

# VUE.JS

## TOOLS & SKILLS



BUILD YOUR OWN SOPHISTICATED WEB APPS

# Vue.js: Tools & Skills

Copyright © 2019 SitePoint Pty. Ltd.

Ebook ISBN: 978-1-925836-28-8

**Cover Design:** Alex Walker

**Project Editor:** James Hibbard

## Notice of Rights

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

## Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

## Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood

VIC Australia 3066

Web: [www.sitepoint.com](http://www.sitepoint.com)

Email: [books@sitepoint.com](mailto:books@sitepoint.com)

## About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit <http://www.sitepoint.com/> to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

# Preface

Since its release in 2014, Vue.js has seen a meteoric rise to popularity and is now considered one of the primary front-end frameworks, and not without good reason. Its component-based architecture was designed to be flexible and easy to adopt, making it just as easy to integrate into projects and use alongside non-Vue code as it is to build complex client-side applications.

This book covers a selection of essential tools and skills you'll need while working with Vue. It contains:

1. *Setting Up a Vue Development Environment* by James Hibbard
2. *Five Top Vue Animation Libraries* by Maria Antonietta Perna
3. *Build Your First Static Site with VuePress* by Ivaylo Gerchev
4. *Five Vue UI Libraries for Your Next Project* by Michiel Mulders
5. *Five Handy Tips when Starting Out with Vue* by David Bush

## Who Should Read This Book?

This book is for developers with experience of JavaScript.

## Conventions Used

### CODE SAMPLES

Code in this book is displayed using a fixed-width font,

like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back at s
```

Where existing code is required for context, rather than repeat all of it, `:` will be displayed:

```
function animate() {
    :
    new_variable = "Hello";
}
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An `↵` indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-
↵design-real-user-testing/?responsive1");
```

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

## TIPS, NOTES, AND WARNINGS

...

### **Hey, You!**

Tips provide helpful little pointers.

### **Ahem, Excuse Me ...**

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

### **Make Sure You Always ...**

... pay attention to these important points.

### **Watch Out!**

Warnings highlight any gotchas that are likely to trip you up along the way.

# Chapter 1: Setting Up a Vue Development Environment

**BY JAMES HIBBARD**

If you're going to do any serious amount of work with Vue, it'll pay dividends in the long run to invest some time in setting up your coding environment. A powerful editor and a few well-chosen tools will make you more productive and ultimately a happier developer.

In this tutorial, I'm going to demonstrate how to configure VS Code to work with Vue. I'm going to show how to use ESLint and Prettier to lint and format your code and how to use Vue's browser tools to take a peek at what's going on under the hood in a Vue app. When you've finished reading, you'll have a working development environment set up and will be ready to start coding Vue apps like a boss.

Let's get to it!

## Installing and Setting Up Your Editor

I said that I was going to be using VS Code for this

tutorial, but I'm afraid I lied. I'm actually going to be using VSCodium, which is an open-source fork of VS Code without the Microsoft branding, telemetry and licensing. The project is under active development and I'd encourage you to check it out.

It doesn't matter which editor you use to follow along; both are available for Linux, Mac and Windows. You can download the latest release of VSCodium here, or download the latest release of VSCode here and install it in the correct way for your operating system.

Throughout the rest of this guide, for the sake of consistency, I'll refer to the editor as VS Code.

## **ADD THE VETUR EXTENSION**

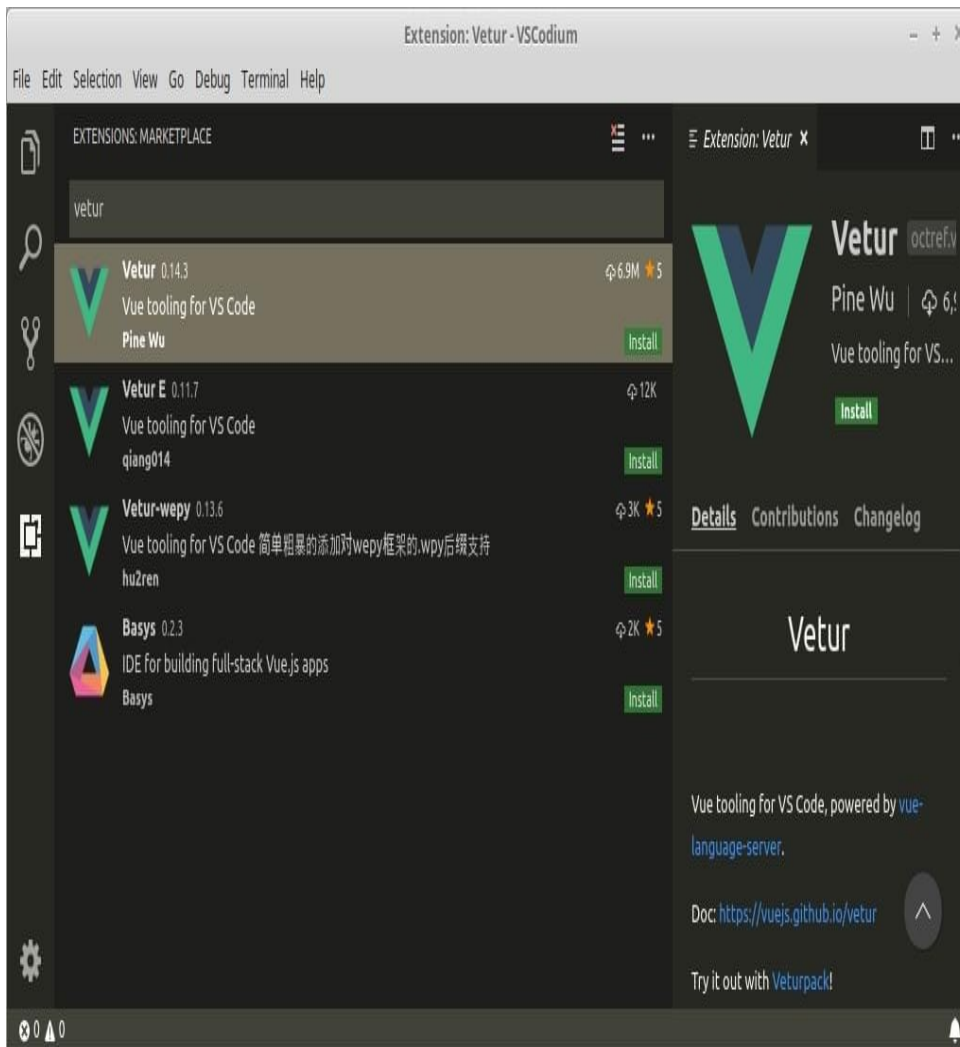
When you fire up the editor, you'll notice a set of five icons in a toolbar on the left-hand side of the window. If you click the bottom of these icons (the square one), a search bar will open up that enables you to search the VS Code Marketplace. Type "vue" into the search bar and you should see dozens of extensions listed, each claiming to do something slightly different.

Depending on your use case, you might find something here to suit you. There are lots available. For example, TSLint for Vue could be handy if you're working on a Vue project involving TypeScript. For now, I'm going to focus on one called Vetur.

Type "Vetur" into the search box and select the package



authored by Pine Wu. Then hit *Install*.



Once the extension has installed, hit *Reload to activate* and you're in business.

## EXPLORING VETUR'S FEATURES

If you visit the project's home page, you'll see that the extension gives you a number of features:

- syntax highlighting
- snippets

- Emmet
- linting/error checking
- formatting
- auto completion
- debugging

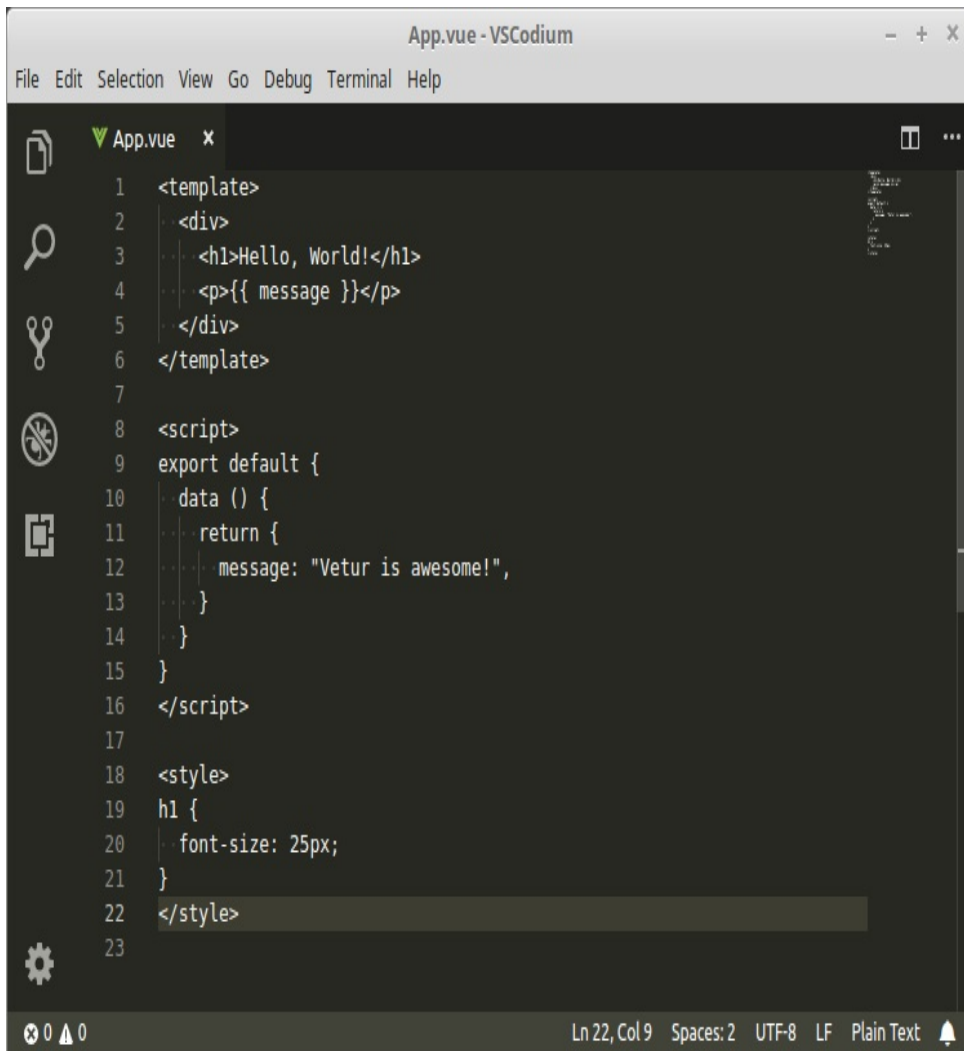
Let's see some of these in action now.

### **Set Up Your Project First**

Note that many of these features only work when you have a project set up. This means you need to create a folder to contain your Vue files, open the folder using VS Code and access the files via VS Code's explorer.

## **Syntax highlighting**

As your app grows, you'll undoubtedly want to make use of single-file components (SFCs) to organize your code. These have a `.vue` ending and contain a template section, a script section and a style section. Without Vetur, this is what an SFC looks like in VS Code:

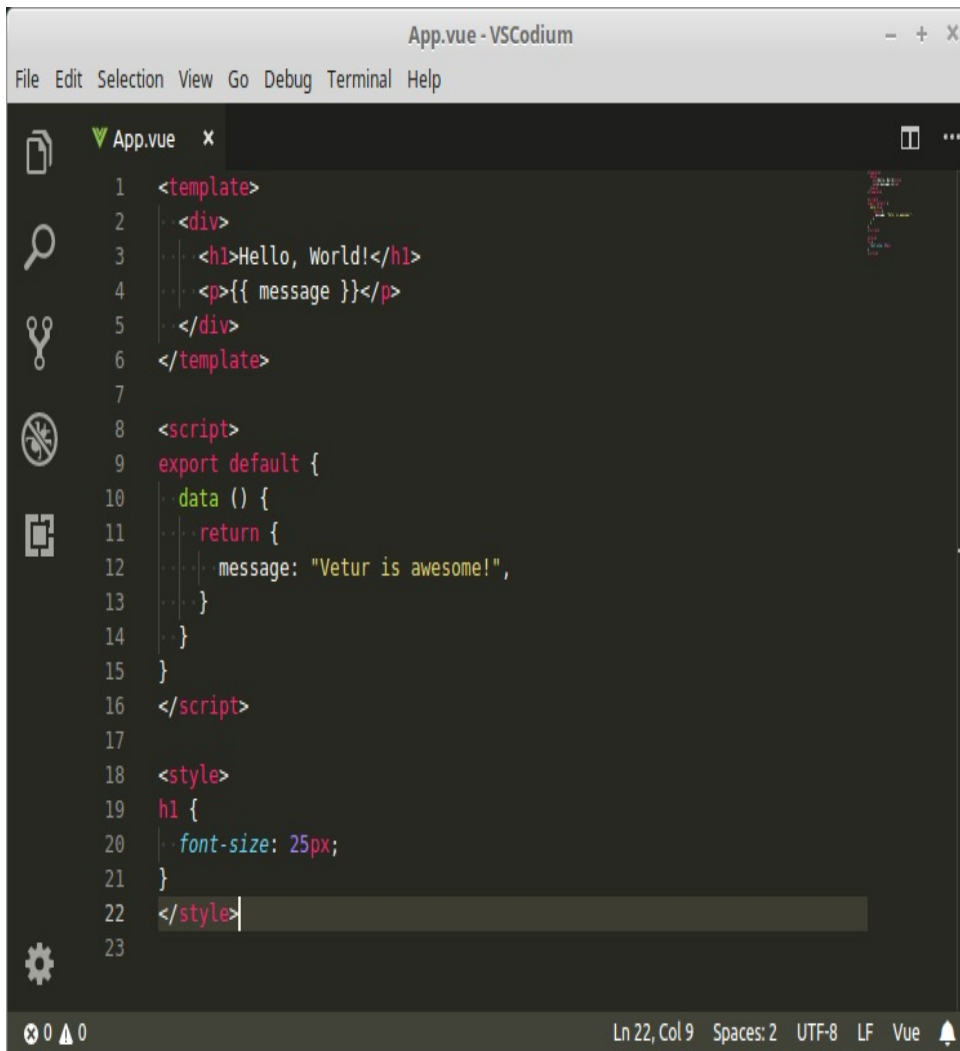


The image shows a screenshot of the Visual Studio Code (VSCode) editor window titled "App.vue - VSCode". The editor displays a Vue.js single-file component (App.vue) with the following code:

```
1 <template>
2   <div>
3     <h1>Hello, World!</h1>
4     <p>{{ message }}</p>
5   </div>
6 </template>
7
8 <script>
9   export default {
10     data () {
11       return {
12         message: "Vetur is awesome!",
13       }
14     }
15   }
16 </script>
17
18 <style>
19   h1 {
20     font-size: 25px;
21   }
22 </style>
23
```

The code is syntax-highlighted, and the editor interface includes standard VSCode icons on the left (Explorer, Search, Source Control, Run and Debug, Extensions) and a status bar at the bottom showing "Ln 22, Col 9", "Spaces: 2", "UTF-8", "LF", "Plain Text", and a bell icon.

However, installing Vetur will make it look like so:



```
App.vue - VSCodeium
File Edit Selection View Go Debug Terminal Help

1 <template>
2   <div>
3     <h1>Hello, World!</h1>
4     <p>{{ message }}</p>
5   </div>
6 </template>
7
8 <script>
9   export default {
10     data () {
11       return {
12         message: "Vetur is awesome!",
13       }
14     }
15   }
16 </script>
17
18 <style>
19   h1 {
20     font-size: 25px;
21   }
22 </style>
23
```

Ln 22, Col 9 Spaces: 2 UTF-8 LF Vue

## Snippets

As you can read on the [VS Code website](#), snippets are templates that make it easier to enter repeating code patterns, such as loops or conditional-statements. Vetur makes it possible for you to use these snippets in single-file components.

It also comes with some snippets of its own. For example, try typing “scaffold” (without the quotes) into an area outside a language region and it will generate all the code you need to get going with an SFC:

```
<template>

</template>

<script>
export default {
}
</script>

<style>

</style>
```

## Emmet

Emmet takes the idea of snippets to a whole new level. If you've never heard of this and spend any amount of time in a text editor, I'd recommend you head over to the [Emmet website](#) and spend some time acquainting yourself with it. It has the potential to boost your productivity greatly.

In a nutshell, Emmet allows you to expand various abbreviations into chunks of HTML or CSS. Vetur ships with this turned on by default.

Try clicking into a `<template>` region and entering the following:

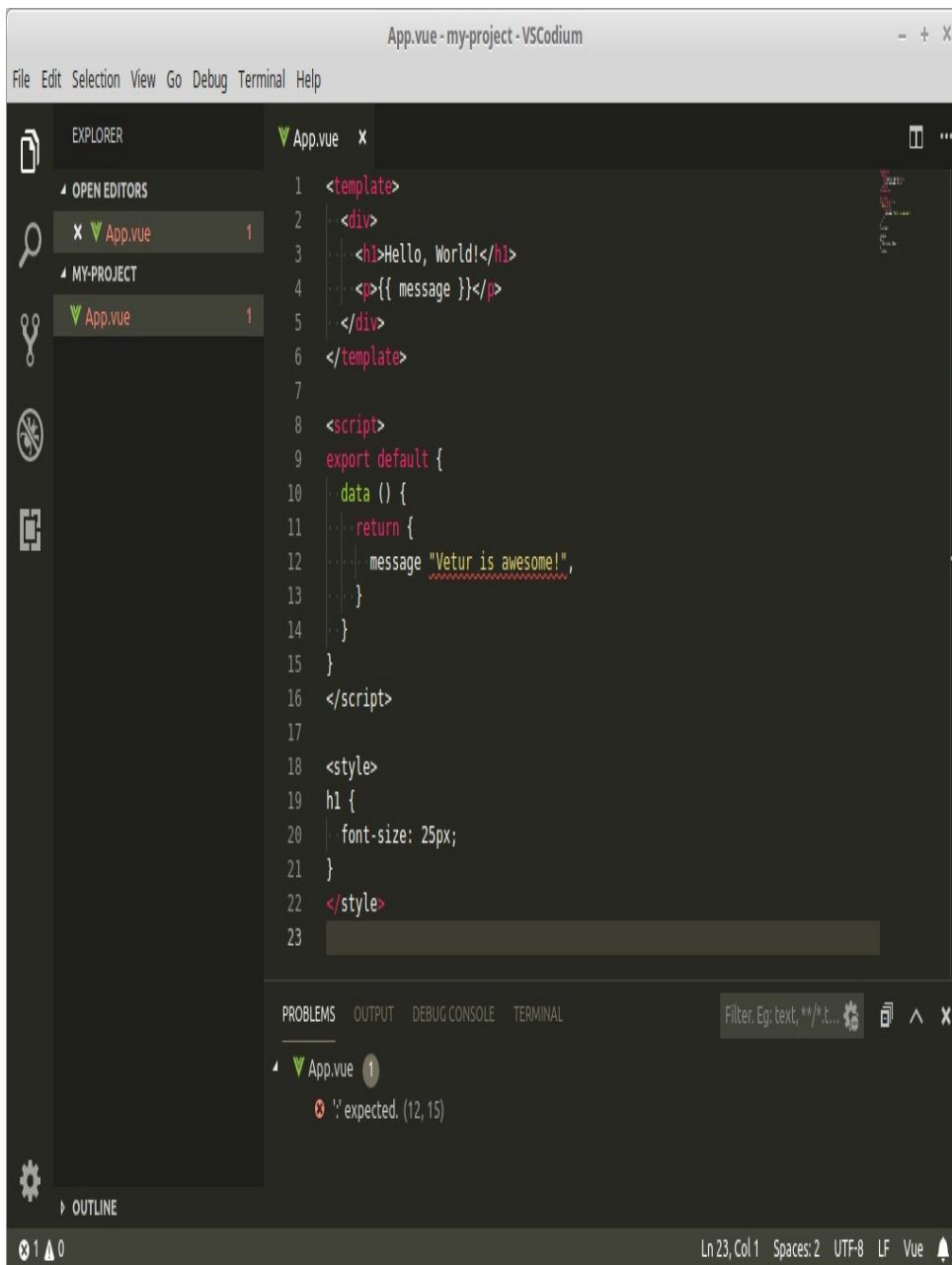
```
div#header>h1.logo>a{site Name}
```

Then press `Tab`. It should be expanded to this:

```
<div id="header">
  <h1 class="logo"><a href="">sitename</a></h1>
</div>
```

## Error Checking/Linting

Out of the box, Vetur offers some basic error checking. This can be handy for spotting typos in your code.



In the above example, Vetur has noticed that I've forgotten the colon following the object property name and has consequently underlined it in red. Opening up the error panel (click on the small cross in the bottom left-hand corner) gives you a description of the error.

Vetur also uses [eslint-plugin-vue](#) to lint templates. We'll

find out more about why linting your code is a good idea in the next section, but for now, let's just see it in action.

Let's add an `items` key to our data property:

```
export default {  
  data () {  
    return {  
      message: "Vetur is awesome!",  
      items: [  
        { message: 'Foo' },  
        { message: 'Bar' }  
      ]  
    }  
  }  
}
```

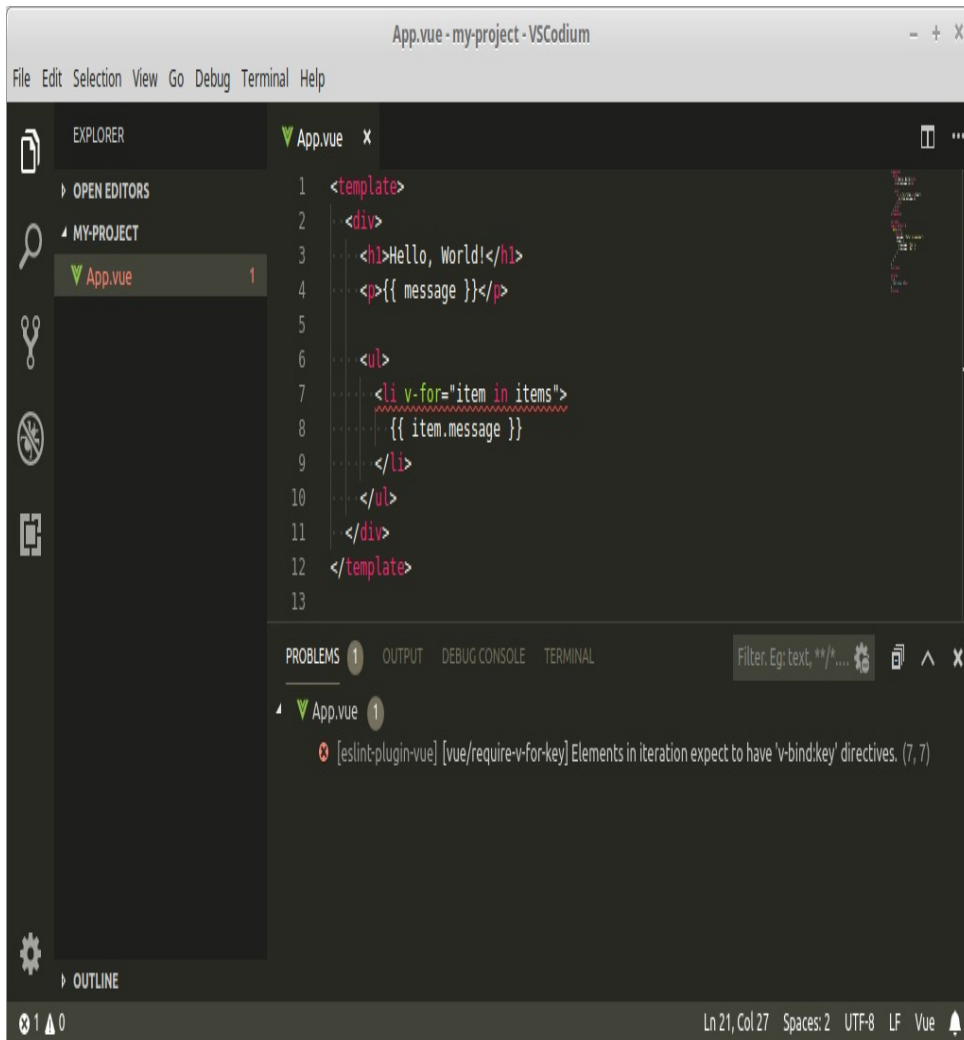
Then add some code to output it in our template:

```
<template>  
  <div>  
    <h1>Hello, World!</h1>  
    <p>{{ message }}</p>  
  
    <ul>  
      <li v-for="item in items">  
        {{ item.message }}  
      </li>  
    </ul>  
  </div>  
</template>
```

Straight away we'll see that `<li v-for="item in items">` is underlined in red. What gives?

Well, you can hover over the offending code, or open the error console to see what's bothering Vetur.





The error appears to be that we've forgotten to declare a key. Let's remedy that:

```
<li v-for="(item, i) in items" :key="i">
  {{ item.message }}
</li>
```

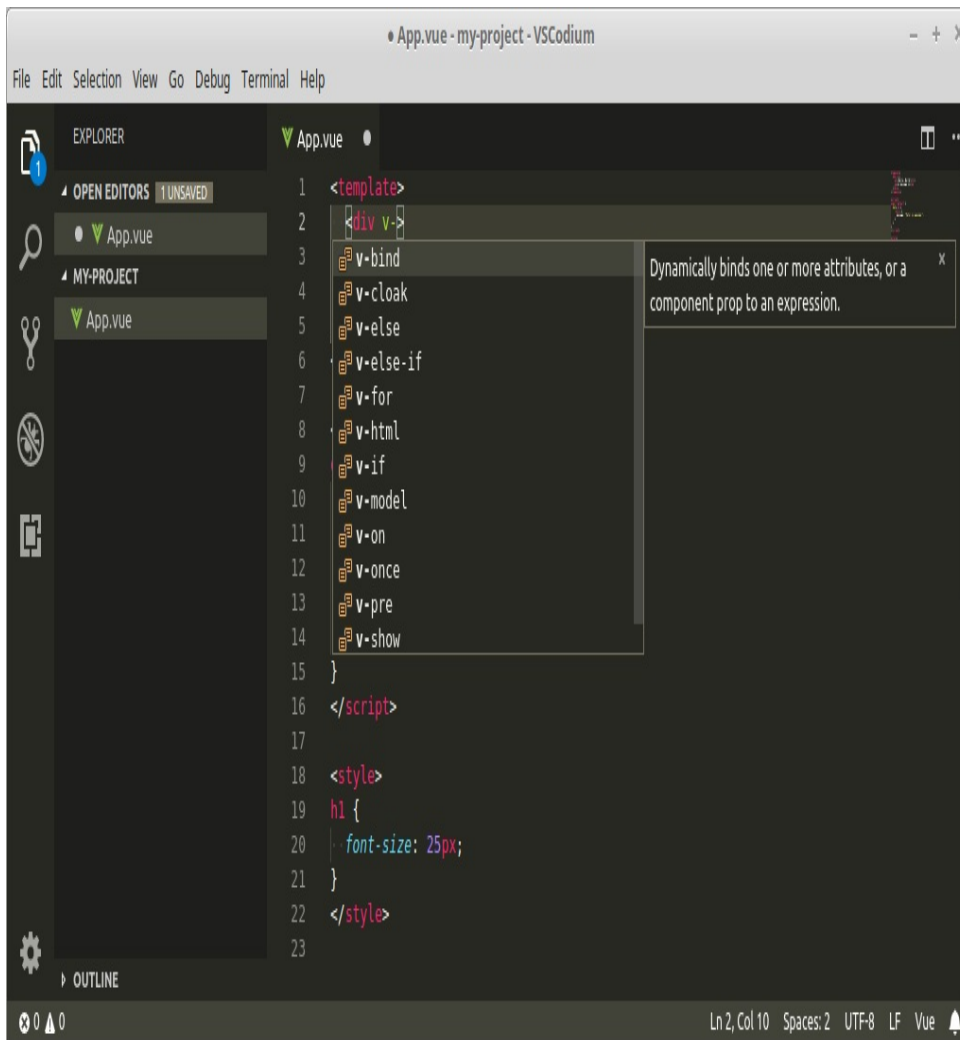
And the error vanishes from our editor.

## IntelliSense

IntelliSense is one of my favorite features in VS Code, but

it's limited to a few formats that the editor can understand. Installing Vetur makes IntelliSense available in your `.vue` file, which is mighty handy.

You can try this out by clicking into the `<template>` region of a Vue component and typing “v-” on any tag (minus the quotes). You should see this:



This allows you to select from any of the listed directives, and also provides you with an explanation of what each does.

That's not all that Vetur can do, but we'll leave the extension there and turn our attention to setting up a project with Vue's CLI.

## An Example Project with Vue CLI

When building a new Vue app, the best way to get up and running quickly is using Vue CLI. This is a command-line utility that allows you to choose from a range of build tools which it will then install and configure for you. It will also scaffold out your project, providing you with a pre-configured starting point that you can build on, rather than starting everything from scratch.

### Getting Up to Speed with Vue CLI

If the CLI is new for you, you might like to check out our tutorial "A Beginner's Guide to Vue CLI" in this Vue series.

To get started, you'll need to have Node installed on your system. You can do this by downloading the binaries for your system from the official website, or using a version manager. I recommend the second of the two methods.

With Node installed, execute the following command:

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

And create a new Vue project with the command:

```
vue create my-project
```

This will open a wizard which asks you to choose a preset. Choose to manually select features, then accept the defaults for everything, apart from when you're asked to pick a linter/formatter config. In this step, be sure to select *ESLint* + *Prettier* and *Lint on save*, and to place config files in `package.json`.

## LINTING YOUR CODE WITH ESLINT

If you open this newly created project and take a peek inside the `package.json` file, you'll notice that the CLI has set up ESLint for you. This is a tool that can automatically check your code for potential problems. This provides many benefits, such as:

- keeping your code style consistent
- spotting potential errors and bad patterns
- enforcing quality when you follow a style guide
- saving time for all of the above reasons

### Learning ESLint

If you'd like to dive deeper into ESLint, check out our article "[Up and Running with ESLint — the Pluggable JavaScript Linter](#)".

If you look at the `devDependencies` property in `package.json`, you'll see that the CLI is also using eslint-plugin-vue. This is the official ESLint plugin for

Vue.js, which is able to detect code problems in your `.vue` files.

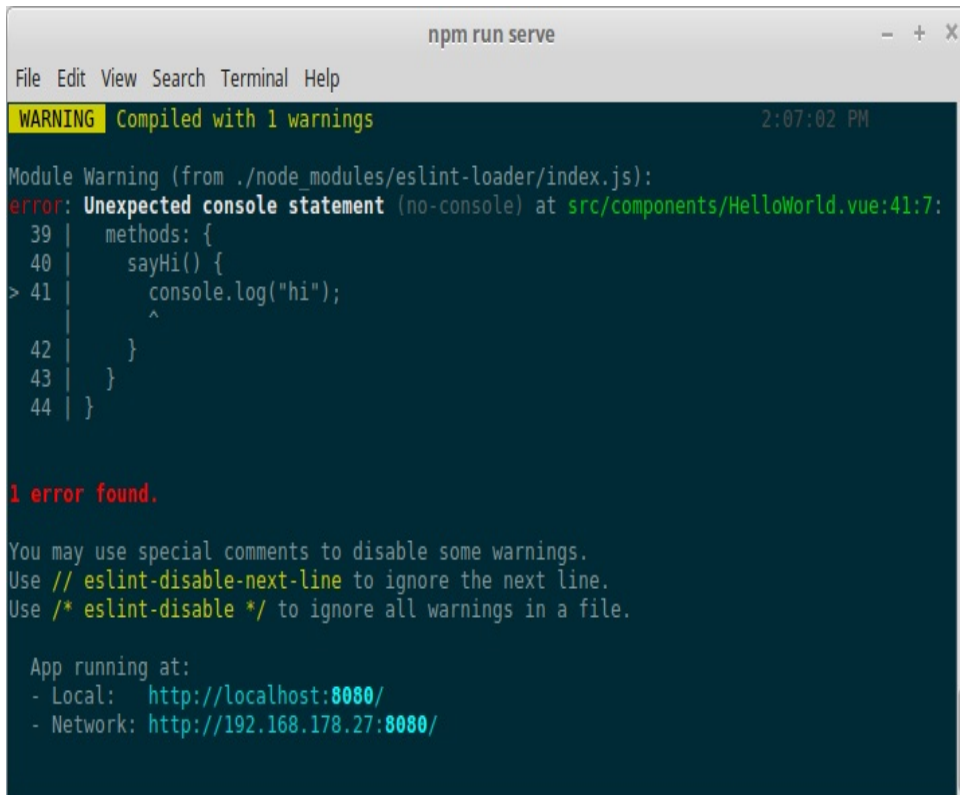
Let's put that to the test.

Start the Vue dev server with `npm run serve` and navigate to localhost:8080.

In VS Code, open up the project you just created with the CLI (*File > Open Folder*), then navigate to `src/components/HelloWorld.vue` in the VS Code explorer. Let's add a method to our Vue instance:

```
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  },
  methods: {
    sayHi() {
      console.log("hi");
    }
  }
}
```

Now, if you look at the terminal window in which the dev server is running, you'll see Vue complaining.



```
npm run serve
File Edit View Search Terminal Help
WARNING Compiled with 1 warnings 2:07:02 PM
Module Warning (from ./node_modules/eslint-loader/index.js):
error: Unexpected console statement (no-console) at src/components/HelloWorld.vue:41:7:
 39 |   methods: {
 40 |     sayHi() {
> 41 |       console.log("hi");
    |       ^
 42 |     }
 43 |   }
 44 | }

1 error found.

You may use special comments to disable some warnings.
Use // eslint-disable-next-line to ignore the next line.
Use /* eslint-disable */ to ignore all warnings in a file.

App running at:
- Local: http://localhost:8080/
- Network: http://192.168.178.27:8080/
```

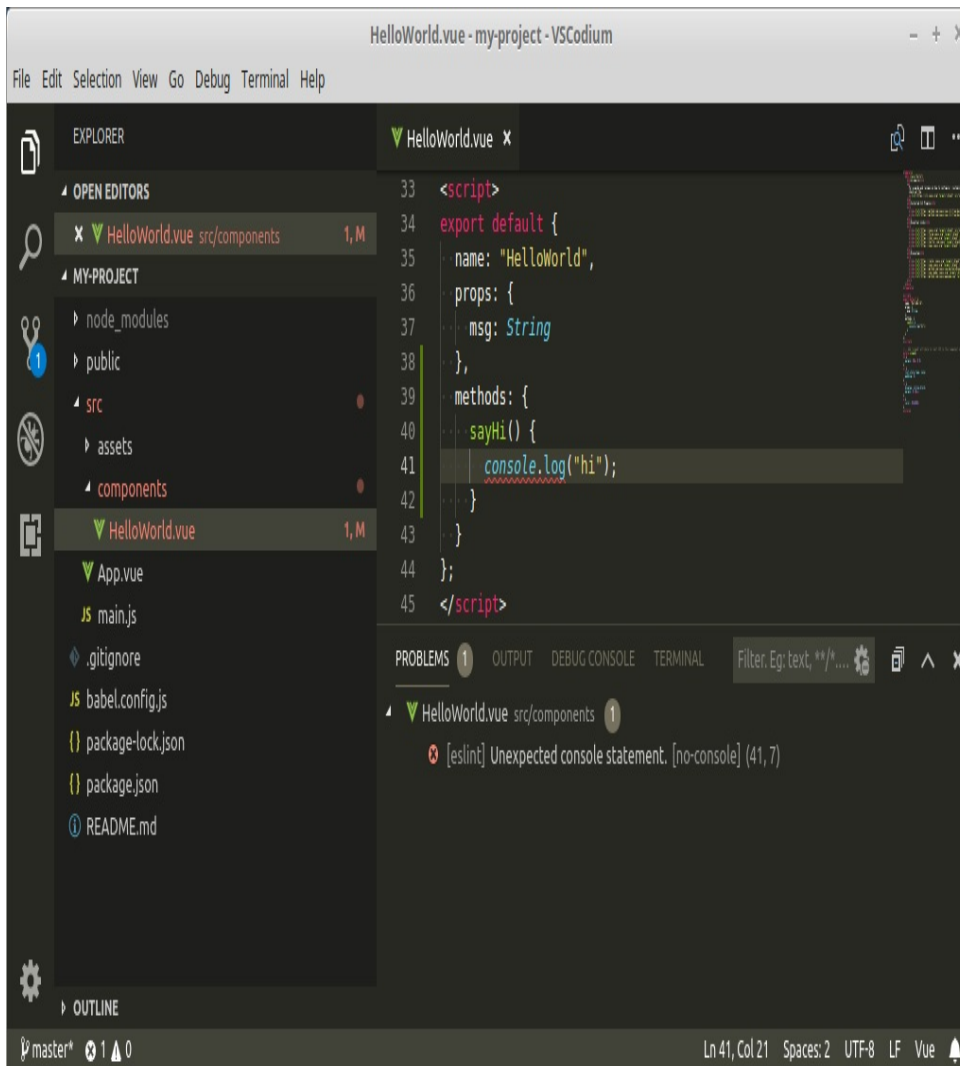
This is because, under the hood, Vue CLI has configured ESLint to use the `eslint:recommended` ruleset. This enables any rules marked with a check mark on the [ESLint rules page](#), of which [no-console](#) is one.

While it's nice that the CLI shows us ESLint errors in the terminal, wouldn't it'd be nicer if we could see them in our editor, too? Well, luckily we can. Hop back into VS code, click the extensions icon and type in "ESLint" (without the quotes). The top result should be for a package named [ESLint](#) by Dirk Baeumer. Install that and restart VS Code.

Finally, you'll need to edit your VS Code preferences. Go to *File > Preferences > Settings* and edit the *User Settings file* and add the following configuration:

```
"eslint.validate": [  
  "vue"  
]
```

This will tell the ESLint plugin we just installed to perform validation for `.vue` files.



Should you desire, you can turn this (or any) rule off in the `rules: {}` section of `package.json`:

```
"eslintConfig": {
```

```
...
"rules": {
  "no-console": 0
},
...
}
```

## FORMATTING YOUR CODE WITH PRETTIER

Prettier is an opinionated code formatter. As you can read on [the project's home page](#), it enforces a consistent style by parsing your code and re-printing it with its own rules that take the maximum line length into account, wrapping code when necessary.

That might sound a little draconian at first, but once you get used to it, you literally never have to think about code formatting again. This is very useful if you're part of a team, as Prettier will halt all the on-going debates over styles in their tracks.

### Look Prettier

If you're not convinced, you can [read more about why you should use Prettier here](#).

The way Prettier works in conjunction with Vue CLI is similar to ESLint. To see it in action, let's remove the semicolon from the end of the `console.log("hi");` statement from our previous example. This should display a warning in the terminal:

---



```
warning: Insert `;` (prettier/prettier) at src/componen
 39 |   methods: {
 40 |     sayHi() {
> 41 |       console.log("hi")
    |                               ^
 42 |     }
 43 |   }
 44 | };

1 error and 1 warning found.
1 warning potentially fixable with the `--fix` option.
```

It will also display a warning in VS Code, thanks to the ESLint plugin we installed previously.

We can also have Prettier fix any formatting errors for us whenever we save a file. To do this, go to *File > Preferences > Settings* and edit the *User Settings* file to add the following configuration:

```
"editor.formatOnSave": true
```

Now when you press save, everything will be formatted according to Prettier's standard rule set.

Note, you can configure a handful of rules in Prettier via a "prettier" key in your `package.json` file:

```
"prettier": {
  "semi": false
}
```

The above, for example, would turn the semicolon rule off.

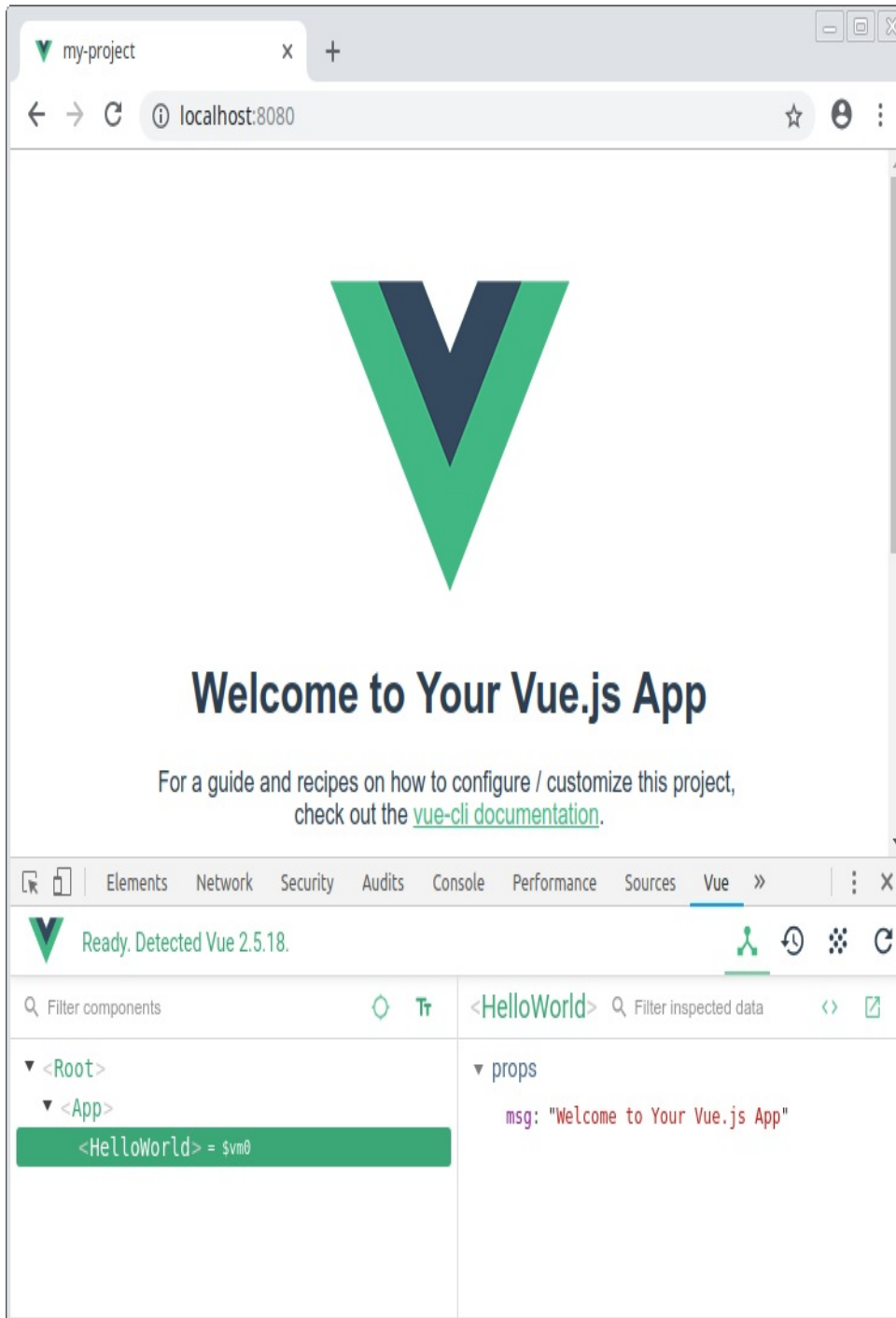
You can read more about configuration options [here](#).

## Vue Browser Tools

In this section, I want to take a look at the the Vue.js devtools, which are available as a browser plugin for both [Chrome](#) and [Firefox](#), and as a [cross-platform Electron app](#), which can also debug Vue apps running on mobile devices.

Once you have them installed, you can access them by visiting a running Vue app, opening your browser's console and hitting the *Vue* button. You'll then see three further sections: *Components*, *Vuex* and *Events*.

The first section gives you a hierarchical view of all the components that make up your application. Selecting a component from the tree allows you to inspect its state (for example, the `props` it received) in the right-hand pane. Some of its values (such as its `data` object) can be dynamically edited while the app runs.



The *Vuex* tab is only active if a Vuex store is detected in the application. (For more information on this, check out “Getting Started with Vuex: a Beginner’s Guide” in this Vue series.) It allows you to examine the state of the store at any point in time, and all the mutations that

have been committed. You can even move back and forth through the mutations, effectively time traveling through the state of your application.

The *Events* tab aggregates all the events emitted by your application, from anywhere in the component tree. Selecting an event will display more information about it in the right-hand pane, allowing you to see which component emitted it and any payload that was sent.

There's a lot more to Vue's browser tools than I've demonstrated here, so I encourage you to install them and experiment with them as your application grows.

## Conclusion

And that's a wrap. In this guide, I've demonstrated how to install the Vetur plugin for VS Code and highlighted several of its features. I've also shown how to use Vue CLI to generate a project and how to use ESLint and Prettier to ensure code quality and consistency. We also took a brief look at Vue's browser tools and saw how to inspect the state of a running Vue application, something which is important for debugging purposes.

This should see you well on the way to having a sensible environment for writing Vue applications and hopefully make you a productive and happy developer.

# Chapter 2: Five Top Vue Animation Libraries

**BY MARIA ANTONIETTA PERNA**

**In this article, we'll look at what Ethereum nodes are, and explore one of the most popular ones, called Geth.**

Animation can be your a powerful tool in your UX toolbox. It keeps web users engaged, and it's lots of fun to create. Vue.js is a progressive JavaScript framework that lets you build user interfaces and powerful web apps.

Follow me as I show you how you can quickly set Vue in motion with five great web animation libraries.

## Why Animation on the Web?

Our brains are hardwired to pay attention and react to things that move around us. This is a survival mechanism, but also part of the way we understand the world. Websites are communication tools, and animation is part of the communication strategy around a web design project. As such, web animation, far from simply being decorative, is functional to the success of a

website's goals, and therefore plays a number of important roles.

## A FEW WORDS OF CAUTION ABOUT WEB ANIMATION

When it comes to web animation, usually less is more. Going overboard with your animations is tempting, but it's often a consequence of poor design. To avoid falling into this trap, it's useful to keep in mind that your animation should have a point and should facilitate what users are expected to do on your website. In other words, animation must not be in the way.

Another aspect of animation you should be aware of is that motion on the Web can have some serious accessibility implications:

It's no secret that a lot of people consider scrolljacking and parallax effects annoying and overused. But what if motion does more than just annoy you? What if it also makes you ill? — *Val Head, Designing Safer Web Animation for Motion Sensitivity*

Val Head refers to visually triggered vestibular disorders, which can cause symptoms of nausea, dizziness, headaches and even worse, at the sight of large-scale motion on screen. One way in which you can minimize discomfort to your users is to give them control over whether they want to start an animation, and at what speed. For more details on this topic, check out More Resources for Accessible Animations by Val Head, which,

together with the article cited above, covers a great deal of ground and goes a long way towards helping you craft animations designed with accessibility and respect for your website's users in mind.

## WHAT WEB ANIMATION IS GOOD FOR

Humans are drawn towards movement, so you can use animation to convey information through motion. For instance, color animation on hover draws users' attention to the link. Animating an area or an element of a web page gives users cues about where they're expected to look or what they're expected to do. A fun loader communicates to users that some process is taking place on the website and keeps them engaged so the perceived waiting time feels shorter.

Also, the way you design an animation contributes to the creation and reinforcement of brand identity while also giving your website a sophisticated touch of professionalism and polish.

From the point of view of front-end devs, there's never been a better time for coding motion experiences on the web. Browsers' handling of animation keeps improving, devices are more and more capable, and there are some amazing tools out there you can use to create web animations.

Let's explore some possibilities in relation to Vue.js.

### Vue.js Transition and Animation

# vue.js Transition and Animation Systems

Applying CSS transitions and animations is a snap with Vue's `<transition>` and `<transition-group>` components. Before I illustrate how these work, let's briefly clarify the difference between CSS transitions and CSS animations.

## CSS TRANSITIONS

CSS transitions involve animating the state of a property from an initial starting point to an end state in response to an event—such as a mouse hover, or a hover event. The browser will take care of interpolating all the in-between states of the animation. Therefore, if your animation consists of just changing a property from A to B, transitions will be the simplest and most efficient way of implementing it.

## CSS ANIMATIONS

CSS animations are great if your animation consists of more than just an initial and final state and therefore you need more control over each stage throughout the movement.

For more details on CSS transitions and animations, their differences and use cases, check out “CSS Transitions 101: Let's Animate a Toggle Button Icon”.

## CSS TRANSITIONS IN VUE



Vue already makes hiding and showing content on the page a painless and quick operation, using the v-if or v-show directives:

```
<div id="app">
  <button v-if="show" @click="show = !show">Show</button>
</div>

<script>
new Vue({
  el: '#app',
  data() {
    return {
      show: true
    }
  }
})
</script>
```

The snippet above shows the button on the page only if the `show` property in the Vue instance is `true`. Clicking the button causes the `show` property to switch to `false`, which in turn leads to the removal of the button element.

However, this approach produces a sudden showing/hiding of the element, which doesn't give users a great experience. Enter CSS transitions and Vue's `<transition>` component: the latter is a convenient wrapper that Vue makes available to let you add transitions to any element or component.

Here's how you'd use it to animate the button in the above example:

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

```
<transition name="fade">
  <button v-if="show" @click="show = !show">Show</button>
</transition>
</div>
```

The button code is wrapped inside a `<transition>` component named *fade*.

In the CSS code, the *fade* CSS transition is built using Vue's `<transition>` component's hooks as follows:

```
.fade-leave-active{
  transition: opacity 2s;
}

.fade-leave-to {
  opacity: 0;
}
```

The `.fade-leave-active` hook is available while the animation is running, and it's here that you can specify the transition properties.

`.fade-leave-to` represents the last positions in the animation. In this case, by setting `opacity` to `0`, the button starts the animation by being visible and ends the animation by disappearing.

#### Live Code

[Here's a live demo](#) with both entering and leaving transition hooks.

You can get the lowdown on all the transition classes

available to you on the [Vue.js docs](#).

Vue also offers a `<transition-group>` component for animating more than one element in one go, which I'm going to illustrate later in this article.

## CSS ANIMATIONS IN VUE

You can add CSS animations to your Vue app using the same `<transition>` component and the same class hooks as CSS transitions, but instead of simply defining a starting point and an end point leaving the browser to figure out the in-between states, you take matters in your own hands using `@keyframes` to define each stage in the animation. The cool thing is that you can code your own animations or just as easily plug in a CSS animation library like `Animate.css` and you're off to the races!

Let's see how you can do just that.

### #1 Animate.css

Animate.css is a handy cross-browser library of sleek and playful CSS animations created by Daniel Eden.

You can quickly put it to use by adding the `animated` class to the element you intend to animate together with the name of the animation of your choice among all the great options this library has to offer. You can also add other animation-specific properties like how many times you want the animation to play, its `duration`, `delay`,

and so on.

For instance, here's a `<div>` element set up to bounce up and down for an infinite number of times:

```
<div class="animated infinite bounce delay-2s"></div>
```

To include `Animate.css` in your Vue app, add the library's classes to the hooks made available inside the `<transition>` component. Here's an example:

```
<div id="app">
  <transition name="rotateSlide" appear
    enter-active-class="animated rotateInUpRight"
    leave-active-class="animated slideOutUp">
    <div v-if="show" class="message" @click="move">
      {{ bubble }}
    </div>
  </transition>
</div>
```

Let's go through what happens in the snippet above:

- The Vue animation is called *rotateSlide*.
- The `appear` attribute ensures that the opening animation is triggered as soon as the page renders in the browser.
- The `animated`, `rotateInUpRight` and `slideOutUp` classes are `Animate.css` classes.
- Adding `rotateInUpRight` to Vue's `enter-active-class` causes the element to appear as it rotates on the right side towards the top of the screen.
- By the same token, adding `slideOutUp` to Vue's `leave-active-class` causes the element to slide to the top of the page and disappear. This second animation is triggered by clicking on the

element.

[Live Code](#)

[Here's the full demo.](#)

## ANIMATE.CSS WITH VUE'S <TRANSITION-GROUP> COMPONENT

The `<transition-group>` component is Vue's way of facilitating the animation of a bunch of elements like list items. Its use is similar to the `<transition>` component. Here's a snippet to show what it looks like with Animate.css:

```
<transition-group
  name="listitems"
  tag="ul"
  appear
  enter-active-class="animated bounceInUp"
>
  <li
    is="app-book"
    v-for="(book, index) in books"
    :key="index"
    :book="book">
  </li>
</transition-group>
```

Above, the `<transition-group>` component named *listitems* encloses a list item component. Notice the `tag` property inside the `transition-group` with a value of *ul*. This automatically generates the `<ul></ul>` tags around the list's `<li></li>` tags of the list component.

When the page first renders, the items in the list will make a smooth bouncing animation thanks to the `Animate.css bounceInUp` class.

#### Live Code

Here's [the full demo on CodePen](#). Feel free to play around with it.

## #2 Animate.js

Animate.js by Gibbok is—

a bunch of cool, fun, and cross-browser animations for you to use in your projects. Great for emphasis, home pages, sliders, and general just-add-water-awesomeness.  
— Animate.js

It's also a porting to the Web Animations API of the `Animate.css` project examined earlier.

It's a great option if you need to add a bit more dynamism to your CSS animations.

Below is a snippet taken from the library's docs to show you the basic usage:

```
<!-- Include the polyfill -->
<script src="//cdn.rawgit.com/web-animations/web-animat

<!-- Include Animate.js -->
<script src="//cdn.rawgit.com/gibbok/animate.js/1.0.3/c

<!-- Set up a target to animate -->
```

```
<h1 id="hello">Hello world!</h1>

<!-- Animate! -->
<script>
  window.animatelo.flip('#hello');
</script>
```

Here are a couple of things to notice:

- Since Animatelo uses the Web Animations API, it needs a polyfill, at least until wider browser support becomes available
- To set an element in motion, the syntax above with your chosen pre-built animation and a reference to your element is all you need. More generally, structure your code like this:  
`window.animatelo.animation(selector, options).`

The options Animatelo makes available are:

- `id` (optional): a DOM string you can use to reference your animation, which is specific to `Animate.css`.
- `delay` (optional): the number of milliseconds you want to delay the start of the animation. It defaults to 0.
- `direction` (optional): you can set this value to `normal` (the animation runs forward), `reverse` (the animation runs backwards), `alternate` (the animation switches direction after each iteration), `alternate-reverse` (the animation runs backwards and switches direction after each iteration).
- `duration` (optional): the number of milliseconds your animation lasts. It defaults to 1000 milliseconds.
- `fill` (optional): you can set this option to `backwards` (you can see the animation's effect before the animation even starts playing), `forwards` (the animation's effects are retained after the animation has finished playing), or `both`, which is the default value.
- `iterations` (optional): the number of times you want your animation to repeat. 1 is the default value, but you can pick any

other number, and if you want your animation to repeat indefinitely, just set it to `infinite`.

You can also use the `onfinish` property to trigger an animation as soon as another one completes:

```
const anim1 = window.animatelo.shake('#headline1', {
  delay: 500,
  duration: 1500
})[0];
anim1.onfinish = function() {
  const anim2 = window.animatelo.wobble('#headline2',
    duration: 1500
  )[0];
};
```

#### Live Code

Since Vue.js is so awesome for animation, making Animatelo work with Vue is also quite straightforward. Check out this cute entrance/exit effect and have fun deconstructing the code: See the Pen [Animatelo + Vue Demo](#)

## #3 Pose.js

Pose.js is an intuitive animation library tailor made for Vue, React and React Native. The documentation's opening lines describe it as follows:

Pose for Vue is a declarative motion system that combines the simplicity of CSS transitions with the power and flexibility of JavaScript.

The way this library works is by defining possible states, or poses, for your component, a bit like CSS animations.



Pose has got tons of features, so let's start with a simple example of a red box that scales up and then down on a click event:

```
new Vue({
  el: '#app',
  data() {
    return {
      isVisible: false,
      moving: true
    }
  },
  components: {
    Box: posed.div({
      pressable: true,
      init: {scale: 1},
      press: {scale: 2}
    })
  }
});
```

Inside your component's JS section, or in the case of this simple demo inside the Vue instance, you need to add a `components` section. You then use `posed.` plus the name of the element you want to animate. The magic of `posed` is that it allows you to create animated versions of any HTML or SVG element. For example, `posed.div()` lets you build an animated div. I called the component that represents this animated div `Box`, but you can call it whatever you prefer.

Next, you add some properties that control your animation. The ones I used above are just a tiny fraction of what Pose.js makes available, and you can already achieve a nice effect with very little code. Let's go through them one by one:

- `pressable` set to `true` indicates that the element can react to mouse and touch down events
- `init` allows you to set an initial CSS property to a certain initial value, which then gets modified through the animation.
- `press` lets you set your chosen CSS property to the value you want the element to animate to when pressed (either by a mouse or touch down event). Pose makes available other cool interactive options like `drag`, `hover`, and `focus`.

In the template section, you add the `Box` component like this:

```
<div id="app">
  <Box class="box"></Box>
</div>
```

#### Live Code

Check out what the animation looks like in this [live demo](#).

You can also configure custom transitions and take full control of the animation:

```
transition: {
  default: { ease: "linear" }
}
```

Inside each transition object you can specify a few properties like `ease`, `duration`, `delay`, and so on. The `ease` property affects the speed of the animation over the course of its duration and it's super important if you

aim for professional-looking animations. Pose offers a number of ease values:

- `linear`
- `easeIn`, `easeOut`, `easeInOut`
- `circIn`, `circOut`, `circInOut`
- `backIn`, `backOut`, `backInOut`
- `anticipate`
- an array of numbers to create a cubic bezier easing function

The default transition is `tween`, but by setting the `type` property you'll also find other fun choices like:

- **Spring**, which maintains velocity between animations to create engaging motions that you can further fine-tune by adjusting `stiffness`, `mass`, and `damping` properties
- **Decay** reduces the velocity of an animation over its duration and it works beautifully for the special `dragEnd` pose that fires when users stop dragging an element on the page
- **Keyframes** allow you to set up a series of tween values
- **Physics** lets you simulate effects like velocity, friction, and acceleration.

Another feature of Pose that I like a lot is the way you can coordinate animations between parent and multiple children with ease. Whenever a posed component changes its `pose`, the change propagates through its children—even those which aren't direct children.

#### [Live Code](#)

[Experiment with this demo](#) to get familiar with how Pose quickly animates all

the list items inside a toggleable sidebar component.

## #4 GreenSock (GSAP)

GreenSock, also known as GSAP (GreenSock Animation Platform) is a fantastic library for the creation of “ultra high-performance, professional-grade animation for the modern web.”

When it comes to web animation, there’s almost nothing that you can’t do with GSAP. Here are some of what I think are its strongest points:

- You can learn it in no time. Its API is intuitive and straightforward.
- The docs are excellent.
- It’s a mature library that’s been around for years.
- It’s got great cross-browser support that goes back to IE6!
- It works like a charm with HTML5, SVG, Canvas, jQuery, React, Vue, and more.
- It’s fast.
- It’s continually being maintained and updated.
- It’s got a super helpful community for support.

The API features you’re going to use the most are:

- TweenLite, which is GSAP’s foundation. An instance of this object handles tweening one or more properties of any object over time.
- TweenMax, which extends TweenLite adding extra bells and whistles.
- TimelineLite, a powerful sequencing tool inside which you can wrap tweens and other timelines to get full control of complex

animations.

- TimelineMax, which extends TimelineLite adding more goodies like `repeat`, `repeatDelay`, `yoyo`, etc.

The syntax you need to use GSAP's API is very similar in all four of the above objects, which really helps getting familiar with the library in a relatively short period of time:

```
/* tween an element from full opacity to opacity 0
in 1 second with a 2 second's delay */
TweenLite.to(element, 1, {opacity: 0, delay: 2});
TweenMax.to(element, 1, {opacity: 0, delay: 2});

/* create an instance of TimelineLite called tl
which repeats the initial animation twice with 1
second's delay in between tweens */
const tl = new TimelineLite({repeat: 2, repeatDelay: 1});
// add a TweenLite instance to the timeline that animates
// an element from full opacity to opacity 0 over 1 second
tl.add( TweenLite.to(element, 1, {opacity: 0}) );

// do the same as above, but using TimelineMax
const tl = new TimelineMax({repeat: 2, repeatDelay: 1});
tl.add( TweenLite.to(element, 1, {opacity: 0}) );
```

You can animate a property towards an end value, in which case you use `.to()` like in the snippets above. If you want to define the starting values instead of the end values, use `.from()`. Finally, to specify both starting and ending values of the tween, use `.fromTo()`.

Check out Getting Started with GSAP on the library's website for a friendly introduction.

[Live Code](#)

#### Live Code

Try playing with [this demo](#) to get a sense of what you can do with GSAP and [Vue](#).

## #5 Anime.js

[Anime.js](#) is a lightweight and fast animation engine created by Julian Garnier. This is also a full-featured library you can painlessly integrate with [Vue.js](#) to create some really cool web animations.

Anime works with CSS properties, CSS transforms, SVGs, DOM attributes and JavaScript objects. It's also very well supported and the [docs](#) are pretty sleek.

Here's a snippet taken from the library's GitHub page to get you started with the syntax:

```
anime({
  targets: 'div',
  translateX: [
    { value: 100, duration: 1200 },
    { value: 0, duration: 800 }
  ],
  rotate: '1turn',
  backgroundColor: '#FFF',
  duration: 2000,
  loop: true
});
```

- `targets` lets you specify the DOM elements you want to animate.
- `translateX`, `rotate`, and `backgroundColor` indicate the properties on the element that you want to animate. Although this is not specific to Anime but to JavaScript more generally, notice how the CSS compound names, which in your CSS document are

usually hyphenated—such as `background-color`—in JavaScript are camelcased: `backgroundColor`.

- Also, notice how `translateX` has been given specific `duration` values, both with respect to the initial and ending states and with respect to the other properties on the same object—for example, `rotate` and `backgroundColor`.
- `duration` indicates how long the tween is going to last.
- `loop` set to `true` means that the animation keeps repeating indefinitely.

### Live Code

[Here's a live demo of Anime.js with Vue in action.](#)

## Conclusion

I find Vue to be a truly animation-friendly framework. You've seen how you can create nice animation effects quite straightforwardly both with and without the use of an external library. However, integrating any one of the libraries listed here with Vue will give you animation super powers. There's no end to what you can come up with and share with the world!

# Chapter 3: Build Your First Static Site with VuePress

BY IVAYLO GERCHEV

In this tutorial, we'll build a static site with [VuePress](#) and we'll deploy it to [GitHub Pages](#). The site will be a simple, technical blog which offers the possibility to organize and navigate the content by collections, categories, and tags.

## Code and Live Site

The code for this tutorial [can be found on GitHub](#). You can also view [the site running live on GitHub Pages](#).

## What Is VuePress?

VuePress is a simple yet powerful static site generator, powered by [Vue.js](#). It was created by Evan You (the creator of Vue), who made it to facilitate the writing of technical, documentation-heavy sites for the Vue ecosystem. But as we'll see in this tutorial, with a little tweaking the use cases for VuePress can be broadened.

A site made with VuePress is served as a single page application (SPA), powered by Vue, [Vue Router](#), and



webpack as underlying technologies. As an SPA, VuePress uses a mixture of Markdown files and Vue templates/components to generate a static HTML site.

## The Benefits of a Static Site.

A static site has some significant benefits:

- **Simplicity.** A static site is written in plain, simple HTML files. As all files are physically present on the server—in contrast to a database-driven site—you can move them, deploy them, and back them up quickly and easily.
- **Speed.** Because the files are already compiled and ready to be served, the page load time of a static site is pretty fast.
- **SEO friendly.** Metadata-rich pages plus human-readable URLs and plain text content make a static site extremely accessible to search engines.
- **Security.** A static site is far more secure than a dynamic one. According to a WP WhiteSecurity report, about 70% of WordPress sites are at risk of getting hacked. Because a static site doesn't rely on a database, CMS and/or plugins, the chances of getting hacked is minimal.
- **Salability.** A static site can be easily scaled up by just increasing the bandwidth.
- **Version Control Support.** Adding version control to a static site is super easy.

## Why Use VuePress?

There's an abundance of static site generators out there. So why might we choose VuePress over the others options?

Well, for me, these are the two strongest selling points of VuePress:

1. VuePress is built with Vue. This means that we can use all the superpowers of Vue while we build our site.
2. We can add a VuePress site to any existing project, at any time.

The above benefits are already strong enough, but let's explore the full feature palette that makes VuePress shine:

- There's close integration between Vue templates/components and Markdown files. We can use Vue templates/components directly in Markdown, which give us great flexibility.
- VuePress uses markdown-it, one of the best Markdown parsers out there, which has a rich plugin ecosystem. Some of its plugins, optimized for technical documentation, are already bundled with VuePress. And we can add more if we want.
- VuePress has a Vue-powered custom theme system, with which we can create a full-featured theme by using Vue SFC (single file components). The sky's the limit.
- VuePress offers multi-language support. So we can turn our site multilingual fairly easily.
- VuePress has Google Analytics integration.
- VuePress has a responsive default theme, specifically tailored for technical documentation, with a great set of features out of the box. These include:

- Optional home page
- Header-based search functionality (optional Algolia search)
- Customizable navbar and searchbar
- Auto-generated GitHub link and page edit links.

## Getting Started



### VuePress Versions

In this tutorial, we'll use the stable VuePress version, which is **0.14.8** at the time of writing. There is also a new version, with great new features, including ones for blogging, but it's still in alpha stage and it's not secure to be used yet.

Before we get started, you'll need a recent version of Node.js installed on your system (version 8 or greater). If you don't have Node installed, you can download the binaries for from the [official website](#), or use a [version manager](#). This is probably the easiest way, as it allows you to manage multiple versions of Node on the same machine.

Next, create a new directory `vuepress-blog` and initialize a new Node project inside:

```
mkdir vuepress-blog && cd vuepress-blog  
npm init -y
```

Next, install VuePress locally, as a dependency:

```
npm install -D vuepress
```

Finally, edit the scripts section of the `package.json` file to include the `dev` and `build` commands:

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

```
"docs:dev": "vuepress dev docs",  
"docs:build": "vuepress build docs"  
},
```

Now we can run the dev server with the following command:

```
npm run docs:dev
```

If you've set up everything correctly, the site will build and be available at <http://localhost:8080>. Currently there isn't a whole lot to see apart from a 404 message, but we'll change that in the next sections.

## EXPLORING THE DIRECTORY STRUCTURE OF THE FINAL PROJECT

Right now, our project is empty. But, in this section I want to give you a glimpse of the directory structure of the final project. Our site will be built inside a `docs` directory. The `docs` directory will contain a `blog` directory and a `.vuepress` directory.

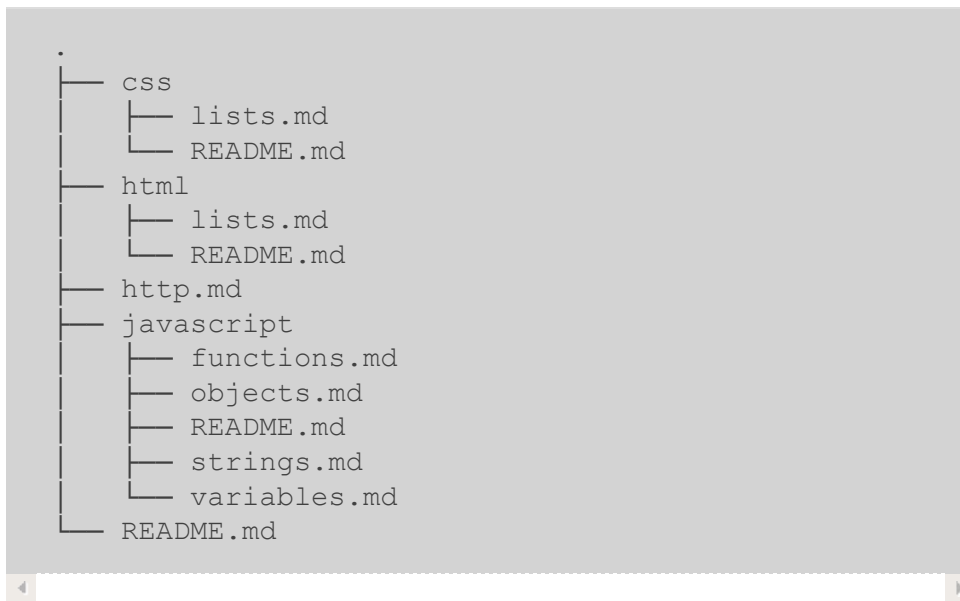
```
.  
├── docs  
│   ├── blog  
│   ├── README.md  
│   └── .vuepress  
├── package.json  
└── package-lock.json
```

As you can see, the `docs` directory, serves as a root of

our VuePress site. A `README.md` file is required for the root directory and for all directories containing Markdown files. It will be automatically transformed into an `index.html` file, which will serve as a starting point/page for the current directory.

## THE BLOG DIRECTORY

The `blog` directory will house all our blog posts and content. This is what it will look like:

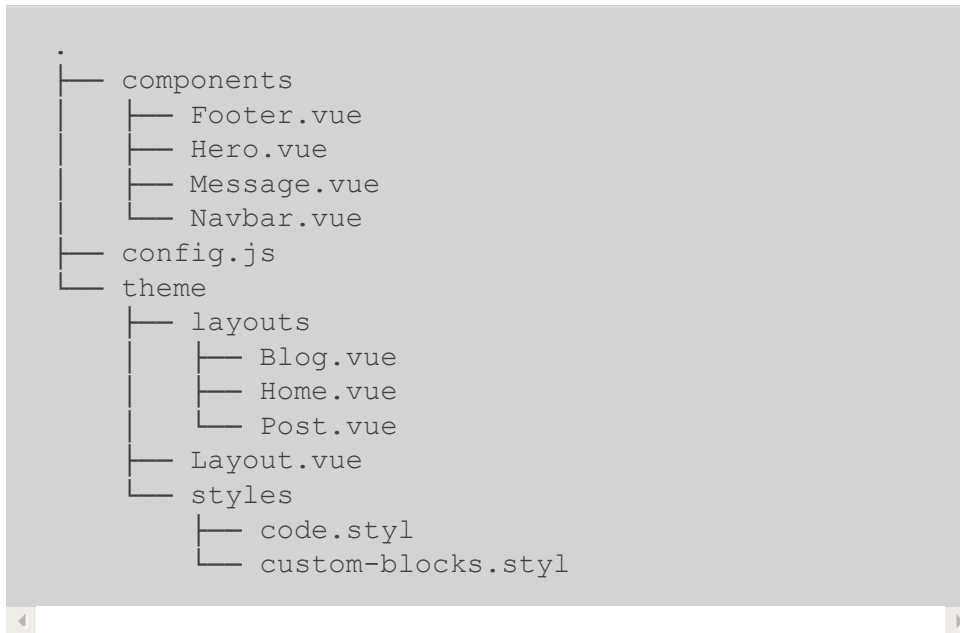


There's not too much going on here. The posts are represented by Markdown files and the three subfolders (`css`, `html` and `javascript`) represent collections. You'll hear more about these later.

## THE `.VUEPRESS` DIRECTORY

This will contain all files and folders necessary for the development and building of our site. This is what it will

look like:



As you can see, the `.vuepress` directory contains the following:

- A `components` directory for our custom components. All components put in this folder are globally registered, and dynamic and async by default.
- A `config.js` file, used for site and theme configuration.
- A `theme` directory to hold our custom theme.

The `theme` directory contains:

- a `layouts` directory, where we put our custom layouts
- a `Layout.vue` file, which is required to create a custom theme
- a `styles` directory for the theme's styles

## FOLLOWING ALONG AT HOME

If you'd like to follow along with this tutorial, it'd be a good idea to create most of these folders and files now. For those of you on 'nix-based systems, you can do that with the following code.

## Project root

```
mkdir -p docs/{blog,.vuepress}  
touch docs/README.md
```

## blog directory

```
mkdir docs/blog/{css,html,javascript}  
touch docs/blog/{css,html}/{lists.md,README.md}  
touch docs/blog/javascript/{functions.md,objects.md,README.md}  
touch docs/blog/{README.md,http.md}
```

Please note that we won't fill all of these posts (Markdown files) with content during the tutorial. That will be left to you.

## .vuepress directory

```
mkdir docs/.vuepress/{components,theme}  
touch docs/.vuepress/components/{Footer.vue,Hero.vue,Navbar.vue}  
touch docs/.vuepress/config.js  
mkdir docs/.vuepress/theme/{layouts,styles}  
touch docs/.vuepress/theme/Layout.vue  
touch docs/.vuepress/theme/layouts/{Blog.vue,Home.vue,Post.vue}  
touch docs/.vuepress/theme/styles/{code.styl,custom-block.styl}
```

Alternatively, you could clone the repo and work with the finished project.

## CONFIGURING YOUR SITE/THEME

Finally, before we start building the blog, let's establish some settings. In `.vuepress/config.js`:

```
module.exports = {
  title: "Front-end Web School",
  description: "Learn HTML, CSS, and JavaScript",
  head: [
    [
      "link",
      {
        rel: "stylesheet",
        href: "https://cdnjs.cloudflare.com/ajax/libs/bulma/0.9.0/css/bulma.min.css"
      }
    ],
    [
      "link",
      {
        rel: "stylesheet",
        href: "https://use.fontawesome.com/releases/v5.11.2/css/all.css"
      }
    ]
  ]
};
```

Here, we provide a site title and description. And we also add Bulma and Font Awesome to the project. They'll be injected into the head tag of each compiled HTML page. We'll use them to develop our custom theme.

## Building Your Site

Now we're going to build our blog. In the version we'll be using, VuePress doesn't offer blogging features (such as tags and categories), but we'll try to improvise.

## DEVELOPING THE BLOG CONTENT



## PREPARING THE BLOG CONTENT

Let's start by preparing some content for our blog.

In the `blog` directory, we have `html`, `css`, and `javascript` subdirectories. Each blog subdirectory will play the role of a collection. So we'll have the ability to organize the blog's content by collections and to explore each collection individually.

The `blog` directory and all of its subdirectories each have a `README.md` file. This file will contain only a Front Matter section with a `title` property containing the name of the directory. So for the `blog` directory, the `README.md` file will contain:

```
---
title: Blog
---
```

For the `css` directory:

```
---
title: CSS
---
```

And so on.

Next, let's look at our blog posts—the Markdown files. Each file must contain a `Front Matter` section, where we put the necessary settings and metadata in YAML format. Here's an example from the `functions.md` file:

```
---
title: Learn JavaScript Functions
date: 2018-12-12
categories: [Intermediate]
tags: [JavaScript, Function, Callback Function]
---

<!-- Your Markdown content here -->
```

## Grabbing the Content

As mentioned above, we'll not fill out all of the blog posts here. If you'd like to copy any of the content from the finished site, [you can find it here](#).

## CREATING THE LAYOUT.VUE COMPONENT

After we get our content done, let's start creating our custom theme. In `.vuepress/theme/Layout.vue` add this:

```
<template>
  <div class="container">
    <Navbar/>
    <Component :is="currentLayout"/>
    <Footer/>
  </div>
</template>

<script>
import Home from "../layouts/Home.vue";
import Blog from "../layouts/Blog.vue";
import Post from "../layouts/Post.vue";
export default {
  components: {
    Home,
    Blog,
    Post
  },
}
```

```
computed: {
  currentLayout() {
    const { path } = this.$page;
    if (path === "/") {
      return "Home";
    } else if (path.endsWith("/")) {
      return "Blog";
    } else if (path.endsWith(".html")) {
      return "Post";
    }
  }
};
</script>
```

`Layout.vue` will be invoked for each Markdown file. It's like a root component in a Vue application. The code in this file checks the current page's path (via the `currentLayout` computed property) and according to the result loads the needed layout. For that, we use Vue's built-in `<Component/>` element, which swaps the displayed component via its `is` property. In the template, we also put `<Navbar/>` and `<Footer/>` components, which we'll be creating in the following section.

VuePress exposes two variables—`this.$page` and `this.$site`—which give us access to data about the page and the site respectively. In `currentLayout`, `this.$page` is used to get the path of the current page.

## CREATING THE PARTIAL COMPONENTS

In this section, we'll look at the partial components, which we'll import into our layout. These partials are in

the `.vuepress/components` folder.

## Navbar.vue

In this component, we use Bulma's Navbar component to create a navbar for our blog:

```
<template>
  <nav class="navbar is-dark" role="navigation" aria-label="Navigation">
    <div class="navbar-brand is-marginless">
      <a class="navbar-item" :href="$withBase('/')">
        <span class="icon is-large has-text-warning">
          <i class="fas fa-2x fa-laptop-code"></i>
        </span>
      </a>
      <a
        role="button"
        class="navbar-burger"
        :class="{ 'is-active': isActive}"
        @click="isActive = !isActive"
        aria-label="menu"
        aria-expanded="false"
        data-target="navbarMenu"
      >
        <span aria-hidden="true"></span>
        <span aria-hidden="true"></span>
        <span aria-hidden="true"></span>
      </a>
    </div>

    <div id="navbarMenu" class="navbar-menu" :class="{ 'is-active': isActive}">
      <div class="navbar-end">
        <a class="navbar-item" :href="$withBase('/')">Home</a>
        <a class="navbar-item" :href="$withBase('/blog')">Blog</a>
        <a
          class="navbar-item"
          :href="$withBase(collection.path)"
          v-for="collection in collections"
        >{{collection.path | formatNavItems}}</a>
      </div>
    </div>
  </nav>
</template>

<script>
export default {
```

```

data: function() {
  return {
    isActive: false
  };
},
computed: {
  collections() {
    let pages = this.$site.pages.filter(page => {
      return page.path.match(/(blog\/).+(\\/$)/);
    });
    return pages;
  }
},
filters: {
  formatNavItems(value) {
    if (!value) return "";
    let pos = value.search(/\\w+\\$/);
    let res = value.slice(pos, value.length - 1);
    return res.toUpperCase();
  }
}
};
</script>

<style>
.navbar {
  padding-right: 1em;
}
</style>

```

In this file, we create a `computed collections` property, which filters all pages and matches only the blog directory and subdirectories. We use that property in the template to populate the navbar with a link for each collection. We also use the `withBase` built-in helper in order to have proper links for our non-root URL when we deploy it. We set the `Home` and `Blog` links manually.

We also create and use a `formatNavItems` filter, which extracts the name of the directory from the path and uppercases it.

The `isActive` data property is used to toggle the navbar menu on mobile by binding it with the `is-active` Bulma class.

If you run the dev server at this point (using `npm run docs:dev`) and navigate to <http://localhost:8080>, you should be able to see the navbar component rendered to the page.

## Hero.vue

In this component, we use the Bulma hero component:

```
<template>
  <section class="hero is-success">
    <div class="hero-body">
      <div class="container">
        <h1 class="title">Front-end Web School</h1>
        <h2 class="subtitle">Learn to code in HTML, CSS, JS</h2>
      </div>
    </div>
  </section>
</template>
```

## Message.vue

In this component, we create a message box, which we'll display on the home later on. We use Bulma's message component for this:

```
<template>
  <article class="message is-info" :class="{hidden: !show}">
    <div class="message-header">
      <p>Welcome to Our Blog</p>
      <button @click="hide" class="delete"></button>
    </div>
    <div class="message-body">Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.</div>
  </article>
</template>
```

```

</article>
</template>

<script>
export default {
  data: function() {
    return {
      dismiss: false
    };
  },
  methods: {
    hide() {
      this.dismiss = true;
    }
  }
};
</script>

<style>
.hidden {
  display: none;
}
</style>

```

## Footer.vue

This is a simple footer created with Bulma's footer component:

```

<template>
  <footer class="footer has-background-grey-darker">
    <div class="content has-text-centered">
      <p>Copyright 2018 All rights reserved.</p>
    </div>
  </footer>
</template>

<style>
.footer {
  margin-top: 2em;
}
</style>

```

## CREATING THE LAYOUT COMPONENTS

Now it's time to create the custom layouts for our blog. In this section, we'll look at the files in the `.vuepress/theme/layouts` folder: `Home.vue`, `Blog.vue`, and `Post.vue`.

### Home.vue

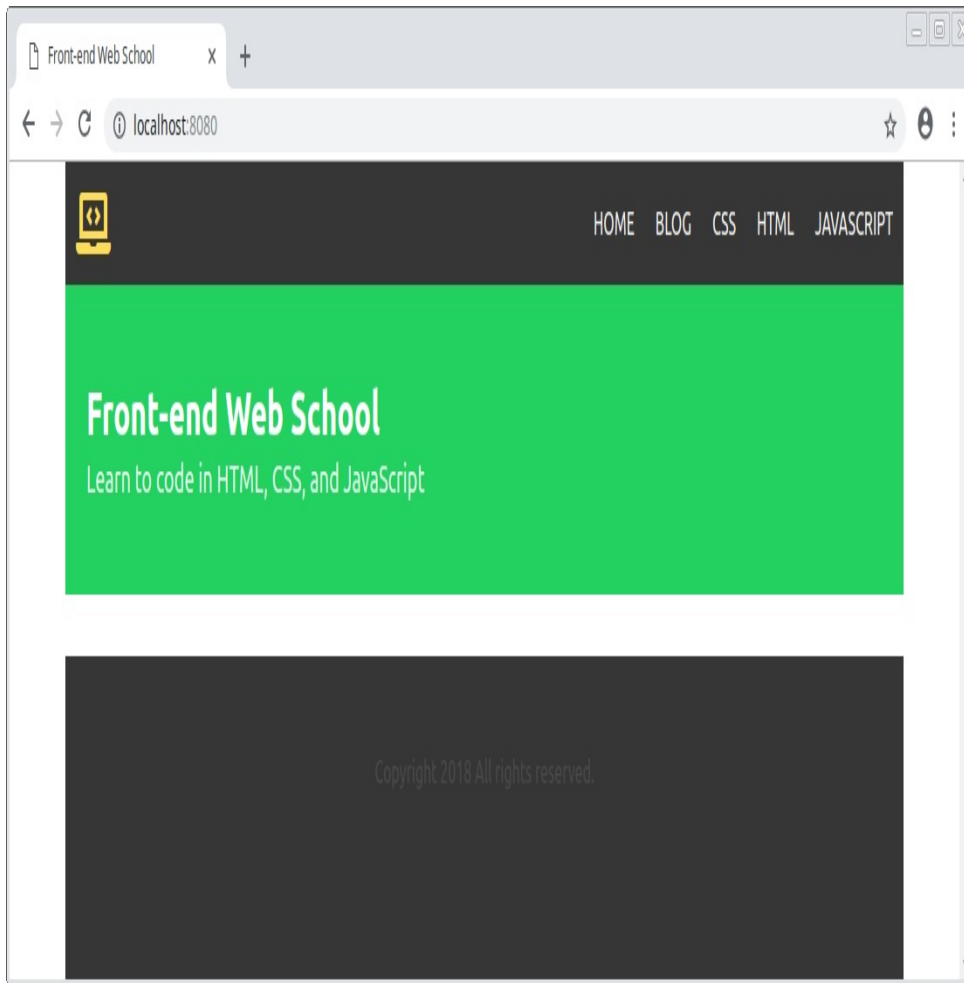
This will be our home template. It's super simple:

```
<template>
  <div>
    <Hero/>
    <Content/>
  </div>
</template>
```

We just add the `<Hero/>` partial and the VuePress' `<Content/>` global component.

Take a second to check your progress at this point. You should see the site being rendered with a nice hero element. If you test it out in your browser, you should see that it's responsive, too.





## Blog.vue

This is our most complex component, which will render the blog posts for each collection. It will also find and filter the taxonomies (categories and tags) we set in the Front Matter sections of the Markdown files:

```
<template>
  <div>
    <div class="content">
      <h1>{{ $page.frontmatter.title }}</h1>
      <Content/>
    </div>
    <div v-if="posts.length">
      <div class="columns">
        <!-- Filter controls -->
      </div>
    </div>
  </div>
</template>
```

```

<div class="column is-3">
  <nav class="panel">
    <p class="panel-heading">Categories</p>
    <a
      class="panel-block is-capitalized"
      :class="{ 'is-active': i == currentCat}"
      v-for="cat, i in findTaxonomy('categories')"
      @click.prevent="filterTaxonomies('categories', cat)"
    >
      <span class="panel-icon">
        <i class="fas fa-tasks"></i>
      </span>
      {{cat}}
    </a>
  </nav>
  <nav class="panel">
    <p class="panel-heading">Tags</p>
    <div class="tags panel-block is-capitalized">
      <a
        v-for="tag, i in findTaxonomy('tags')"
        @click.prevent="filterTaxonomies('tags', tag)"
      >
        <span class="tag" :class="{ 'has-text-info': i == currentTag}">
          {{tag}}
        </span>
      </a>
    </div>
  </nav>
</div>
<!-- Blog posts -->
<div class="column is-9">
  <transition-group tag="ul" name="posts">
    <li class="box" v-for="post in posts" :key="post.id">
      <router-link :to="post.path">
        <div class="box">
          
        </div>
        <p class="title is-3">{{post.frontmatter.title}}</p>
        <p class="subtitle is-6">
          <span class="icon">
            <i class="fas fa-calendar-alt"></i>
          </span>
          {{post.frontmatter.date | formatDate}}
          <span class="icon">
            <i class="fas fa-tags"></i>
          </span>
          <span v-for="tag in post.frontmatter.tags">{{tag}}</span>
        </p>
      </router-link>
    </li>
  </transition-group>
</div>

```

```

    </div>
  </div>
</div>
</template>

<script>
export default {
  data: function() {
    return {
      posts: [],
      currentCat: null,
      currentTag: null
    };
  },
  created() {
    this.posts = this.allPosts;
  },
  computed: {
    allPosts() {
      let currentCollection = this.$page.path;
      let posts = this.$site.pages
        .filter(page => {
          return page.path.match(
            new RegExp(`(${currentCollection})(?=.html)`);
          );
        })
        .sort((a, b) => {
          return new Date(b.frontmatter.date) - new Date(a.frontmatter.date);
        });
      return posts;
    }
  },
  filters: {
    formatDate(value) {
      let d = new Date(value);
      return d.toDateString();
    }
  },
  methods: {
    filterTaxonomies(type, tax) {
      let currentCollection = this.$page.path;
      let posts = this.$site.pages.filter(post => {
        if (post.frontmatter[type]) {
          return post.frontmatter[type].includes(tax);
        }
      });
      this.posts = posts;
    },
    findTaxonomy(type) {
      let tax = [];
    }
  }
}

```

```

        let posts = this.$site.pages.forEach(function(post) {
            if (post.frontmatter[type]) {
                tax = tax.concat(post.frontmatter[type]);
            }
        });
        let uniqTax = [...new Set(tax)];
        return uniqTax;
    }
}
};
</script>

<style>
.tag {
    margin: 0 0.3em;
}

.posts-move {
    transition: transform 1s;
}

.posts-enter-active,
.posts-leave-active {
    transition: all 0.5s;
}

.posts-enter,
.posts-leave-to {
    opacity: 0;
    transform: translateY(30px);
}
</style>

```

In this file, we create a `posts` data property to hold our blog posts. Then, we create an `allPosts` computed property to get all the posts from the current collection and sort them by date. We use the `created` hook to populate the sorted posts in the `posts` data property.

We then go through the posts with a `v-for` directive and populate each one by extracting data from its `Front Matter`. We use a `formatDate` filter to display the date

in human-readable format.

We implement tags and category features by creating two filters:

- `filterTaxonomies` filters all pages which have a particular taxonomy type and search term.
- `findTaxonomy` goes through each page and collects the desired taxonomy. Then, it extracts and returns only the unique terms from the collected taxonomies.

The above two filters are used in the template to create the `Categories` and `Tags` panels, which display all existing categories and tags.

Finally, we use the `<transition-group>` component to add a smooth transition when we filter the blog posts, as well as `currentCat` and `currentTag` data properties to assign proper active classes to the selected tag or category.

## Post.vue

Now, let's create the layout for the single post:

```
<template>
  <div class="content">
    <h1>{{ $page.frontmatter.title }}</h1>
    <Content/>
  </div>
</template>

<style>
.content {
  margin-top: 2em;
  padding: 0 2em;
}
```

```
}  
</style>
```

In the single post layout we just display the title and the content.

## ADDING SOME STYLES

For the Markdown extensions to be properly displayed, we'll need some styling. We'll steal the necessary styles from the default theme and we'll put them at the end of our `Layout.vue` component:

```
<style src="prismjs/themes/prism-tomorrow.css"></style>  
<style lang="stylus">  
  /* colors */  
  $accentColor = #3eaf7c;  
  $textColor = #2c3e50;  
  $borderColor = #eaecef;  
  $codeBgColor = #282c34;  
  $arrowBgColor = #ccc;  
  
  /* code */  
  $lineNumbersWrapperWidth = 3.5rem;  
  $codeLang = js ts html md vue css sass scss less stylus  
  
  @import './styles/custom-blocks.styl';  
  @import './styles/code.styl';  
</style>
```

First, we add the necessary CSS file for the syntax highlighting. Then, we add some Stylus variables and import two Stylus files. The Stylus imports are very long, so I don't want to list their contents here. Rather, you can grab them from our repo: [custom-blocks.styl](#) and [code.styl](#).

At this point, you should be able to click around your blog and see all of the posts displayed under the different categories.

## USING VUE COMPONENTS IN MARKDOWN

As I mentioned at the beginning, we can use templates and components directly in our Markdown. To add the `<Message/>` component we created earlier to the home page, we just add it in the home's `README.md` like this (in `docs/README.md`):

```
---
title: Home
---

<Message/>
```

Now, when you visit <http://localhost:8080>, you should see a nicely styled `Message` component.

## USING MARKDOWN EXTENSIONS

VuePress comes with some Markdown extensions optimized for technical documentation. Here are some of them. You can add them to `docs/README.md` to see them in action straight away, or view them on the [blog's home page](#):

- **Table of contents** (you'll actually need some content to see this in action):

```
[[toc]]
```

- **Code blocks**, with line highlighting and optional line numbers:

```
```js{2}
function add(a, b) {
  return a * b; // Line highlighting is cool!
}
```
```

- **Custom containers and emoji:**

```
::: warning
Be careful! VuePress is highly addictive! :wink
:::
```

- To display the **line numbers**, we have to add the following to the `.vuepress/config.js` file:

```
markdown: {
  lineNumbers: true
}
```

## BUILD THE SITE

Phew, this was a long ride. So finally our blog is ready to go and we can build it.

Run the following in the terminal:

```
npm run docs:build
```

This will generate a `dist` folder, inside the `.vuepress` directory, with all files compiled and ready to be deployed.



## Internationalize Your Site

As I mentioned before, we can make our site multilingual. However, I'm not going to explore this feature here. For more information, please [consult the documentation](#).

## Deploy Your Site to GitHub Pages

Once your site is done, you have plenty of options for deploying. You can read about the different choices [in the documentation](#). I deployed my version of the blog to GitHub Pages. Here is how.

1. I created a repo `vuepress`.
2. Then, I set the `base` property in the `config.js` to be `/vuepress/`.
3. Next, I created a `deploy.sh` file with the following code:

```
# abort on errors
set -e

# build
npm run docs:build

# navigate into the build output directory
cd docs/.vuepress/dist

# if you are deploying to a custom domain
# echo 'www.example.com' > CNAME

git init
git add -A
git commit -m 'deploy'

# if you are deploying to https://<USERNAME>.github.io/<REPO>
# git push -f git@github.com:<USERNAME>:<REPO>

# if you are deploying to https://<USERNAME>.github.io/
git push -f git@github.com:codeknack/vuepress.git
```

A terminal window with a light gray background. The text 'cd -' is displayed in a monospaced font. Below the text is a horizontal scrollbar with a light beige track and a gray slider. The scrollbar is positioned at the left end of the track.

```
cd -
```

4. Finally, I ran it. And that's all.

## Conclusion

In this tutorial, we learned what VuePress is, why we should use it, and how to create a simple blog with it. We also deployed the blog to GitHub Pages. All this gave us a solid understanding of the VuePress system and its capabilities. So now we're ready to apply this knowledge in our future projects, creating more beautiful VuePress-powered sites.

# Chapter 4: Five Vue UI Libraries for Your Next Project

**BY MICHIEL MULDER**

According to [the official Vue.js website](#), Vue is a progressive framework for building user interfaces. It's designed from the ground up to be incrementally adoptable, and the core library is focused on the view layer only. This makes it easy for devs to pick it up and quickly become productive.

When starting a new project, designing a whole set of UI elements can be a time-consuming business. In this guide, I'm going to introduce you to five UI libraries that you can integrate with Vue in a few lines of code.

## WHAT IS A UI LIBRARY?

Before we get into things, let's pause for a quick definition.

A UI library usually consists of easy-to-use components which you can include in your Vue application using custom elements. You benefit from using a library like this, as you can create elements such as forms much

faster, with most libraries offering a great range of input components with added functionality and events.

Besides that, most libraries come with a proper and simple design which you can easily modify to your needs. In short, a UI library saves you time and helps you develop your application much faster.

## 1. Ant Design for Vue

### Pros:

- Its components support a lot of functionality and modifications via API options.
- It has a very minimal but beautiful design.
- It offers a wide range of data entry and display components.
- It has great ES6 support.

### Cons:

- While it can be used in mobile applications via its Col and Row components, it's more suitable for desktop web applications.

With over 5,000 stars on GitHub, Ant Design for Vue is part of the bigger Ant Design library covering other frameworks like Angular and React. In my opinion, Ant Design has a very minimal but beautiful design which you can customize if needed. Ant Design is best known for its wide variety of data entry and data display components. Just their data table page shows more than 15 different data table examples you can implement in your web application.

Ant Design for Vue is great if you like to work with the JavaScript ES6 standard. Besides that, it comes with a webpack-based debug build solution supporting ES6.

In addition, Ant Design for Vue is known for its great documentation. It provides tons of API options that influence the behavior or design of your component. For example, a simple checkbox has 5 properties:

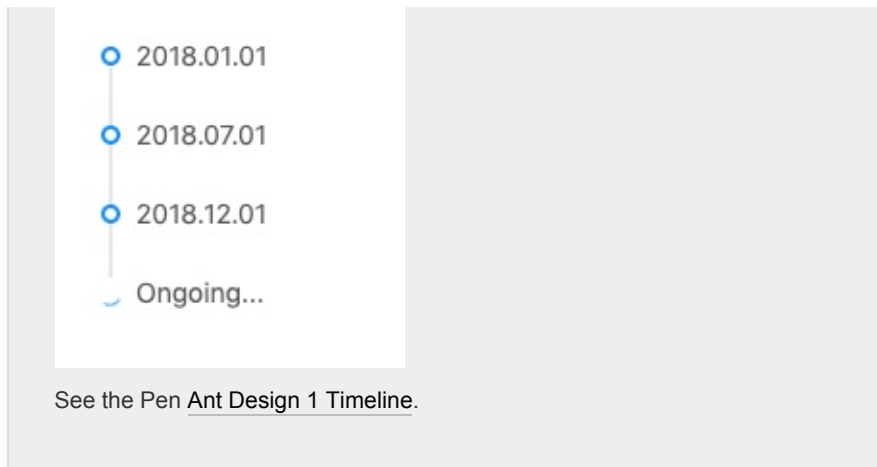
`autoFocus`, `checked`, `disabled`, `defaultChecked`, and `intermediate` (which can help you to achieve a “check all” effect).

Also, the documentation always provides an event section for each component, which displays the type of events you can catch and what values are emitted. Every property in the documentation mentions its data type, if there are any default values, and if the value is required.

As you can see, it's a pleasure to work with Ant Design for Vue as the documentation is so accurate and helps you to be very productive while developing your front end.

The first snippet I selected is to demonstrate the nice design of Ant Design. It's a timeline which has a loading state at the end. It's a very useful component for showing the history of certain objects.

[Live Code](#)



The second element shows you an upload button that registers files, and you can even push the files to an API endpoint that stores them. It's a very useful component which is mostly hard to configure correctly, but Ant Design just gets it right.



## 2. Element UI

### Pros:

- It has a very large range of components.
- It offers extensive documentation with version management.

## Cons:

- The default language of Element is Chinese. If you want to use another language, you'll need to do some i18n configuration.

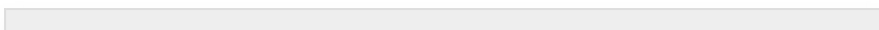
With over 35,000 stars on GitHub, Element is one of the leading UI frameworks for Vue.js 2.0. Element is focused on web applications. Built upon a strong community of over 400 contributors, the library provides a rich selection of customizable components along with a full style guide and more resources.

Element is quite similar to Ant Design, as it provides the same minimal style with a beautiful touch. But Element provides better capabilities for designers to change the style and layout of all components using an easy style guide where you can change things such as the primary color of the theme.

Element offers 16 components for building nice, easy-to-understand forms. Besides that, it offers several components for things such as data display, navigation, and message.

## CODE SNIPPETS

The first element I want to bring to your attention is the tree select. It allows you to display and select data in a hierarchical structure. Here's an example implementation of this tree component in Vue which shows how to load node data asynchronously.



### Live Code

Last selected: zone9

- ▼ ☒ Root1
  - ▶ ☒ zone1
  - ▼ ☐ zone2
    - ▶ ☐ zone5
    - ▶ ☐ zone6
- ▼ ☒ Root2
  - ▼ ☒ zone3
    - ▶ ☐ zone7
  - ▼ ☒ zone8
    - ▶ ☒ zone9
    - ▶ ☐ zone10
  - ☐ zone4

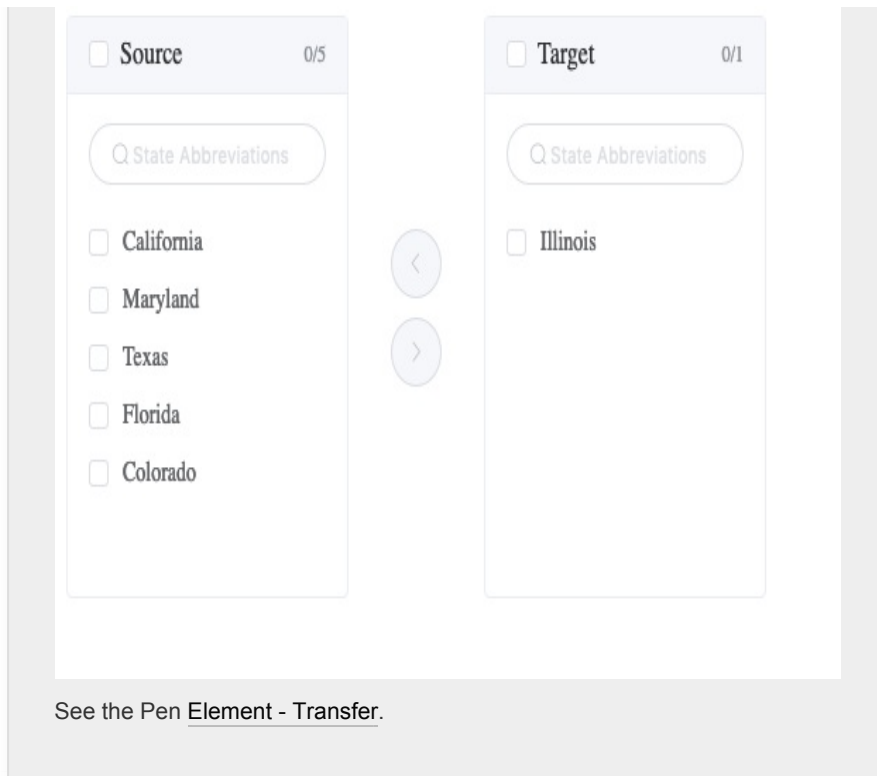
See the Pen [Element - Tree select](#).

This is a simple tree element that allows you to select data entries in this hierarchical data set. There are even more crazy implementations that let you filter this tree, move elements from one branch to another, or append and delete data in the tree. In my opinion, it's quite a unique component for a UI library that can be a good alternative way for displaying and interacting with data.

The next component is also quite special for a UI library. This transfer component lets you move data from one list to another. It also allows you to select or deselect all data in one list so you can quickly move it.

### Live Code





Again, many advanced options are available, like disabling the transfer of certain elements, [filtering list](#), or even executing operations on your selected data. It's a great alternative for selecting data in a user-friendly way.

### 3. Vue Material

#### Pros:

- Allows you to mimic [Google elements](#) in your application.
- It's mobile focused.
- Its documentation.
- It includes [Code Sandbox](#) integration, so you can immediately experiment with the code.

#### Cons:

- Currently 109 open issues, of which most have not yet been addressed by the developers.

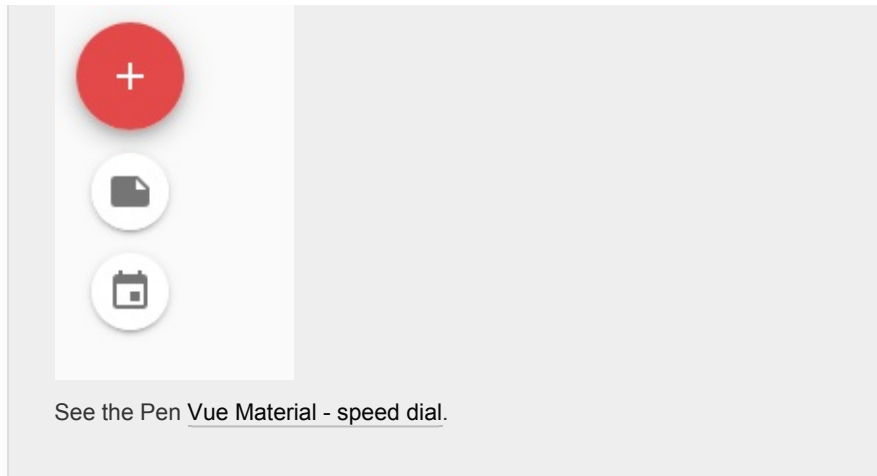
Vue Material is a simple and lightweight design library built exactly according to the Google Material Design specs. They offer a wide range of form elements to build your mobile application. For example, the Input & Textarea section offers many input options, such as password fields, fields with a maximum length, built-in error validation, or prefixes and suffixes. All of this is well documented with all properties the element accepts and the events it fires.

Vue Material even provides different classes to change the layout of your component. The toolbar section is a good example of this. You can add an offset to your title, remove the hamburger icon, or add some custom section ending.

## CODE SNIPPETS

The first snippet shows a simple speed dial—a common component for mobile applications—which not many UI libraries offer. You can add multiple icons to the speed dial that each have their own action attached to it.

[Live Code](#)

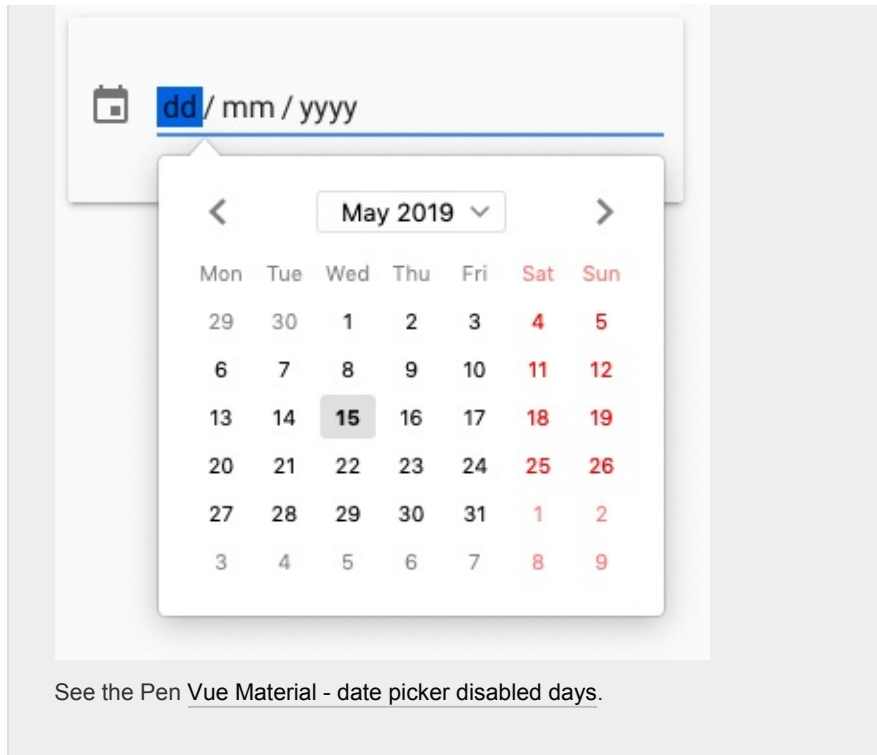


The second item I selected is the calendar input. It's a great example of Google Material Design. I experimented a bit with the options and was able to disable all Saturdays and Sundays by declaring a `disabledDates` data property:

```
new Vue({
  el: '#app',
  name: 'DisabledDatesDatepicker',
  data: () => ({
    selectedDate: null,
    disabledDates: date => {
      const day = date.getDay()
      return day === 6 || day === 0
    }
  })
})
```

It's a good example of controlling the input of your users with a smart UI.

[Live Code](#)



## 4. iView

### Pros:

- It offers clean animations.
- It offers a wide range of components with amazing data table components.
- It has an easily customizable theme.

### Cons:

- Some components don't have many variants.
- The default language is Chinese. If you want to use English, you'll need to do some i18n configuration.

With over 20,000 stars on GitHub, iView is one of the leading UI libraries for Vue. The style of the theme can

be easily changed in the Less file, which holds a set of variables that can be customized. It's recommended that you override these variables by importing a custom theme file. The style of the theme is very minimalistic but beautiful, and includes many simple animations that help the library to stand out.

For some elements, the library doesn't offer so many variants. For example, the Card component just offers a few basic card designs, while other libraries have card designs that include action bars, images, etc. However, some elements are very elaborate, like the data table components. Let's take a look at them in the code snippets section.


## CODE SNIPPETS

The first data table snippet includes a view and delete button to display or delete rows in the table. It's quite a basic example.

[Live Code](#)

| Name       | Age | Address                  | Action  |
|------------|-----|--------------------------|---|
| John Brown | 18  | New York No. 1 Lake Park | <button>View</button> <button>Delete</button> |
| Jim Green  |     |                          | <button>View</button> <button>Delete</button> |
| Joe Black  |     |                          | <button>View</button> <button>Delete</button> |
| Jon Snow   |     |                          | <button>View</button> <button>Delete</button> |

User Info

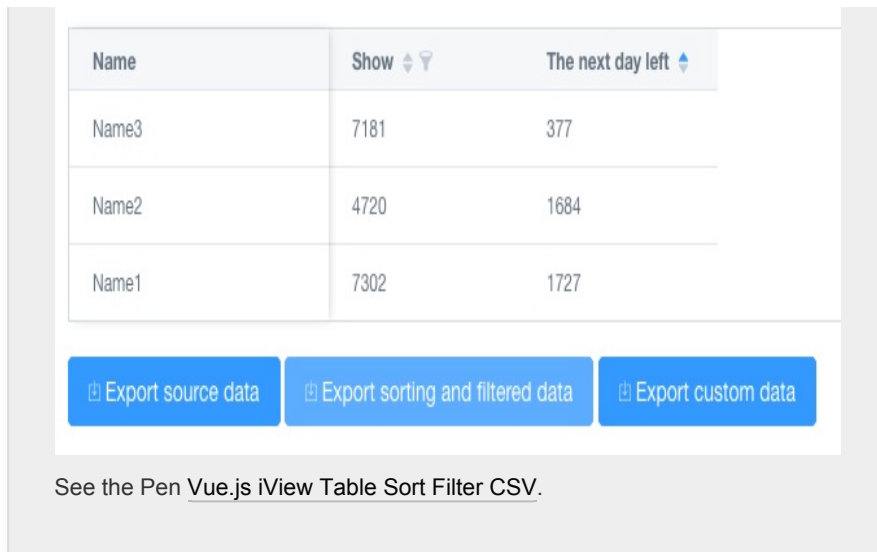
 Name: Jim Green  
Age: 24  
Address: London No. 1 Lake Park

OK

See the Pen [iVew Data Table Component](#).

Next, we have a more elaborate data table example that includes filters and sorting for each column. The data table comes with the ability to export the data to CSV. What's special here is that you can even export the filtered data and also select the columns you want to include.

[Live Code](#)



## 5. Mint-UI

### Pros:

- It includes uncommon components like lazy load, action sheet, swipe or infinite scroll.
- It's mobile focused.

### Cons:

- The design is a bit too simple.
- There's no documentation on how to change or improve the design.
- There are no events listed in API specs.

With over 13,000 stars on GitHub, Mint-UI is a popular Chinese UI library for Vue. The library includes some more uncommon components like an action sheet, cell swipe, index list or infinite scroll, all of which are mobile focused. However, the design is too simplistic and doesn't have any styling.

Besides that, the library doesn't list events for the components. For example, the checklist component returns an array of checked options, but it's not clear how to access this value or where it's coming from. For a Vue beginner, this can be very confusing!

## CODE SNIPPETS

The first code snippet is an example of an index list. It's an uncommon example but still a useful element for including in your mobile application.

**Live Code**

|         |   |
|---------|---|
| A       |   |
| Aaron   |   |
| Alden   |   |
| Austin  |   |
| B       |   |
| Baldwin |   |
| Braden  | A |
| Z       | B |
| Zack    | Z |
| Zane    |   |
|         |   |



See the Pen [Mint-UI 1 Proper index list](#).

The second code snippet shows a year-month picker to indicate a range. It includes a comparison function that makes sure the second value is always greater than the first item in the range comparison. Again, a simple but useful element.

#### Live Code

2015-02    2015-03

2015-03    2015-04

2015-04 - 2015-05

2015-05    2015-06

2015-06

See the Pen [Mint-UI 2 Picker](#).

## The Bottom Line

There are many great UI libraries out there for Vue and this was only a small selection of my favorites.

Of the libraries presented, iView offers the greatest design of all libraries but not all elements are as elaborate as the data tables section. Ant Design is an ideal choice if you want a nice and minimal design with most of the data input and data display components covered in the library. It also has a number of other

useful components, such as a progress bar with steps. In my opinion, despite its mobile-friendly approach, Mint-UI is probably the least useful library from this selection as the design is very basic and the events aren't well documented in the API.

However, saying this, whichever library you pick will depend on your project requirements. They all have their strengths and weaknesses, and there isn't one that covers all the bases alone.

# Chapter 5: Five Handy Tips when Starting Out with Vue

**BY DAVID BUSH**

I originally came to the world of Vue from ReactJS. Learning to divide my page up into small, reusable components and how to leverage the component lifecycle represented a shift in approach to problems, but I quickly got proficient.

React Native allowed me to write apps for both major smartphone platforms, which made it a slam dunk for me. But I kept hearing about how great Vue was. Crucially, I noticed no one I spoke to had anything bad to say about it, and experience tells me that the absence of multiple people having qualms with a language or framework means it should be investigated immediately, so I did.

These are the things I wish I'd known when I was starting out.

## Incremental Adoption

In my many years of programming, I've seldom found it a trivial undertaking to switch out an entire view layer. While I was keen to try Vue, I wasn't prepared to initiate a whole rewrite to do so. I was exceedingly happy to learn that, with Vue, you don't have to! You can incrementally add it to projects with a piecemeal rewrite simply by including the library file and adding components. This allows you to add new features and leverage its many strengths without having to throw out your current code, so there's no excuse not to try it!

## Devtools

One thing I found useful was the addition of Vue Devtools to my browser (available on both Firefox and Chrome). It renders superfluous those cumbersome `console.log()` debug calls by instead showing you the contents of each component in your devtools pane. It also affords you the ability to jump directly to a component in your editor (admittedly with a little tweaking) which I've found to be an absolutely killer feature, particularly in larger projects. Here's a screenshot of it in action:

sitepoint-example

127.0.0.1:3080

3000 4000

# r/vuejs

Sort by Best

10

How to make Router wait for Vuex Async?

2

I would appreciate feedback for my first project in Vue.

51

Reasons Why Vue.js Is Getting More Traction Every Month

1401

Console Inspector Debugger Style Editor Performance Memory Network Vue

Ready. Detected Vue 2.5.21.

Filter components

<Root>

<App>

<Story>

<Story>

<Story>

<Story>

<Story>

<Story>

<Story> = \$vm0

<Story>

<Story>

<Story>

Filter inspected data

<Story>

props

story: Object

approved\_at\_utc: null

approved\_by: null

archived: false

author: "Vue-had-me-at-Trello"

author\_flair\_background\_color: null

author\_flair\_css\_class: null

author\_flair\_richtext: Array[0]

author\_flair\_template\_id: null

author\_flair\_text: null

author\_flair\_text\_color: null

author\_flair\_type: "text"

As you can see, everything passed down to the component is immediately visible; there's no need for debugging code of any sort. But that's not all! Much like you might be used to doing with rendered HTML or CSS, you can edit Vue components directly in the browser: add values to your props, modify them, or perhaps remove them entirely to see how your app responds. I do almost all of my development in a terminal window, but I'm finding myself taking advantage of this in-browser, instant feedback loop more and more frequently. To my mind, there's no higher praise than willingly modifying your own workflow to leverage a tool.

Additionally, component navigation is outstanding. Not only is there an *Inspect Vue component* option in the context menu, there's also a fuzzy search to filter your components by name, which is a very nice touch. You can also see the history of your Vuex state and "time travel" back to any given point (which is an invaluable tool when debugging issues rooted in complex state changes).

Another useful feature is a tab to track Custom Events from your own components (those of the `this.$emit('YourEvent')` variety), but lamentably not native events like `<button v-on:click="counter += 1">Add 1</button>`:

sitepoint-example

127.0.0.1:8080

3000 4000

# r/vuejs

Emit Event

10

How to make Router wait for Vuex Async?

2

I would appreciate feedback for my first project in Vue.

51

Reasons Why Vue.js Is Getting More Traction Every Month

140M

1.1K

1.1K

Console Inspector Debugger {} Style Editor Performance Memory Network Vue

Ready. Detected Vue 2.5.21.

Filter events

Hi Sitepoint reader! \$emit by <App> 13:01:21

Hi Sitepoint reader! \$emit by <App> 13:01:27

Hi Sitepoint reader! \$emit by <App> 13:01:31

Hi Sitepoint reader! \$emit by <App> 13:01:31

event info

name: "Hi Sitepoint reader!"

type: "\$emit"

source: "<app>"

payload: Array[0]

## AVOID EVENT BUSES

I find Vue's native mechanism of passing state to be excellent, but inevitably, as your app scales, you're going to need the same data in multiple different places and that's going to be unwieldy. The typical example of this is the instance of data sharing between sibling components, or even two components that aren't related in any way. One project I worked on recently needed to access a `category` field, both on a list item and in a "grouping" view elsewhere. Vuex is frequently an elegant solution to problems of this nature, because it adroitly sidesteps the constraints inherent in passing data via parent and child relationships.

I must admit that there was a time in my life where I would reach straight for an event bus rather than delve into Flux architecture. I had read from a few people on Hacker News that Flux, and, by extension, Redux and Vuex, were complicated to pick up, and as a result, avoided them for as long as possible. Finally, I sat down to get my head around it and I wish someone had told me this: "It's just the Command Pattern". That's pretty much much all there is to it. As you may know, you write commands (or "Actions" in Flux parlance) which are executed (by the "Dispatcher" in this case), and the result is passed to a receiver (a Store) at a later time. I'm pleased to report that once I need to manage any degree of state that's more complex than "pass props down the hierarchy", I reach for Vuex. Try it: it's really not as



complicated as you may have heard!

## VUE CLI

Vue CLI is a relatively new addition to the Vue roster, but I can't speak highly enough of it. I've lost many hours over the last couple of years fiddling with configurations and integrating libraries correctly, so to be able to effortlessly scaffold an app and be up and running is a breath of fresh air. Simply run `vue create <app_name>` and you'll be prompted to select various options which are installed and configured for you. Out of the box, it will set you up with Babel and ESLint (both are absolute *musts* to my mind), but you can customize your setup to include a variety of extras out of the box. Here's a quick rundown of the options available to you (as of version 3.2.1):

- impose a degree of type safety on your project with Typescript
- leverage Progressive Web Apps for mobile support
- seamlessly route your single page app with Vue Router
- manage more complex levels of state with Vuex
- make writing CSS more pleasant with CSS Preprocessors, including Sass/SCSS, Less and Stylus flavors
- pick your linting options from Typescript, various degrees of ESLint strictness (ranging from “error prevention only” all the way to my personal favorite, the Airbnb config)
- ensure code stability with Mocha and its BDD cousin Chai, or Facebook's Jest
- there's even end-to-end testing options in Cyprus or Nightwatch

This setup even gives you the option to “Save this as a

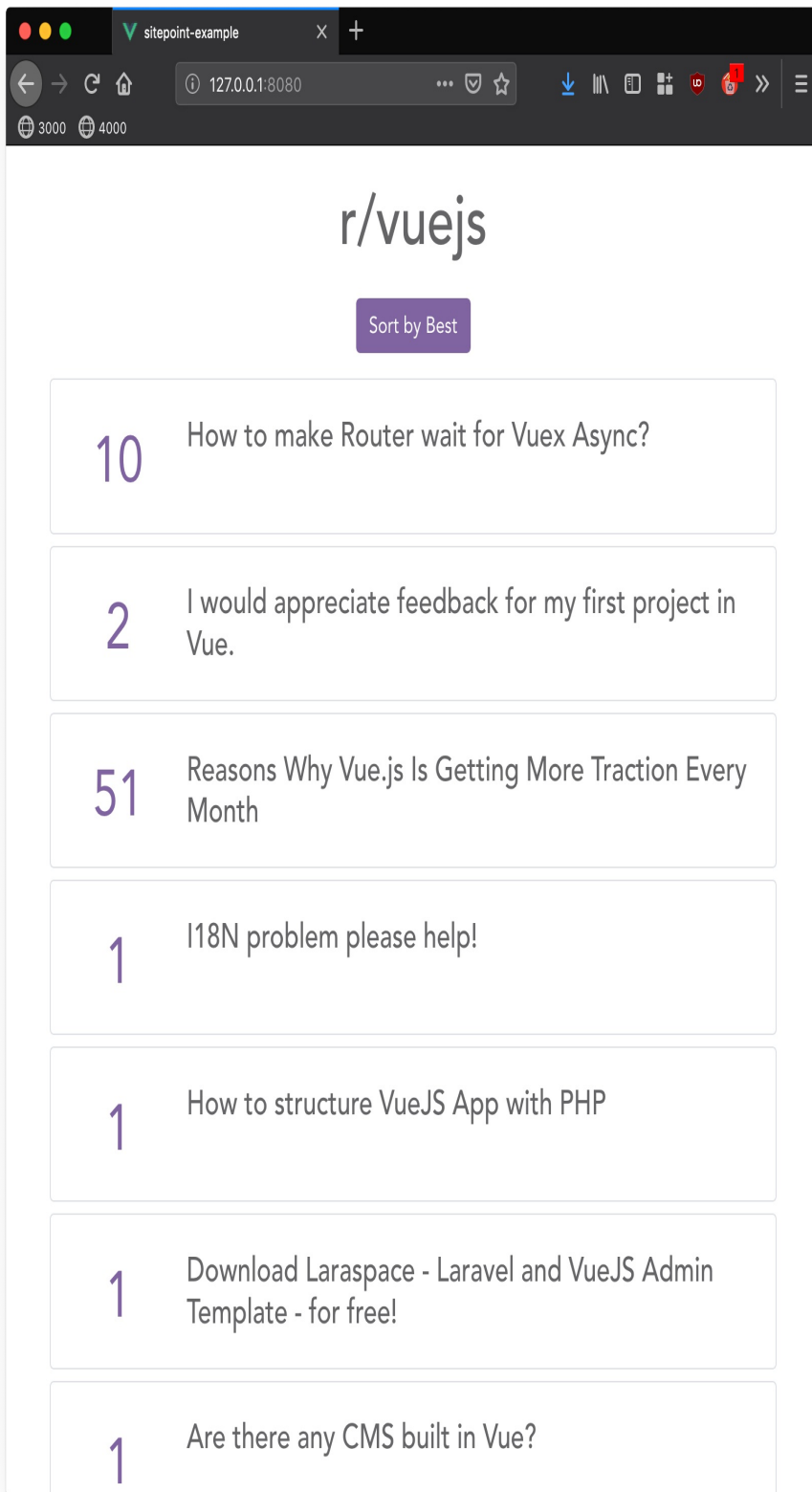
preset for future projects”, which is useful once you’ve developed a preferred stack. Check out “Getting Started with Vuex: a Beginner’s Guide” in this Vue series for more Vue CLI goodness.

## COMPUTED PROPERTIES

When I first started using Vue, I put all my rendering helpers in functions and called them from the template. As my projects grew, this quickly became unwieldy and I searched for a better way. Enter computed properties, which are the idiomatic way of declaring more complex logic relating to calculation of anything to be rendered on the page. They have the added benefit of de-cluttering your template (and while it may be an unpopular opinion in modern JS, I firmly believe in separating the concerns of my presentation logic and rendering code where possible). Let’s walk through it with a quick example.

Here’s the VueJS subreddit. I’m going to `wget https://www.reddit.com/r/vuejs.json` and put that in a file, but if you’re playing along at home, there’s nothing to stop you using this as a basis for writing your own Vue-based Reddit interface and registering for an API key.

Let’s render it on the page:



So far, so good. Now let's add an option to sort by score. Here's my code, in case you'd like to emulate it (you'll want to add Bootstrap to your project).

## App.vue

```
<template>
  <div id="app" class="container">
    <h1 class="list-title">r/vuejs</h1>
    <button v-on:click="sortHottest = !sortHottest" class="btn">
      {{ sortHottest ? "Sort by Best" : "Sort by Hottest" }}
    <div v-for="story in sortHottest ? sortByHottest : sortByBest">
      <Story v-bind:story="story.data" />
    </div>
  </div>
</template>

<script>
import Story from './components/Story.vue'
import stories from './vue_stories.json'

export default {
  name: 'app',
  data: () => {
    return {
      // Submissions from JSON file (but you can switch to API)
      stories: stories.data.children,
      // Initial sort order of stories:
      sortHottest: true
    }
  },
  components: {
    Story
  },
  computed: {
    sortByBest: function () {
      // Sorts by score, descending. We use slice() to avoid mutating the array
      return this.stories.slice().sort((a, b) => b.data.score - a.data.score)
    },
    sortByHottest: function () {
      // The real Reddit 'Hottest' algorithm is a function of score, time, and upvotes
      // We'll skip all that for this example and just sort by 'hottest'!
      return this.stories
    }
  }
}
```

```

    }
  }
</script>

<style>
#app {
  font-family: 'Avenir', Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #656568;
  margin-top: 1.5em;
}
.list-title {
  margin-bottom: 0.5em;
  font-size: 3em;
}

.btn-sort {
  margin-bottom: 0.6em;
  background-color: #8064A2 !important;
  border-color: #8064A2 !important;
}
</style>

```

## components/Story.vue

```

<template>
  <div class="row border rounded story">
    <div class="col-2">
      <p class="score">{{ story.score }}</p>
    </div>
    <div class="col-10 text-left title">
      <h4>{{ story.title }}</h4>
    </div>
  </div>
</template>

<script>
export default {
  name: 'Story',
  props: {
    story: Object
  }
}
</script>

```

```
<style scoped>
.score {
  margin-left: 0.3em;
  margin-top: 0.5em;
  font-size: 2.8em;
  color: #8064A2;
}
.title {
  margin-top: 1.5em;
}

.story {
  margin-top: 0.5em;
  border-width: 3px;
}
</style>
```

Now although my sorting example is pretty concise (thanks, ES6!), the benefits should be readily apparent: with this implementation, we enjoy uncluttered content in the template and neatly compartmentalized, self-documenting (mostly!) code in the script. This comes with the usual advantages of Vue's data model in that, by avoiding side effects (like mutating the `stories` data), we can leverage Vue's data binding to ensure this is fully reactive. For example, in the event a comment got upvoted sufficiently to overtake the one above it, this would be reflected in our `sortByBest()` result! So far, this is the same as calling a function directly in your template code, but this is where Vue shines: computed properties are only evaluated in the event that their dependencies change (in this case, `stories`). If you called this as a function, it would be run on every render, which could be costly in any number of ways (frequently rendered component, large lists to sort, expensive

computations on list items, etc.). This is often an easy win when optimizing slow pages, so I always look out for it.

## Wrapping Up

Those are the five things I wish I'd known earlier when learning Vue. I hope they help you!