saved the changes, we can run the `git status` command again. This is what we will get back inside the console:

```
On branch master
Changes not staged for commit:
  (use 'git add <file>...' to update what will be committed)
  (use 'git checkout -- <file>...' to discard changes in working
  directory)

        modified:   src/App.vue
no changes added to commit (use 'git add' and/or 'git commit -a')
```

Now, we can create another *snapshot* of the condition our app was in at this point in time. To do that, we need to do the following two things:

- First, we need to add all our changes to the *staging area* (this is the second Git tree).
- Second, we need to commit those changes, so that they get their own commit SHA (unique identifier consisting of a string of 40 pseudo-random characters).

## Committing changes to our app

Let's add all the changes with this command:

```
git add --all
```

Next, we'll commit the changes by giving a commit message as follows:

```
git commit -m "Update App.vue's template"
```

Now, let's run the `git log` command again to see the output. This time, we'll also pass it a flag, `--oneline`, as follows:

```
git log --oneline
```

This is the result that we get back inside the console:

```
e469374 (HEAD -> master) Update App.vue's template
ee07b48 init
```

Note that, this time, the information we get back is a lot more succinct. The 40-character SHA string is now abbreviated to the first eight letters and numbers, and the author details are skipped. The `--oneline` flag is very useful when we have many commits to look into. You can even narrow down the number of results by passing another flag, starting with the minus character, and then the number of commits you want to see. Whichever number you specify, you'll get back that many of the most recent commits.

For example, we currently have two commits, so we could run this command to see the last two commits:

```
git log --oneline -2
```

We could also run this command to see only the last commit:

```
git log --oneline -1
```

Next, we'll look at the three trees concept.

## The three trees concept

There is a concept of three trees in Git. It is important to note that this concept doesn't have anything to do with the tree data structure. Here, the tree is just a metaphor we use to refer to a certain status of our files. The three trees include the following:

- Working directory
- Staging area (also known as staging index)
- The repository

The concept of three trees might sound confusing at first, but it's actually very simple. Here is a basic Git workflow:

1. We install an app using `vue create <appname>`.
2. As part of the installation, our app is initialized as a Git repository.
3. Once the installation is complete, we have all our files (working directory) and a single commit (with the message that reads "init", meaning a repository has been initialized).
4. We then make some changes to our working directory.
5. Before committing these changes, we need to add them to our *staging index*. The staging index is just a tree of updated files that we are preparing for a commit. If we haven't added any files with the `git add <filename>` command, our staging index is empty.
6. Once you've staged all the files into the staging area, you then commit them (using the `git commit -m` syntax, followed with the specific commit message inside quotes).
7. When you've committed your changes, you simply repeat the process.

Besides the working directory and the staging area, the third tree is simply the repository. You can use the `checkout` command to get to *load* whatever part of the repository you want into your working directory. Then you can simply repeat the process if you like.

## Branching in Git

Another important concept in Git is branching. For science fiction fans, branching could be described as sort of a multiverse: a bunch of alternative realities in which your project can, for example, have a navbar in the footer, or no navbar at all.



When we want to add a new feature to our project, we use branching. That way, each branch represents a separate feature of our project.

Every Git project starts off with a single branch: `master`.

Once we decide we need another branch for a special feature we need to work on, we simply run this command:

```
git checkout -b "branch name here"
```

The preceding command means the following: create a new branch, call it `branch name here`, and switch to this newly added branch.

To view a listing of all the branches in our project, we simply run this command:

```
git branch
```

This command will print out all the branches we have in our project, as well as putting an asterisk in front of the current branch. Here's an example output:

```
branch one  
branch two  
* master
```

The preceding output would mean that we have a total of three branches in our project, and that we're currently on the `master` branch.

## Merging branches

One more thing to mention before we set up our project in GitHub is the concept of merging branches.

The workflow goes like this: we check out a new branch using a sensible name (for example, `add-navbar`). Next, we add all the features that we want to this branch, and commit our changes. Once we're done, we switch back to another branch, for example, `master`. Then we simply run the following command:

```
git merge add-navbar
```

Doing this will result in all the code from the `add-navbar` branch getting pulled into the `master` branch. Our `master` branch and our `add-navbar` branch now share the same code.



Note that you cannot switch branches if you have changes in your working directory that haven't been committed! If you don't want to commit your changes, but still need to switch to a different branch, a workaround is to use the `git stash` command.

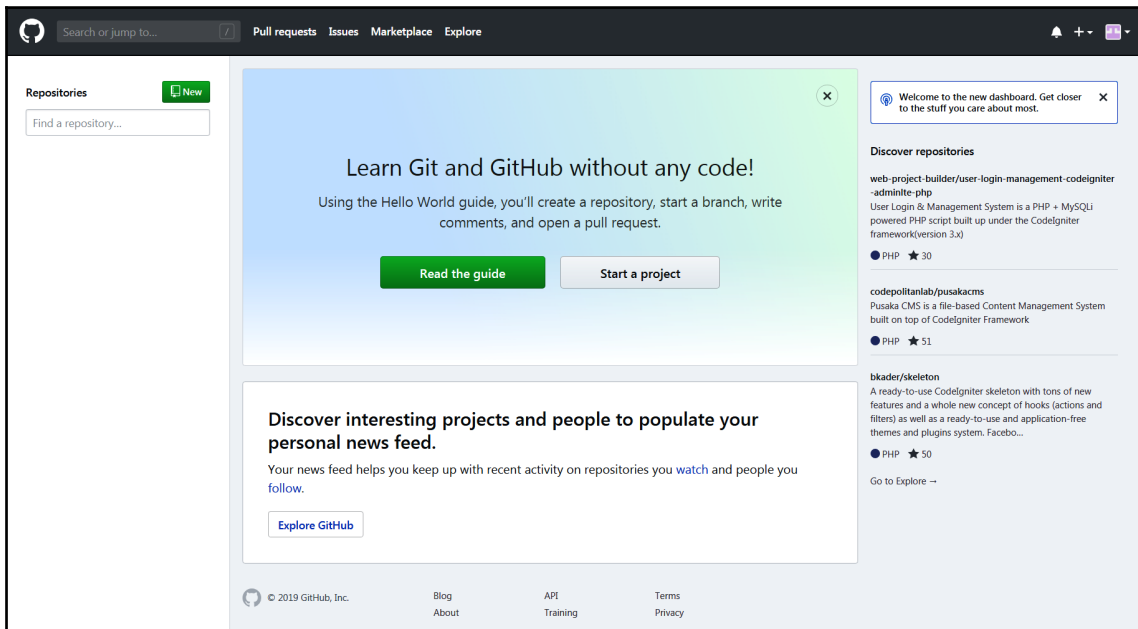
This concludes our quick introduction to some basic concepts of Git. However, this is nowhere near a complete introduction. We have only mentioned some basic features that we will be using in our project. There are still many more things to learn.

## Registering a new GitHub account

We have installed GitHub Desktop. We also know enough about Git to be able to understand the rest of this chapter. Now we'll register for a GitHub account.

Navigate to <https://github.com> and sign up for a new account. Make sure to check your email and confirm the registration.

Once you've signed up and logged in, you'll see an interface like this:



This is the place where our *remote* GitHub repository will live. To add to our remote repository, we need to push our changes there. The remote repository is referred to as *origin/master*.

Luckily, we have GitHub Desktop installed, and using it is the simplest way possible to interact with GitHub. You can easily create new repositories, update existing ones, add branches, and push to the remote, all from GitHub Desktop.

We already have the `chapter8` Vue project, with one commit added to it. Let's now use GitHub Desktop to add it to remote.

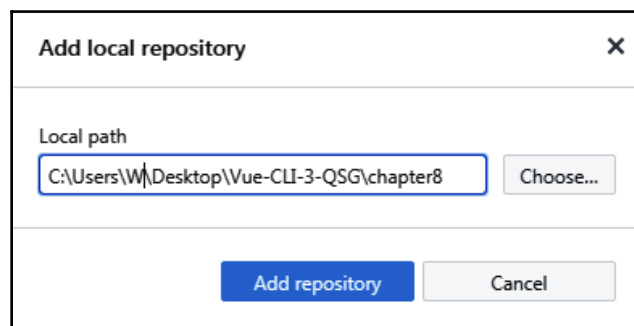
## Adding origin/master using GitHub Desktop

Let's open the GitHub Desktop app. We'll start using it by adding our `chapter8` project. To do that, click on the **File** dropdown in the main application menu at the very top. Next, click on the **Add local repository** command.

A popup will appear. This popup's title is **Add local repository**. In the body of the popup, we can see a single input field with a placeholder that reads **repository path**. Next to it is the button to choose a local repository. Let's click the **Choose...** button.

Clicking the **Choose...** button will open Windows Explorer, where we can navigate to the location of our `chapter8` folder. Once you've selected the `chapter8` folder, simply click the **Select Folder** button at the very bottom of the Windows Explorer window.

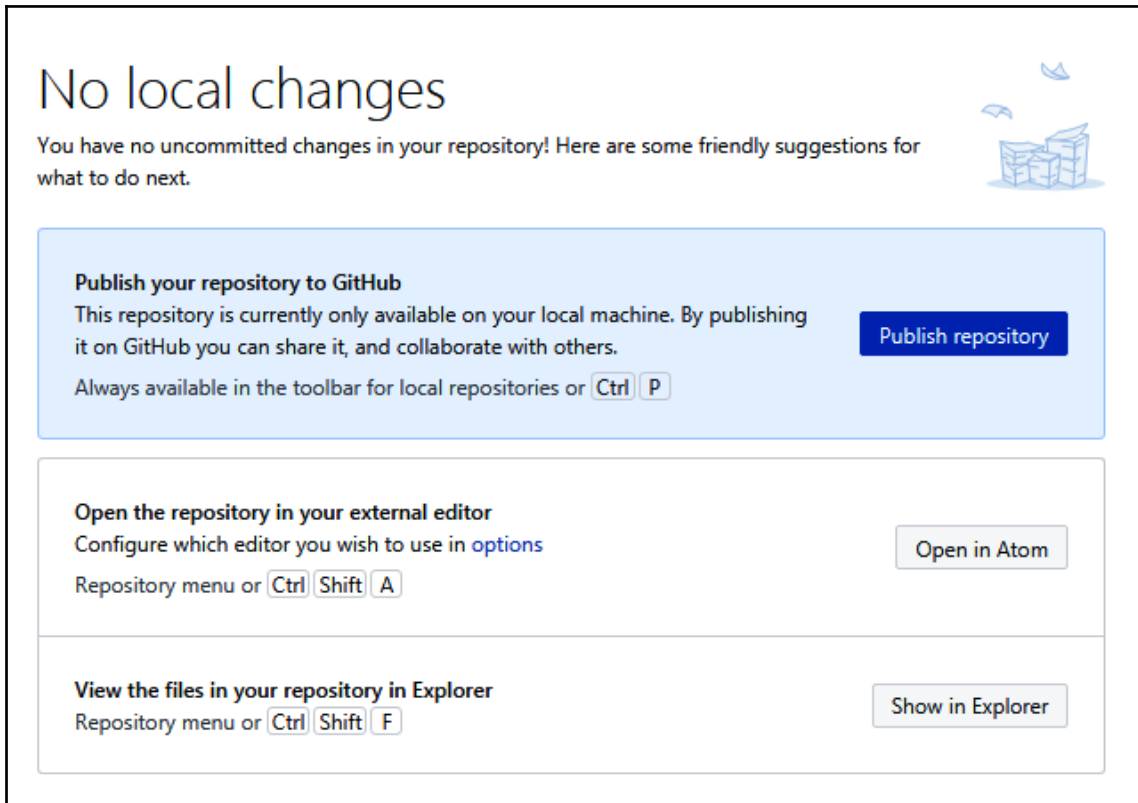
Next, simply click the **Add repository** button in the popup as follows:



That's it! You've just added your local repository. Now, you're ready to publish it.

## Publishing your local repository

To publish our local repository, we need to simply click the **Publish repository** button as follows:

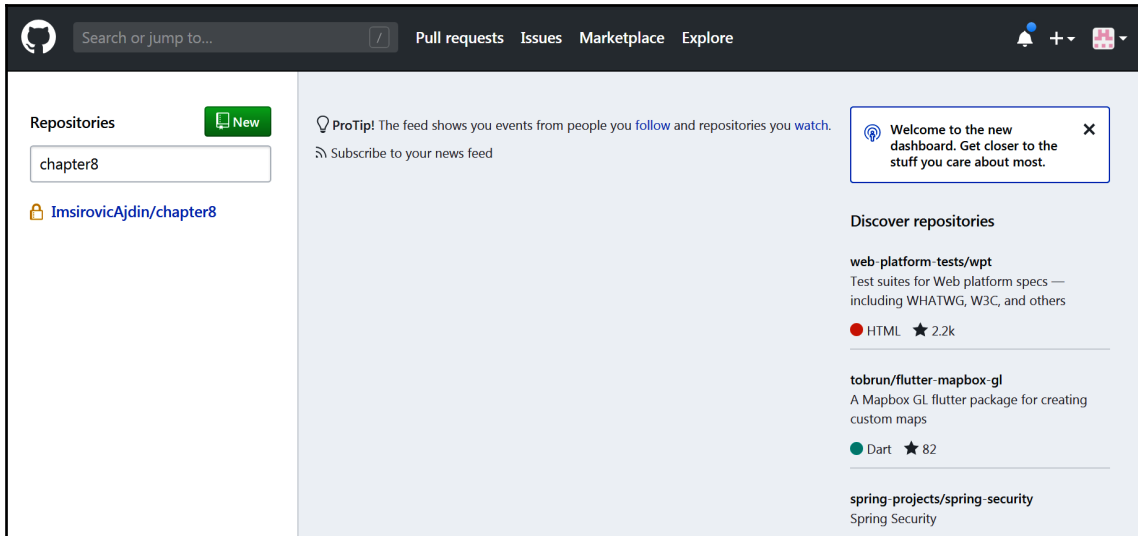


Once we click the **Publish repository** button, a new popup will appear. There are only the following default settings already added to the **Publish repository** popup window:

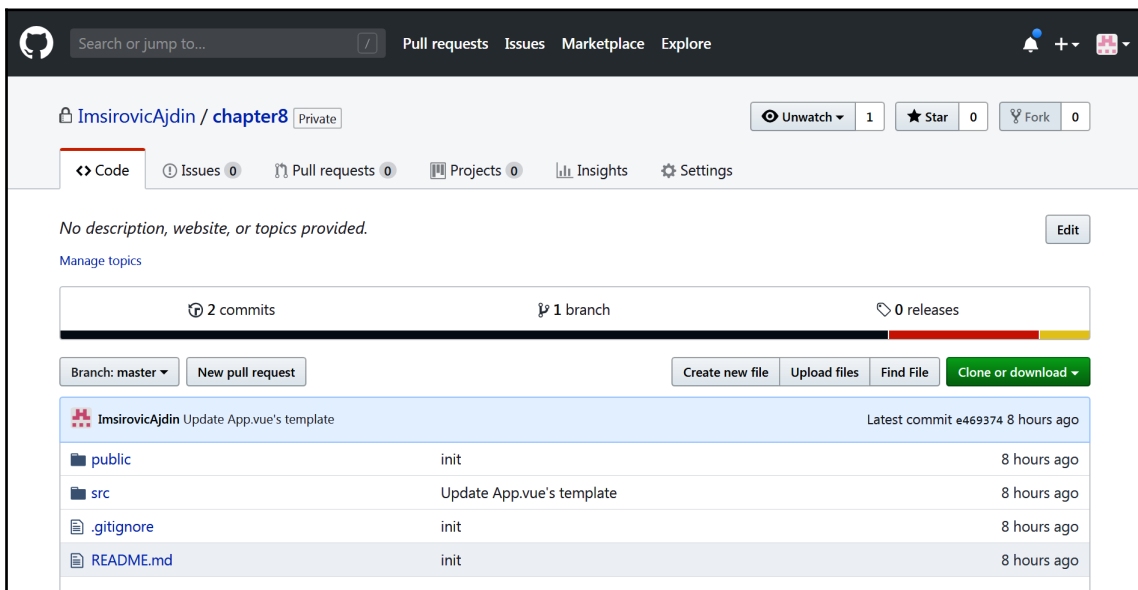
- Repository name (as it will appear in GitHub)
- Description (blank by default)
- Keep this code private checkbox (checked)

Let's accept the defaults and click the **Publish repository** button. We will now have a new private repository in our GitHub account.

Back in GitHub, here's our remote repository:



If we click on the link to `chapter8` in the left-hand sidebar, we'll be taken to our repository as follows:



Here, we can see our entire project.

We can also see that there are currently **2 commits** in our repository. Clicking on the **2 commits** link will take us to a sort of visual Git log of commits, where you can also click on each individual commit to inspect the changes made at that specific time in our project's history.

Next, we'll publish our project so that it can be seen on GitHub Pages.

## Publishing our project on GitHub Pages

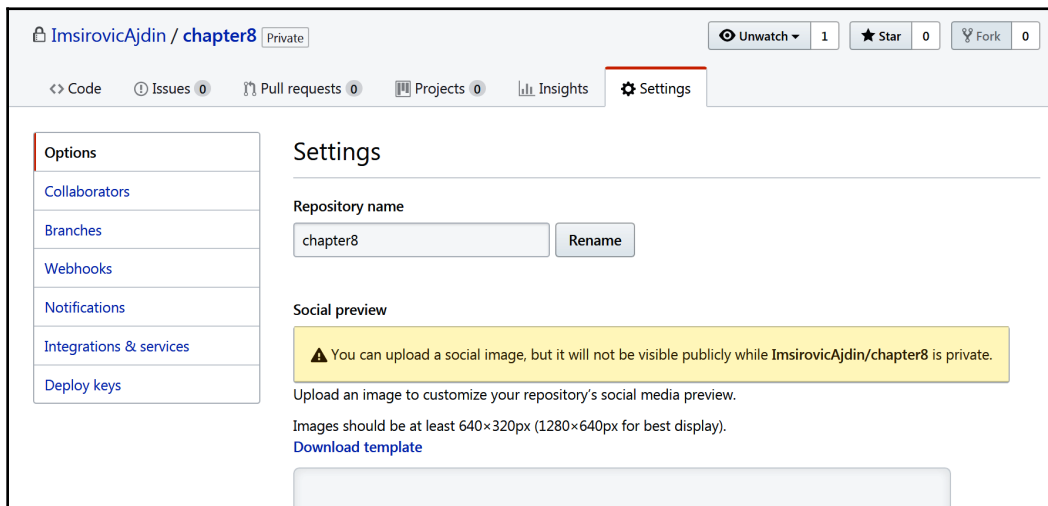
GitHub Pages is a hosting service. You can host static files there. This means that you can't host a database with a backend, so projects written in backend technologies, such as PHP, Node, .NET, and so on, can't be added.

Still, it's interesting how far you can take a simple project consisting of static files. You can even host a blog using Jekyll, GitHub's serverless blogging platform.

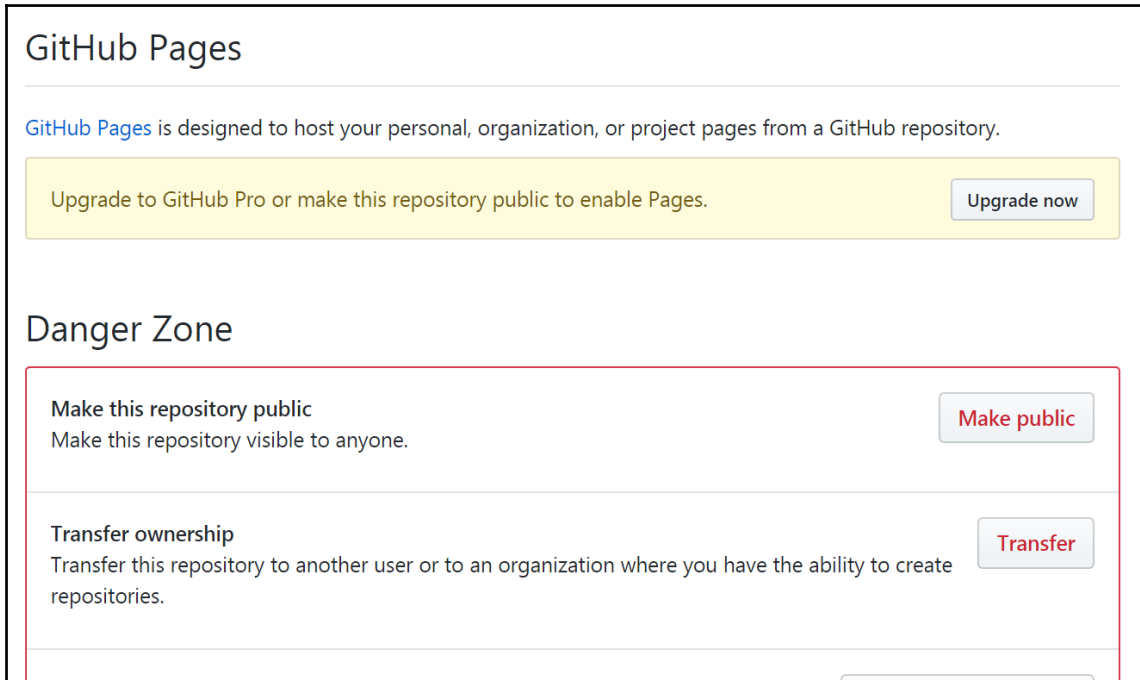
However, the focus of this section is publishing our simple Vue app, so let's get ready to do that.

## Getting our project ready to be published

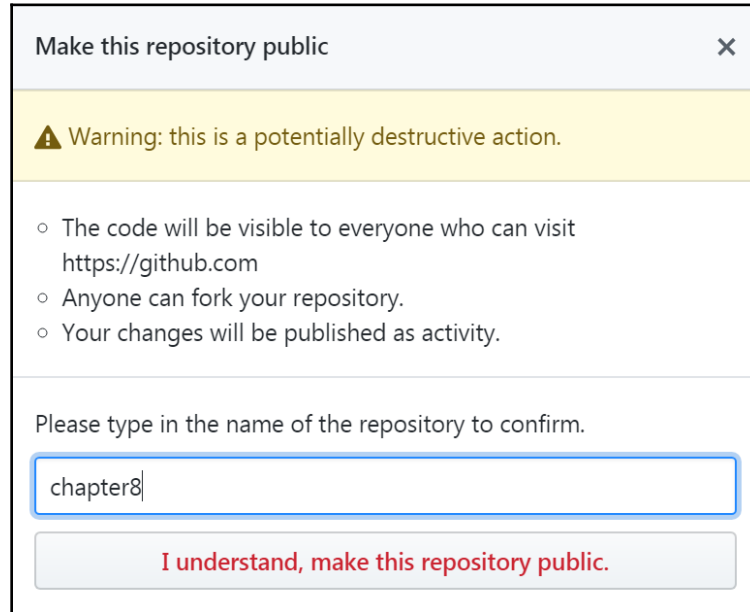
We'll begin in our repository dashboard on GitHub. Let's navigate to the **Settings** tab as follows:



Next, we need to scroll down to the **GitHub Pages** section as follows:



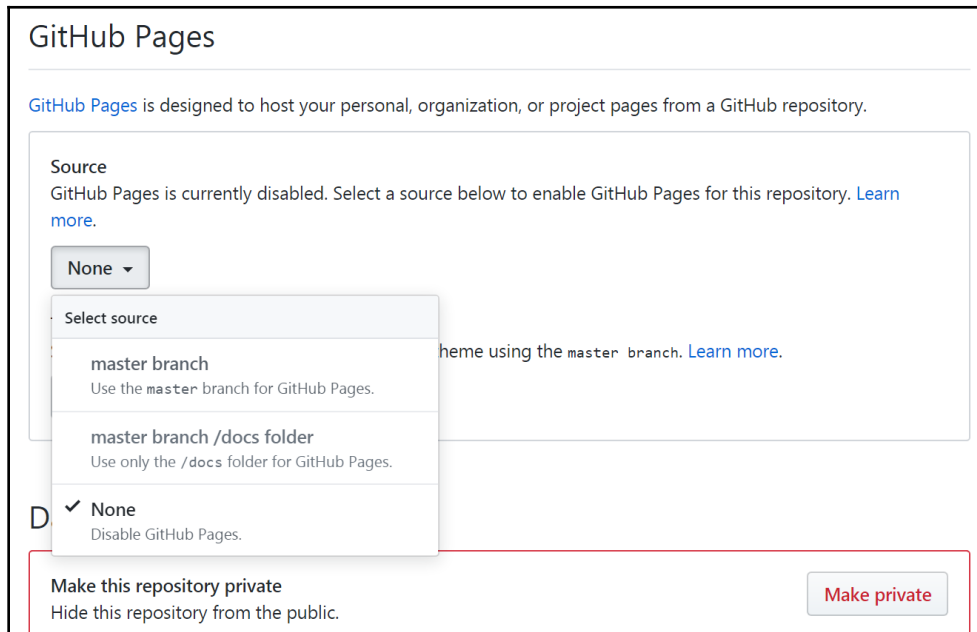
Looking at the preceding screenshot, we can see that we need to make our repository public before we can publish it on **GitHub Pages**. To do that, we simply need to click the **Make public** button (the first button under the **Danger Zone** title) as follows:



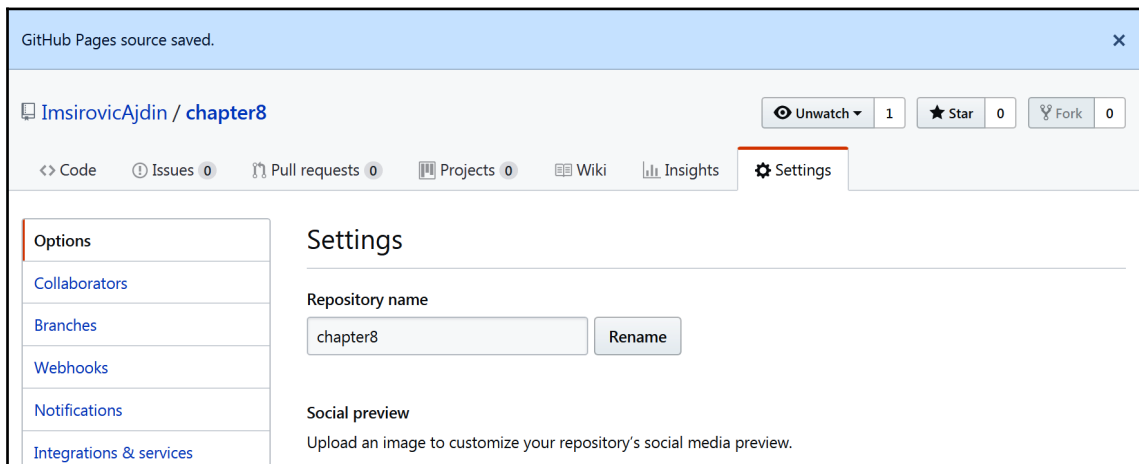
In the popup that appears, we'll type the name of our repository to confirm our choice, then click the **I understand, make this repository public** button. It is possible that GitHub will require you to retype your password at this stage.



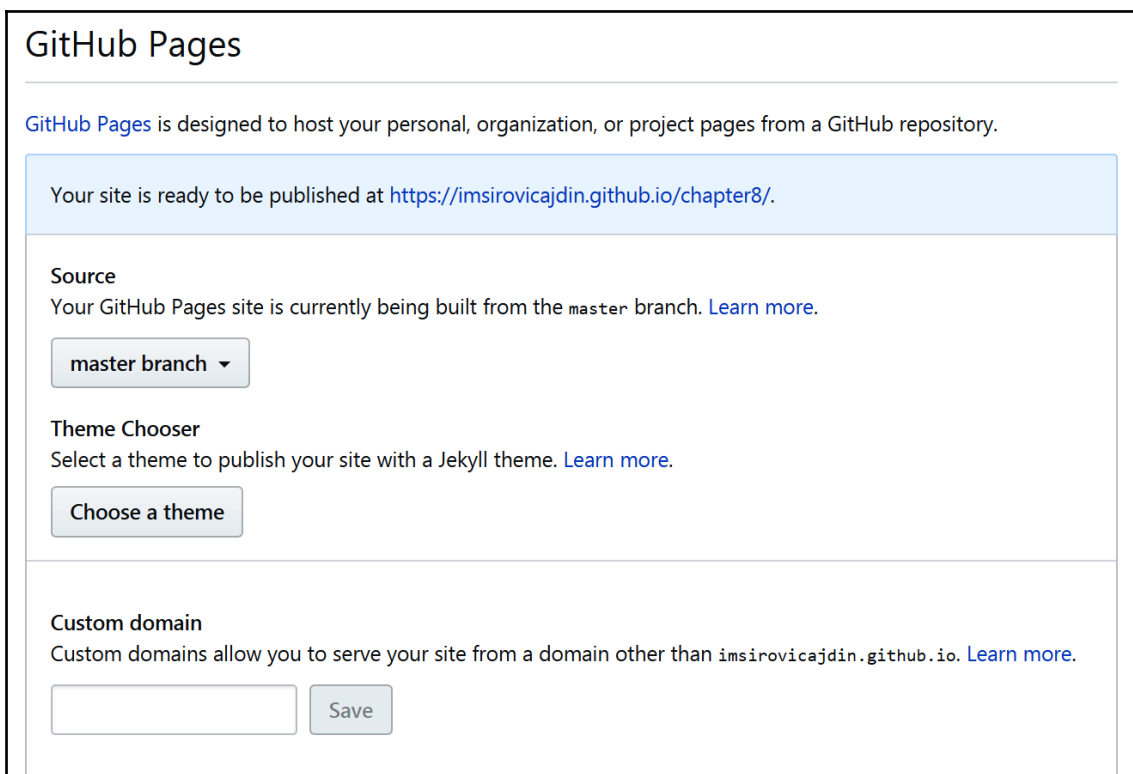
Now, you need to scroll down to the **GitHub Pages** section again, and click the drop-down button. Choose **master branch** in the drop-down list as follows:



Once you've chosen the master branch, an alert will appear under the main menu, and it will read "GitHub Pages source saved", as follows:



Finally, scroll down to the GitHub Pages section again, and you'll see this message:



Everything is now ready for publishing on GitHub Pages, but we still need to configure our Vue app.

## Updating the Vue configuration file

With all the config files we've seen in this book, it should come as no surprise that Vue itself has one! The name of this file is `vue.config.js` and it lives in the root of our project. This file is optional, so `vue.config.js` doesn't come with our project preinstalled. Let's add it to the root of our project, and add the following code to it:

```
// vue.config.js
module.exports = {
  // options...
}
```

The first option we'll give our Vue config file is `publicPath`, as follows:

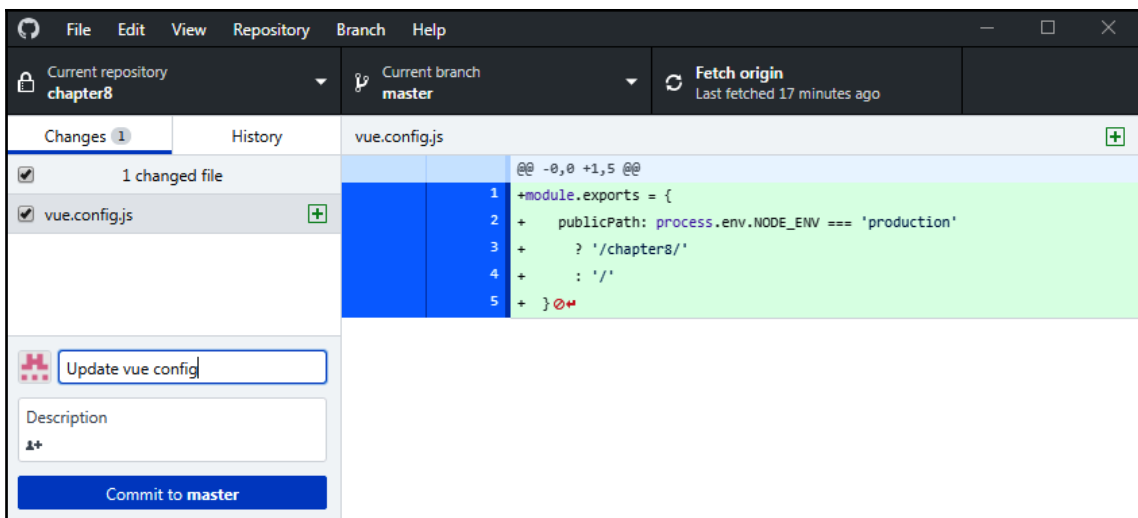
```
module.exports = {
  publicPath: process.env.NODE_ENV === 'production'
    ? '/chapter8/'
    : '/'
}
```

You can read more about this setup at the following URL: <https://cli.vuejs.org/guide/deployment.html#github-pages>.

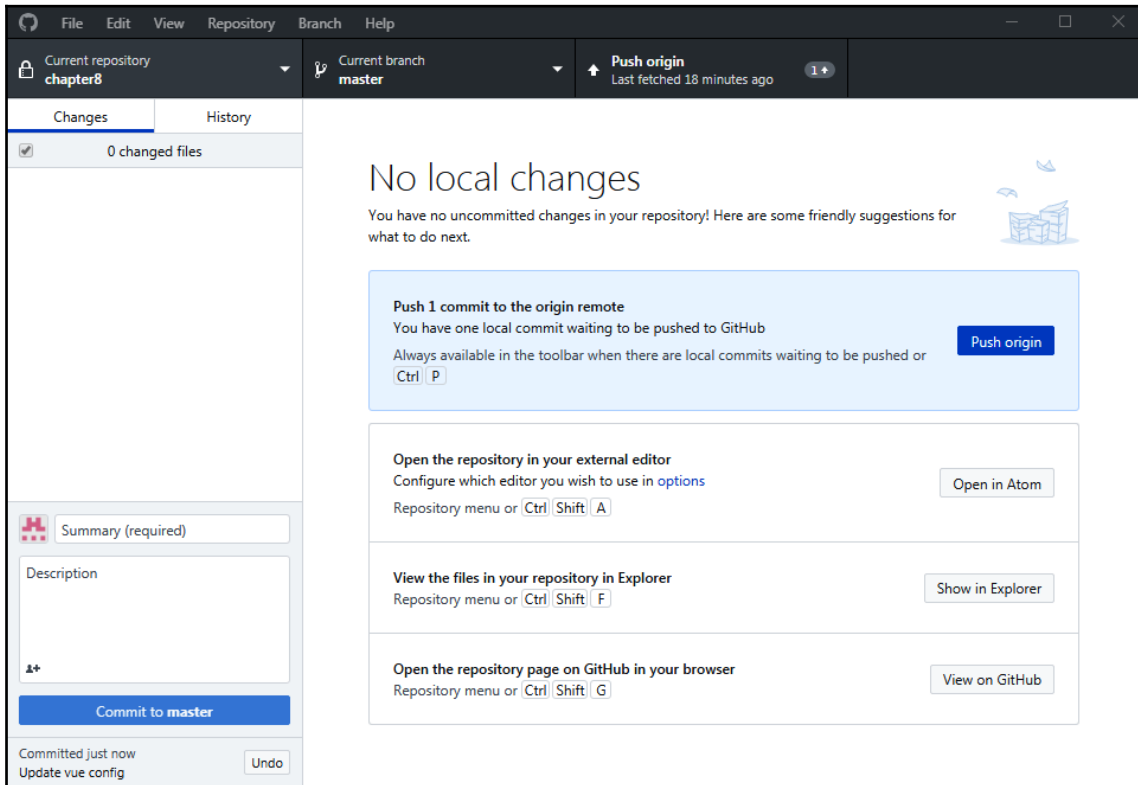
Let's save our changes and use GitHub Desktop to commit them and push them to the remote repository.

## Saving changes and pushing them to the remote repository with GitHub Desktop

Open GitHub Desktop and you'll see your changes listed. Simply type your commit message in the lower-left corner of the app, and then click the **Commit to master** button (the blue button in the very lower-left corner of the GitHub Desktop app, as shown in the following screenshot):



Once you click the **Commit to master** button, you need to push your changes to origin (another name for the remote repository) as follows:



Unfortunately, our repository will still not be published. To publish it, we need to open the `.gitignore` file in our root, and comment out the `/dist` folder so that it's no longer ignored, as follows:

```
.DS_Store
node_modules
#/dist
```

Next, we'll build our project with this command:

```
npm run build
```

We'll see the following message:

```
Build deployed. The dist directory is ready to be deployed.
```

Commit changes in GitHub Desktop with a message, for example: Prepare dist directory. After that, like before, click the **Push origin** button.

## Debugging your GitHub pages deployment

Sometimes, things just don't work like they're supposed to. In that case, you can simply locate the hidden `.git` folder inside your project, and just delete it. This will remove the local repository. Next, update the `vue.config.js` file, like this:

```
module.exports = {
  publicPath: process.env.NODE_ENV === 'production'
    ? '/chp8/'
    : '/'
}
```

Back in GitHub Desktop, the app will notice your changes and offer you to publish the repository. The offered repository name is `chapter8`, but you need to rename it to `chp8`. Back in GitHub, locate your new `chp8` repository, and set it to public. Note that you could have done that from the pop-up window in GitHub Desktop. Regardless of how you did it, it's important to make the repository public. Next, scroll down to the GitHub Pages section in the **Settings** tab of your GitHub repository's dashboard, and publish a new GitHub Pages project on the master branch, just like we did before.

At this point, it is not perfectly clear in Vue documentation how to easily deploy a website on GitHub Pages. What currently works is publishing your `dist` folder as a subtree repository. Here are the commands used to achieve this:

```
git init # with /dist un-ignored in .gitignore
npm run build
git add dist
git commit -m "Init dist"
git status
git subtree push --prefix dist origin master
git add --all
git commit -m "Publish all"
create mode 100644 README.md
create mode 100644 babel.config.js
create mode 100644 package-lock.json
create mode 100644 package.json
create mode 100644 public/favicon.ico
create mode 100644 public/index.html
create mode 100644 src/App.vue
create mode 100644 src/assets/logo.png
create mode 100644 src/components/HelloWorld.vue
```

```
create mode 100644 src/main.js
# make some additional changes, follow it up with 'npm run build'
git add dist
git commit -m "Publish chp8 title"
git subtree push --prefix origin master
```

The repository for this chapter can be found at <https://github.com/ImsirovicAjdin/chp8>. If you look at the file structure, you'll notice that your repository looks like this:

```
css/
├── app.11c93175.css
img/
├── logo.82b9c7a5.png
js/
├── app.8c7f6fb3.js
├── app.8c7f6fb3.map
├── chunk-vendors.ecd76ec1.js
└── chunk-vendors.ecd76ec1.js.map
favicon.ico
index.html
```

Obviously, the `git subtree push --prefix dist origin master` command pushed only the contents of the `dist` directory to GitHub. To better understand how the repository for this chapter works, refer to the `chapter8` folder in this book's code, and once you open it, run the `git log` command on it inside a command-line program.

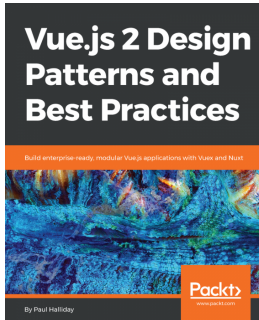
## Summary

In this chapter, we've learned about the very basics of Git and GitHub. We've seen how to set up and use GitHub Desktop, and how to use it to quickly and easily push a local repository to a remote on GitHub.

Further, we've looked at how to add branches to our app and how to pull in these branches to merge them with the master branch. Finally, we've discussed the tweaks we need to make to our project in order to publish it on GitHub Pages.

# Other Books You May Enjoy

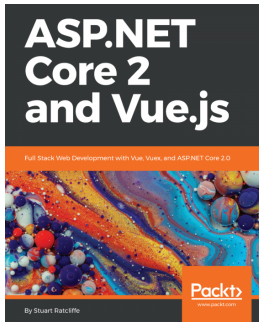
If you enjoyed this book, you may be interested in these other books by Packt:



## **Vue.js 2 Design Patterns and Best Practices** Paul Halliday

ISBN: 978-1-78883-979-2

- Understand the theory and patterns of Vue.js
- Build scalable and modular Vue.js applications
- Take advantage of Vuex for reactive state management
- Create single page applications with vue-router
- Use Nuxt for FAST server-side rendered Vue applications



## **ASP.NET Core 2 and Vue.js**

Stuart Ratcliffe

ISBN: 978-1-78883-946-4

- Setup a modern development environment for building both client-side and server-side code
- Use Vue CLI to scaffold front-end applications
- Build and compose a set of Vue.js components
- Setup and configure client-side routing to introduce multiple pages into a SPA
- Integrate popular CSS frameworks with Vue.js to build a product catalogue



## **Leave a review - let other readers know what you think**

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

# Index

## @

### @vue/cli-plugin-babel

- Babel 7 61
- features 62
- fundamentals 61
- role of @vue/babel-preset-app 62
- role of babel-loader 62

## A

Asynchronous Module Definition (AMD) 35

## B

### Babel

- about 58, 59, 60
- babel-loader, adding to webpack configuration 68, 69
- methods option, updating to ES6 syntax 68
- NPM packages, installing 63, 64, 65
- webpack configuration, updating 70
- webpack, running with ES5 code 66, 67
- webpack-dev-server, adding 67

### bootstrap-vue plugin

- using, to add styles to project 144, 146

### BootstrapVue

- adding, as plugin 143
- installing 142

### branching 161

### Browserify

- URL 35

## C

### Chart.js library

- URL 26

### cli-plugin-babel

- features 61

Command-Line Interface (CLI) 8

content delivery network (CDN) 25

Cypress 97

### Cypress plugin

- adding, to project 98
- package.json updates, verifying 98, 99, 100

### Cypress tests

- updating, in Vue app 100, 101

## D

Document Object Model (DOM) structure 55

DVCS (distributed version control system) 157

dynamic segment 122

## E

### end-to-end tests

- writing, with Cypress 97

### ESLint configuration

- setting up, in Vue CLI 3 GUI 137, 138

### ESLint

- .eslintrc.js config file structure 133
- about 124
- configuring 125, 126
- configuring, in Vue CLI 3 136
- error level rules, updating 135
- init command 131, 132
- installing globally 130, 131
- running 131
- running, on JavaScript file 134, 135
- using, with Prettier, in Vue CLI 3 UI project 139, 140, 141
- working with 129, 130

## F

fatigue level zero 16

## G

### Git

- branches, merging 162
- branching 161
- changes, committing to app 160
- project, tracking with 158
- three trees concept 161
- working 157

### GitHub account

- registering 163, 164

### GitHub Desktop app

- installing 156

### GitHub Desktop

- installing 156
- local repository, publishing 165, 166, 167
- used, for adding origin/master 164
- using 164

### GitHub Pages

- about 167
- project, publishing on 167

### global packages 125

## H

### Hot Module Replacement (HMR)

- about 24, 54
- updates, observing 55, 56, 57

## I

### IIFEs

- about 26
- issues 26

## J

### jQuery library

- URL 26

### JS language

- evolution 25
- IIFEs 26
- JavaScript task runners 30, 31
- modules 32
- NPM scripts 30, 31
- script tag 25
- third-party libraries, sharing in team environment
  - with NPM 27, 28, 29, 30

## L

### Linux

- URL 27
- Long-term Support (LTS) 10

## M

### main.js file

- app, serving in production mode 114, 115
- App.vue file 112
- build task, running from UI 114
- contents 112
- router-link navigation 112
- Vue instance, rendering 113

### module bundler 34, 35

### modules, JavaScript

- about 32
- module bundlers 34

### modules

- working with, in Node.js 32, 33, 34

## N

### Node Package Manager (NPM) 27

### Node Version Manager (NVM)

- installing, on Windows 9, 10
- need for installation 9
- used, for installing Node.js 9

### Node.js

- installing, with NVM 9
- NPM package runner 129

### npm packages

- global installation, versus local installation 125

### NuGet

- URL 27

## P

### parallelization 62

### Parcel

- URL 35

### preset options

- configuring, through VS Code's command line 105

### Prettier

- about 126
- index.html, running 128

- installed npm modules, running locally on
  - command line 126, 127
- installing globally 128
- running, with npx 129
- URL 126

## R

- red-green-refactor cycle 95
- rollup.js
  - URL 35
- router-view tag
  - about 116
  - dynamic routes, working with 122
  - lazy loading routes 122
  - named routes, using 116
  - nested routes, working with 119, 120
  - route, navigating by triggering method 117, 118, 119
  - testing 116
- routes, in vue-router
  - default routes, inspecting 110, 111
  - project files, inspecting 111, 112
  - working with 110
- routes
  - basic of working 115
- Ruby
  - URL 27

## S

- Sassmeister
  - URL 149
- SCSS
  - bootstrap-vue styles, overriding 148
  - errors, dealing with 146
  - interpolation, using 149, 153
  - mixins, using 149, 152
  - practicing, Sassmeister used 149, 151
  - using, in project 146
  - variables, using 149
  - writing, in project 147
- shorthand function 74

## T

- test-driven development (TDD) 75
- test-driven development, Vue CLI 3 95

- tests, Vue CLI 3
  - improving 95
  - watching 96, 97
- this keyword
  - issues 71
  - one counter app, adding 71, 72, 73, 74
  - recommended solutions 71
- three trees concept 161
- toolchain 16

## V

- VS Code
  - installing 12
- Vue app testing
  - button click, triggering 93, 94
  - code, adding 84, 85
  - component, mounting 91
  - failing assertion, writing 92
  - Jest plugin, adding 79, 80, 81, 82
  - mount, importing from test-utils 90
  - passing assertion, writing 93
  - performing 76
  - project, adding with Vue CLI UI 76, 77, 78, 79
  - unit test, running inside Vue CLI UI 87, 88, 89, 90
  - unit test, writing 91
  - unit test, writing with Jest 86
  - updates, examining 83, 84
- Vue app, publishing on GitHub Pages
  - about 167
  - changes, pushing to remote repository 173
  - changes, saving 172
  - GitHub pages deployment, debugging 174
  - preparing 167, 168, 169, 170, 171
  - Vue configuration file, updating 171
- Vue CLI 3 app
  - creating, via command line 16, 17, 18
  - creating, via UI 18, 20, 23
- Vue CLI 3
  - benefits 16
  - installing 10, 11
  - setting up 8, 9
  - test-driven development 95
  - webpack 24
- Vue CLI

- using, with no configuration 12, 13, 15

Vue plugins 75, 76

Vue project

- .vue file, adding as JavaScript module 49, 50
- adding, via NPM 44
- adding, with with vue-router and vuex 103, 104
- changes, tracking with Git 158
- configuration 107
- ESLint and Prettier, adding 107
- HTML file, adding 47
- HTML file, outputting 48, 49
- HTML files, delivering via webpack 53, 54
- index.html file issue, fixing 52, 53
- JavaScript modules, compiling with webpack 46
- options, adding with webpack configuration file 47
- plugins, installing 108
- serve task, running from UI 109, 110
- serving, from Vue CLI UI 108, 109
- Vue component, adding as JavaScript module 45, 46

- vue-router and vuex, adding 106
- webpack loader, adding with .vue files 50, 51, 52
- webpack, using on it 45

vue-router plugins 61

Vue

- Hot Module Replacement 54

vuex 61

## W

webpack loader 50

webpack

- about 35, 36, 37
- and Babel, using in Vue 2 without Vue CLI 63
- project, building 37
- running, on project 38, 39, 40, 41, 42, 43, 44
- URL 35
- working 37

Windows

- NVM, installing 9, 10