THIS DOT

A COMPLETE GUIDE TO VueJS

BILAL HAIDAR

The Ultimate Guide to Learn Vue.js

By Bilal Haidar

About This Dot Labs

Founded in 2016, This Dot Labs is a modern web consultancy focused on helping companies realize their digital transformation efforts. For expert architectural guidance, training, or consulting in React, Angular, Vue, Web Components, GraphQL, Node, Bazel, or Polymer, contact us today at hi@thisdot.co.

Press Contact

Tracy Lee
CEO, This Dot Labs
thisdotlabs.com
408-506-9660

Table of Contents

Introduction

From the onset, I'd like to make it clear that this is not a tutorial, but a comprehensive resource on Vue.js for anyone wanting to deeply understand this framework. It will contain detailed information on internal structures, complementary frameworks, components, and how to integrate everything to make Vue.js work best for you. Those who are familiar with my articles, and follow my work, may expect me to be writing a step-by-step tutorial, but this guide will feel very different from my typical publications.

This guide will help you begin your journey of learning and coding in the Vue.js framework- specifically version v2. I won't assume any prior knowledge in Vue.js or JavaScript, and I will provide the resources you need to learn it in order to develop stunning Web apps.

It's preferable, however, that you have some experience in, or basic knowledge of HTML and CSS.

The guide is lengthy, and contains many external links to supplement your learning. However, you should feel free to go from section to section, or chapter to chapter as you need, or as your experience allows.

Section 1: Learning JavaScript

Considering JavaScript is the cornerstone of building Web apps, it is crucial to know it well before attempting to learn Vue.js, or any other JavaScript Framework like React or Angular. In short, JavaScript is a programming language with a just-in-time compilation, and adheres to the scripting language specifications known as ECMAScript.

I've come across hundreds of articles, videos, and books on this topic, many of which are remarkable. Before beginning my own guide, I feel compelled to share them with you.

Free JavaScript resources

- Learning JavaScript - Full course (https://www.youtube.com/watch?v=PkZNo7MFNFg) by freeCodeCamp.org (https://www.freecodecamp.org/learn)

- The Modern JavaScript Tutorial - A comprehensive guide to learning JavaScript, from fundamental to advanced topics, by JavaScript.info (https://javascript.info/)

- JavaScript Tutorial - A complete reference on the JavaScript language by w3schools.com (https://www.w3schools.com/js/). I always keep this reference by my side in case I forget a function or object name. It will even show you how to use certain functions in JavaScript.

- Learn ES6 [EcmaScript 2015] - A free resource to learn ES6 by FreeCodeCamp (https://www.freecodecamp.org/news/want-to-learn-es6-take-this-free-23-part-course-and-become-a-javascript-ninja-55002db1ff74/).

- Learn ES6 [EcmaScript 2015] - A free resource to learn ES6 by FreeCodeCamp (https://www.freecodecamp.org/news/want-to-learn-es6-take-this-free-23-part-course-and-become-a-javascript-ninja-55002db1ff74/).

Paid JavaScript resources

- The Modern JavaScript Bootcamp - A complete reference to learning JavaScript by Andrew Mead (https://www.udemy.com/course/modern-javascript/). By far, this is my favorite author when it comes to learning JavaScript.

- Modern JavaScript FromThe Beginning - by Brad Traversy (https://www.udemy.com/course/modern-javascript-from-the-beginning/). Brad always has the latest free video tutorials on his YouTube channel (https://www.youtube.com/channel/UC29ju8bIPH5as8OGnQzwJyA).

- JavaScript: Understanding the Weird Parts - An advanced course by Anthony Alicea (https://www.udemy.com/course/understand-javascript/). It provides an in-depth look at Javascripts internals, and how the engine executes functions and programs.

For those who prefer reading a physical book, here are my top recommendations:

- Web Design with HTML, CSS, JavaScript and jQuery Set (https://www.amazon.com/Web-Design-HTML-JavaScript-jQuery-dp-1118907442/dp/1118907442/ref=mt_paperback?_encoding=UTF8&me=&qid=)

- Eloquent JavaScript (https://eloquentjavascript.net/)

- You Don't Know JS Book Series (https://www.amazon.com/gp/bookseries/B01N9EBP9V?ref_=dbs_m_mng_rwt_0000_ft)

Choose whatever resource(s) you want. Make sure you have grasped some basic knowledge of JavaScript before you proceed with learning Vue.js.

Section 2: Learning Vue.js

Vue.js is a progressive framework for building Web apps. It is a layer on top of JavaScript, offering services and development paradigms to facilitate app building. Evan You created this framework, initially released it in 2014,, and is currently maintaining it with the help of a core team.

The Vue.js team has done a great job of producing thorough documentation (https://vuejs.org/v2/guide/), which I personally used when I first started learning about Vue.js. The documentation is well maintained, and is updated with every new release.

There are a ton of available online resources to help you learn Vue.js. Here's the list of courses and books I highly recommend:

Free Vue.js resources

- Vue JS 2 Tutorial (https://www.youtube.com/playlist?list=PL4cUxeGkcC9gQcYgjhBoeQH7wiAyZNrYa) by The Net Ninja. This channel is another essential resource. He offers high-quality courses on many other topics as well.

- Learn Vue.js for free (https://scrimba.com/g/glearnvue) by Scrimba. Scrimba offers a large catalog of uniquely interactive courses. While watching a video course, you can pause it with a single mouse click, and *voila*, you have entered a live coding editor to practice what you are learning.

Paid Vue.js resources

- Vue JS 2 - The Complete Guide (https://www.udemy.com/course/vuejs-2-the-complete-guide/). The most comprehensive course on Vue.js by Maximilian Schwarzmüller. It took me weeks to finish this course!

- Vue Mastery (https://www.vuemastery.com/courses). This is a subscription-based set of courses that fully cover Vue from the beginner level to advanced concepts.

Make sure that you check out all, or some, of the above courses before continuing with the rest of this guide. Practice, practice, and then practice some more. Understanding theory is only one part of the journey. You really get to know all of the framework's aspects when you are willing to jump in and get your hands dirty.

Section 3: Code Editors

When it comes to code editors, there are a bunch of excellent ones out there. On one hand, it's important to pick the right code editor based on the features and plugins, and on the other, personal preference plays a role too.

Here's a selection of the most widely used code editors for Vue.js and JavaScript:

- Visual Studio Code (https://code.visualstudio.com/)

- Bracket (http://brackets.io/)

- Atom (https://atom.io/)

- Sublime (https://www.sublimetext.com/)

These code editors are also used as normal text editors. Find one that best suits you.

For instance, VS Code has the Vetur (https://marketplace.visualstudio.com/items?itemName=octref.vetur) extension. I cannot imagine developing Vue.js apps without installing this plugin. This extension offers the following features, and more:

- Syntax-highlighting
- Snippet
- Emmet
- Linting / Error Checking
- Formatting
- Auto Completion
- Debugging

To install Vetur extension, follow the basic steps below as shown in **Figure 1**:

1. Open VS Code
2. Click the Extensions menu
3. Type `vetur` and hit Enter
4. Once located, click the orange `Install` button

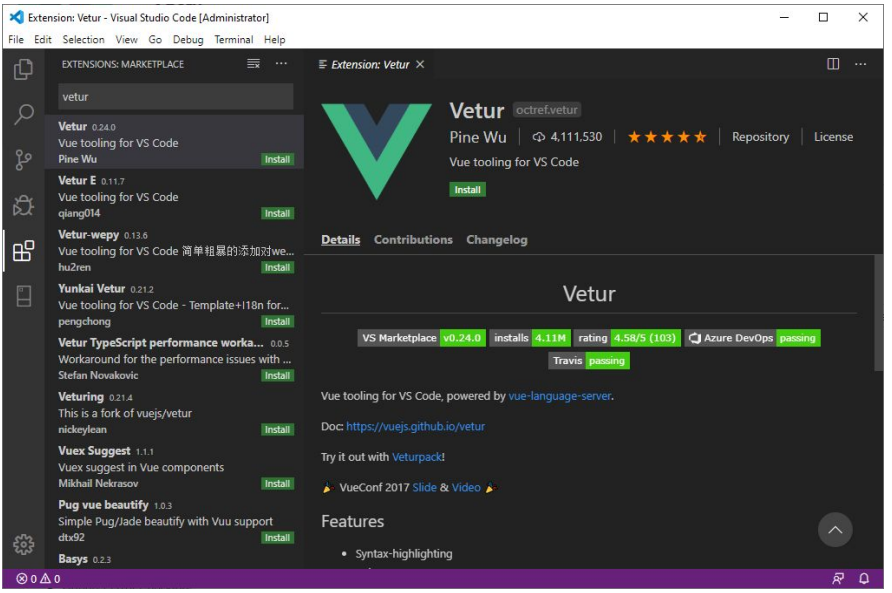

Figure 1: Install Vetur plugin inside Visual Studio Code

Section 4: Vue CLI

The Vue CLI (https://cli.vuejs.org/) is provided and maintained by the Vue.js team. It allows you to start rapidly developing with Vue.js, by taking care of creating the Vue.js project structure for you. It comes in two flavors: Command Line CLI, and Graphical UI CLI.

The CLI supports out-of-the-box Babel, TypeScript, ESLint, PostCSS, PWA, Unit Testing, and End-to-end Testing. It has a built-in plugin system that allows developers to extend its functionality by developing their own plugins. Moreover, the CLI is highly configurable, which means that there is no need to eject its WebPack (https://webpack.js.org/) configuration when a developer wants to customize it further.

The CLI requires Node.js version 8.9 or above (8.11.0+ recommended). Make sure it's installed on your computer before you proceed.

It's also recommended that you install Git (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git) on your computer. Soon, we will explore this topic more deeply.

Installing the CLI

To install the Vue CLI, run the following command on a Terminal or Windows Command Line:

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

You can check the version of the installed CLI by running the following command:

```
vue --version
```

Creating first project

To create a new Vue.js project, run the following command:

```
vue create vue-first-app
```

The CLI asks a few questions to help you customize your app according to your requirements. For the scenario at hand, I will select the following options:

First, select the manual option to allow you to customize the features of this app as shown in **Figure 2**:

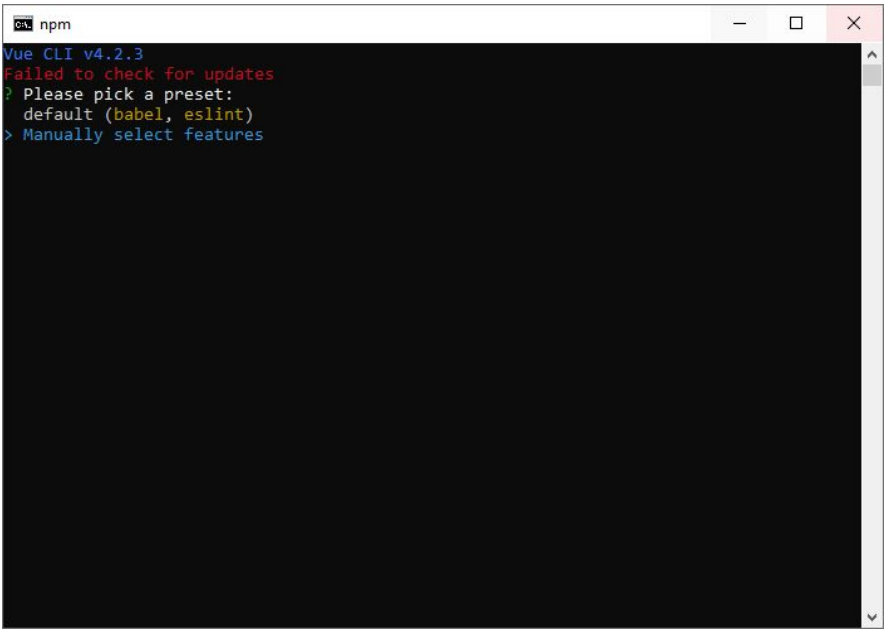

Figure 2: Installing Vue CLI - Default option

Next, select all the options shown in **Figure 3**:

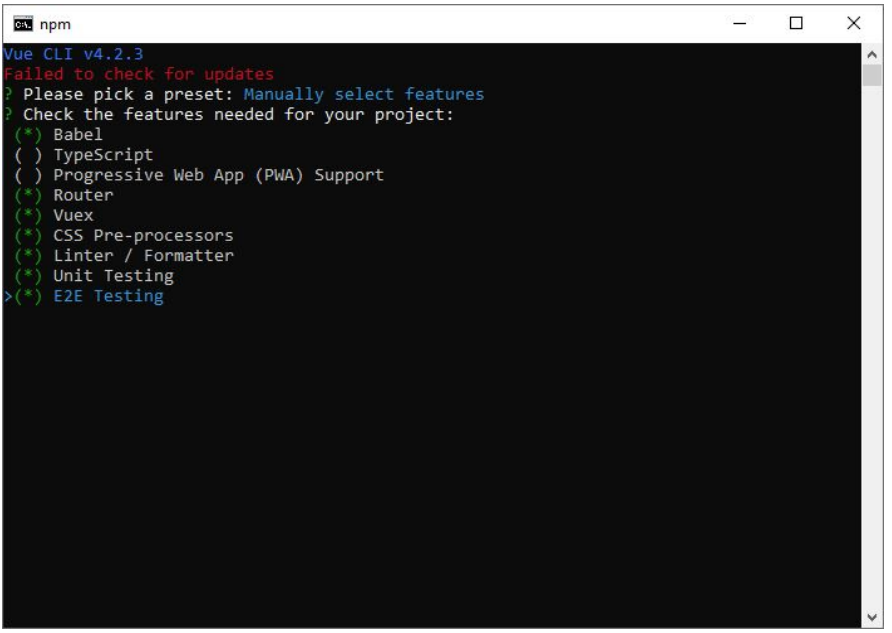

Figure 3: Vue CLI installation - select options

Having `Babel` installed, and configured, allows you to write JavaScript ES6 code.

We will cover the following topics in-depth as we go along, or as they come up.

The `Router` option adds routing capabilities to the app.

The `Vuex` option adds state management capabilities to the app.

The `CSS Pre-processors` option adds capabilities to use Sass, Less, and Stylus, in the app.

The `Linter/Formatter` option adds support for both ESLint, and Prettier, in the app.

The `Unit Testing` and `E2E Testing` options add testing capabilities to the app.

Once done with your selections, hit the {Enter} key on your keyboard.

Type `n` to use the history mode for the router as shown in **Figure 4**:

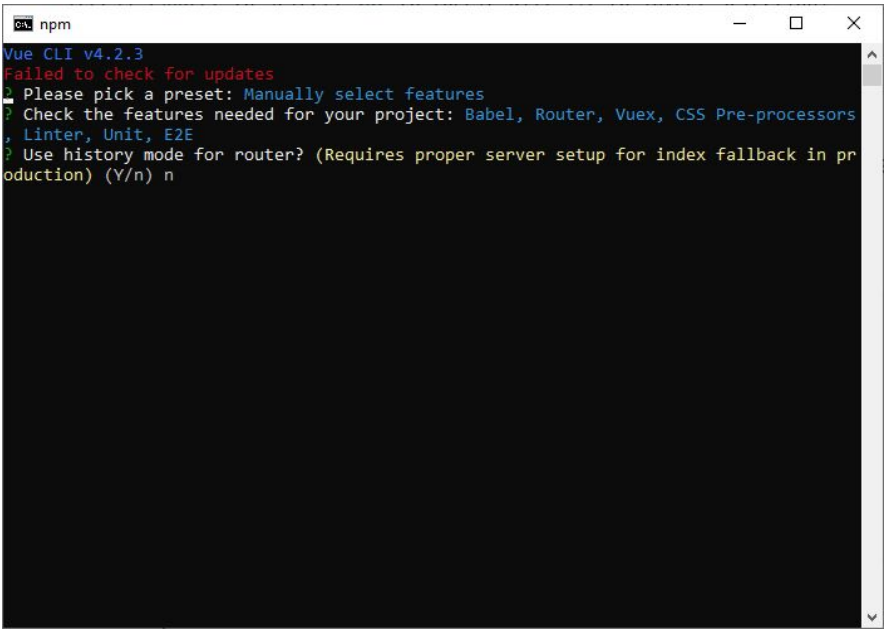

Figure 4: Vue CLI installation - router

Hit the {Enter} key on your keyboard.

Select the `Sass/SCSS (with node-sass)` option for the CSS pre-processor, as shown in **Figure 5**:

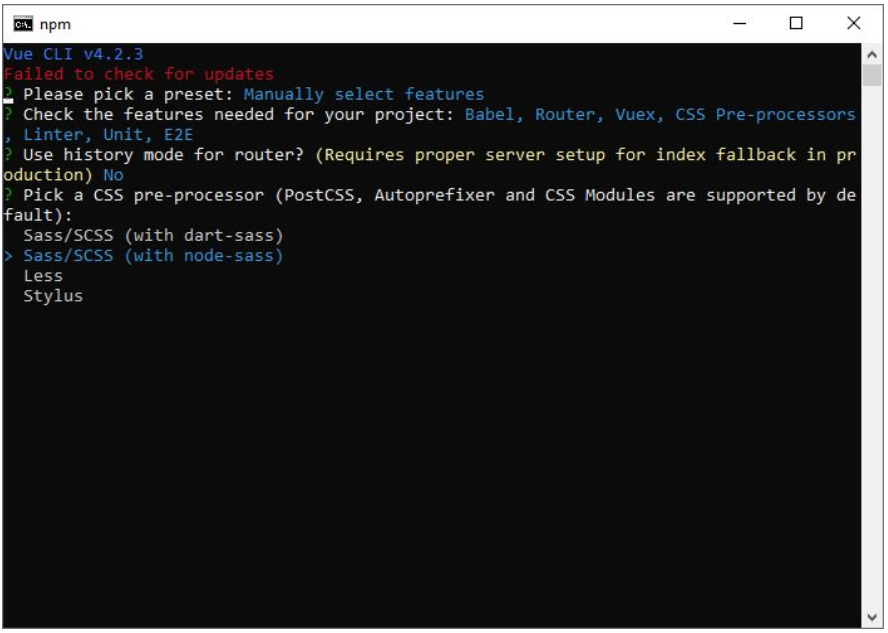

Figure 5: Vue CLI installation - Sass

Hit the {Enter} key on your keyboard.

Select the `ESLint + Prettier` option for the linter/formatter config as shown in **Figure 6**. This enables you to use, and benefit from, both linting, and formatting for your code:

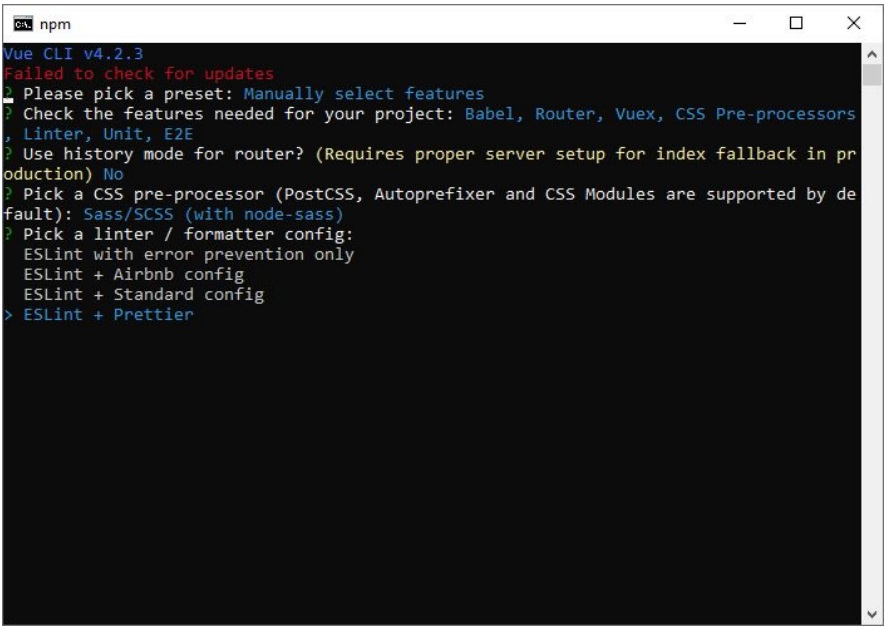

Figure 6: Vue CLI installation - ESLint + Prettier

Hit the `Enter` key on your keyboard.

Select the `Lint on save` option for the additional lint features, as shown in **Figure 7**. When you save your code files, the CLI runs the linter, and formatter, on any saved file. It will immediately report any problems on the command line / terminal. This is the easiest option to use when starting with Vue.js. Later on, I recommend using the `Lint and fix on commit` option. The linter, and formatter, in this case, will only check the commit payload, will report the problems, and will fix any formatting discrepancies. This is much faster than checking all of the project files!

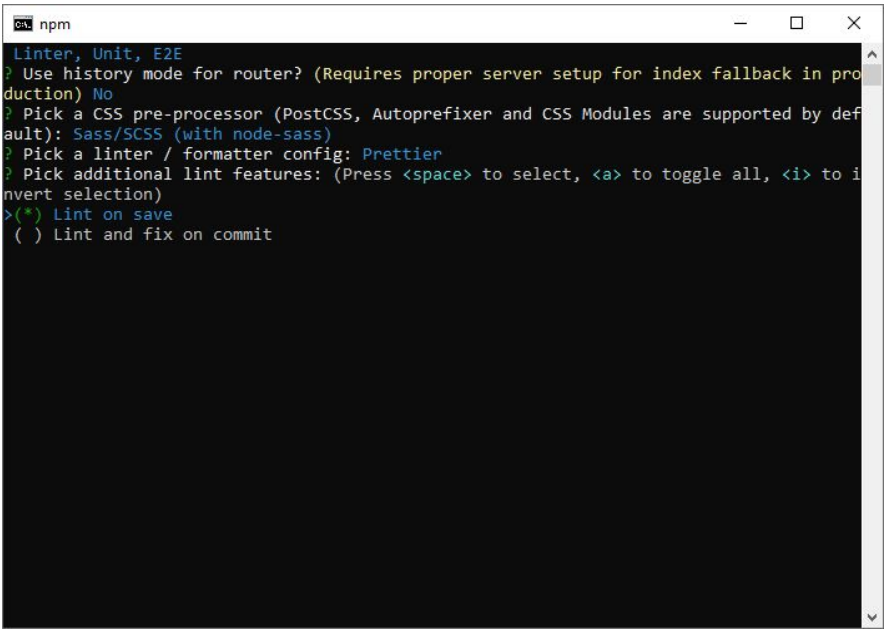

Figure 7: Vue CLI installation - Linting

Hit the {Enter} key on your keyboard.

Select the `Jest` option for the unit testing solution as shown in **Figure 8**. You may choose another option, like `Mocha and Chai`. However, in this guide, I will only cover Jest:

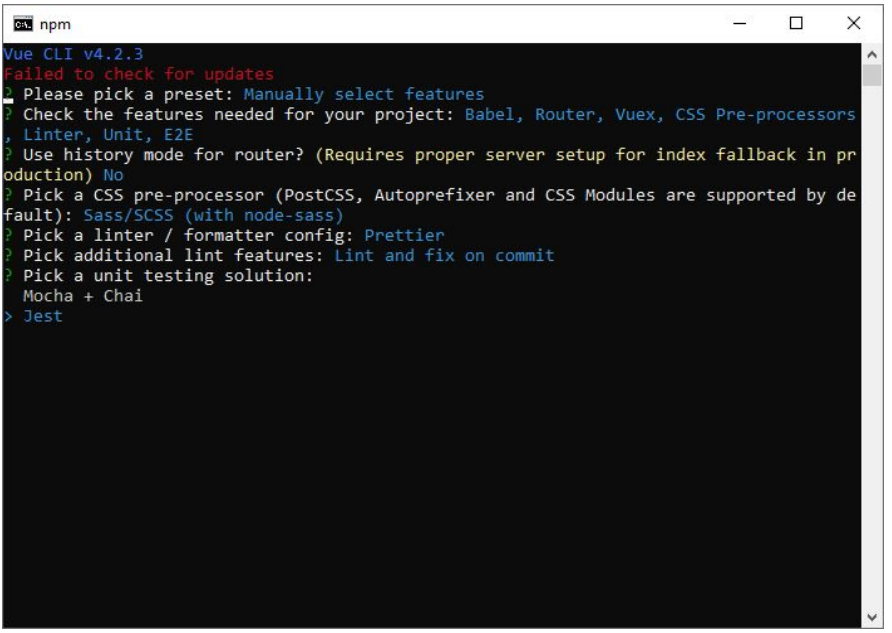

Figure 8: Vue CLI installation - Jest for unit testing

Hit the {Enter} key on your keyboard.

Select the `Cypress` option for the E2E testing solution as shown in **Figure 9**. Cypress is the most widely used framework for building and running End-2-End tests. We will cover this topic, in more depth, down the line.

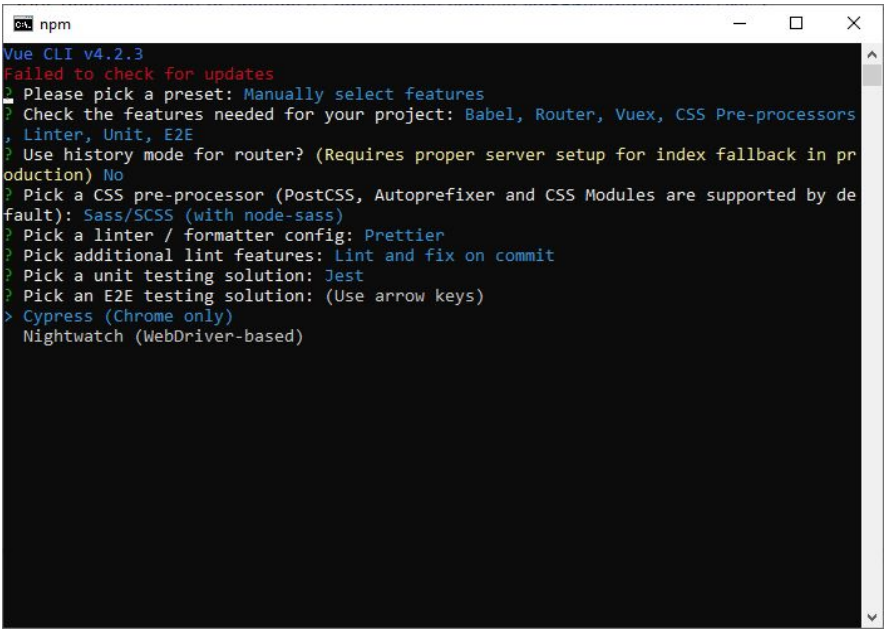

Figure 9: Vue CLI installation - Cypress.io for E2E testing

Hit the {Enter} key on your keyboard.

Select the `In dedicated config files` option to save all the configuration settings inside of their own config files as shown in **Figure 10**. This is better for separating concerns, and keeping each service configuration isolated from the other.

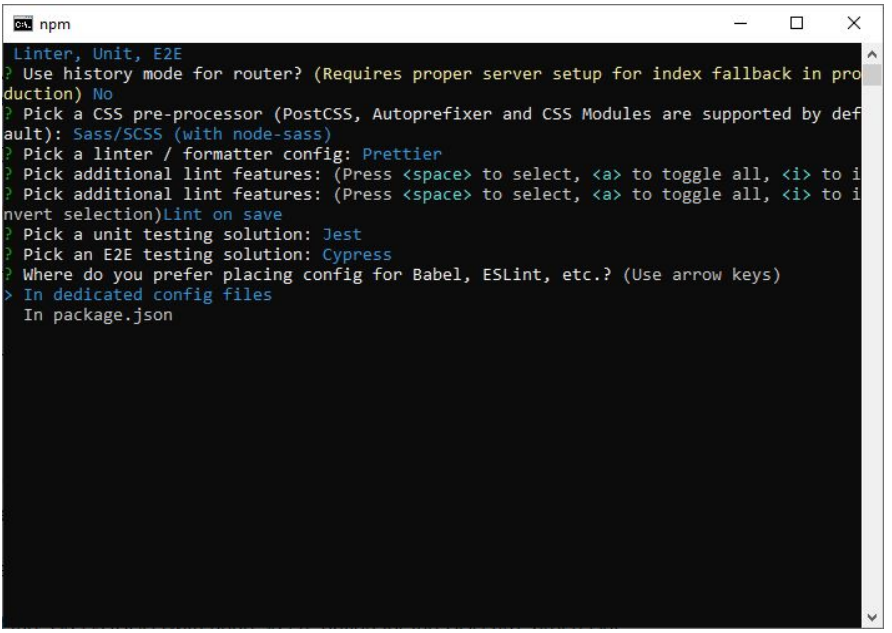

Figure 10: Vue CLI installation - save configuration

Hit the {Enter} key on your keyboard.

Finally, type `y` to save all the settings as a template. This can be reused later to create more Vue.js projects as shown in **Figure 11**. This saves you the time of choosing the same options again and again.

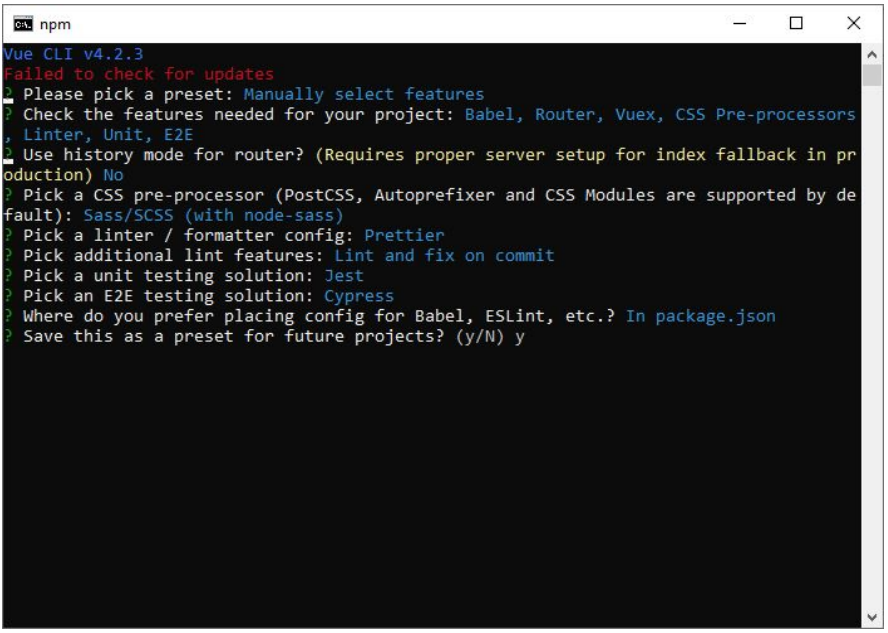

Figure 11: Vue CLI installation - save as a preset

Hit the {Enter} key on your keyboard.

You will be prompted to name this new template. I've used `best_vue_settings` as its name as shown in **Figure 12**.

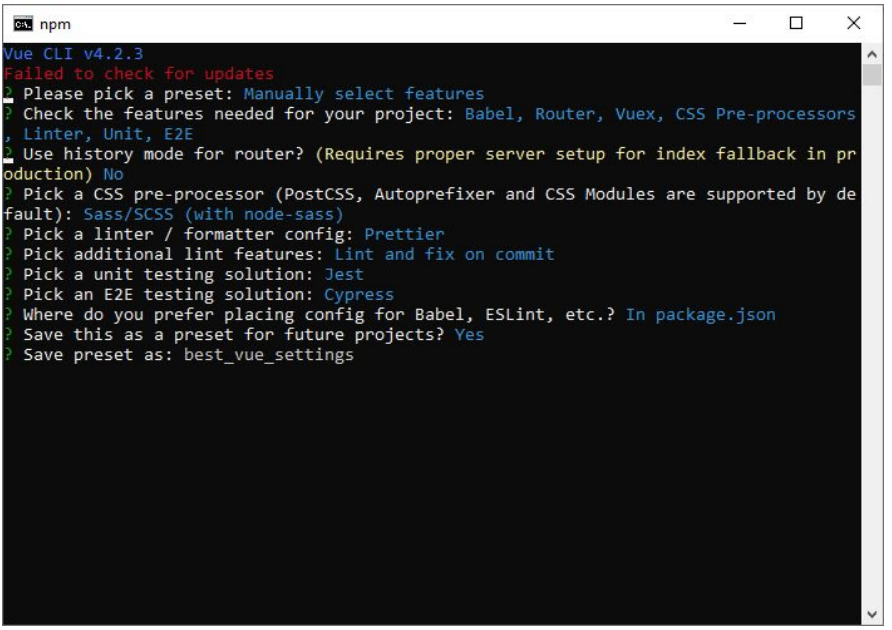

Figure 12: Vue CLI installation - naming the preset

Hit the `Enter` key on your keyboard.

The Vue CLI starts scaffolding your project, taking into consideration all of your options, while running the CLI.

The CLI takes a few minutes before readying your project for use.

Now that the project is created, navigate to the app folder, and run it using the command as shown in **Figure 13**.

```
cd vue-first-app
npm run serve
```

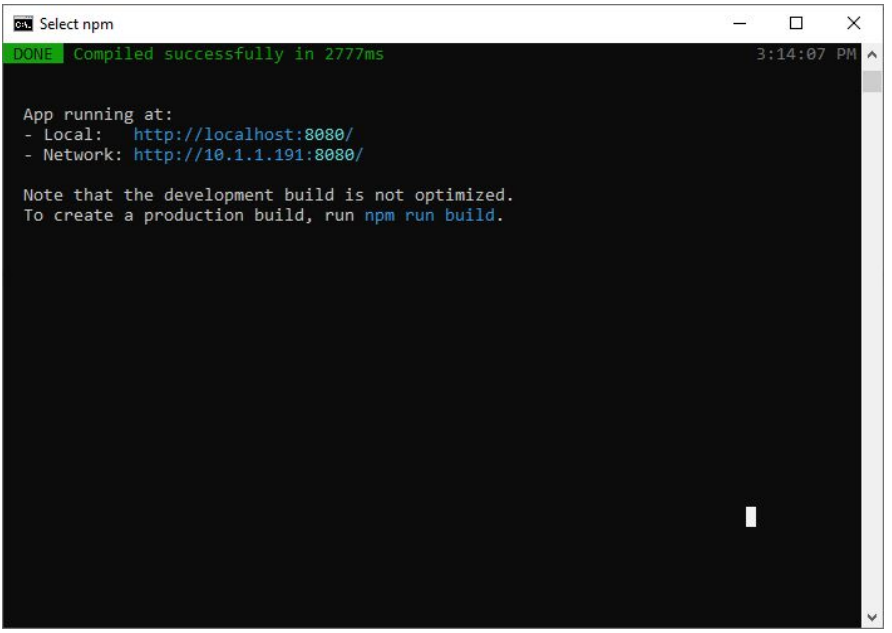

Figure 13: Running the Vue.js app

Now that the app is running, view it live in a browser, by following the steps:

1. Open a new browser instance. You can select Google Chrome, Microsoft Edge, Firefox, etc.

2. Paste the url `http://localhost:8080/`

3. Hit the {Enter} key on your keyboard to open the app.

Figure 14 shows the app running inside of a browser.

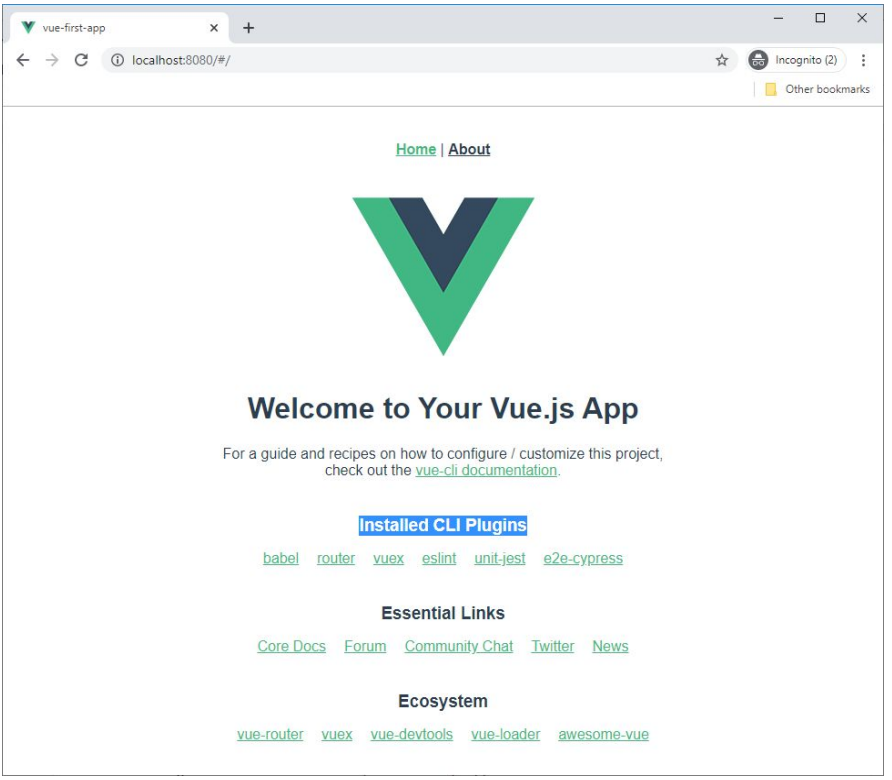

Figure 14: Vue.js app running inside a browser

Congratulations! You've created your first Vue.js app.

Section 5: ESLint and Prettier

ESLint is a static analysis tool, and more specifically, a linting tool. It looks through your code, and helps you identify, and report, problematic code patterns, and enforces code style guidelines.

For instance, if you forget to add a semicolon at the end of a JavaScript statement, ESLint will report this as a problem.

These problems don't stop the code from running. However, they don't adopt certain code styling conventions for readability, and maintenance of the code, either.

ESLint is highly flexible and configurable. Every aspect can be adapted to fit a project's needs. In addition, you can write your own ESLint rules, and use hundreds of ready-made rules.

Prettier, on the other hand, is an opinionated code formatter. When it runs, it formats your code with proper spacing and indentations.

ESLint and Prettier integrate together, so Prettier problems can be reported as ESLine problems.

When we created our first Vue.js app before, we picked `ESLint + Prettier` as our linters for the app. Vue.js has its own set of ESLint rules already baked into the Vue CLI.

Let's see what the CLI has created for us with regard to ESLint and Prettier.

Locate the *.eslintrc.js* file at the root of the app folder. This file contains the ESLint and Prettier configuration settings for this app.

```
module.exports = {
  root: true,
  env: {
    node: true,
  },
  extends: [
    "plugin:vue/essential",
    "eslint:recommended",
    "@vue/prettier",
  ],
  parserOptions: {
    parser: "babel-eslint",
  },
  rules: {
    "no-console":
      process.env.NODE_ENV === "production" ? "error" : "off",
    "no-debugger":
      process.env.NODE_ENV === "production" ? "error" : "off",
  },
  overrides: [
    {
      files: [
        "**/__tests__/*.{j,t}s?(x)",
        "**/tests/unit/**/*.spec.{j,t}s?(x)",
      ],
      env: {
        jest: true,
      },
    },
  ],
};
```

This configuration file includes a set of ESLint plugins. Typically, every plugin contains a defined set of ESLint custom rules to apply to JavaScript files, or any other file extension.

For instance, the `plugin:vue/essential` is the official ESLint plugin for Vue.js. It allows us to check the `<template>`, and `<script>`, of .vue files with ESLint.

The `eslint:recommended` represents the set of recommended ESLint rules to use.

Finally, the `@vue/prettier` plugin is the ESLint plugin for Prettier formatting for Vue Single File Components(SFC).

ESLint is highly configurable and accommodates the needs of any app. By default, the ESLint parser is compatible with ECMAScript 5 syntax. However, Vue.js components, and objects, are written with Babel. Therefore, it adds the `babel-eslint` as the default parser for this app.

VS Code integration

ESLint and Prettier both run, and report results, on the command line / terminal. Therefore, to let VS Code know about their results, we need to install two more VS Code extensions.

Open VS Code, and locate the Extensions page. Search for both ESLint, and Prettier extensions, and install them.

While inside VS Code, go to Preferences, and then click Settings. Make sure you select the Workspace tab, and click the Open Settings(JSON) button located at the top right side.

Paste the following JSON content:

```
{
    "eslint.validate": [
        "vue",
        "javascript",
    ],
    "editor.formatOnSave": false,
    "vetur.validation.template": false,
    "editor.codeActionsOnSave": {
        "source.fixAll.eslint": true
    }
}
```

The Workspace Settings specifies the following:

- Enable ESLint for Vue, and JavaScript code files
- Disable the `editor.formatOnSave` setting
- Disable the vetur validation for the Vue.js `<template>`. In this case, vetur will only validate the `<style>`, and `<script>`.
- Allow ESLint to auto fix any errors on `editor.codeActionsOnSave`. Formatting usually relates to inserting and removing whitespace, while a source code action removes unnecessary imports, unused variables, etc.

You have just integrated ESLint and Prettier with VS Code. Start enjoying an incredible experience of allowing VS Code to lint and format your code automatically!

Section 6: Source Code Repository Management System

GitHub (https://github.com/), BitBucket (https://bitbucket.org/), GitLab (https://about.gitlab.com/), and Azure Repos (https://azure.microsoft.com/en-us/services/devops/repos/) are just four popular source code repository management (SCRM) systems out there. If you don't already use them, I highly recommend you start.

With a SCRM system in place, you and your team members can all share access to the app source code at any time you wish.

For example, my colleagues and I might add or modify code, and push the changes to the centralized repository. That way, developers joining the team will clone the centralized up-to-date repository, and can start coding instantaneously.

A SCRM also contains best practices and workflows. For instance, **Figure 15** depicts the workflow of implementing a new feature in your application.

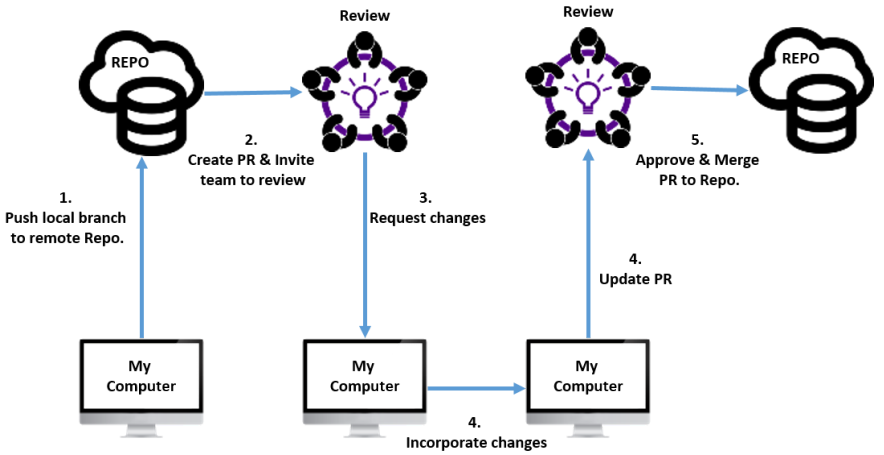

Figure 15: A typical SCRM workflow

Start by cloning or branching to the `master`, or `dev`, branch.

1. Create a new branch name relevant to the feature you are adding.
    a. Add your code, and commit the changes locally.

    b. Push the local branch to a SCRM system- GitHub for instance.

2. Create a Pull Request based on the new branch, and invite your team members to review and comment.

3. The team Requests changes on the code

4. Incorporate the changes and Update PR code with the latest

5. The team Approves & Merges PR into either the `master`, or `dev`, branch on your repo.

The above is a typical SCRM Workflow that I use on a daily basis. It's quite simple, and straight to the point, yet organizes the development process with all team members without losing code, or overlapping.

Learning Git

Git (https://git-scm.com/) is at the core of learning and using a SCRM system. It's an open source distributed version control system. GitHub and friends are all based on Git. They only differ in the User Interface, capabilities and services, and, of course, pricing. Once you learn Git, you can use any of these SCRM systems. There are many resources to help you learn Git. However, here's a set of resources I've used before:

- Git Tutorial for Beginners - Crash Course (https://academind.com/learn/web-dev/git-the-basics/) by Maximilian Schwarzmüller

- Learn Git Branching (https://learngitbranching.js.org/)- an interactive tutorial on Git branching.

- Git and GitHub for Beginners (https://www.youtube.com/playlist?list=PL4cUxeGkcC9goXbgTDQ0n_4TBzOO0ocPR) by Net Ninja tutorials.

- Git tutorial (https://www.youtube.com/playlist?list=PL3Sf6euafegw5C-V4zMCmsVl7GjiXNV0E). This tutorial vividly illustrates the concept of Git Rebase, a Git feature I use daily.

- Git Tutorial & Reference (https://www.atlassian.com/git/tutorials/what-is-git) by Atlassian BitBucket (https://www.atlassian.com/git/tutorials)

Push your code to GitHub

Vue CLI automatically initializes a new repository upon a new app's creation. It automatically detects that Git is locally installed on the computer, and initializes a local Git repository.

To prove that, navigate to the app root folder, open a Command Line / Terminal, and run the following command:

```
git status
```

The result is:

```
On branch master
nothing to commit, working tree clean
```

The message says you are on `master` branch, and nothing to commit. In other words, you haven't changed anything that Git needs to track.

Let's go to Github, and create a new repository. If you don't have an account on GitHub, you need to register first.

First, login to GitHub, and then navigate to creating a new repository (https://github.com/new).

Fill in the details of the new GitHub repository as shown in **Figure 16**:

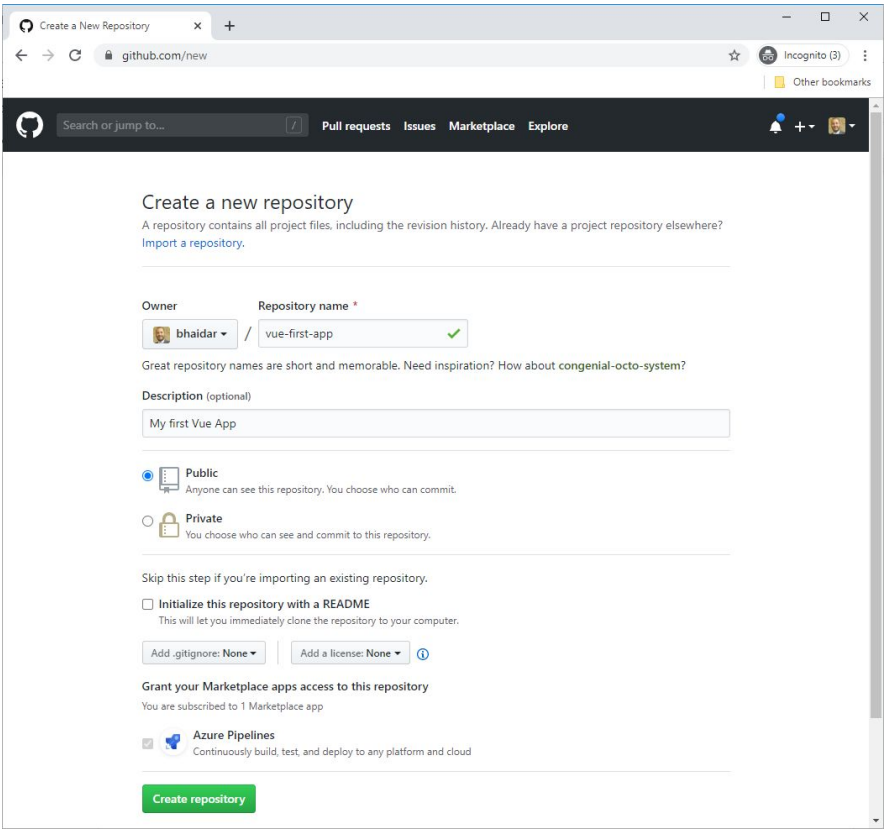

Figure 16: Create a new GitHub repository

Click the Create repository button. GitHub allows the creation of either public or private repositories.

GitHub then takes you to the repository info page as shown in **Figure 17**:

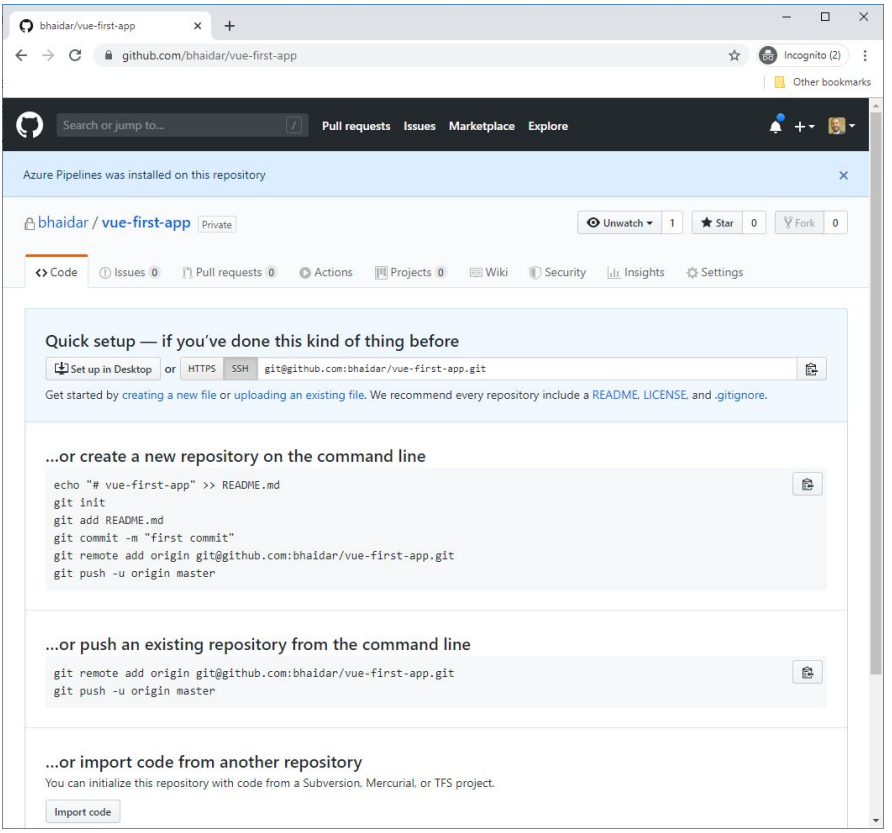

Figure 17: GitHub repository home page

Some important facts worth stopping for on this page:

Repository Url: https://github.com/bhaidar/vue-first-app.git and git@github.com:bhaidar/vue-first-app.git. Both HTTPS, and SSH Urls, are provided.

To connect your local Git repository to the GitHub repository, you need to follow one of these two guidelines:

If you haven't yet created your local Git repository, then you need to follow the steps below:

```
…or create a new repository on the command line (change an origin url to your own)
echo "# vue-first-app" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin git@github.com:bhaidar/vue-first-app.git
git push -u origin master
```

If you have created a local Git repository (which is our case now), you need to follow the steps:

```
…or push an existing repository from the command line (change an origin url to your own)
git remote add origin git@github.com:bhaidar/vue-first-app.git
git push -u origin master
```

Switch back to the Vue app, navigate to the app root folder, open a Command Line / Terminal, and run the following command:

```
git remote add origin git@github.com:bhaidar/vue-first-app.git
```

This command connects your local Git repository to the `remote` branch on GitHub.

```
git push -u origin master
```

This command pushes the local `master` branch to GitHub.

Now, you can go to another computer, pull down the repository, and start working on it.

With every new feature you build into your app, follow the Git Workflow explained above to deal with Git and GitHub.

Section 7: CSS Frameworks

Working with CSS has never been pleasant. CSS, itself, is a very powerful tool for beautifying your Web apps, and laying out the content on your pages. However, things become complicated when you start considering building adaptive and responsive apps to render on different mobile

and desktop devices. In addition to that, one must write CSS to run across multiple and different browsers, hence maintaining cross-browser compatibility is the hardest by far.

That’s why CSS frameworks like Bootstrap, Tailwind, and the like, have emerged to help developers build rich, adaptive, cross-browser User Interfaces more quickly. In this guide, I will focus on Bootstrap and Tailwind only.

Bootstrap is, by far, the most popular HTML, CSS, and JS framework for developing responsive, mobile-first projects on the web. Tailwind is different from Bootstrap in that it's not a UI kit. Instead, it’s a utility-first CSS framework for rapidly building custom designs. It doesn't have a default theme, and there are no built-in UI components. It comes with a set of predesigned widgets with which to build your site.

Bootstrap is a more opinionated framework than Tailwind. The major differences between the two are:

- Preprocessors: Bootstrap ships with both vanilla CSS, and the source code that’s built in CSS preprocessors, and Sass. You can use the precompiled CSS right away, or build on the source.

- One framework, every device: Bootstrap is well-known for making your website adaptive, and cross-browser, with a single code base, using CSS media queries. Bootstrap is well known for the Grid system to build layouts.

- Full of features: Bootstrap offers extensive, and beautiful documentation for common HTML elements, dozens of custom HTML and CSS components, and awesome jQuery plugins.

On the other hand, Tailwind CSS provides low-level utility classes that let you build completely custom designs. It’s way different from Bootstrap in terms of providing:

- No default theme

- No built-in UI components

- No opinion about how your site should look

Now that you know what both frameworks can offer you, I will leave it up to you to choose which one with which to work. However, I will show you how you can configure both frameworks in your Vue app.

Using Bootstrap

You can find the source code used in this section under the Git branch named *using-bootstrap*.

If you are going to use the Bootstrap styles only, then run the following command:

```
npm install bootstrap
```

The command installs the Bootstrap NPM package.

If you want to use Bootstrap's built-in UI components, then you need to install two more NPM packages.

```
npm install bootstrap jquery popper.js
```

The jQuery, and Popper.js packages, are required for the UI components to function properly.

Next, open the *src\main.js* file in your Vue app, and add the following line of code:

```
import "bootstrap/dist/css/bootstrap.min.css";
```

If you want to use Bootstrap's built-in UI components, then you need to add an additional line of code:

```
import "bootstrap";
import "bootstrap/dist/css/bootstrap.min.css";
```

That's all! Now you can start Bootstrap in your Vue app.

Using Tailwind

You can find the source code used in this section under the Git branch named *using-tailwind*.

Tailwind is packaged as a PostCSS plugin. To start using it in your Vue app, you first need to install the plugin, and then instruct Vue to use this plugin.

In brief, PostCSS transforms the CSS you write in your app into a CSS that browsers can understand, using JavaScript plugins.

One of the most popular PostCSS plugins is the CSSNext (https://cssnext.github.io/). It helps you use the latest CSS syntax today. It transforms new CSS specs (https://www.xanthir.com/b4Ko0) into more compatible CSS so you don't need to wait for browser support.

The Vue CLI uses Webpack under the hood to bundle your app, and make it browser-ready. In order for Webpack to load the Vue components, the Vue CLI team created the official Webpack loader for Vue.js components Vue Loader (https://github.com/vuejs/vue-loader).

Internally, the Vue Loader makes use of PostCSS. Therefore, we can easily install, and use, Tailwind in our app by following the steps.

Let's start by installing the Tailwind NPM package by running:

```
npm install tailwindcss --save-dev
```

Now, add a *postcss.config.js* JavaScript file to the root folder of your app, and paste this content:

```
const autoprefixer = require('autoprefixer');
const tailwindcss = require('tailwindcss');

module.exports = {
  plugins: [
    tailwindcss,
    autoprefixer,
  ],
};
```

We are adding Tailwind as a PostCSS plugin. We are also adding the Autoprefixer (https://github.com/postcss/autoprefixer) PostCSS plugin. This plugin automatically adds vendor prefixes to your CSS. Tailwind doesn't have any autoprefixer backed inside, so we need to add the Autoprefixer plugin.

By adding these two plugins, we are telling PostCSS to apply these plugins to our CSS. More specifically, the Tailwind PostCSS plugin will locate Tailwind directives, and process them as you will see shortly.

Our next step is to add Tailwind to our CSS. Inside the path *src\assets*, create a new CSS file named *styles.css* inside a new folder named *styles*. Inside this file, use the `@tailwind` directive to inject Tailwind's base, components, and utilities styles, into your CSS:

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

Tailwind will swap these directives out at build time with all of its generated CSS.

The final step, before you are able to use Tailwind, is to import the CSS file above into the *src/main.js* file as follows:

```
import "./assets/styles/styles.css";
```

That's it! Now you can enjoy using Tailwind in your app.

I highly recommend checking out the Tailwind Docs (https://tailwindcss.com/course/setting-up-tailwind-and-postcss) for more information on installation, customization, and how to use Tailwind.

Section 8: Vue UI Libraries

Modern Web Frameworks promote building smaller, single-focused UI Components, for use in applications. This allows you to reuse components within the same application, or across multiple applications.

A UI Component library follows the same trend, and offers its developers a wide range of building blocks. They include buttons, input elements, scrolling lists, calendars, and grids, to

mention just a few. They are built in such a way that you can easily install them in your app to start using them right away.

We have seen how you can integrate Bootstrap and Tailwind in your Vue app. These frameworks work well, and are great. However, they behave as an external layer on top of Vue.js, rather than being baked into Vue.js context, and environment. You've got to do a fair amount of work to integrate Bootstrap's components into Vue.js context.

That's why Vue-based UI Component libraries exist! These libraries offer a set of UI Components that you can use to build the User Interface or UI of your app. They are built as Vue.js Components, so they effortlessly slide into your app. They make development faster, and more consistent!

In this section, I will show you how you can integrate two well-known, and proven Vue.js UI Component libraries into your app.

Veutify: Material Design Component Framework

Veutify claims to be the No.1 UI Component library for Vue.js. It provides users with everything needed to build rich, and engaging, web applications using the Material Design specification (https://material.io/guidelines/).

Here's where you can browse their API Explorer (https://vuetifyjs.com/en/components/api-explorer/), and see all of their components in action.

Let's see how you can install this library, and start using it in your Vue.js app.

You can find the source code used in this section under the Git branch named add-vuetify (https://github.com/bhaidar/vue-first-app/tree/add-vuetify).

Navigate to your app root folder, and run the following command:

```
vue add veutify
```

This command installs Veutify as a Vue CLI Plugin (https://github.com/vuetifyjs/vue-cli-plugins).

You can read more about Vue CLI Plugins using this link:
https://cli.vuejs.org/guide/plugins-and-presets.html

Once the installation is done, you are prompted to install a Veutify Preset.

```
✔  Successfully installed plugin: vue-cli-plugin-vuetify

? Choose a preset: (Use arrow keys)
❯ Default (recommended)
  Prototype (rapid development)
  Configure (advanced)
```

A preset is a bunch of styles and components to get you started, and demonstrates how to use Veutify in your app.

Let's keep the default selection, and hit `Enter`.

You can read more about Veutify Presents here:
(https://vuetifyjs.com/en/customization/presets/)

The new plugin installed adjusts your app by creating a new *src/plugins* folder. In addition, it creates the *veutify.js* plugin file inside the folder with the following content:

```
import Vue from "vue";
import Vuetify from "vuetify/lib";

Vue.use(Vuetify);

export default new Vuetify({});
```

This instructs Vue to make use of the Veutify plugin.

Then, back to the *src/main.js*:

```
...
import vuetify from "./plugins/vuetify";

...

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

You can see how the Veutify plugin is imported to the main Vue instance.

Switch back to the *src/App.vue* component, and notice how its content was replaced by installing the Veutify default preset:

```
<template>
  <v-app>
    <v-app-bar app color="primary" dark>
      <div class="d-flex align-center">
        <v-img
          alt="Vuetify Logo"
          class="shrink mr-2"
          contain
          src="https://cdn.vuetifyjs.com/images/logos/vuetify-logo-dark.png"
          transition="scale-transition"
          width="40"
        />

        <v-img
          alt="Vuetify Name"
          class="shrink mt-1 hidden-sm-and-down"
          contain
          min-width="100"
          src="https://cdn.vuetifyjs.com/images/logos/vuetify-name-dark.png"
          width="100"
        />
      </div>

      <v-spacer></v-spacer>

      <v-btn
        href="https://github.com/vuetifyjs/vuetify/releases/latest"
        target="_blank"
        text
      >
        <span class="mr-2">Latest Release</span>
        <v-icon>mdi-open-in-new</v-icon>
      </v-btn>
    </v-app-bar>

    <v-content>
      <HelloWorld />
    </v-content>
  </v-app>
</template>
```

With Veutify, you have a bunch of Vue.js Components that you can make use of inside the Vue.js templates.

For instance, the `v-app` component acts like a page/component container. The `v-app-bar` component acts like a page/component header.

For a complete list of Veutify components, I suggest going through their API Explorer (https://vuetifyjs.com/en/components/api-explorer/), and checking all of them.

Bootstrap Vue

This is the Vue.js version of Bootstrap. You already have an idea of what Bootstrap is, and what it can provide. BootstrapVue comes with over 40 plugins, more than 80 custom components, and over 300 icons. It provides one of the most comprehensive implementations of Bootstrap v4 components, and grid system, for Vue.js.

Here's the documentation to get started working with BootstrapVue:
https://bootstrap-vue.js.org/docs

Let's see how you can install this library, and start using it in your Vue.js app.

You can find the source code used in this section under the Git branch named *add-bootstrapvue* (https://github.com/bhaidar/vue-first-app/tree/add-bootstrapvue).

Navigate to your app root folder, and run the following command:

```
vue add bootstrap-vue
```

This command installs BootstrapVue as a Vue CLI Plugin (https://www.npmjs.com/package/vue-cli-plugin-bootstrap-vue).

Once done, you are prompted to use `babel/polyfill` so that you can support older browsers.

```
? Use babel/polyfill? (Y/N)
```

Enter the letter Y and hit `Enter`.The installation continues.

The new plugin installed adjusts your app by creating a new *src/plugins* folder. In addition, it creates the *bootstrap-vue.js* plugin file inside the folder with the following content:

```
import Vue from "vue";

import BootstrapVue from "bootstrap-vue";
import "bootstrap/dist/css/bootstrap.min.css";
import "bootstrap-vue/dist/bootstrap-vue.css";

Vue.use(BootstrapVue);
```

This instructs Vue to make use of the BootstrapVue plugin.

Then, back to the *src/main.js*:

```
import "@babel/polyfill";
import "mutationobserver-shim";
import "./plugins/bootstrap-vue";
...

new Vue({
  …
  render: h => h(App)
}).$mount("#app");
```

The BootstrapVue plugin is imported together with all of the supporting imports needed.

Switch back to the *src/App.vue* component, and embed the BootstrapVue `Alert` component as follows:

```
<template>
  <div id="app">
    <div id="nav">
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link>
    </div>
    <b-alert show>Default Alert</b-alert>
    <router-view />
  </div>
</template>
```

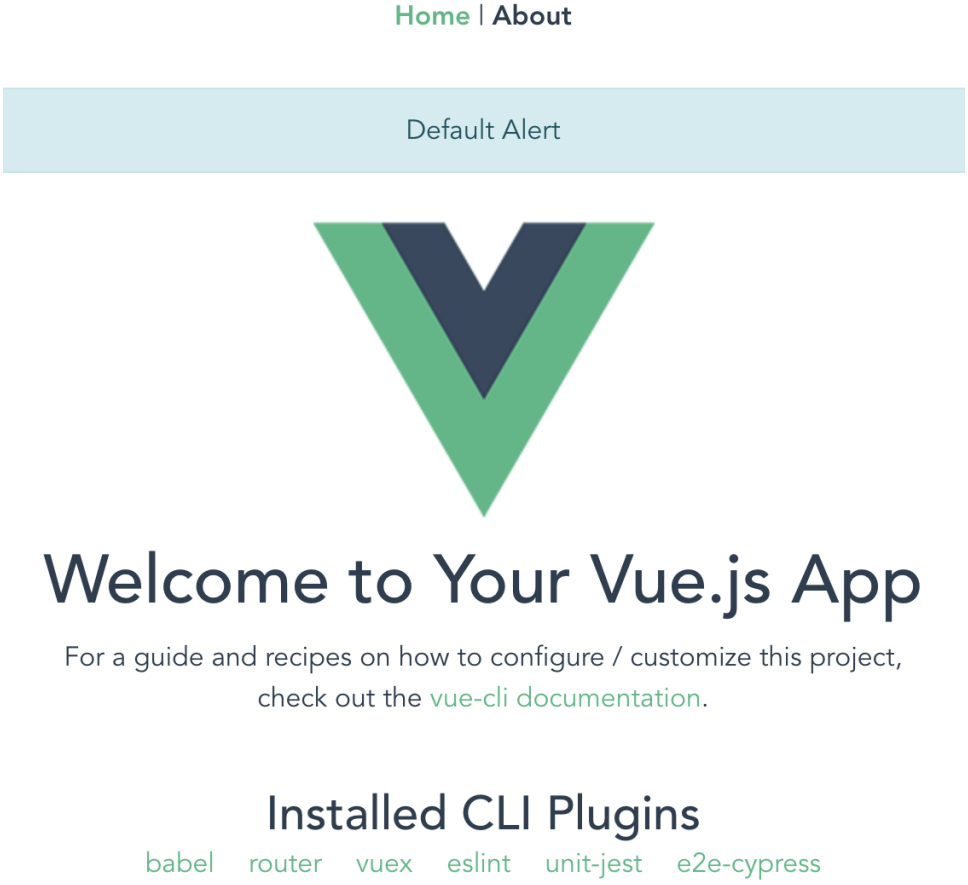

Figure 18: Using BoostrapVue in a Vue.js app

BootstrapVue provides many more Vue components that you can use in your app. Check the entire suite of Components (https://bootstrap-vue.js.org/docs/components/) on their website.

Section 9: Third-party Vue.js Modules

The Vue.js core framework can help you begin developing apps right away. However, once you add more features, more pages, and begin consuming data from remote Web APIs, your app gets bigger. You need to look for third-party modules to help you get your work done following well known standards, best practices, and above all, excellent testing methods! Don't reinvent the wheel!

Here, we will discover a few modules that are in any Vue.js app. Of course, there are tons of them out there that might be more specific, or what simply target certain aspects of an app.

Axios

Axios (https://github.com/axios/axios) is a Promise-based HTTP Client for the browser, and Node.js. It offers the following features:

1. Makes XMLHttpRequests (https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) from the browser.
2. Makes http (http://nodejs.org/api/http.html) requests from Node.js.
3. Supports the Promise API (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise).
4. Intercepts request and response.
5. Transforms request and response data.
6. Cancels requests.
7. Automatic transforms for JSON data.
8. Client side support for protecting against XSRF (http://en.wikipedia.org/wiki/Cross-site_request_forgery).

Vue.js doesn't ship with a library to perform HTTP requests out of the box. Therefore, we need to use a third-party module to help us with that.

One of the options is to use Axios. There is another module that I won't be detailing here, but in case you want to learn more about it, it's the Vue-Resource

(https://github.com/pagekit/vue-resource) module. Personally, I prefer using Axios, especially when I can use the same library with Vue.js, and Node.js.

To start using Axios in your Vue.js app, install the Axios NPM package by running the command:

```
npm install axios --save
```

Inside your components, you import Axios as follows:

```
import axios from 'axios';
```

Then, in order to use Axios to communicate with a remote Web API, you use one of the many functions available on Axios. Typically, you query for remote data inside the `created()` life-cycle hook in Vue.js component:

```
created() {
    axios.get(`http://jsonplaceholder.typicode.com/posts`)
    .then(response => {
      // JSON responses are automatically parsed.
      this.posts = response.data
    })
    .catch(e => {
      this.errors.push(e)
    });
}
```

Axios exposes more functionality. For instance, the `axios.post()` is used to post data to the server. You can play with a full list of functions available in Axios by checking Axios examples (https://github.com/axios/axios/blob/master/examples/README.md).

That was just an introduction to using Axios in a Vue app. I recommend reading the following article to gain more knowledge about Axios: Vue.js REST API Consumption with Axios (https://alligator.io/vuejs/rest-api-axios/).

Vue Router

Client-side routing is at the core of Single Page Apps (SPA). With routing, you can navigate to other screens, or pages, without reloading the entire page.

Vue Router (https://router.vuejs.org/) is the official router for Vue.js. It deeply integrates with Vue.js core to make building SPAs with Vue.js a breeze. The features include:

1. Nested route/view mapping.
2. Modular, component-based router configuration.
3. Route parameters, query, and wildcards.
4. View transition effects powered by Vue.js' transition system.
5. Fine-grained navigation control.
6. Links with automatic active CSS classes.
7. HTML5 history mode or hash mode, with auto-fallback in IE9.
8. Customizable Scroll Behavior.

The Vue.js team provides extensive support and documentation on Vue Router. It's the most useful resource to learn about Vue Router, and explore its API.

There are hundreds of examples that you can explore in order to deepen your understanding. You can access them by following this link: https://github.com/vuejs/vue-router/tree/dev/examples.

Back in Section #4: Vue CLI, we added the option to install Vue Router while creating a new Vue.js app. Let's see how the Vue Router is enabled, and installed, in the app.

Locate the *src/router/index.js* file content:

```
import Vue from "vue";
import VueRouter from "vue-router";
import Home from "../views/Home.vue";

Vue.use(VueRouter);

const routes = [
  {
    path: "/",
    name: "Home",
    component: Home
  },
  {
    path: "/about",
    name: "About",
    component: () =>
      import(/* webpackChunkName: "about" */ "../views/About.vue")
  }
];

const router = new VueRouter({
  routes
});

export default router;
```

The code installs Vue Router as a Vue Plugin. Then, it defines a set of routes to be used by Vue Router. You should customize, add, and remove routes as per your app. Finally, the code creates a new instance of the `VueRouter` object with the routes defined. The `VueRouter` instance is exported at the end of the file.

Switch back to the *src/main.js* file, and notice the following:

```
...

new Vue({
  router,
  ...
}).$mount("#app");
```

To make the Vue app aware of the routes, you need to pass the `VueRouter` instance to the main `Vue` object representing your app.

That's it for configuring and initializing the Vue Router plugin.

To start using it, you will need to be aware of two major components:

- RouterView (https://router.vuejs.org/api/#router-view)
- RouterLink (https://router.vuejs.org/api/#router-link)

The `RouterView` component is a placeholder that you add to indicate, to the Vue Router plugin, where to load components to which the user is navigating.

The `RouterLink` component is used to link to one of the predefined routes in the app.

```
<template>
  <div id="app">
    <div id="nav">
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link>
    </div>
    <router-view />
  </div>
</template>
```

That's all! Make sure to read the documentation to grasp the ins and outs of the Vue Router.

Vuex

Vuex (https://vuex.vuejs.org/) is a state management library for Vue.js. It serves as a single source of truth for any managed state in your app. It stores the state of all the components in your app in a centralized store. Components can communicate with the store to either update the state, or read it.

The ultimate goal of Vuex is to control mutating the state of your app. With Vuex, mutating the state becomes predictable and regulated. This gives way to a conceptual pattern that's more popular in SPAs, which is the `one-way-data-flow`. This pattern is illustrated in **Figure 19**:

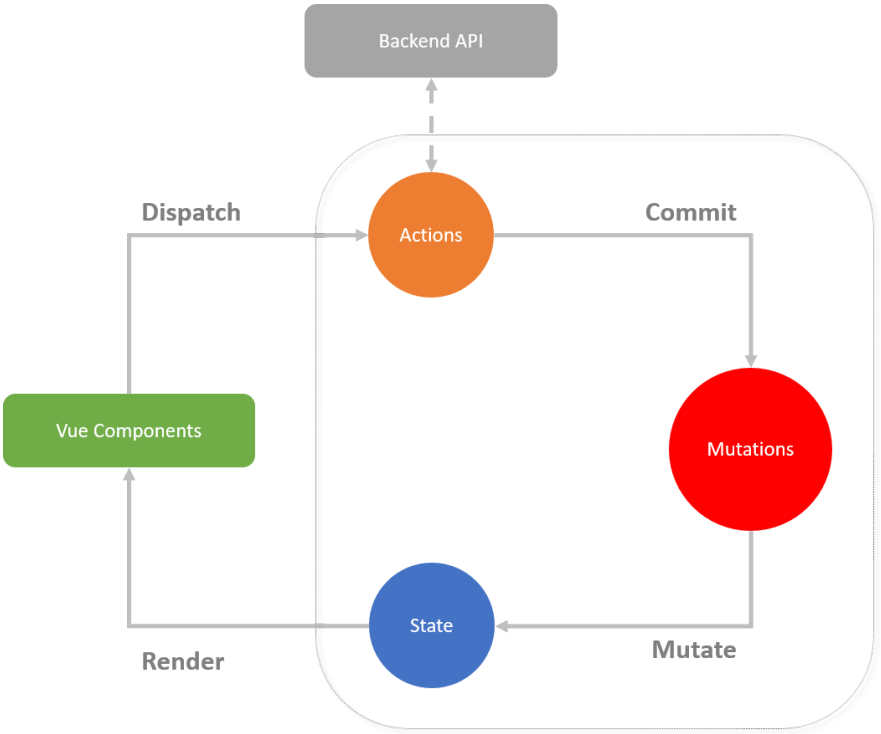

Figure 19: Vuex - one way data flow

1. A Vue Component dispatches a Vuex Action.

2. The Action communicates with a remote Web API. It then commits the results into a Vuex Mutation. A Vuex Action is not allowed to mutate the state. It does so by means of a Vuex Mutation.

3. The Mutation mutates the Vuex Store's State object. The State is shared across all components in a Vue app.

4. The State is now mutated. Vuex triggers the Vuex Reactive Getters to read fresh data from the State.

5. The Vue engine detects a change in the value of the Vuex Getters; it renders the new data, and refreshes the affected components.

This is just a brief explanation of how one-way-data-flow occurs inside a Vue app.

The team behind Vuex did a great job in offering well-structured, and detailed documentation, to explain the intrinsic aspects of Vuex. Make sure you read the Vuex documentation (https://vuex.vuejs.org/guide/).

Back in Section #4: Vue CLI, we added the option to install Vuex while creating a new Vue.js app. Let's look at how the Vuex is enabled, and installed, in the app.

Locate the *src/store/index.js* file content:

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

Vue.use(Vuex);

export default new Vuex.Store({
  state: {},
  mutations: {},
  actions: {},
  modules: {}
});
```

This code installs Vuex as a Vue Plugin. Then, it creates a new instance of `Vuex.Store` object.

The store object contains four main properties:

- State: This object holds all the state elements managed by the store.
- Mutations: This object holds all the mutation functions available by the store.
- Actions: This object holds all the action functions available by the store.
- Modules: This object holds all the different modules created as part of the store. You can layout and structure your store by means of independent modules. That's an advanced topic, and I suggest consulting with the Vuex documentation (https://vuex.vuejs.org/guide/modules.html) to learn more about Vuex Modules.

You can go through this basic Vuex Counter app (https://jsfiddle.net/n9jmu5v7/1269/) to understand how Vuex works.

Switch back to the *src/main.js* file, and notice the following:

```
...

new Vue({
  …
store,
  ...
}).$mount("#app");
```

To make the Vue app aware of the Vuex store, you need to pass the `Vuex.Store` instance to the main Vue object, representing your app.

That's all it takes to configure and initialize the Vuex plugin.

Vuelidate

Form validation is an essential task you carry on in each and every Web app. In principle, you cannot trust your users and you've got to protect your app by validating whatever input you receive from them. It has always been advisable to perform data validation on the client-side and the server-side. Since we are talking about Vue.js, let's focus our attention on client-side validation. This however, doesn't mean server-side validation is not important!

In Vue.js, you have two options available to validate HTML Forms:

1. To handle the validation manually and that's doable, but might become troublesome with time especially when the forms become bigger and more involved in terms of validation logic.

2. To make use of a third-party validation library that could take this burden off your hands and replace the manual validation with a more systematic and robust one.

Vuelidate(https://vuelidate.js.org/#getting-started) is one of the best validation libraries currently available for Vue.js. It's simple and is lightweight model-based validation for Vue.js 2.0. It'is characterized by the following features:

- Model based
- Decoupled from templates
- Dependency free minimalistic library
- Support for collection validations
- Support for nested models

- Contextified validators
- Support for function composition
- Validates different data sources: Vuex getters, computed values, etc.
- High test coverage

Here's the documentation to get started working with Vuelidate:
https://vuelidate.js.org/#getting-started

Before we dive into this library, let me demonstrate how to manually validate a form in Vue.js and then switch to Vuelidate.

You can find the source code used in this section under the Git branch named *add-validation* (https://github.com/bhaidar/vue-first-app/tree/add-validation).

Let's consider this HTML Form shown in **Figure 20**.

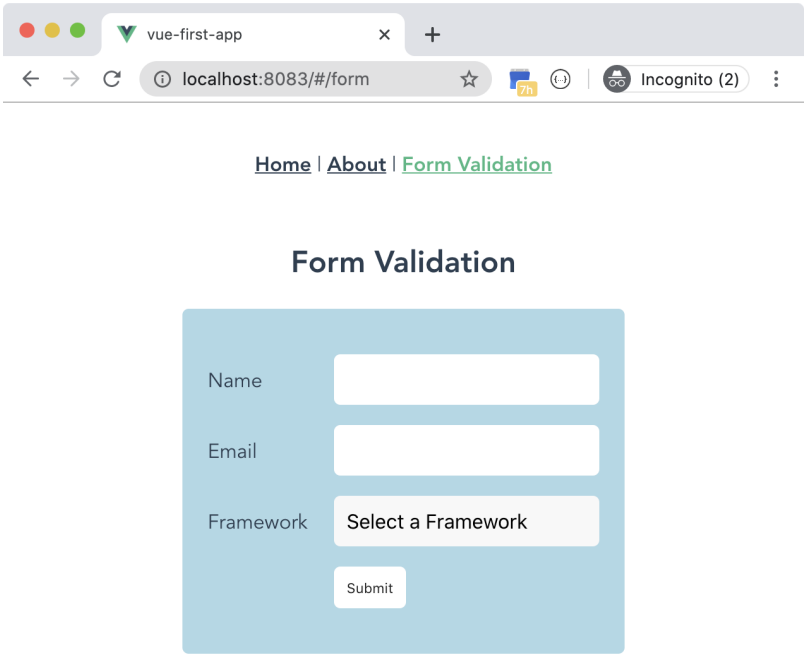

Figure 20: Form Validation

The Form has three fields. Below is the HTML template for this Form:

```
<form @submit.prevent="save" novalidate="true" class="form">
  <div v-show="errors.length">
    <b>Please correct the following error(s):</b>
    <ul>
      <li v-for="(error, index) in errors" :key="index">{{ error }}</li>
    </ul>
  </div>

  <p>
    <label for="name">Name</label>
    <input type="text" name="name" id="name" v-model="name" />
  </p>

  <p>
    <label for="email">Email</label>
    <input type="email" name="email" id="email" v-model="email" />
  </p>

  <p>
    <label for="framework">Framework</label>
    <select
      name="framework"
      id="framework"
      aria-label="Select a Framework"
      v-model="framework"
    >
      <option value="null" selected>Select a Framework</option>
      <option value="1">Angular</option>
      <option value="2">React</option>
      <option value="3">Svelte</option>
      <option value="4">Vue</option>
    </select>
  </p>

  <p>
    <input type="submit" value="Submit" />
  </p>
</form>
```

Notice how we disable the default HTML Form validation by setting `novalidation="true"` on the Form element.

Let's submit the Form and go over the `save()` function.

```
save: function(e) {
  this.errors = [];

  if (!this.name) this.errors.push("Name required.");

  if (!this.email) {
    this.errors.push("Email required.");
  } else if (!this.validEmail(this.email)) {
    this.errors.push("Valid email required.");
  }

  if (!this.errors.length) return true;

  e.preventDefault();
}
```

The function uses basic conditional statements to check for the existence of a value for the `Name` and `Email` fields. It checks if the email entered is a valid email. While doing so, it populates an array of errors that is eventually displayed for the user in case there are invalid fields. **Figure 21** shows those errors.

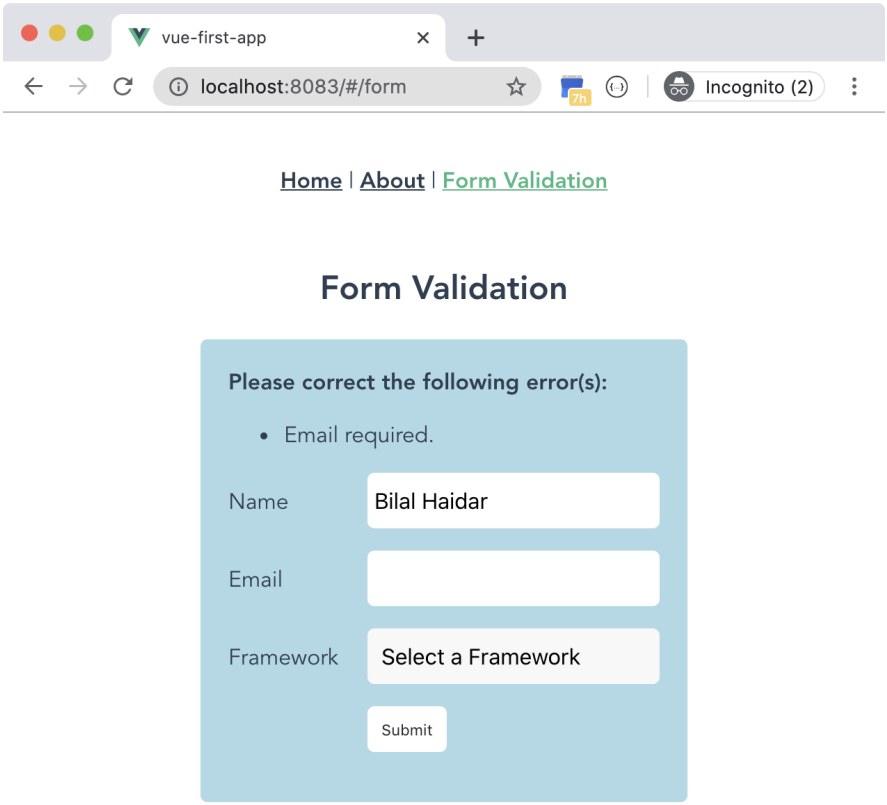

Figure 21: Form with errors

That's it!

For a complete reference on native Form validation in Vue.js make sure you read the documentation (https://vuejs.org/v2/cookbook/form-validation.html).

Ok! Let's look at Vuelidate.

You can find the source code used in this section under the Git branch named *add-vuelidate* (https://github.com/bhaidar/vue-first-app/tree/add-vuelidate).

Run the following command to install the Vuelidate NPM package (https://www.npmjs.com/package/vuelidate) into your app:

```
npm install vuelidate --save
```

Vuelidate is a Vue.js Plugin. To add it into the app, locate the *src/main.js* file and replace its content with the script below:

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

import Vuelidate from "vuelidate";
Vue.use(Vuelidate)

Vue.config.productionTip = false;

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

To install a Vue.js Plugin we make use of `Vue.use()` function.

Let's revisit the `FormValidation` component. Below you can see the `<script>` section of this component.

```
import { required, minLength, email } from "vuelidate/lib/validators";

export default {
  name: "FormValidation",
  data() {
    return {
      name: null,
      email: null,
      framework: null,
      submitted: false
    };
  },
  validations: {
    name: {
      required,
      min: minLength(4)
    },
    email: {
      required,
      email
    }
  },
  methods: {
    handleSubmit: function(e) {
      this.submitted = true;

      // stop here if form is invalid
      this.$v.$touch();

      if (this.$v.$invalid) {
        return;
      }

      e.preventDefault();
    }
  }
};
```

We start by importing the Vueldate built-in validator mainly the `required`, `minLength`, and `email` validators.

For a complete list of validators make sure to visit the Vuelidate Builtin Validators (https://vuelidate.js.org/#sub-builtin-validators).

Vuelidate integrates tightly with the Vue.js Options API. It adds a new property named `validations`. It's there, you define your validations.

For instance, the code above makes the `name` field required with a minimum length of 4 characters. The `email` field is made a required one and accepts only valid email addresses.

The handleSubmit() function handles the Form submission as follows:

1. It sets `submitted` to `true` meaning that the Form is now submitted.
2. It triggers the Form validation by calling `this.$v.$touch()` method, defined by Vuelidate.
3. The result of validation is saved inside `this.$v.invalid` property, defined by Vuelidate. If the Form is invalid, the method returns and errors are now displayed for the user. Otherwise, it continues normal processing.

Now let's have a look at the HTML Form:

```
<form @submit.prevent="handleSubmit" novalidate="true" class="form">
  <div class="field">
    <label for="name">Name</label>
    <input type="text" name="name" id="name" v-model="name" />
    <div v-if="submitted && !$v.name.required" class="invalid-feedback">
      Name is required
    </div>
  </div>

  <div class="field">
    <label for="email">Email</label>
    <input type="email" name="email" id="email" v-model="email" />
    <div v-if="submitted && !$v.email.required" class="invalid-feedback">
      Email is required
    </div>
    <div v-if="submitted && !$v.email.email" class="invalid-feedback">
      Email is invalid
    </div>
  </div>

  <div class="field">
    <label for="framework">Framework</label>
    <select
      name="framework"
      id="framework"
      aria-label="Select a Framework"
      v-model="framework"
    >
      <option value="null" selected>Select a Framework</option>
      <option value="1">Angular</option>
      <option value="2">React</option>
      <option value="3">Svelte</option>
      <option value="4">Vue</option>
    </select>
  </div>

  <p>
    <input type="submit" value="Submit" />
  </p>
</form>
```

This is the same HTML Form we used previously however, I removed the errors section and added error handling under each field to be validated.

For example, the “Name is Required” error message is shown only when the field is required, it’s value is empty and the Form is submitted.

Vuelidate attaches the validation information on a field and allows you to access them via `$v.name.required`, or `$v.email.required`, or `$v.email.email`, etc.

Now let's submit the Form without filling any information and see how the error messages are displayed under the field. **Figure 22** will demonstrate this.

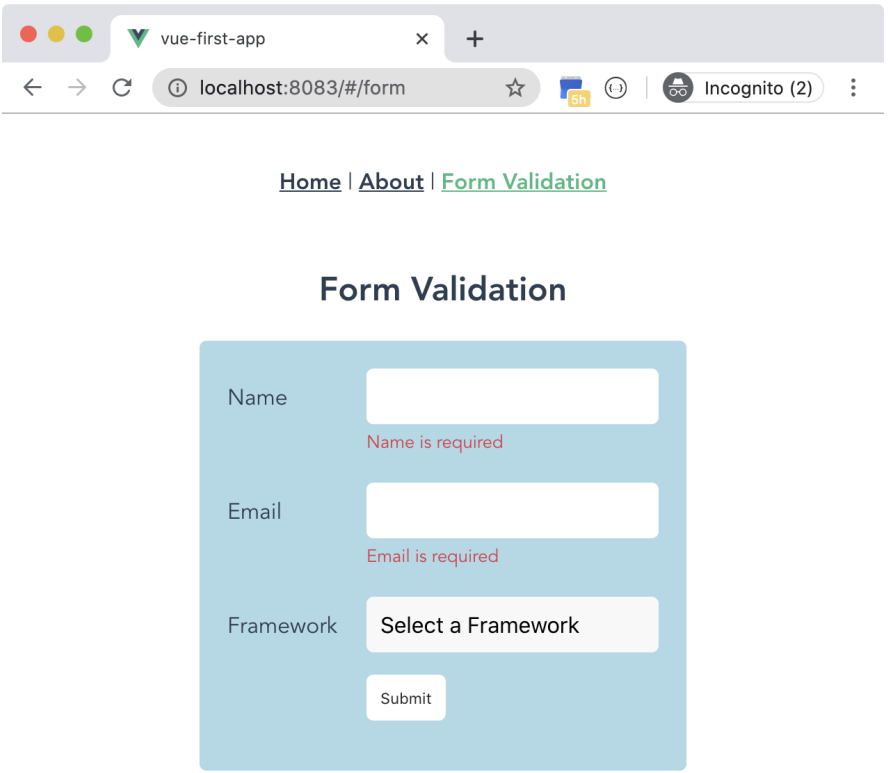

Figure 22: Form with Vuelidate errors

The error messages start to disappear when you fill in the fields with valid data.

Simple really.

I've only scratched the surface when using Vuelidate in a Vue.js app. This library has a rich set of features and I highly recommend going through their documentation website to learn more. (https://vuelidate.js.org/#examples)

Vue-i18n

Internationalization and localization of an app, in general, is a difficult task. It entails many aspects starting with localizing content to be displayed on the Web app, localizing routes, handling document language and layout direction Right To Left (RTL) and Left To Right (LTR), deciding on the database structure for localized data and many other facts.

The Vue-i18n (https://github.com/kazupon/vue-i18n) plugin is by far the most comprehensive internationalization Vue.js plugin out there. It offers translation file management, message formatting, date and number formatting, and much more.

You can find the source code used in this section under the Git branch named *add-i18n* (https://github.com/bhaidar/vue-first-app/tree/add-i18n).

Let's start by installing the Vue.js plugin!

Run the following command to install the Vue-i18n NPM package (https://www.npmjs.com/package/vue-i18n) into your app:

```
vue add i18n
```

The command downloads and installs the Vue-i18n library. During installation you are prompted to answer a few questions to install and customize the plugin the correct way. The questions are shown in **Figure 23**.

```
? The locale of project localization. en
? The fallback locale of project localization. en
? The directory where store localization messages of project. It's stored under `src` directory. locales
? Enable locale messages in Single file components ? Yes
```

Figure 23: Vue-i18n installation

Figure 23 is a simple question/answer wizard and is self-explanatory.

In addition to installing the Vue-i18n plugin, the command does a few things behind the scenes mainly to do with the configuration and to make the plugin ready to use without any additional effort on your part. I won't go into the details but I will recommend a great resource for this.

The Ultimate Guide to Vue Localization with Vue I18n (https://phrase.com/blog/posts/ultimate-guide-to-vue-localization-with-vue-i18n). This is by far the most comprehensive and current tutorial online for the Vue-i18n plugin.

For now, locate the *src/locales/en.json* file. This was added by the plugin installation process. This file will eventually contain all localization messages for the English language in this app.

```
{
  "message": "hello i18n !!"
}
```

Currently, it has a single localization key named `message`. You can add as many as you want and you can even nest them inside each other as you see fit for your app.

Locate and open the *src/components/HelloWorld.vue* component. Replace this line:

```
<h1>{{ msg }}</h1>
```

With this:

```
<h1>{{ $t('message') }}</h1>
```

It's that simple.! By using the special function `$t()` provided by the Vue-i18n plugin, you are requesting a translation for the localization key named `message`.

When the current app language is set to English (default), the `$t()` function returns a translation for the `message` key from the *src/locales/en.json* file. When the current app language is set to French for instance, the function returns a translation for the `message` key from the `src/locales/fr.json` file (The file is not there, this is just for demonstration purposes).

I've only just touched on using Vue-i18n in a Vue.js app. This library has much more to offer on this subject and I recommend going through their documentation website to learn more. (https://kazupon.github.io/vue-i18n/started.html).

Vue Apollo

“GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. It provides a complete and coherent description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.”

If you are planning to use GraphQL in your Vue app, you should be checking the Vue Apollo (https://vue-apollo.netlify.com/) Plugin. It integrates Apollo (https://www.apollographql.com/) into your Vue app.

Apollo is a GraphQL implementation standard for the client-side, and server-side. You can implement GraphQL on the backend/server using Apollo Server (https://www.apollographql.com/docs/apollo-server/). On the client-side (React, Vue, or Angular), you can use Apollo Client (https://www.apollographql.com/docs/react/) to communicate with the Apollo Server.

To install Vue Apollo, check this detailed guide on how to do so: Vue Apollo Installation (https://vue-apollo.netlify.com/guide/installation.html#_1-apollo-client).

To start using Vue Apollo in your Vue app, check this detailed guide: Using Vue Apollo (https://vue-apollo.netlify.com/guide/apollo/#apollo-options).

Section 10: Storybook and Component-based development

Developers and designers spend most of their time designing and building the frontend of an app. A common trend is to build small UI components, and then assemble them together inside a Page or View. This brings modularity to the code you write, and allows you to build things faster.

Following this trend, several libraries emerged to help document, and test, components. For instance, Storybook (https://storybook.js.org/), a library that is used with React.js, Vue.js and Angular frameworks. With Storybook, you build your UI components, and then write Storybook stories to consume those components. Think of it as doing Component Unit Testing. An example of a Storybook story is shown below:

```
import Vue from 'vue';
import MyButton from './Button.vue';

export default { title: 'Button' };

export const withText = () => '<my-button>with text</my-button>';

export const withEmoji = () => '<my-button>😀 😎 👍 💯 </my-button>';

export const asAComponent = () => ({
  components: { MyButton },
  template: '<my-button :rounded="true">rounded</my-button>'
});
```

Given a `Button` component, the story renders three instances of this component, each time with different text content.

Storybook reads all stories in your application, and renders them inside the browser outside the context of the application. This means that you can have team one working on the components in a separate project, and have another team building the pages, and consuming the components.

In addition to having the ability to view all your basic components in a Component Explorer, you are also indirectly testing your components via Storybook stories. Each and every store represents a single state of a component. You will write stories for basic components, and also stories for composite components, hence the ability to test your components outside the app context.

Install Storybook

Let's add Storybook to our Vue.js app by following the steps below. You can find the source code used in this section under the Git branch named *add-storybook*.

Run the command below to install the Storybook NPM package (https://www.npmjs.com/package/@storybook/vue) into your app:

```
npm install @storybook/vue @babel-preset-vue --save-dev
```

We need to install further NPM packages to let Storybook load our Vue.js components, and all the SCSS inside these components. Run the following command:

```
npm install vue-loader vue-style-loader vue-template-compiler
@babel/core babel-loader babel-preset-vue --save-dev
```

Once all the packages are installed, create a new *.storybook* folder at the root folder of the app. Note the dot before the name of the folder.

Add a *.storybook/config.js* JavaScript file with the following content:

```
import { configure } from "@storybook/vue";

// Automatically import all files ending in *.stories.js
configure(
  require.context("../src/stories", true, /\.stories\.js$/),
  module
);
```

With this, you are instructing Storybook to load all story files with names, ending in *.stories.js*, from within the *src/stories* folder.

Then, create a new *src/stories* folder. This folder holds all the Storybook stories in your app.

Back in Section #4: Vue CLI, we added the option to use SCSS while creating a new Vue.js app. Therefore, we need to instruct Storybook to load any SCSS code it finds in our Vue.js components. To do this is, we will create a *.storybook/webpack.config.js* JavaScript file, and will add the following content:

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

module.exports = {
  module: {
    rules: [
      {
        test: /\.scss$/,
        use: [
          {
            loader: "vue-style-loader",
          },
          {
            loader: "css-loader",
          },
          {
            loader: "sass-loader",
          },
        ],
        include: path.resolve(__dirname, "../src")
      }
    ]
  }
};
```

The file instructs Webpack to deal with any file that ends with an extension of *.scss* and `<style lang="scss">` blocks in *.vue* files. For these files and blocks, Webpack will use three loaders, starting with `vue-style-loader`, then `css-loader`, and finally the `sass-loader`.

Now our Vue.js components will render perfectly inside the Storybook stories.

Create a Storybook story

Inside the *src/stories* folder, add our first Storybook story by creating a new file named *HelloWorld.stories.js* with the following content:

```
import HelloWorld from "./../components/HelloWorld";

export default {
  title: "Hello World"
};

export const withDefault = () => ({
  components: {
    HelloWorld
  },
  template: `
  <div style="margin: 30px auto; padding: 0 20px; border: solid 1px blue;">
    <HelloWorld></HelloWorld>
  </div>`
});
```

The story:

- Imports the `HelloWorld` component.
- Defines an ES6 default export with a title.
- Exports a single story named `withDefault`. You can think of a story as a Vue.js Component. It takes the shape of a Vue.js component, and renders any component under test. For example, the story defines a property named `components` that includes the `HelloWorld` component. It also defines a property called `template` that makes use of this component.

Before we can actually run this story, open the *package.json* file at the root of the app folder. Add the following convenient script to run Storybook in your app:

```
"scripts": {
    ...
    "storybook": "start-storybook"
  }
```

To run Storybook, run the command:

```
npm run storybook
```

A new browser opens and shows all the stories defined in the app as shown in **Figure 24**:

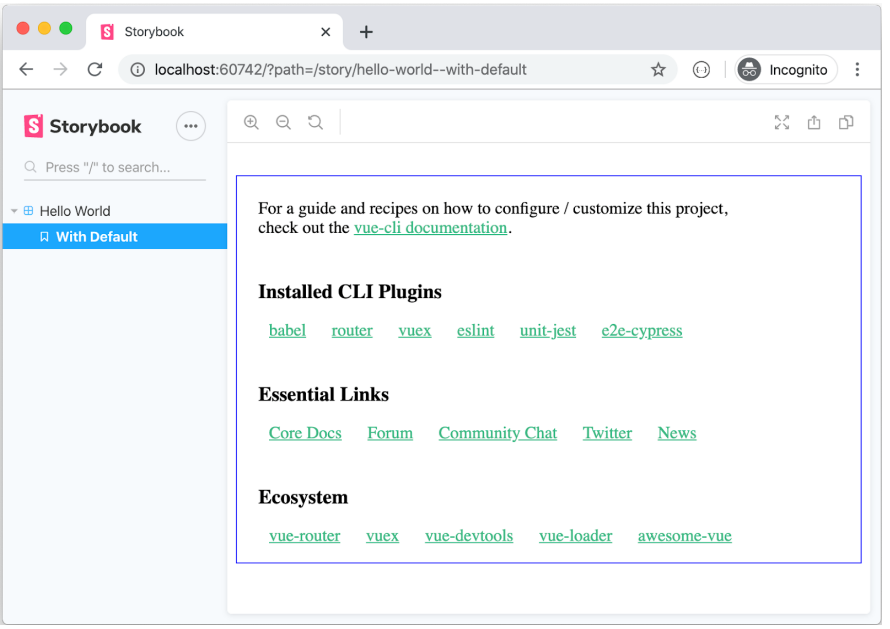

Figure 24: Running Storybook stories

How to use Vue Router inside a Storybook story

Sometimes the Vue.js components make use of the Vue Router. For instance, a component uses a `RouterView` or `RouterLink` component, or a component is watching changes to certain route parameters. You need a way to introduce Storybook to the Vue Router. How?

First of all, install the storybook-vue-router (https://www.npmjs.com/package/storybook-vue-router) NPM package by running this command:

```
npm install storybook-vue-router --save-dev
```

Then, install the Storybook addons NPM packages:

```
npm install @storybook/addon-actions --save-dev
npm install @storybook/addon-links --save-dev
```

After that, create a new file inside *.storybook/addons.js* with the content:

```
import "@storybook/addon-actions/register";
import "@storybook/addon-links/register";
```

Finally, adjust the Storybook story to load the `StoryRouter` (https://github.com/gvaldambrini/storybook-router) object via Storybook decorators (https://storybook.js.org/docs/addons/introduction/). The `StoryRouter` allows you to use routing-aware components in your stories.

```
import StoryRouter from "storybook-vue-router";
import HelloWorld from "./../components/HelloWorld";

export default {
  title: "Hello World",
  decorators: [StoryRouter({})]
};

export const withDefault = () => ({
  components: {
    HelloWorld
  },
  template: `
  <div style="margin: 30px auto; padding: 0 20px; border: solid 1px blue;">
    <HelloWorld></HelloWorld>
  </div>`
});
```

That's it!

How to use Vuex inside a Storybook story

Storybook stories want to render components that make use of Vuex. How do you integrate Vuex with Storybook? That's very simple!

Locate, and open the *.storybook/config.js* file, and add the following:

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

Vue.use(Vuex);

store.dispatch("getUser");

store.commit(
  "setUser",
  JSON.parse(
    '[{ "id": 1, "name": "Bilal Haidar" }] }]'
  )
);
```

You just import `Vue` and `Vuex` objects, then add the Vuex plugin to the Vue instance. You can dispatch store actions, or commit store mutations from within the config file. Most of the time, I find myself committing mutations with hardcoded data to test my components. Nothing new here! You are populating the store with some data to be used later when the Storybook stories load.

Now, switch back to our *Helloworld.stories.js* file, and add:

```
import store from './../store';
```

Start by importing the `store` object into the story file.

Then, you need to inject the `store` object into the story as follows:

```
export const withDefault = () => ({
  components: {
    HelloWorld
  },
  store,
  template: `
  <div style="margin: 30px auto; padding: 0 20px; border: solid 1px blue;">
    <HelloWorld></HelloWorld>
  </div>`
});
```

That's all! If the `HelloWorld` component uses the Vuex store, it will run smoothly, without any issues, inside Storybook runner.

How to load CSS libraries inside Storybook

Most of the time, you configure and use one of the CSS libraries in your components. To make those components render properly inside Storybook, with all the styling and UI features, you need to let Storybook know about the CSS library your components are using.

If you are using the Bootstrap library in your app, you can import the *bootstrap.scss* file into the *.storybook/config.js* as follows:

```
import '../node_modules/bootstrap/scss/bootstrap.scss';
```

Storybook provides a neater solution which consists of adding the *.storybook/preview-head.html* (https://storybook.js.org/docs/configurations/add-custom-head-tags/) file. This file gets loaded for each and every Storybook story.

Inside this file, you can link to Bootstrap via CDN.

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css">
```

You may use the *preview-head.html* file to include additional stuff to render together with the Storybook stories.

Including Tailwind (https://tailwindcss.com/) is a little bit more involved. Either you use the *prev-head.html*, and include the Tailwind on CDN

(https://tailwindcss.com/docs/installation/#using-tailwind-via-cdn) option, or you use a more advanced technique that involves Webpack. Here are two resources that will help you do so:

- Tailwind-storybook (https://github.com/audunru/tailwind-storybook/blob/master/.storybook/webpack.config.j)

- How to configure Tailwind CSS for Storybook (https://github.com/storybookjs/storybook/issues/4038#issuecomment-475083406).

Section 11: Debugging Vue.js apps

We spend a big chunk of our time debugging our code! It's expected, and part and parcel of the process. In this section, I will share a few debugging methods that I use.

1. Debugging Vue.js apps inside VS Code. This guide (https://vuejs.org/v2/cookbook/debugging-in-vscode.html) explains what you need to configure your VS Code editor to allow for debugging Vue.js apps.

2. Debugging Vue.js apps inside Chrome DevTools 101. This guide (https://www.youtube.com/watch?v=H0XScE08hy8) explains, in detail, how to use Chrome devtools to debug any JavaScript code, and is not limited to Vue.js only. In addition, you can check the FireFox DevTools for debugging (https://developer.mozilla.org/en-US/docs/Tools) that details how to use FireFox devtools to debug any JavaScript code, including, but not limited to, Vue.js.

3. Vue DevTools (https://github.com/vuejs/vue-devtools) is a browser devtools extension for debugging Vue.js applications. This is the best way to inspect your Vue.js components from inside of the browser. You will be able to see what components your view is composed of, what the internal state of a component is, and view the data properties and computed properties, among other things. Also, if you use the Vuex store in your app, you can view its state by viewing the data, actions, mutations, and getters in the state.

Finally, watch the 3 Ways to Debug Vue.js apps (https://www.youtube.com/watch?v=lyGt1TmleoU) video. It's comprehensive, and covers three different ways for debugging. The methods include: VS Code, Chrome devtools, and Vue devtools debugging.

And that wraps up debugging for Vue.js apps!

Section 12: Testing Vue apps

Software testing has become an integral part of every software development methodology. Testing your code guarantees a high level of confidence in the apps you are delivering to your clients.

In this section, I will introduce the different types of software testing, and then explore the different testing options for Vue.js apps.

Testing types

Figure 25 shows the traditional Software Testing Pyramid, and the different types of testing available through the development life cycle.

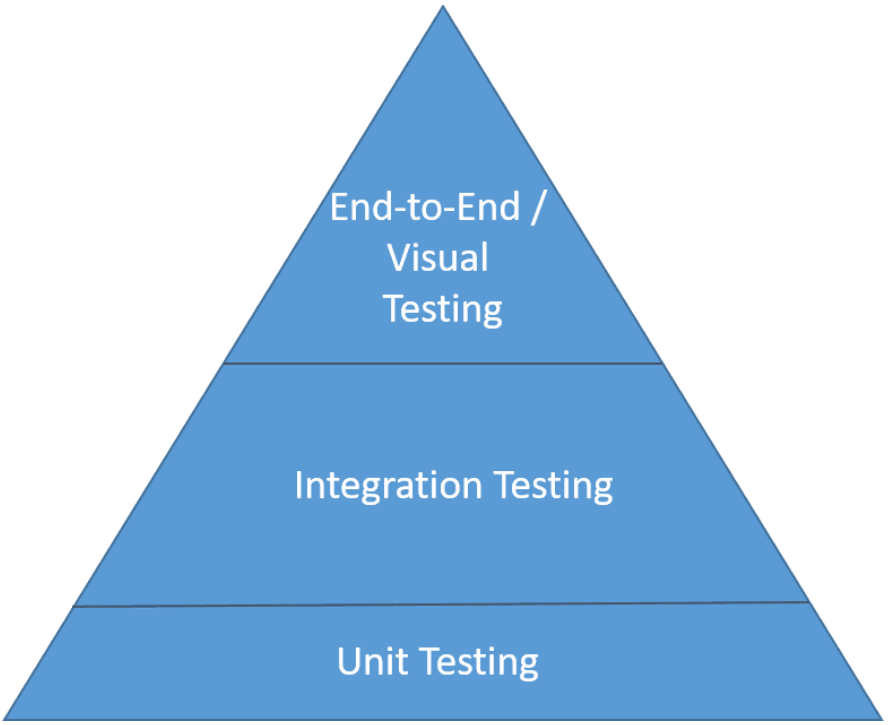

Figure 25: Software Testing Pyramid

Unit Testing

Unit testing is a method that looks at individual units of code. It's a crucial part of Continuous Integration (CI) (https://codeship.com/continuous-integration-essentials). It isolates individual functions to check inputs against outputs, making sure each part of the software functions as expected.

Integration Testing

While unit testing focuses on individual functions in the source code, integration testing (http://softwaretestingfundamentals.com/integration-testing/) is done on a higher level of abstraction. Integration testing checks the communication between:

- One component and another, or one page and another.
- Your application, and the backend database engine.
- One subsystem/module, and another

End-to-End Testing

Functional testing- also known as End-to-End or E2E testing- sits at the top of the pyramid. When writing a functional test script, you focus on testing the logic of the application, as well as verifying, and implementing, all the features requested by your client. Functional testing usually follows a scenario testing path, taken in your application, starting from a certain point, and ending at another point. For instance, a possible functional test could test the entire Login process:

- The user navigates to the Login page.
- They enter their credentials, and submit the form.
- If the credentials are correct, the user is redirected to the secure access pages.
- Otherwise, the user is denied access, and is left with an error message stating a possible reason for the login failure.

Only when the test runs, and succeeds, will you have verified, and fulfilled one functional requirement, requested by your client. Then, you move on to write the next tests.

Visual Testing

With the way technology is evolving, the need for visual testing has never been stronger. It's unacceptable to have any type of bug in your application. The general public is unforgiving. This challenge is growing exponentially with the constant introduction of new devices, operating systems, browsers, and screen resolutions.

With unit, integration, and functional testing, you can obtain a certain level of confidence in the function of the application, and to what extent it serves its purpose. However, with visual testing, you can be sure your application works across all platforms without any hiccups. If your apps run smoothly on all platforms, you should expect consumer confidence.

The special relationship between functional and visual testing distinguishes the two among all forms of testing. Functional testing has been around for a long time. It's reputable, and the testing frameworks supporting functional testing are robust.

Visual testing takes advantage of existing functional testing infrastructure and builds its tools on top of it. Given a functional test, you can easily integrate a visual testing framework by injecting function calls inside of your tests. These function calls serve the purpose of taking snapshots, sending them to the backend servers for analysis and comparison, and finally, generating a report of the results.

If you are interested in learning about visual testing, I've written thirteen articles (https://applitools.com/blog/author/bilalhaidar) exploring Applitools as a visual testing framework.

All of the above testing types come in the form of automated tests. By leveraging them, you benefit from using regression testing to ensure the validity of your app at all levels.

Testing Vue.js apps

Vue.js app, like any other app, can benefit the most from the different testing types available. What's even more important is the fact that the Vue CLI bakes-in all the recipes needed to perform testing.

You can find the source code, used in this section, under the Git branch named "add-tests" (https://github.com/bhaidar/vue-first-app/tree/add-tests).

Component / Unit Testing

In Vue.js apps, unit testing maps to component testing. A component in a Vue.js app can be thought of as the smallest unit of code! The Vue team provides the Vue Test Utils (https://vue-test-utils.vuejs.org/guides/)- the official unit testing utility library for Vue.js. Reading this guide is essential before attempting component testing.

In brief, the Vue Test Utils library offers the following features:

- It tests Vue components by mounting them in isolation, mocking the necessary inputs (props, injections and user events), and asserting the outputs (render result, emitted custom events).

- Simulates User Interaction by allowing you to trigger a Button inside the component under test.

- It has all of the necessary utilities to test single-file-components (https://vuejs.org/v2/guide/single-file-components.html), including those using Vuex (https://vuex.vuejs.org/) and Vue Router (https://router.vuejs.org/).

To learn how to do unit testing in your Vue.js app, I suggest reading the three articles, provided in the links, to gain a greater understanding of what must be tested, and how.

- Unit Testing in Vue: Your First Test (https://www.vuemastery.com/blog/Unit-Testing-in-Vue-Your-First-Test/)

- Unit Test your First Vue.js Component (https://frontstuff.io/unit-test-your-first-vuejs-component)

- Knowing What to Test - Vue Component Unit Testing (https://vuejsdevelopers.com/2019/08/26/vue-what-to-unit-test-components/)

While creating a new Vue.js app back in Section #4: Vue CLI, we added the option to use unit testing in the sample app, and also made use of Jest (https://jestjs.io/), a well-established JavaScript testing framework with so many goodies. The Vue CLI adds all the necessary bits to allow writing, and running, unit tests in the app.

Each component you want to test should have a corresponding *spec* file that represents the test suite of a component. The spec file is named after the component name. For instance, a component named `Navigation` will have a corresponding spec file named *navigation.spec.js*. The spec file should be placed under the *tests/unit* folder in your app as well.

Switch back to the Vue.js app, and checkout the Git branch named add-tests (https://github.com/bhaidar/vue-first-app/tree/add-tests).

Locate the *Counter.vue* component inside the *src/components* folder.

```
<template>
  <div>
    <span class="count">{{ count }}</span>
    <button @click="increment">Increment</button>
  </div>
</template>

<script>
export default {
  data() {
    return { count: 0 };
  },
  methods: {
    increment() {
      this.count++;
    }
  }
};
</script>
```

Locate the *tests/unit/counter.spec.js* file. It's a basic test suite with two tests, written as follows:

```
import { shallowMount } from "@vue/test-utils";
import Counter from "@/components/Counter";

describe("Counter.vue", () => {
  it("mounts a component and prints the counts", () => {
    // Mount the component, and you have the wrapper
    const wrapper = shallowMount(Counter);

    // Access the actual Vue instance via `wrapper.vm`
    const vm = wrapper.vm;

    // display the value of the count
    console.log(vm.count);
  });

  it("button click should increment the count", () => {
    // Mount the component, and you have the wrapper
    const wrapper = mount(Counter);

    expect(wrapper.vm.count).toBe(0);

    const button = wrapper.find("button");
    button.trigger("click");

    expect(wrapper.vm.count).toBe(1);
  });
});
```

To run all unit test scripts in your app, run the following command:

```
npm run test:unit
```

The tests are run, and results are displayed on the screen as shown in **Figure 26**.

```
vue-guide/vue-first-app [add-tests] » npm run test:unit

> vue-first-app@0.1.0 test:unit /Users/bilalhaidar/Projects/vue-guide/vue-first-app
> vue-cli-service test:unit

 PASS  tests/unit/counter.spec.js
  ● Console

    console.log tests/unit/counter.spec.js:15
      0

 PASS  tests/unit/example.spec.js

Test Suites: 2 passed, 2 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        2.616s
Ran all test suites.
```

Figure 26: Unit tests results

Using `shallowMount()`, which is provided by the Vue Test Utils, is essential to writing this test. This method allows you to create an isolated instance of the component without loading its children.

End-to-End Testing

Whether the app is written in Vue.js, or some other framework, functional, or E2E, testing works the same.

While creating a new Vue.js app back in Section #4: Vue CLI, we added the option to use Cypress (https://www.cypress.io/) as an E2E testing solution. The other option was Nightwatch (https://nightwatchjs.org/), which I won't cover in this guide. However, I'll include some recommended resources for you.

Before you can start writing functional tests, I suggest reading or watching the following resources in order:

1. Introduction to Cypress (https://testautomationu.applitools.com/cypress-tutorial/?utm_term=&utm_source=web-r

eferral&utm_medium=blog&utm_content=blog&utm_campaign=&utm_subgroup=): a free and detailed video course for beginners by Gil Tayar.

2. Cypress Tutorial for Beginners: Getting started with End to End Testing (https://www.valentinog.com/blog/cypress/): An informative article on E2E testing in Cypress.

3. Quick Start Vue.js With Cypress End-To-End Testing For Beginners (https://www.youtube.com/watch?v=eM7fjPzHVa4): A video tutorial on using Cypress inside of a Vue.js app.

4. End To End Testing our VueJS application with Cypress (https://www.youtube.com/watch?v=_f61LfPHyAo) A detailed video on writing Cypress E2E tests.

5. (Optional) [Intro to Nightwatch.js and e2e testing](https://www.youtube.com/watch?v=zzofkNaoPYE) To learn Nightwatch.

Now, going back to the Vue.js app, checkout the Git branch named add-tests (https://github.com/bhaidar/vue-first-app/tree/add-tests).

Locate the *tests/e2e* folder. This folder represents the Cypress setup folders that the Vue CLI scaffolds for us while creating the Vue.js app. You will place all of your E2E spec files inside the *tests/e2e/*specs folder.

For now, let's explore the single spec file named *test.js*, as shown in **Figure 27**.

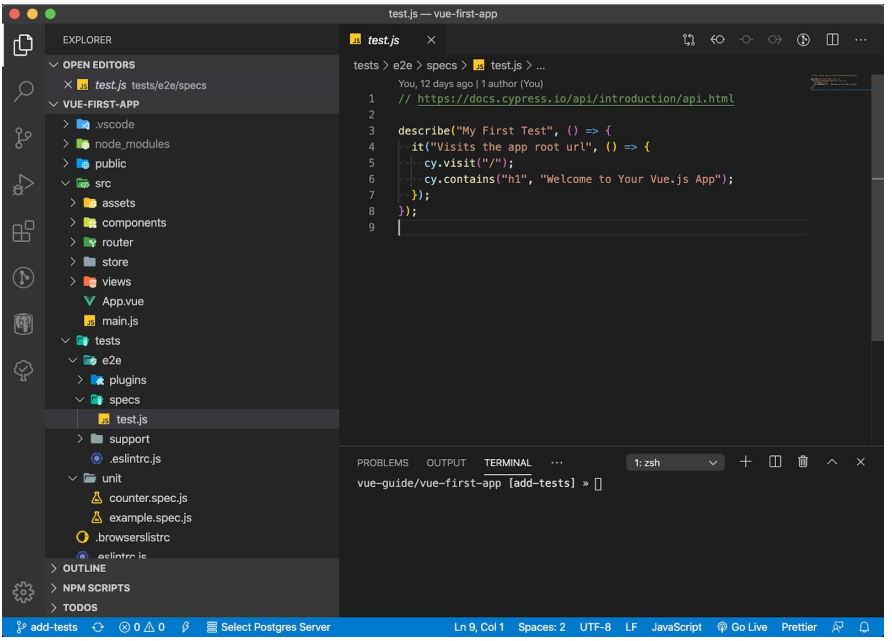

Figure 27: Cypress E2E test script

Although basic, this example is enough to help us understand how to write Cypress E2E tests, and run them inside of a Vue.js app.

The test suite defines a single test that:

1. Navigates the browser to the root page of the currently running app

2. Asserts that the HTML element `h1` contains the text `Welcome to Your Vue.js App`

Cypress gives you the chance to go crazy, and query the underlying DOM, or simulate user interactions like filling an input field, or clicking a button, as well as many other techniques (https://docs.cypress.io/guides/core-concepts/introduction-to-cypress.html).

To run all e2e test scripts in your app, run the command:

```
npm run test:e2e
```

This command internally runs the Cypress command behind the scenes. **Figure 28** shows the Cypress test runner app open for the first time.

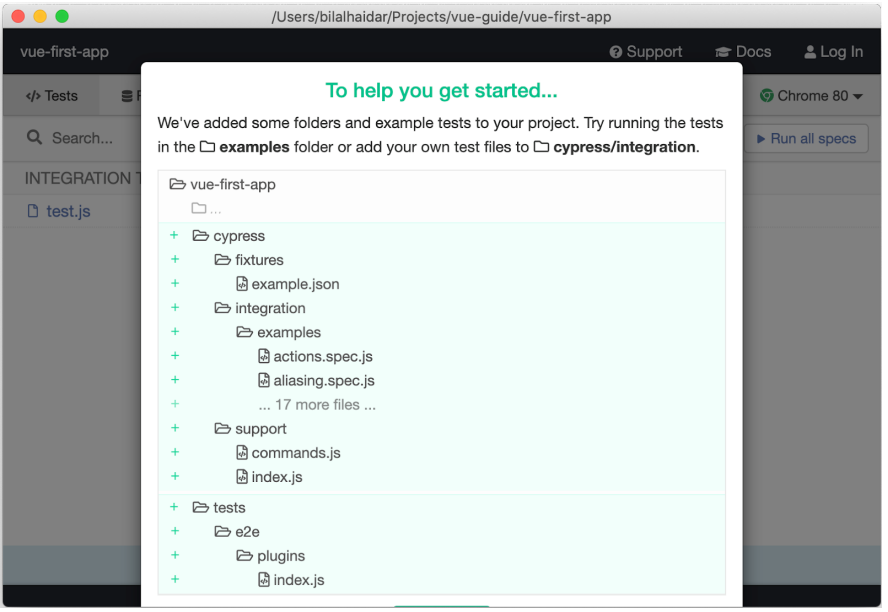

Figure 28: Cypress test runner app

Since we are running the test runner for the first time, you receive a message notification saying that Cypress plugin has added a few sample tests for your reference to help you get started.

Close the popup window to access the single E2E test in our app, as shown in **Figure 29**.

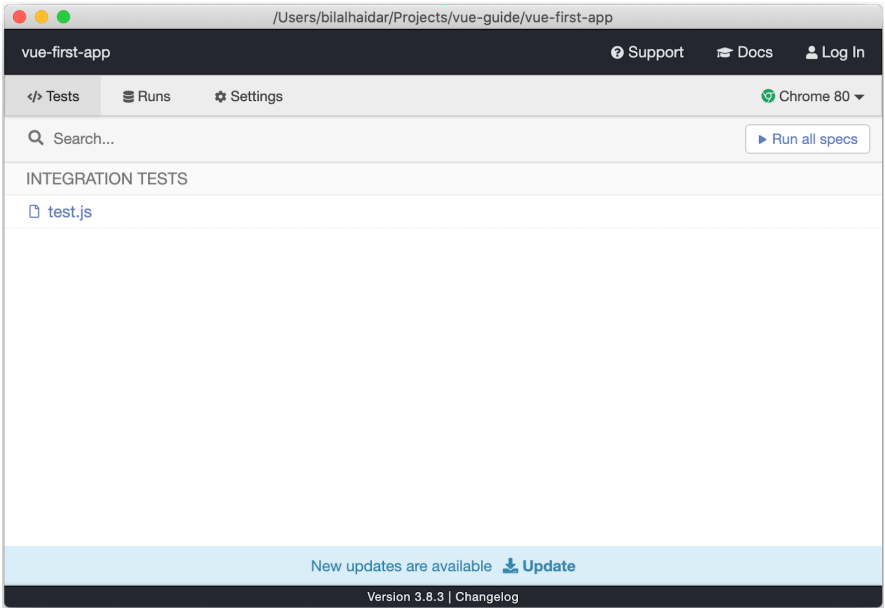

Figure 29: E2E tests listing

Locate the single *test.js* test script, and click on it. Cypress will launch a new instance of a Google Chrome browser, and will run the clicked test script.

Figure 30 shows the full steps performed to run the single E2E test script in hand.

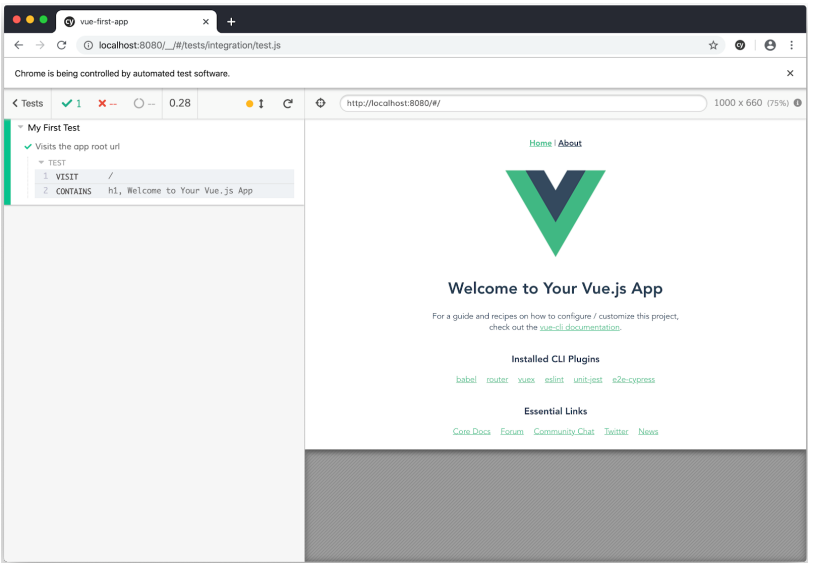

Figure 30: Full E2E test script run

The left side displays all of the steps/actions you've specified when writing the test script. On the right side, you see the result of executing the steps/actions of the test script.

This is just the tip of the iceberg for writing E2E test scripts. The resources I shared with you are enough to get you started writing functional/E2E tests for your Vue.js apps.

Visual Testing

Visual testing is one of my favorite testing topics. The beauty of it is in the ability to catch any single change in the HTML and CSS code, that even the naked eye cannot see.

To start using Visual testing in your app, you need to use a framework, such as Applitools (https://applitools.com/), or Percy (https://percy.io/). I will call on Applitools for this guide.

With Applitools, you can introduce visual testing into your app by means of two methods:

1. Start by writing E2E tests with Cypress, and then upgrade the E2E tests to visual tests. I've written about this method here: How to visually test VueJS apps using Cypress.io and Applitools (https://applitools.com/blog/test-vuejs-cypress-io-applitools).

2. If you are using Storybook in your app, then you can test the stories with visual testing. I've written about this method here: Visually Test Vue.js application using Applitools and Storybook (https://applitools.com/blog/visually-test-javascript-vue-storybook-applitools).

I won't delve into any more detail, but the two articles quoted are more than enough to get you started.

Section 13: Continuous Integration

Continuous Integration, or CI, is a fully-automated development methodology that seamlessly integrates your daily code changes into a centralized source code repository management system (SCRM). CI ensures a high level of code integrity by enforcing your own software development process for any code changes.

When employing CI in your development workflow, creating a new PR in your SCRM system triggers the CI workflow via a webhook more often than not.

The CI workflow is depicted in **Figure 31**:

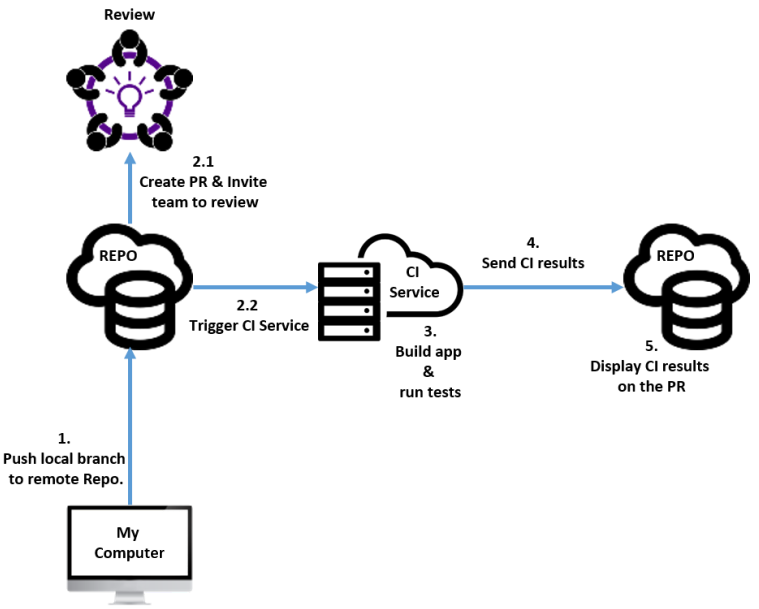

Figure 31: SCRM and CI Service Workflow

1. Create a new branch name, relevant to the feature you are adding.
    a. Add your code, and commit the changes locally.
    b. Push the local branch to a SCRM system- GitHub, for instance.

2. Create a Pull Request based on the new branch, and invite team members to review, and comment. In parallel, the SCRM triggers the CI Service.

3. The CI Service:
    a. Pulls the branch with PR.
    b. Builds the branch.
    c. Runs the tests, such as unit tests, e2e, and visual tests.

4. The CI Service reports the results of building the app, and running the tests, back to the SCRM.

5. The SCRM displays the CI Service results on the active PR.

If the CI Service reports errors, the PR is not merged. The developer is required to fix any issues, and push the branch again. Otherwise, the PR is approved, and is ready to be merged.

There are several CI Services you can use such as:

- CircleCI (https://circleci.com/). This is currently used by the Vue.js Github repository (https://github.com/vuejs/vue)

- Jenkins (https://jenkins.io/)

- Travis CI (https://travis-ci.com/)

- Azure DevOps (https://azure.microsoft.com/en-us/services/devops/)

- Google CI (https://cloud.google.com/solutions/continuous-integration)

The process of connecting a SCRM to a CI Service is lengthy and detailed. Therefore, I will direct you to a few resources to get started with CI, and how to add CI to your Vue.js app.

1. How Visual UI Testing can speed up DevOps flow (https://applitools.com/blog/visual-ui-speeds-devops). I've written this article to demonstrate using CircleCI, a CI Service, with GitHub, a SCRM, to run E2E tests with Cypress, and visual tests with Applitools. To see more, visit the Demo section. There, I give a detailed tutorial on how to create a new account at CircleCI, connect GitHub to it, connect CircleCI support to your app, and create a new PR to trigger the CircleCI.

2. Deploying Vue with CircleCI 2.0 (https://rossta.net/blog/deploying-vue-with-circleci-2.html). This short article assumes a fair bit of knowledge about CircleCI, and demonstrates a practical example of using Vue.js and CircleCi.

3. CircleCI config for Vue.js with S3 upload and node_modules caching (https://medium.com/@rmossakowski/circleci-config-for-vue-js-with-s3-upload-7503057b435c): This article demonstrates a more advanced CircleCI configuration setting to deploy a Vue.js app.

4. Vue JS CI/CD with Azure Devops to Azure App Service (https://www.youtube.com/watch?v=WPAvsj7H0l0). This video tutorial demonstrates adding CI support for a Vue.js app, using the Azure DevOps CI Service.

This concludes my recommendations, and suggested resources.

Now, let's take a quick look into the future with Vue 3.

Bonus: Vue 3

A guide on learning Vue wouldn't be complete without mentioning Vue 3! Its release has been set for later this year (2020).

To stay updated on upcoming Vue 3 news, you can refer to this blog link: Vue 3 – A roundup of infos about the new version of Vue.js (https://madewithvuejs.com/blog/vue-3-roundup).

Vue features, and enhancements, are driven by two factors: The Vue Core team, and The Community! If you follow the Vue RFC (https://github.com/vuejs/rfcs) GitHub repository, you can see a list of the Request For Comments (RFC) the Vue Core team is proposing for inclusion in the upcoming release. They are inviting developers all over the world to participate by commenting, and discussing, the RFC before they make final decisions on any new implementations that go out for shipping in the framework.

The Vue 3 features coming are non-breaking changes from the perspective of a developer. The syntax you use for Vue 2 can be applied to building Vue 3 apps, with no major changes to note. The major shift apparent in Vue 3 is found within the internal framework. Here is where the bulk of the changes have been made.

You can start using, and exploring, Vue 3 features thanks to the Vue CLI Plugin for Vue Next (https://github.com/vuejs/vue-cli-plugin-vue-next?ref=madewithvuejs.com). To use this plugin, start by creating a new Vue app with the existing Vue CLI (Refer to Section 4 on creating Vue apps).

To install the Vue CLI Vue Next plugin, run this command inside your app:

```
vue add vue-next
```

The result is summarized as follows:

- Adds Vue 3 alpha and `@vue/compiler-sfc` to the project dependencies.
- Configures webpack to compile .vue files with the new Vue 3 compiler.
- A simple codemod that automatically migrates some global API changes mentioned in RFC#0009 - Global API Change (https://github.com/vuejs/rfcs/blob/master/active-rfcs/0009-global-api-change.md).

Here's the lowdown on a few of the new features with the relevant links to help get you more informed.

First of all, check out this video: Intro to Vue 3 features (https://www.youtube.com/watch?v=HmdKqXP8JR8) by Dan Vega. This video gives you a comprehensive look at the upcoming Vue 3 framework.

Global API Tree Shaking

This feature is illustrated in this RFC: RFC#0004 Global API Tree Shaking (https://github.com/vuejs/rfcs/blob/master/active-rfcs/0004-global-api-treeshaking.md).

Tree-shaking is achieved by making the Vue runtime tree-shakable. We do this by exposing as many APIs through [named exports] as possible. The Vue Core team worked thoroughly on this, ensuring as much exposure as is feasible.

Named and default exports features belong to ES 6 Modules (https://exploringjs.com/es6/ch_modules.html).

By using named exports, and with the help of Webpack 4 Tree Shaking (https://webpack.js.org/guides/tree-shaking/), the Vue runtime can do the tree-shaking to eliminate any non-reachable, or dead code, to minimize the size of the final package.

Global API Changes

This feature is illustrated in this RFC: RFC#0009 Global API Change (https://github.com/vuejs/rfcs/blob/master/active-rfcs/0009-global-api-change.md).

This change affects bootstrapping a Vue app. The intention of this change is to minimize cluttering the main Vue instance by shifting registering global stuff on the main Vue instance to seperate app instances. Currently, Vue 2 doesn't have the concept of an app. For example, you register global mixins (https://vuejs.org/v2/guide/mixins.html) or global components on the main Vue instance. Basically, if you want to serve two apps, both will share the same global settings.

Vue 3 changes this behavior by introducing the concept of standalone apps.

Below, you can find a typical Vue 2 bootstrapping code:

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

Vue.config.ignoredElements = [/^app-/]
Vue.use(/* ... */)
Vue.mixin(/* ... */)
Vue.component(/* ... */)
Vue.directive(/* ... */)

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

While the same bootstrapping code in Vue 3 becomes:

```
import { createApp } from 'vue'
import App from './App.vue'

const app = createApp(App)

app.config.isCustomElement = tag => tag.startsWith('app-')
app.use(/* ... */)
app.mixin(/* ... */)
app.component(/* ... */)
app.directive(/* ... */)

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

You import a new function named `createApp` from the `vue` package. Then, you use this function to create a new application named `app` by passing a root component.

Then you start configuring the `app` instance with plugins, mixins, components, directives, etc.

This way, you can create multiple Vue apps following this technique.

Another benefit of this change is to isolate Vue Tests in your app without cluttering them with any pre-configured stuff.

Make sure to read the RFC referenced above for a more detailed explanation on the motives behind this change.

Composition API

This feature is illustrated in this RFC: RFC#00013 Composition API (https://github.com/vuejs/rfcs/blob/master/active-rfcs/0013-composition-api.md).

Using the Component Options API (https://vuejs.org/v2/api/#Options-Data) has been the traditional way of building components adopted in Vue 2. A component has several options a developer can use to implement certain features inside of it. With several options, a developer can use, and implement, a certain feature inside the component.

The new Composition API is Vue's approach to building Components in Vue 3. It's a set of additive, function-based APIs, that allow flexible composition of component logic.

With the new approach in hand, you group all the code required for one feature inside a single composable function. This function can live inside the component itself, or in its own file. Thus, your component gets composed out of smaller functions/features.

Therefore, a feature implemented in a Vue app is spread across multiple Options in the current Vue 2 runtime. While, in Vue 3, the same feature is built inside a single composable function. This way, readability and maintainability are greatly improved!

I won't delve into more details on the new Composition API. However, I invite you to read my article on this topic: Vue 3 Composition API, do you really need it? (https://labs.thisdot.co/blog/vue-3-composition-api-do-you-really-need-it)

Fragments AKA Multiple Root Nodes

This feature is mentioned in this RFC: RFC#00012 Custom Directives API Change (https://github.com/vuejs/rfcs/blob/a2fb9703dc93ab0c4f5edd9ea2b0baa8c40792c8/active-rfcs/0012-custom-directive-api-change.md#usage-on-components).

Currently, in Vue 2, you cannot have multiple roots inside a single `<template>`. Most of the time we overcome this limitation by introducing an extra `<div>` to act as a single root.

Vue 3 introduces the Fragment API to allow having multiple root nodes in a single `<template>`.

This short video by Dan Vega introduces Vue 3 Fragments:
(https://www.youtube.com/watch?v=iC9smVwm7GE)

New Reactivity System

Vue 2 runtime accomplishes reactivity through getters, and setters, as defined in the Object.defineProperty (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty) method. Let's check a very simplified version of code that explains how reactivity works in Vue 2:

```
Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get:function(){
       return value;
    },
    set:function(newValue){
        if(value !== newValue){
           value = newValue;
    notifyValueChanged(); // notify the value has changed
        }
    }
});
```

Vue 2 reads the object returned from the Component `data()` function, loops over the keys of the object, and defines them as properties on the Vue instance.

This method has its own caveats. Two well-known issues are:

1. `Object.defineProperty` doesn't work with arrays, hence, changing an array by index is not reactive in Vue 2.
2. Dynamic properties are not reactive. You've got to use `Vue.set(this.$data,'PROPERTY NAME, PROPERTY VALUE);` to add a new reactive property to a Vue Component.

Vue 3 implements reactivity via Proxies (http://exploringjs.com/es6/ch_proxies.html#sec_proxies-explained). Proxies were introduced in ES6 or ES2015. By using them, the above two caveats no longer exist! Here's an example of defining a Proxy to wrap an array defined on the object returned by the `data()` function:

```
let data = {
   cities:[]
};

data.cities = new Proxy(data.cities, {
    Set: function(obj, prop, value) {
            If (obj[prop] !== value) {
                obj[prop] = value;
                notifyValueChanged(); // notify the value has changed
            }
        }
});
```

I highly recommend this article on Reactivity in Vue 2 vs. Vue 3 (https://blog.cloudboost.io/reactivity-in-vue-js-2-vs-vue-js-3-dcdd0728dcdf).

In Closing

Thank you for choosing to purchase or download a copy of The Ultimate Guide to VueJS. While it is my hope that readers will use this guide as a springboard to launch their further exploration into this amazing framework, I also believe that this will be a helpful resource to keep on hand in order to reference core concepts in VueJS in the future. If you have any questions about concepts discussed in this book, please do not hesitate to reach out to me on Twitter (https://twitter.com/bhaidar).

I also want to express my sincere appreciation to everyone at This Dot Labs, and beyond, who supported me in bringing this book to the web development community. If you would like to learn more about the work that This Dot Labs does to support enterprise web development teams, as well as all of the free educational resources they make available through This Dot Media, visit https://labs.thisdot.co/.

About the Author

Graduating with majors in Computer Engineering and Computer Science and a distinction from the Lebanese American University, it was clear, early on, that Bilal Haidar would be an expert in his field.

After graduating, he lectured, mentored, and served as a guest speaker at the very University from which he graduated.

He then went on to become a published author (Professional ASP.NET 3.5 Security, Membership, and Role Management with C# and VB) at just 26, before being picked up by Microsoft, and awarded 10 consecutive annual MVP awards in ASP.NET. Additionally, he earned a Microsoft Certified Trainer tag, and a Project Management Professional PMP in 2012.

Today, as a Full Stack Engineer, he is highly regarded in his field. Fluent in ASP.NET Core, Node.js, PHP Laravel, Angular, Vue.js, JavaScript, GraphQL, Databases, among other technologies, he has become a highly sought after consultant.

With well over 100 CODE Magazine articles, online forums, and private clients, he has extensive knowledge of all things code and continues to grow and thrive in all aspects of his career.