



This book is provided in digital form with the permission of the rightsholder as part of a Google project to make the world's books discoverable online.

The rightsholder has graciously given you the freedom to download all pages of this book. No additional commercial or other uses have been granted.

Please note that all copyrights remain reserved.

About Google Books

Google's mission is to organize the world's information and to make it universally accessible and useful. Google Books helps readers discover the world's books while helping authors and publishers reach new audiences. You can search through the full text of this book on the web at <http://books.google.com/>

Working with Vue.js

Copyright © 2019 SitePoint Pty. Ltd.

Ebook ISBN: 978-1-925836-32-5

Cover Design: Alex Walker

Project Editor: James Hibbard

Notice of Rights

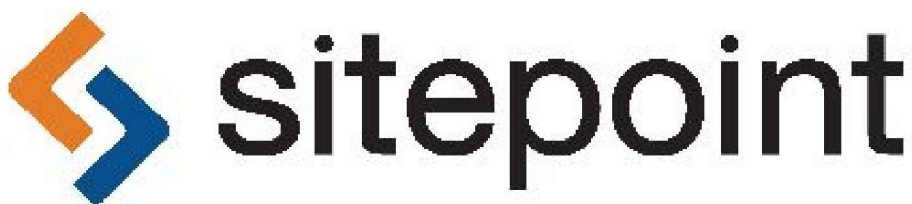
All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

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

Email: 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 is a collection of three books covering crucial Vue topics. It contains:

- *Working with Vue.js*
- *11 Practical Vue.js Projects*
- *Vue.js: Tools & Skills*

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 school.</p>
```

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!

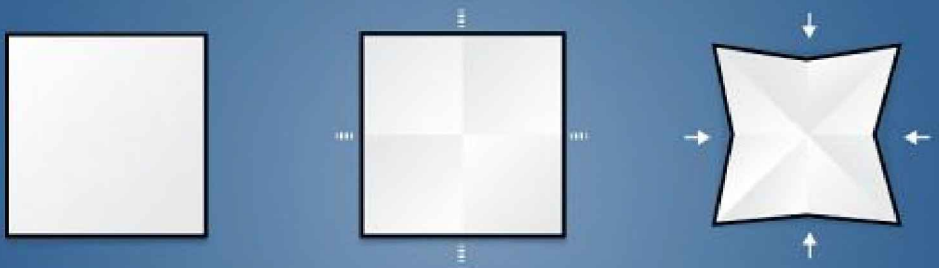
Warnings highlight any gotchas that are likely to trip you up along the way.

Book 1: Working with Vue.js



WORKING WITH VUE.JS

A BEST PRACTICE GUIDE



BUILD YOUR OWN SOPHISTICATED WEB APPS

Chapter 1: Getting up and Running with the Vue.js 2.0 Framework

by Jack Franklin

As soon as the popular JavaScript framework Vue.js released v2, I was eager to give it a spin and see what it's like to work with. As someone who's pretty familiar with Angular and React, I was looking forward to seeing the similarities and differences between them and Vue.

Vue 2 sports excellent performance stats, a relatively small payload (the bundled runtime version of Vue weighs in at 30KB once minified and gzipped), along with updates to companion libraries like vue-router and Vuex, the state management library for Vue. There's far too much to cover in just one article, but keep an eye out for some later articles where we'll look more closely at various libraries that couple nicely with the core framework.

Inspiration from Other Libraries

As we go through this tutorial, you'll see many features that Vue has that are clearly inspired by other frameworks. This is a good thing; it's great to see new frameworks take some ideas from other libraries and improve on them. In particular, you'll see Vue's templating is very close to Angular's, but its components and component lifecycle methods are closer to React's (and Angular's, as well).

One such example of this is that, much like React and nearly every framework in JavaScript land today, Vue uses the idea of a virtual DOM to keep rendering efficient. Vue uses a fork of snabbdom, one of the more popular virtual DOM libraries. The Vue site includes documentation on its Virtual DOM rendering, but as a user all you need to know is that Vue is very good at keeping your rendering fast (in fact, it performs better than React in many cases), meaning you can rest assured you're building on a solid platform.

Components, Components, Components

Much like other frameworks these days, Vue's core building block is the component. Your application should be a series of components that build on top of each other to produce the final application. Vue.js goes one step further by suggesting (although not enforcing) that you define your components in a single `.vue` file, which can then be parsed by build tools (we'll come onto those shortly). Given that the aim of this article is to fully explore Vue and what it feels like to work with, I'm going to use this convention for my application.

A Vue file looks like so:

```
<template>
  <p>This is my HTML for my component</p>
</template>

<script>
  export default {
    // all code for my component goes here
  }
</script>

<style scoped>
  /* CSS here
   * by including `scoped`, we ensure that all CSS
   * is scoped to this component!
   */
</style>
```

Alternatively, you can give each element a `src` attribute and point to a separate HTML, JS or CSS file respectively if you don't like having all parts of the component in one file.

Setting Up a Project

Whilst the excellent Vue CLI exists to make setting up a full project easy, when starting out with a new library I like to do it all from scratch so I get more of an understanding of the tools.

These days, webpack is my preferred build tool of choice, and we can couple that with the vue-loader plugin to support the Vue.js component format that I mentioned previously. We'll also need Babel and the `env` preset, so we can write all our code using modern JavaScript syntax, as well as the `webpack-dev-server`, which will update the browser when it detects a file change.

Let's initialize a project and install the dependencies:

```
mkdir vue2-demo-project
cd vue2-demo-project
npm init -y
npm i vue
npm i webpack webpack-cli @babel/core @babel/preset-env babel-loader vue-loader vue-template-compiler
```

Then create the initial folders and files:

```
mkdir src
touch webpack.config.js src/index.html src/index.js
```

The project structure should look like this:

```
.
├── package.json
├── package-lock.json
├── src
│   ├── index.html
│   └── index.js
└── webpack.config.js
```

Now let's set up the webpack configuration. This boils down to the following:

- Tell webpack to use the `vue-loader` for any `.vue` files
- Tell webpack to use Babel and the `env` preset for any `.js` files
- Tell webpack to generate an HTML file for the dev-server to serve, using `src/index.html` as a template:

```
//webpack.config.js
const VueLoaderPlugin = require('vue-loader/lib/plugin')
const HtmlWebPackPlugin = require("html-webpack-plugin")

module.exports = {
  module: {
    rules: [
      {
        test: /\.vue$/,
        loader: 'vue-loader',
      },
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env']
          }
        }
      }
    ]
  },
  plugins: [
    new VueLoaderPlugin(),
    new HtmlWebPackPlugin({
      template: "./src/index.html"
    })
  ]
}
```

Finally, we'll add some content to the HTML file and we're ready to go!

```
<!-- src/index.html -->
<!DOCTYPE html>
<html>
```



```
<head>
  <title>My Vue App</title>
</head>
<body>
  <div id="app"></div>
</body>
</html>
```

We create an empty `div` with the ID of `app`, as this is the element that we're going to place our Vue application in. I always prefer to use a `div`, rather than just the `body` element, as that lets me have control over the rest of the page.

Writing Our First Vue.js App

We're going to stay true to every programming tutorial ever and write a Vue application that puts "Hello, World!" onto the screen before we dive into something a bit more complicated.

Each Vue app is created by importing the library and then instantiating a new `Vue` instance:

```
import Vue from 'vue'

const vm = new Vue({
  el: '#app',
})
```

We give `Vue` an element to render onto the page, and with that, we've created a Vue application! We pass a selector for the element that we want `Vue` to replace with our application. This means when `Vue` runs it will take the `div#app` that we created and replace it with our application.

The reason we use the variable name `vm` is because it stands for "View Model". Although not strictly associated with the "Model View View-Model" (MVVM) pattern, `Vue` was inspired in part by it, and the convention of using the variable name `vm` for `Vue` applications has stuck. Of course, you can call the variable whatever you'd like!

So far, our application isn't doing anything, though, so let's create our first component, `App.vue`, that will actually render something onto the page.

`Vue` doesn't dictate how your application is structured, so this one is up to you. I ended up creating one folder per component, in this case `App` (I like the capital letter, signifying a component), with three files in it:

- `index.vue`
- `script.js`
- `style.css`

```
mkdir src/App
touch src/App/{index.vue,script.js,style.css}
```

The file structure should now be:

```
.
├── package.json
├── package-lock.json
├── src
│   ├── App
│   │   ├── index.vue
│   │   ├── script.js
│   │   └── style.css
│   ├── index.html
│   └── index.js
└── webpack.config.js
```

`App/index.vue` defines the template, then imports the other files. This is in keeping with the structure recommended in the [What About Separation of Concerns?](#) section of `Vue`'s docs.

```
<!-- src/App/index.vue -->
<template>
  <p>Hello, World!</p>
```

```
</template>
<script src="./script.js"></script>
<style scoped src="./style.css"></style>
```

I like calling it `index.vue`, but you might want to call it `app.vue` too so it's easier to search for. I prefer importing `App/index.vue` in my code versus `App/app.vue`, but again you might disagree, so feel free to pick whatever you and your team like best.

For now, our template is just `<p>Hello, World!</p>`, and I'll leave the CSS file blank. The main work goes into `script.js`, which looks like so:

```
export default {
  name: 'App',
  data() {
    return {}
  },
}
```

Doing this creates a component which we'll give the name `App`, primarily for debugging purposes, which I'll come to later, and then defines the data that this component has and is responsible for. For now, we don't have any data, so we can just tell Vue that by returning an empty object. Later on, we'll see an example of a component using data.

Now we can head back into `src/index.js` and tell the Vue instance to render our `App` component:

```
import Vue from 'vue'

import AppComponent from './App/index.vue'

const vm = new Vue({
  el: '#app',
  components: {
    app: AppComponent,
  },
  render: h => h('app'),
})
```

Firstly, we import the component, trusting webpack and the vue-loader to take care of parsing it. We then declare the component. This is an important step: by default, Vue components are not globally available. Each component must have a list of all the components they're going to use, and the tag that it will be mapped to. In this case, because we register our component like so:

```
components: {
  app: AppComponent,
}
```

This means that in our templates we'll be able to use the `app` element to refer to our component.

Finally, we define the `render` function. This function is called with a helper — commonly referred to as `h` — that's able to create elements. It's not too dissimilar to the `React.createElement` function that React uses. In this case, we give it the string `'app'`, because the component we want to render is registered as having the tag `app`.

More often than not (and for the rest of this tutorial) we won't use the `render` function on other components, because we'll define HTML templates. But the Vue.js guide to the render function is worth a read if you'd like more information.

Once we've done that, the final step is to create an npm script in `package.json`:

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

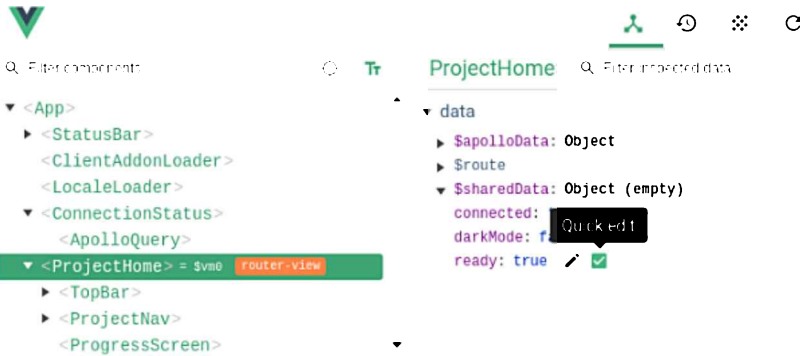
Now, run `npm run start`. Your default browser should open at `http://localhost:8080/` and you should see "Hello, World!" on the screen.

Try editing `src/index.vue` to change the message to something else. If all has gone correctly, webpack-dev-server should refresh the page to reflect your changes.

Yay! We're up and running with Vue.js.

Vue Devtools

Before we dive into a slightly more complicated app with Vue, now is a good time to mention that you should definitely get the Vue devtools installed. These sit within the Chrome developer tools and give you a great way to look through your app and all the properties being passed round, state that each component has, and so on.



Building the App

As an example application, we're going to be using the GitHub API to build an application that lets us enter a username and see some GitHub stats about that user. I've picked the GitHub API here as it's familiar to most people, usable without authenticating, and gives us a fair amount of information.

Before starting an application I like to have a quick think about what components we'll need, and I'm thinking that our `App` component will render two further components: `GithubInput`, for taking input from the user, and `GithubOutput`, which will show the user's information on the screen. We'll start with the input.

Github

You can find all the code on GitHub and even check out the application running online.

Initial Setup

Create folders for the `GithubOutput` and `GithubInput` components within the `src` directory:

```
mkdir src/{GithubInput,GithubOutput}
```

Add the necessary files to each:

```
touch src/GithubInput/{index.vue,script.js,style.css}
touch src/GithubOutput/{index.vue,script.js,style.css}
```

The structure of the `src` folder should now look like so:

```
.
├── App
│   ├── index.vue
│   ├── script.js
│   └── style.css
├── GithubInput
│   ├── index.vue
│   ├── script.js
│   └── style.css
├── GithubOutput
│   ├── index.vue
│   ├── script.js
│   └── style.css
├── index.html
└── index.js
```

Forms in Vue.js

Let's start with the GithubInput component. As with the App component, the index.vue file should contain the template, as well as loading in the script and CSS file. The template simply contains `<p>github input</p>` for now. We'll fill it in properly shortly. I like putting in some dummy HTML so I can check I've got the template wired up properly when creating a new component:

```
<!-- src/GithubInput/index.vue -->
<template>
  <p>github input</p>
</template>

<script src="./script.js"></script>
<style scoped src="./style.css"></style>
```

When creating this component the one thing we do differently is create a piece of data that's associated with the component. This is very similar to React's concept of state:

```
// src/GithubInput/script.js
export default {
  name: 'GithubInput',
  data() {
    return {
      username: '',
    }
  }
}
```

This says that this component has a piece of data, username, that it owns and is responsible for. We'll update this based on the user's input shortly.

Finally, to get this component onto the screen, I need to register it with the App component, as it's the App component that will be rendering it.

To do this, I update src/App/script.js and tell it about GithubInput:

```
// src/App/script.js
import GithubInput from '../GithubInput/index.vue'

export default {
  name: 'App',
  components: {
    'github-input': GithubInput,
  },
  data() {
    return {}
  },
}
```

And then I can update the App component's template:

```
<!-- src/App/index.vue -->
<div>
  <p>Hello World</p>
  <github-input></github-input>
</div>
```

A restriction of Vue components (which is also true in Angular and React) is that each component must have one root node, so when a component has to render multiple elements, it's important to remember to wrap them all in something, most commonly a div.

Tracking a Form Input

Our GithubInput component will need to do two things:

- Keep track of the current value of the input
- Communicate that the value has changed, so that other components can know and therefore update their state.

We can do the first version by creating a form with an input element in it. We can use Vue's built-in directives that enable us to keep

track of form values. The template for `GithubInput` looks like so:

```
<form v-on:submit.prevent="onSubmit">
  <input type="text" v-model="username" placeholder="Enter a github username here" />
  <button type="submit">Go!</button>
</form>
```

There are two important attributes that you'll notice: `v-on` and `v-model`.

`v-on` is how we bind to DOM events in Vue and call a function. For example, `<p v-on:click="foo">Click me!</p>` would call the component's `foo` method every time the paragraph was clicked. If you'd like to go through event handling in greater detail, I highly recommend the Vue documentation on event handling.

`v-model` creates a two-way data binding between a form input and a piece of data. Behind the scenes, `v-model` is effectively listening for change events on the form input and updating the data in the Vue component to match.

Taking our template above into consideration, here's how we're using `v-on` and `v-model` to deal with the data in the form:

- `v-on:submit.prevent="onSubmit"` binds the method `onSubmit` to be run when the form is submitted. By adding `.prevent` that means that Vue will automatically prevent the default action from occurring. (If Vue didn't do this, we could call `event.preventDefault()` in our code, but we might as well take advantage of Vue's feature.)
- `v-model:username` binds the input's value to a value, `username`, in our code. For those of you familiar with Angular you may recognize this as very similar to `ng-model`. When we created `GithubInput` we declared that it had a piece of data, `username`, and here we've bound that piece of data to the input field. The two will automatically be kept in sync.

Now, back in our component's JavaScript, we can declare the `onSubmit` method. Note that the name here is entirely arbitrary — you can choose whatever you'd like — but I like to stick with the convention of naming the function after the event that will trigger it:

```
export default {
  name: 'GithubInput',
  data() {
    return { username: '', }
  },
  methods: {
    onSubmit(event) {
      if (this.username && this.username !== '') {
      }
    }
  }
}
```

We can refer to data directly on `this`, so `this.username` will give us the latest value of the text box. If it's not empty, we want to let other components know that the data has changed. For this, we'll use a message bus. These are objects that components can emit events on and use to listen to other events. When your application grows larger you might want to look into a more structured approach, such as `Vuex`. For now, a message bus does the job.

The great news is that we can use an empty Vue instance as a message bus. To do so, we'll create `src/bus.js`, which simply creates a Vue instance and exports it:

```
import Vue from 'vue'
const bus = new Vue()

export default bus
```

In the `GithubInput` component we can then import that module and use it by emitting an event when the username changes:

```
import bus from '../bus'

export default {
  ...,
  methods: {
    onSubmit(event) {
      if (this.username && this.username !== '') {
        bus.$emit('new-username', this.username)
      }
    }
  }
}
```

```

    }
  },
  ...
}

```

With that, our form is done, and we're ready to start doing something with the resulting data.

Displaying Results From GitHub

The `GithubOutput` component has the same structure as our other two components. In `GithubOutput/script.js` we also import the `bus` module, as we'll need it to know when the username changes. The data that this component will be responsible for will be an object that maps GitHub usernames to the data we got from the GitHub API. This means we won't have to make the request to the API every single time; if we've already fetched the data previously we can simply reuse it. We'll also store the last username we were given, so we know what data to display on screen:

```

// src/GithubOutput/script.js
import bus from '../bus'
import Vue from 'vue'

export default {
  name: 'GithubOutput',
  data() {
    return {
      currentUsername: null,
      githubData: {}
    }
  }
}

```

When the component is created, we want to listen for any `new-username` events that are emitted on the message bus. Thankfully, Vue supports a number of lifecycle hooks, including `created`. Because we're responsible developers, let's also stop listening for events when the component is destroyed by using the `destroyed` event:

```

export default {
  name: 'GithubOutput',
  data: { ... },
  created() {
    bus.$on('new-username', this.onUsernameChange)
  },
  destroyed() {
    bus.$off('new-username', this.onUsernameChange)
  }
}

```

We then define the `onUsernameChange` method, which will be called and will set the `currentUsername` property:

```

methods: {
  onUsernameChange(name) {
    this.currentUsername = name
  }
},

```

Note that we don't have to explicitly bind the `onUsernameChange` method to the current scope. When you define methods on a Vue component, Vue automatically calls `myMethod.bind(this)` on them, so they're always bound to the component. This is one of the reasons why you need to define your component's methods on the `methods` object, so Vue is fully aware of them and can set them up accordingly.

Conditional Rendering

If we don't have a username — as we won't when the component is first created — we want to show a message to the user. Vue has a number of conditional rendering techniques, but the easiest is the `v-if` directive, which takes a condition and will only render the element if it exists. It also can be paired with `v-else`:

```

<!-- src/GithubOutput/index.vue-->

```

```

<template>
  <div>
    <p v-if="currentUsername == null">
      Enter a username above to see their GitHub data
    </p>
    <p v-else>
      Below are the results for {{ currentUsername }}
    </p>
  </div>
</template>
<script src="./script.js"></script>
<style scoped src="./style.css"></style>

```

Once again, this will look very familiar to any Angular developers. We use double equals rather than triple equals here because we want the conditional to be true not only if `currentUsername` is `null` but also if it's `undefined`, and `null == undefined` is `true`.

Fetching from GitHub

Vue.js doesn't ship with a built-in HTTP library, and for good reason. These days the `fetch` API ships natively in many browsers (although at the time of writing, not IE11, Safari or iOS Safari). For the sake of this tutorial I'm not going to use a polyfill, but you can easily polyfill the API in browsers if you need to. If you don't like the `fetch` API there are many third-party libraries for HTTP, and the one mentioned in the Vue docs is `Axios`.

I'm a big proponent of frameworks like Vue not shipping with HTTP libraries. It keeps the bundle size of the framework down and leaves it to developers to pick the library that works best for them, and easily customize requests as needed to talk to their API. I'll stick to the `fetch` API in this article, but feel free to swap it out for one that you prefer.

If you need an introduction to the `fetch` API, check out Ludovico Fischer's post on SitePoint, which will get you up to speed.

To make the HTTP request, we'll give the component another method, `fetchGithubData`, that makes a request to the GitHub API and stores the result. It will also first check to see if we already have data for this user, and not make the request if so:

```

methods: {
  ...
  fetchGithubData(name) {
    // if we have data already, don't request again
    if (this.githubData.hasOwnProperty(name)) return

    const url = `https://api.github.com/users/${name}`
    fetch(url)
      .then(r => r.json())
      .then(data => {
        // in here we need to update the githubData object
      })
  }
}

```

We then finally just need to trigger this method when the username changes:

```

methods: {
  onUsernameChange(name) {
    this.currentUsername = name
    this.fetchGithubData(name)
  },
  ...
}

```

There's one other thing to be aware of, due to the way that Vue keeps track of the data you're working with so that it knows when to update the view. There is a great [Reactivity guide](#) which explains it in detail, but essentially Vue isn't able to magically know when you've added or deleted a property from an object, so if we do:

```
this.githubData[name] = data
```

Vue won't recognize that and won't update our view. Instead, we can use the special `Vue.set` method, which explicitly tells Vue that we've added a key. The above code would then look like so:

```
Vue.set(this.githubData, name, data)
```

This code will modify this `githubData`, adding the key and value that we pass it. It also notifies Vue of the change so it can rerender.

Now our code looks like so:

```
const url = `https://api.github.com/users/${name}`
fetch(url)
  .then(r => r.json())
  .then(data => {
    Vue.set(this.githubData, name, data)
  })
```

Finally, we need to register the `GitHubOutput` component with the `App` component:

```
// src/App/script.js
import GithubInput from '../GithubInput/index.vue'
import GithubOutput from '../GithubOutput/index.vue'

export default {
  name: 'App',
  components: {
    'github-input': GithubInput,
    'github-output': GithubOutput,
  },
  data() {
    return {}
  },
}
```

And include it in the template:

```
<!-- src/App/index.vue -->
<template>
  <div>
    <github-input></github-input>
    <github-output></github-output>
  </div>
</template>
```

Although we haven't yet written the view code to show the fetched data on screen, you should be able to fill in the form with your username and then inspect the Vue devtools to see the data requested from GitHub. This shows how useful and powerful these devtools are; you can inspect the local state of any component and see exactly what's going on.

Showing Some Stats in the View

We can now update the template to show some data. Let's wrap this code in another `v-if` directive so that we only render the data if the request has finished:

```
<!-- src/GithubOutput/index.vue -->
<p v-if="currentUsername == null">
  Enter a username above to see their GitHub data
</p>
<p v-else>
  Below are the results for {{ currentUsername }}
  <div v-if="githubData[currentUsername]">
    <h4>{{ githubData[currentUsername].name }}</h4>
    <p>{{ githubData[currentUsername].company }}</p>
    <p>Number of repos: {{ githubData[currentUsername].public_repos }}</p>
  </div>
</p>
```

With that, we can now render the GitHub details to the screen, and our app is complete!

Refactors

There are definitely some improvements we can make. The above bit of HTML that renders the GitHub data only needs a small part of it – the data for the current user. This is the perfect case for another component that we can give a user’s data to and it can render it.

Let’s create a `GithubUserData` component, following the same structure as with our other components:

```
mkdir src/GithubUserData
touch src/GithubUserData/{index.vue,script.js,style.css}
```

There’s only one tiny difference with this component: it’s going to take a property, `data`, which will be the data for the user. Properties (or, “props”) are bits of data that a component will be passed by its parent, and they behave in Vue much like they do in React. In Vue, you have to explicitly declare each property that a component needs, so here I’ll say that our component will take one prop, `data`:

```
// src/GithubUserData/script.js
export default {
  name: 'GithubUserData',
  props: ['data'],
  data() {
    return {}
  }
}
```

One thing I really like about Vue is how explicit you have to be; all properties, data, and components that a component will use are explicitly declared. This makes the code much nicer to work with and, I imagine, much easier as projects get bigger and more complex.

In the new template, we have exactly the same HTML as before, although we can refer to `data` rather than `githubData[currentUsername]`:

```
<!-- src/GithubUserData/index.vue -->
<template>
  <div v-if="data">
    <h4>{{ data.name }}</h4>
    <p>{{ data.company }}</p>
    <p>Number of repos: {{ data.public_repos }}</p>
  </div>
</template>
<script src="./script.js"></script>
<style scoped src="./style.css"></style>
```

To use this component we need to update the `GithubOutput` component. Firstly, we import and register `GithubUserData`:

```
// src/GithubOutput/script.js
import bus from '../bus'
import Vue from 'vue'
import GithubUserData from '../GithubUserData/index.vue'

export default {
  name: 'GithubOutput',
  components: {
    'github-user-data': GithubUserData,
  },
  ...
}
```

You can use any name for the component when declaring it, so where I’ve placed `github-user-data`, you could place anything you want. It’s advisable that you stick to components with a dash in them. Vue doesn’t enforce this, but the W3C specification on custom elements states that they must contain a dash to prevent naming collisions with elements added in future versions of HTML.

Once we’ve declared the component, we can use it in our template:

```
<!-- src/GithubOutput/index.vue -->
<p v-else>
  Below are the results for {{ currentUsername }}:
  <github-user-data :data="githubData[currentUsername]"></github-user-data>
</p>
```

The crucial part here is how I pass the `data` property down to the component:

```
:data="githubData[currentUsername]"
```

The colon at the start of that attribute is crucial; it tells Vue that the attribute we're passing down is dynamic and that the component should be updated every time the data changes. Vue will evaluate the value of `githubData[currentUsername]` and ensure that the `GitHubUserData` component is kept up to date as the data changes.

If you find `:data` a bit short and magical, you can also use the longer `v-bind` syntax:

```
v-bind:data="githubData[currentUsername]"
```

The two are equivalent, so use whichever you prefer.

Conclusion

With that, our GitHub application is in a pretty good state! You can find all the code on GitHub and even check out the application running online.

I had high hopes when getting started with Vue, as I'd heard only good things, and I'm happy to say it really met my expectations. Working with Vue feels like taking the best parts of React and merging them with the best parts of Angular. Some of the directives (like `v-if`, `v-else`, `v-model` and so on) are really easy to get started with (and easier to immediately understand than doing conditionals in React's JSX syntax), but Vue's component system feels very similar to React's.

You're encouraged to break your system down into small components, and all in all I found it a very seamless experience. I also can't commend the Vue team highly enough for their documentation: it's absolutely brilliant. The guides are excellent, and the API reference is thorough yet easy to navigate to find exactly what you're after.

Chapter 2: Getting Started with Vuex: a Beginner's Guide

by Michael Wanyoike

In single-page applications, the concept of *state* relates to any piece of data that can change. An example of state could be the details of a logged-in user, or data fetched from an API.

Handling state in single-page apps can be a tricky process. As an application gets larger and more complex, you start to encounter situations where a given piece of state needs to be used in multiple components, or you find yourself passing state through components that don't need it, just to get it to where it needs to be. This is also known as "prop drilling", and can lead to some unwieldy code.

Vuex is the official state management solution for Vue. It works by having a central store for shared state, and providing methods to allow any component in your application to access that state. In essence, Vuex ensures your views remain consistent with your application data, regardless of which function triggers a change to your application data.

In this article, I'll offer you a high-level overview of Vuex and demonstrate how to implement it into a simple app.

A Shopping Cart Example

Let's consider a real-world example to demonstrate the problem that Vuex solves.

When you go to a shopping site, you'll usually have a list of products. Each product has an *Add to Cart* button and sometimes an *Items Remaining* label indicating the current stock or the maximum number of items you can order for the specified product. Each time a product is purchased, the current stock of that product is reduced. When this happens, the *Items Remaining* label should update with the correct figure. When the product's stock level reaches 0, the label should read *Out of Stock*. In addition, the *Add to Cart* button should be disabled or hidden to ensure customers can't order products that are currently not in inventory.

Now ask yourself how you'd implement this logic. It may be trickier than you think. And let me throw in a curve ball. You'll need another function for updating stock records when new stock comes in. When the depleted product's stock is updated, both the *Items Remaining* label and the *Add to Cart* button should be updated instantly to reflect the new state of the stock.

Depending on your programming prowess, your solution may start to look a bit like spaghetti. Now, let's imagine your boss tells you to develop an API that allows third-party sites to sell the products directly from the warehouse. The API needs to ensure that the main shopping website remains in sync with the products' stock levels. At this point you feel like pulling your hair out and demanding why you weren't told to implement this earlier. You feel like all your hard work has gone to waste, as you'll need to completely rework your code to cope with this new requirement.

This is where a state management pattern library can save you from such headaches. It will help you organize the code that handles your front-end data in a way that makes adding new requirements a breeze.

Prerequisites

Before we start, I'll assume that you:

- have a basic knowledge of Vue.js
- are familiar with ES6 and ES7 language features

You'll also need to have a recent version of Node.js that's not older than version 6.0. At the time of writing, Node.js v10.13.0 (LTS) and npm version 6.4.1 are the most recent. If you don't have a suitable version of Node installed on your system already, I recommend using a version manager.

Finally, you should have the most recent version of the Vue CLI installed:

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

Build a Counter Using Local State

In this section, we're going to build a simple counter that keeps track of its state locally. Once we're done, I'll go over the fundamental concepts of Vuex, before looking at how to rewrite the counter app to use Vue's official state management solution.

Getting Set Up

Let's generate a new project using the CLI:

```
vue create vuex-counter
```

A wizard will open up to guide you through the project creation. Select *Manually select features* and ensure that you choose to install **Vuex**.

Next, change into the new directory and in the `src/components` folder, rename `HelloWorld.vue` to `Counter.vue`:

```
cd vuex-counter
mv src/components/HelloWorld.vue src/components/Counter.vue
```

Finally, open up `src/App.vue` and replace the existing code with the following:

```
<template>
  <div id="app">
    <h1>Vuex Counter</h1>
    <Counter/>
  </div>
</template>

<script>
import Counter from './components/Counter.vue'

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

You can leave the styles as they are.

Creating the Counter

Let's start off by initializing a count and outputting it to the page. We'll also inform the user whether the count is currently even or odd. Open up `src/components/Counter.vue` and replace the code with the following:

```
<template>
  <div>
    <p>Clicked {{ count }} times! Count is {{ parity }}.</p>
  </div>
</template>

<script>
export default {
  name: 'Counter',
  data: function() {
    return {
      count: 0
    };
  },
  computed: {
    parity: function() {
      return this.count % 2 === 0 ? 'even' : 'odd';
    }
  }
}
</script>
```

As you can see, we have one state variable called `count` and a computed function called `parity` which returns the string `even` or `odd` depending on the whether `count` is an odd or even number.

To see what we've got so far, start the app from within the root folder by running `npm run serve` and navigate to `http://localhost:8080`.

Feel free to change the value of the counter to show that the correct output for both `count` and `parity` is displayed. When you're satisfied, make sure to reset it back to 0 before we proceed to the next step.

Incrementing and Decrementing

Right after the computed property in the `<script>` section of `Counter.vue`, add this code:

```
methods: {
  increment: function () {
    this.count++;
  },
  decrement: function () {
    this.count--;
  },
  incrementIfOdd: function () {
    if (this.parity === 'odd') {
      this.increment();
    }
  },
  incrementAsync: function () {
    setTimeout(() => {
      this.increment()
    }, 1000)
  }
}
```

The first two functions, `increment` and `decrement`, are hopefully self-explanatory. The `incrementIfOdd` function only executes if the value of `count` is an odd number, whereas `incrementAsync` is an asynchronous function that performs an increment after one second.

In order to access these new methods from the template, we'll need to define some buttons. Insert the following after the template code which outputs the count and parity:

```
<button @click="increment" variant="success">Increment</button>
<button @click="decrement" variant="danger">Decrement</button>
<button @click="incrementIfOdd" variant="info">Increment if Odd</button>
<button @click="incrementAsync" variant="warning">Increment Async</button>
```

After you've saved, the browser should refresh automatically. Click all of the buttons to ensure everything is working as expected.

Live Code

Vue Counter Using Local State shows what you should have ended up with.

The counter example is now complete. Let's move and examine the fundamentals of Vuex, before looking at how we would rewrite the counter to implement them.

How Vuex Works

Before we go over the practical implementation, it's best that we acquire a basic grasp of how Vuex code is organized. If you're familiar with similar frameworks such as Redux, you shouldn't find anything too surprising here. If you haven't dealt with any Flux-based state management frameworks before, please pay close attention.

The Vuex Store

The store provides a centralized repository for shared state in Vue apps. This is what it looks like in its most basic form:

```
// src/store/index.js

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

```

Vue.use(Vuex)

export default new Vuex.Store({
  state: {
    // put variables and collections here
  },
  mutations: {
    // put synchronous functions for changing state e.g. add, edit, delete
  },
  actions: {
    // put asynchronous functions that can call one or more mutation functions
  }
})

```

After defining your store, you need to inject it into your Vue.js application like this:

```

// src/main.js
import store from './store'

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

```

This will make the injected store instance available to every component in our application as `this.$store`.

Working with State

Also referred to as the *single state tree*, this is simply an object that contains all front-end application data. Vuex, just like Redux, operates using a single store. Application data is organized in a tree-like structure. Its construction is quite simple. Here's an example:

```

state: {
  products: [],
  count: 5,
  loggedInUser: {
    name: 'John',
    role: 'Admin'
  }
}

```

Here we have `products` that we've initialized with an empty array, and `count`, which is initialized with the value 5. We also have `loggedInUser`, which is a JavaScript object literal containing multiple fields. State properties can contain any valid datatype from Booleans, to arrays, to other objects.

There are multiple ways to display state in our views. We can reference the store directly in our templates using `$store`:

```

<template>
  <p>{{ $store.state.count }}</p>
</template>

```

Or we can return some store state from within a computed property:

```

<template>
  <p>{{ count }}</p>
</template>

```

```

<script>
export default {
  computed: {
    count() {
      return this.$store.state.count;
    }
  }
}

```

```
}  
</script>
```

Since Vuex stores are reactive, whenever the value of `$store.state.count` changes, the view will change as well. All this happens behind the scenes, making your code look simple and cleaner.

The `mapState` Helper

Now, suppose you have multiple states you want to display in your views. Declaring a long list of computed properties can get verbose, so Vuex provides a `mapState` helper. This can be used to generate multiple computed properties easily. Here's an example:

```
<template>  
  <div>  
    <p>Welcome, {{ loggedInUser.name }}.</p>  
    <p>Count is {{ count }}.</p>  
  </div>  
</template>  
  
<script>  
import { mapState } from 'vuex';  
  
export default {  
  computed: mapState({  
    count: state => state.count,  
    loggedInUser: state => state.loggedInUser  
  })  
}</script>
```

Here's an even simpler alternative where we can pass an array of strings to the `mapState` helper function:

```
export default {  
  computed: mapState([  
    'count', 'loggedInUser'  
  ])  
}
```

This version of the code and the one above it do exactly the same thing. You should note that `mapState` returns an object. If you want to use it with other computed properties, you can use the spread operator. Here's how:

```
computed: {  
  ...mapState([  
    'count', 'loggedInUser'  
  ]),  
  parity: function() {  
    return this.count % 2 === 0 ? 'even' : 'odd'  
  }  
}
```

Getters

In a Vuex store, getters are the equivalent to Vue's computed properties. They allow you to create *derived state* that can be shared between different components. Here's a quick example:

```
getters: {  
  depletedProducts: state => {  
    return state.products.filter(product => product.stock <= 0)  
  }  
}
```

Results of getter handlers (when accessed as properties) are cached and can be called as many times as you wish. They're also reactive to state changes. In other words, if the state it depends upon changes, the getter function is automatically executed and the new result is cached. Any component that has accessed a getter handler will get updated instantly. This is how you can access a

getter handler from a component:

```
computed: {
  depletedProducts() {
    return this.$store.getters.depletedProducts;
  }
}
```

The mapGetters Helper

You can simplify the getters code by using the mapGetters helper:

```
import { mapGetters } from 'vuex'

export default {
  //..
  computed: {
    ...mapGetters([
      'depletedProducts',
      'anotherGetter'
    ])
  }
}
```

There's an option for passing arguments to a getter handler by returning a function. This is useful if you want to perform a query within the getter:

```
getters: {
  getProductById: state => id => {
    return state.products.find(product => product.id === id);
  }
}

store.getters.getProductById(5)
```

Do note that each time a getter handler is accessed via a method, it will always run and the result won't be cached.

Compare:

```
// property notation, result cached
store.getters.depletedProducts

// method notation, result not cached
store.getters.getProductById(5)
```

Changing State with Mutations

An important aspect of the Vuex architecture is that components never alter the state directly. Doing so can lead to odd bugs and inconsistencies in the app's state.

Instead, the way to change state in a Vuex store is by committing a *mutation*. For those of you familiar with Redux, these are similar to *reducers*.

Here is an example of a mutation that increases a count variable stored in state:

```
export default new Vuex.Store({
  state: {
    count: 1
  },
  mutations: {
    increment(state) {
      state.count++
    }
  }
})
```



```
}}
```

You can't call a mutation handler directly. Instead, you trigger one by "committing a mutation" like this:

```
methods: {
  updateCount() {
    this.$store.commit('increment');
  }
}
```

You can also pass parameters to a mutation:

```
// store.js
mutations: {
  incrementBy(state, n) {
    state.count += n;
  }
}

// component
updateCount() {
  this.$store.commit('incrementBy', 25);
}
```

In the above example, we're passing the mutation an integer by which it should increase the count. You can also pass an object as a parameter. This way, you can include multiple fields easily without overloading your mutation handler:

```
// store.js
mutations: {
  incrementBy(state, payload) {
    state.count += payload.amount;
  }
}

// component
updateCount() {
  this.$store.commit('incrementBy', { amount: 25 });
}
```

You can also perform an *object-style commit* that looks like this:

```
store.commit({
  type: 'incrementBy',
  amount: 25
})
```

The mutation handler will remain the same.

The mapMutations Helper

Similar to `mapState` and `mapGetters`, you can also use the `mapMutations` helper to reduce the boilerplate for your mutation handlers:

```
import { mapMutations } from 'vuex'

export default{
  methods: {
    ...mapMutations([
      'increment', // maps to this.increment()
      'incrementBy' // maps to this.incrementBy(amount)
    ])
  }
}
```

On a final note, mutation handlers must be synchronous. You can attempt to write an asynchronous mutation function, but you'll come

to find out later down the road that it causes unnecessary complications. Let's move on to actions.

Actions

Actions are functions that don't change the state themselves. Instead, they commit mutations after performing some logic (which is often asynchronous). Here's a simple example of an action:

```
//..  
actions: {  
  increment(context) {  
    context.commit('increment');  
  }  
}
```

Action handlers receive a context object as their first argument, which gives us access to store properties and methods. For example:

- `context.commit`: commit a mutation
- `context.state`: access state
- `context.getters`: access getters

You can also use *argument destructuring* to extract the store attributes you need for your code. For example:

```
actions: {  
  increment({ commit }) {  
    commit('increment');  
  }  
}
```

As mentioned above, actions can be asynchronous. Here's an example:

```
actions: {  
  incrementAsync: async({ commit }) => {  
    return await setTimeout(() => { commit('increment') }, 1000);  
  }  
}
```

In this example, the mutation is committed after 1,000 milliseconds.

Like mutations, action handlers aren't called directly, but rather via a dedicated `dispatch` method on the store, like so:

```
store.dispatch('incrementAsync')  
  
// dispatch with payload  
store.dispatch('incrementBy', { amount: 25})  
  
// dispatch with object  
store.dispatch({  
  type: 'incrementBy',  
  amount: 25  
})
```

You can dispatch an action in a component like this:

```
this.$store.dispatch('increment')
```

The `mapActions` Helper

Alternatively, you can use the `mapActions` helper to assign action handlers to local methods:

```
import { mapActions } from 'vuex'  
  
export default {  
  //..  
  methods: {  
    ...mapActions([
```

```

    'incrementBy', // maps this.increment(amount) to this.$store.dispatch(increment)
    'incrementAsync', // maps this.incrementAsync() to this.$store.dispatch(incrementAsync)
    add: 'increment' // maps this.add() to this.$store.dispatch(increment)
  })
}
}

```

Re-build Counter App Using Vuex

Now that we've had a look at the core concepts of Vuex, it's time to implement what we've learned and rewrite our counter to make use of Vue's official state management solution.

If you fancy a challenge, you might like to have a go at doing this yourself before reading on ...

When we generated our project using `Vue CLI`, we selected `Vuex` as one of the features. A couple of things happened:

1. `Vuex` was installed as a package dependency. Check your `package.json` to confirm this.
2. A `store.js` file was created and injected into your `Vue.js` application via `main.js`.

To convert our “local state” counter app to a `Vuex` application, open `src/store.js` and update the code as follows:

```

import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)

export default new Vuex.Store({
  state: {
    count: 0
  },
  getters: {
    parity: state => state.count % 2 === 0 ? 'even' : 'odd'
  },
  mutations: {
    increment(state) {
      state.count++;
    },
    decrement(state) {
      state.count--;
    }
  },
  actions: {
    increment: ({ commit }) => commit('increment'),
    decrement: ({ commit }) => commit('decrement'),
    incrementIfOdd: ({ commit, getters }) => getters.parity === 'odd' ? commit('increment') : false,
    incrementAsync: ({ commit }) => {
      setTimeout(() => { commit('increment') }, 1000);
    }
  }
});

```

Here we can see how a complete `Vuex` store is structured in practice. Please go back over the theory part of this article if anything here is unclear to you.

Next, update the `src/components/Counter.vue` component by replacing the existing code within the `<script>` block. We'll switch the local state and functions to the newly created ones in the `Vuex` store:

```

import { mapState, mapGetters, mapActions } from 'vuex'

export default {
  name: 'Counter',
  computed: {
    ...mapState([

```

```

    'count'
  ]),
  ...mapGetters([
    'parity'
  ])
},
methods: mapActions([
  'increment',
  'decrement',
  'incrementIfOdd',
  'incrementAsync'
])
}

```

The template code should remain the same, as we're sticking to the previous variable and function names. See how much cleaner the code now is.

If you don't want to use the state and getter map helpers, you can access the store data directly from your template like this:

```

<p>
  Clicked {{ $store.state.count }} times! Count is {{ $store.getters.parity }}.
</p>

```

After you've saved your changes, make sure to test your application. From an end-user perspective, the counter application should function exactly the same as before. The only difference is that the counter is now operating from a Vuex store.

Live Code

See the [Pen Vue Counter Using Vuex](#).

Conclusion

In this article, we've looked at what Vuex is, what problem it solves, how to install it, as well as its core concepts. We then applied these concepts to refactor our counter app to work with Vuex. Hopefully this introduction will serve you well in implementing Vuex in your own projects.

It should also be noted that using Vuex in such a simple app is total overkill. However, in another article—*Build a Shopping List App with Vue, Vuex and Bootstrap Vue*—I'll build a more complicated app to demonstrate a more real-world scenario, as well as some of Vuex's more advanced features.

Chapter 3: A Beginner's Guide to Vue CLI

by Ahmed Bouchefra

When building a new Vue app, the best way to get up and running quickly is to use 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.

The most recent version of Vue CLI is version 3. It provides a new experience for Vue developers and helps them start developing Vue apps without dealing with the complex configuration of tools like webpack. At the same time, it can be configured and extended with plugins for advanced use cases.

Vue CLI v3 is a complete system for rapid Vue.js development and prototyping. It's composed of different components, such as the CLI service, CLI plugins and recently a web UI that allows developers to perform tasks via an easy-to-use interface.

Throughout this article, I'll introduce the latest version of Vue CLI and its new features. I'll demonstrate how to install the latest version of Vue CLI and how to create, serve and build an example project.

Vue CLI v3 Installation and Requirements

In this section, we'll look at the requirements needed for Vue CLI v3 and how to install it.

Requirements

Let's start with the requirements. Vue CLI v3 requires Node.js 8.9+, but v8.11.0+ is recommended.

You can install the latest version of Node.js in various ways:

- By downloading the binaries for your system from the official website.
- By using the official package manager for your system.
- Using a version manager. This is probably the easiest way, as it allows you to manage multiple versions of Node on the same machine. If you'd like to find

out more about this approach, please see our quick tip [Installing Multiple Versions of Node.js Using nvm](#).

Vue creator, Evan You, described version 3 of the CLI as a “completely different beast” from its predecessor. As such, it’s important to uninstall any previous version of the CLI (that is, 2.x.x) before proceeding with this tutorial.

If the `vue-cli` package is installed globally on your system, you can remove it by running the following command:

```
npm uninstall vue-cli -g
```

Installing Vue CLI v3

You can now install Vue CLI v3 by simply running the following command from your terminal:

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

Fixing Permissions

If you find yourself needing to add `sudo` before your command in macOS or Debian-based systems, or to use an administrator CMD prompt in Windows in order to install packages globally, then you should fix your permissions. The npm site has a guide on how to do this, or just use a version manager and you avoid the problem completely.

After successfully installing the CLI, you’ll be able to access the `vue` executable in your terminal.

For example, you can list all the available commands by executing the `vue` command:

```
vue
```

You can check the version you have installed by running:

```
vue --version  
$ 3.2.1
```

Creating a Vue Project

After installing Vue CLI, let’s now look at how we can use it to quickly scaffold

complete Vue projects with a modern front-end toolset.

Using Vue CLI, you can create or generate a new Vue app by running the following command in your terminal:

```
vue create example-vue-project
```

Naming a Project

`example-vue-project` is the name of the project. You can obviously choose any valid name for your project.

The CLI will prompt you for the preset you want to use for your project. One option is to select the default preset which installs two plugins: Babel for transpiling modern JavaScript, and ESLint for ensuring code quality. Or you can manually select the features needed for your project from a set of official plugins. These include:

- Babel
- TypeScript
- Progressive Web App support
- Vue Router
- Vuex (Vue's official state management library)
- CSS Pre-processors (PostCSS, CSS modules, Sass, Less & Stylus)
- Linter/ Formatter using ESLint and Prettier
- Unit Testing using Mocha or Jest
- E2E Testing using Cypress or Nightwatch

Whatever you choose, the CLI will download the appropriate libraries and configure the project to use them. And if you choose to manually select features, at the end of the prompts you'll also have the option to save your selections as a preset so that you can reuse it in future projects.

Now let's look at the other scripts for serving the project (using a webpack development server and hot module reloading) and building the project for production.

Navigate inside your project's folder:

```
cd example-vue-project
```

Next, run the following command to serve your project locally:

```
npm run serve
```

The command will allow you to run a local development server from the `http://localhost:8080` address. If you use your web browser to navigate to this address, you should see the following page:



Welcome to Your Vue.js App

For a guide and recipes on how to configure / customize this project, check out the [vue-cli documentation](#).

Installed CLI Plugins

[babel](#) [eslint](#)

Essential Links

[Core Docs](#) [Forum](#) [Community Chat](#) [Twitter](#) [News](#)

Ecosystem

[vue-router](#) [vuex](#) [vue-devtools](#) [vue-loader](#) [awesome-vue](#)

The development server supports features like hot code reloading, which means you don't need to stop and start your server every time you make any changes to your project's source code. It will even preserve the state of your app!

And when you've finished developing your project, you can use the following command to build a production bundle:

```
npm run build
```

This will output everything to a `dist` folder within your project. You can read more about deployment [here](#).

What is the Vue CLI Service?

The Vue CLI Service is a run-time dependency (`@vue/cli-service`) that abstracts webpack and provides default configurations. It can be upgraded, configured and extended with plugins.

It provides multiple scripts for working with Vue projects, such as the `serve`, `build` and `inspect` scripts.

We've seen the `serve` and `build` scripts in action already. The `inspect` script allows you to inspect the webpack config in a project with `vue-cli-service`. Try it out:

```
vue inspect
```

As you can see, that produces a lot of output. Later on we'll see how to tweak the webpack config in a Vue CLI project.

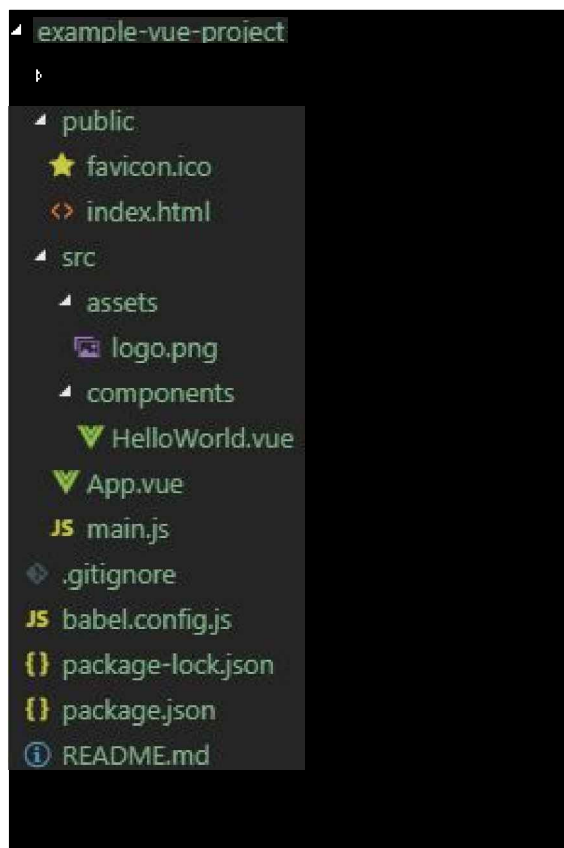
The Project Anatomy

A Vue project generated with the CLI has a predefined structure that adheres to best practices. If you choose to install any extra plugins (such as the Vue router), the CLI will also create the files necessary to use and configure these libraries.

Let's take a look at the important files and folders in a Vue project when using the default preset.

- `public`. This folder contains public files like `index.html` and `favicon.ico`. Any static assets placed here will simply be copied and not go through webpack.
- `src`. This folder contains the source files for your project. Most work will be done here.
- `src/assets`. This folder contains the project's assets such as `logo.png`.
- `src/components`. This folder contains the Vue components.
- `src/App.vue`. This is the main Vue component of the project.
- `src/main.js`. This is the main project file which bootstraps the Vue application.
- `babel.config.js`. This is a configuration file for Babel.
- `package.json`. This file contains a list of the project's dependencies, as well as the configuration options for ESLint, PostCSS and supported browsers.
- `node_modules`. This folder contains the installed npm packages.

This is a screenshot of the project's anatomy:



Vue CLI Plugins

Vue CLI v3 is designed with a plugin architecture in mind. In this section, we'll look at what plugins are and how to install them in your projects. We'll also look at some popular plugins that can help add advanced features by automatically installing the required libraries and making various settings—all of which would otherwise have to be done manually.

What a Vue Plugin Is

CLI Plugins are just npm packages that provide additional features to your Vue project. The `vue-cli-service` binary automatically resolves and loads all plugins listed in the `package.json` file.

The base configuration for a Vue CLI 3 project is webpack and Babel. All the other features can be added via plugins.

There are official plugins provided by the Vue team and community plugins developed by the community. Official plugin names start with `@vue/cli-`

plugin-, and community plugin names start with `vue-cli-plugin-`.

Official Vue CLI 3 plugins include:

- Typescript
- PWA
- Vuex
- Vue Router
- ESLint
- Unit testing etc.

How to Add a Vue Plugin

Plugins are either automatically installed when creating the project or explicitly installed later by the developer.

You can install many built-in plugins in a project when initializing your project, and install any other additional plugins in the project using the `vue add my-plugin` command at any point of your project.

You can also install plugins with presets, and group your favorite plugins as reusable presets that you can use later as the base for other projects.

Some Useful Vue Plugins

There are many Vue CLI plugins that you might find useful for your next projects. For example, the Vuetify UI library is available as a plugin, as is Storybook. You can also use the Electron Builder plugin to quickly scaffold out a Vue project based on Electron.

I've also written a couple of plugins which you can make use of:

- `vue-cli-plugin-nuxt`: a Vue CLI plugin for quickly creating a universal Vue application with Nuxt.js
- `vue-cli-plugin-bootstrap`: a Vue CLI plugin for adding Bootstrap 4 to your project

If you'd like to find out more about plugins, check out this great article on Vue Mastery: [5 Vue CLI 3 plugins for your Vue project](#).

What About webpack?

webpack is abstracted away by the Vue CLI and the different APIs it provides to

access and mutate the webpack configuration.

Most project configuration for Vue CLI is abstracted into plugins and is merged into the base configuration at runtime. But in some situations you might want to manually tweak the webpack configuration for your project. In that case, you can either:

- Create a `vue.config.js` file in your project root and then make any configuration within a `configureWebpack` option:

```
module.exports = {
  configureWebpack: {
    // custom config here
  }
}
```

- Mutate the webpack configuration using tools like `webpack-chain`

You can find out more about working with Vue CLI and webpack [here](#).

Vue CLI UI

Let's now look at the Vue CLI UI, covering how to launch it and the different views used to create and manage projects a graphical user interface.


Vue CLI v3 provides a modern web interface that allows you to create and manage projects without using terminal commands. You can launch the UI as follows:


```
vue ui
```

The UI should be available from the `http://localhost:8000` address.

Vue Project Manager

 Projects

 Create

 Import



No existing projects

You create a new project from the *Create* tab. Browse for the location where you want to create your project, then click on the *+ Create a new project here* button.

Vue Project Manager







 Projects


 Create

 Import

 C: Users ahmed



-  .config
-  .vscode
-  .vue-cli-ui
-  Contacts
-  Desktop
-  Documents
-  Downloads
-  Favorites

 Create a new project here

You'll be taken to a new interface where you need to enter different details about your project such as the name, the project's location, the package manager and whether or not you want to initialize a Git repository.

Create a new project

Details Presets Features Configuration

Project folder

myvueproject

C:/Users/ahmed/myvueproject

Package manager

npm

Additional options

Overwrite target folder if it exists

Git repository

Initialize git repository (recommended)

Initial commit message (optional)

Cancel Next

Enter the details and click on the *Next* button. You'll be taken to the *Presets* tab where you can specify the preset for your project.

You can choose:

- *Default preset* for a default preset with Babel and ESLint plugins
- *Manual* for manually selecting plugins
- *Remote preset* for using a remote preset from a Git repository

Let's continue with the default preset:

Create a new project

Details Presets Features Configuration

A preset is an association of plugins and configurations. After you've selected features, you can optionally save it as a preset so that you can reuse it for future projects, without having to reconfigure everything again.

Select a preset

- Default preset
babel, eslint
- Manual
Manually select features
- Remote preset
Fetch a preset from a git repository

← Previous

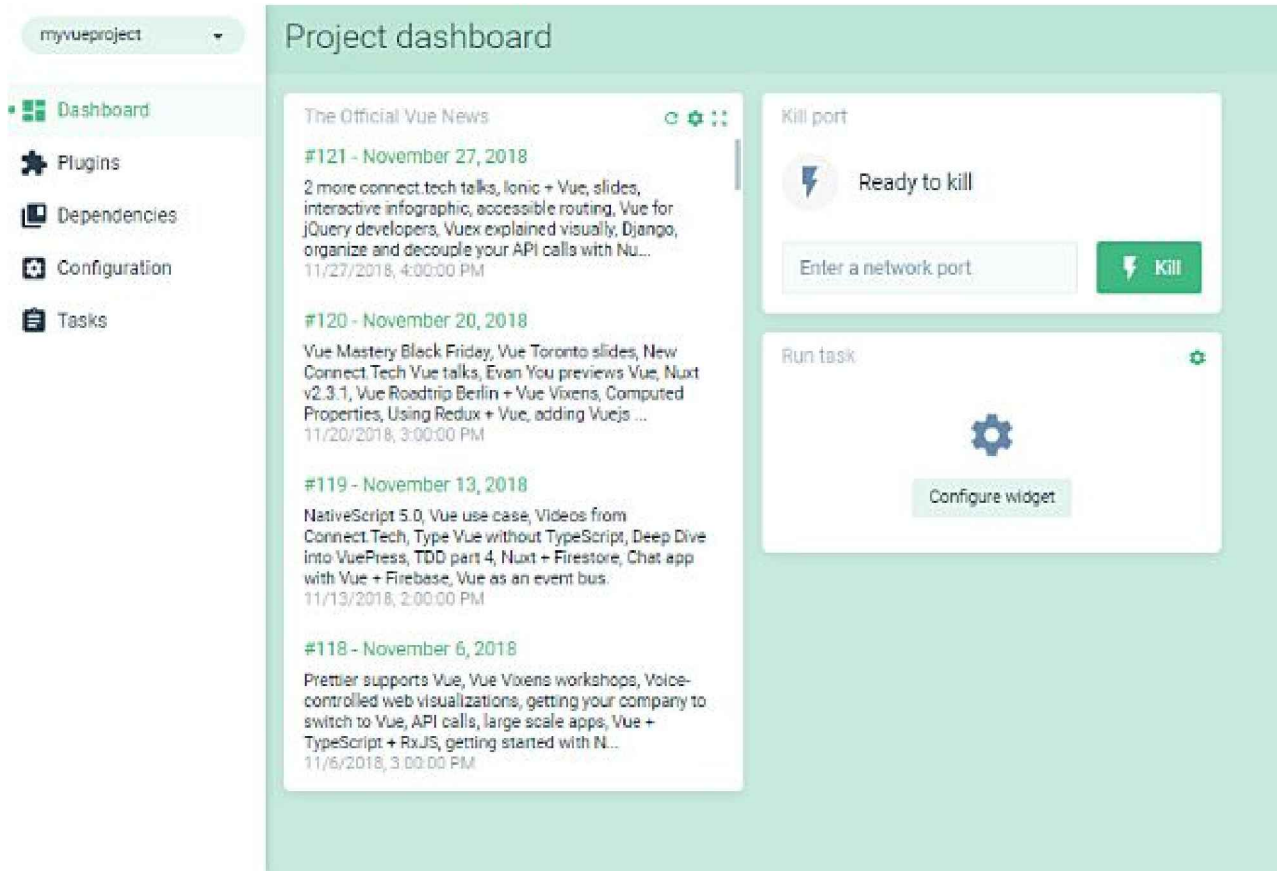
✓ Create Project

Presets

A preset is an association of plugins and configurations.

Next, you can click on the *Create Project* button to start generating your project. You'll be taken to a new interface that shows you the progress of your project generation.

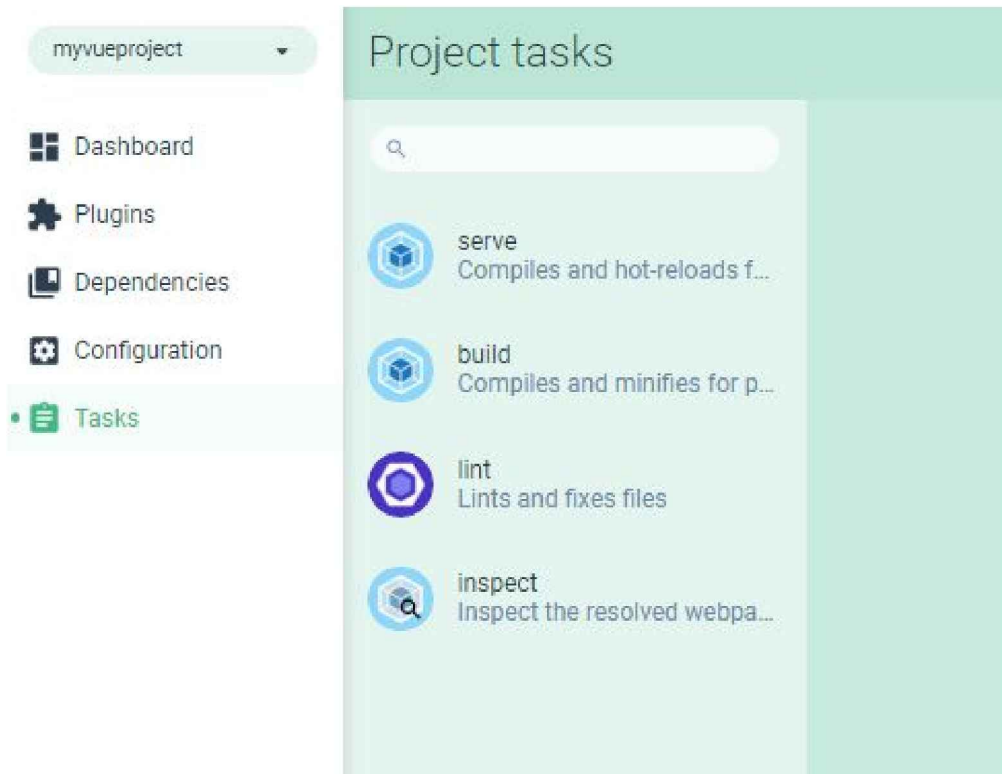
Next, you'll be taken to the project dashboard—where you can put widgets, which you can add using the *Customize* button at the top right of the page, after which they'll be automatically saved.



On the left of the dashboard you can find different pages:

- *Plugins* for adding new Vue CLI plugins
- *Dependencies* for managing the packages
- *Configuration* for configuring the tools
- *Tasks* for running scripts

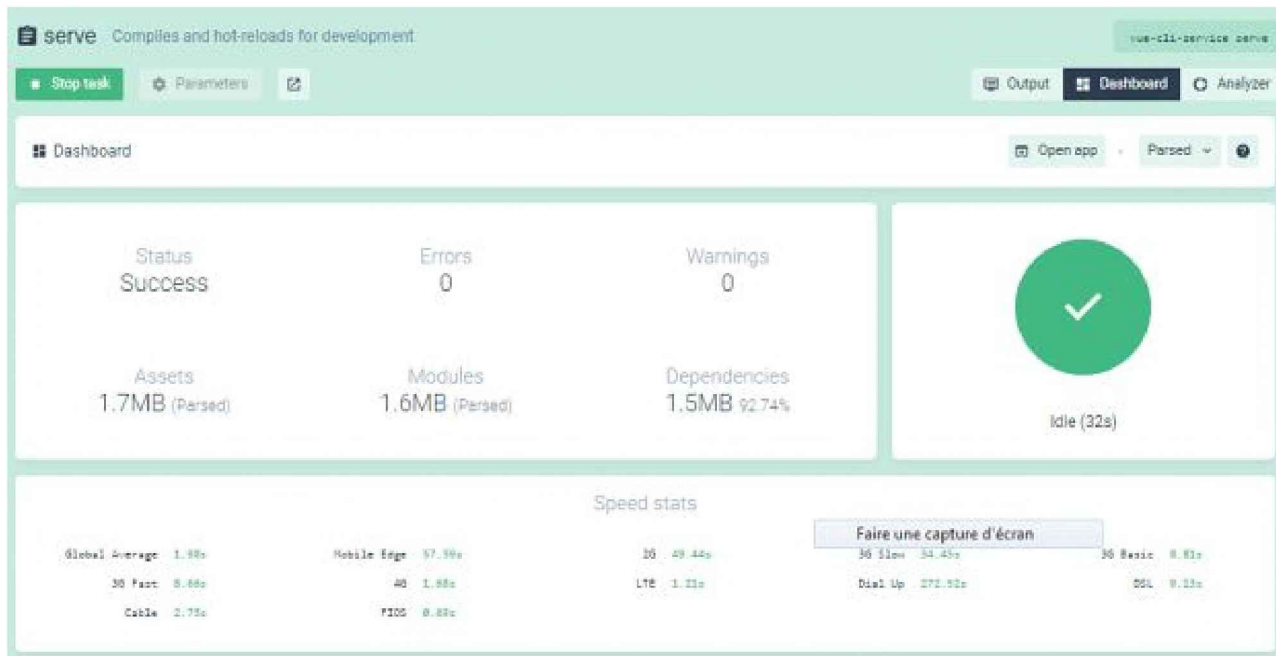
Switch to the *Tasks* page.



Next, click on the *serve* button and then on the *Run task* button to serve your project.



You can stop serving the project using the *Stop task* button. You can also open the application from this interface and see information about the project, such as the size of assets, modules and dependencies, speed statistics and so on.



Conclusion

In this article we've seen an overview of the new Vue CLI version, which provides a whole host of developer-friendly features such as interactive project scaffolding, a rich collection of official plugins integrating the best tools in the front-end ecosystem, and a full graphical user interface to create and manage Vue.js projects.

The CLI is a powerful tool in the hands of Vue developers, but in cases when you don't need all of its features, it might be preferable to use Vue.js in your project without the CLI. You can see how to do this in our tutorial *Getting up and Running with the Vue.js 2.0 Framework*.

Chapter 4: A Beginner's Guide to Working With Components in Vue

by Kingsley Silas

One of the great things about working with Vue is its component-based approach to building user interfaces. This allows you to break your application into smaller, reusable pieces (components) which you can then use to build out a more complicated structure.

In this guide, I'll offer you a high-level introduction to working with components in Vue. I'll look at how to create components, how to pass data between components (via both props and an event bus) and how to use Vue's `<slot>` element to render additional content within a component.

Each example will be accompanied by a runnable CodePen demo.

How to Create Components in Vue

Components are essentially reusable Vue instances with a name. There are various ways to create components within a Vue application. For example, in a small- to medium-sized project you can use the `Vue.component` method to register a global component, like so:

```
Vue.component('my-counter', {
  data() {
    return {
      count: 0
    }
  },
  template: '<div>{{ count }}</div>'
})

new Vue({ el: '#app' })
```

The name of the component is `my-counter`. It can be used like so:

```
<div id="app">
  <my-counter></my-counter>
</div>
```

When naming your component, you can choose kebab case (`my-custom-component`) or Pascal case (`MyCustomComponent`). You can use either variation when referencing your component from within a template, but when referencing it directly in the DOM (as in the example above), *only* the kebab case tag name is valid.

You might also notice that, in the example above, `data` is a function which returns an object literal (as opposed to being an object literal itself). This is so that each instance of the component receives its own `data` object and doesn't have to share one global instance with all other instances.

There are several ways to define a component template. Above we are using a template literal, but we could also use a `<script tag>` marked with `text/x-template` or an in-DOM template. You can read more about the different ways of defining templates here.

Single-file Components

In more complex projects, global components can quickly become unwieldy. In such cases, it makes sense to craft your application to use single-file components. As the name suggests, these are single files with a `.vue` extension, which contain a `<template>`, `<script>` and `<style>` section.

For our example above, an App component might look like this:

```
<template>
  <div id="app">
    <my-counter></my-counter>
  </div>
</template>

<script>
import myCounter from './components/myCounter.vue'

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

<style></style>
```

And a MyCounter component might look like this:

```
<template>
  <div>{{ count }}</div>
</template>

<script>
export default {
  name: 'my-counter',
  data() {
    return {
      count: 0
    }
  }
}
</script>

<style></style>
```

As you can see, when using single-file components, it's possible to import and use these directly within the components where they're needed.

In this guide, I'll present all of the examples using the `Vue.component()` method of registering a component.

Using single-file components generally involves a build step (for example, with Vue CLI). If this is something you'd like to find out more about, please check out "A Beginner's Guide to Vue CLI" in this Vue series.

Passing Data to Components Via Props

Props enable us to pass data from a parent component to child component. This makes it possible for our components to be in smaller chunks to handle specific functionalities. For example, if we have a blog component we might want to display information such as the author's details, post details (title, body and images) and comments.

We can break these into child components, so that each component handles specific data, making the component tree look like this:

```
<BlogPost>
  <AuthorDetails></AuthorDetails>
  <PostDetails></PostDetails>
```

```
<Comments></Comments>
</BlogPost>
```

If you're still not convinced about the benefits of using components, take a moment to realize how useful this kind of composition can be. If you were to revisit this code in the future, it would be immediately obvious how the page is structured and where (that is, in which component) you should look for which functionality. This declarative way of composing an interface also makes it much easier for someone who isn't familiar with a codebase to dive in and become productive quickly.

Since all the data will be passed from the parent component, it can look like this:

```
new Vue({
  el: '#app',
  data() {
    return {
      author: {
        name: 'John Doe',
        email: 'jdoe@example.com'
      }
    }
  }
})
```

In the above component, we have the author details and post information defined. Next, we have to create the child component. Let's call the child component `author-detail`. So our HTML template will look like this:

```
<div id="app">
  <author-detail :owner="author"></author-detail>
</div>
```

We're passing the child component the `author` object as props with the name `owner`. It's important to note the difference here. In the child component, `owner` is the name of the prop with which we receive the data from the parent component. The data we want to receive is called `author`, which we've defined in our parent component.

To have access to this data, we need to declare the props in the `author-detail` component:

```
Vue.component('author-detail', {
  template: `
    <div>
      <h2>{{ owner.name }}</h2>
      <p>{{ owner.email }}</p>
    </div>
  `,
  props: ['owner']
})
```

We can also enable validation when passing props, to make sure the right data is being passed. This is similar to `PropTypes` in React. To enable validation in the above example, change our component to look like this:

```
Vue.component('author-detail', {
  template: `
    <div>
      <h2>{{ owner.name }}</h2>
      <p>{{ owner.email }}</p>
    </div>
  `,
  props: {
    owner: {
```

```
    type: Object,
    required: true
  }
}
})
```

If we pass the wrong prop type, you'll see an error in your console that looks like what I have below:

```
"[Vue warn]: Invalid prop: type check failed for prop 'text'. Expected Boolean, got String.
(found in component <>)"
```

There's an official guide in the Vue docs that you can use to learn about prop validation.

Live Code

See the Pen [Vue Componets - Props](#).

Communicating From a Child to Parent Component via an Event Bus

Events are handled by creating wrapper methods that are triggered when the chosen event takes place. By way of a refresher, let's build on our original counter example, so that it increases each time a button is clicked.

This is what our component should look like:

```
new Vue({
  el: '#app',
  data() {
    return {
      count: 0
    }
  },
  methods: {
    increment() {
      this.count++
    }
  }
})
```

And our template:

```
<div id="app">
  {{ count }}
  <div>
    <button @click="increment"></button>
  </div>
</div>
```

This is hopefully simple enough. As you can see, we're hooking into the `onClick` event to trigger a custom `increase` method whenever the button is clicked. The `increase` method is then incrementing our `count` data property. Now let's expand the example to move the counter button into a separate component and display the count in the parent. We can do this using an event bus.

Event buses come in handy when you want to communicate from a child component to parent component. This is contrary to the default method of communication, which happens from parent to child. You can make use of an event bus if your application isn't big enough to require the use of Vuex. (You can read more about that in "Getting Started with Vuex: a Beginner's Guide" in this Vue series.)

So here's what we want to do: the `count` will be declared in the parent component and passed down to a child component. Then in the child component, we want to increment the value of `count` and also ensure that the value is

updated in the parent component.

The App component will look like this:

```
new Vue({
  el: '#app',
  data() {
    return {
      count: 0
    }
  }
})
```

Then in the child component, we want to receive the count via props and have a method to increment it. We don't want to display the value of count in the child component. We only want to do the increment from the child component and have it reflected in the parent component:

```
Vue.component('counter', {
  template: `
    <div>
      <button @click="increment"></button>
    </div>
  `,
  props: {
    value: {
      type: Number,
      required: true
    }
  },
  methods: {
    increment() {
      this.count++
    }
  }
})
```

Then our template will look like this:

```
<div id="app">
  <h3>
    {{ count }}
  </h3>
  <counter :count="count" />
</div>
```

If you try incrementing the value like that, it won't work. To make it work, we have to emit an event from the child component, send the new value of count and also listen for this event in the parent component.

First, we create a new instance of Vue and set it to eventBus:

```
const eventBus = new Vue();
```

We can now make use of the event bus in our component. The child component will look like this:

```
Vue.component('counter', {
  props: {
    count: {
      type: Number,
      required: true
    }
  }
})
```

```

    },
    methods: {
      increment() {
        this.count++
        EventBus.$emit('count-incremented', this.count)
      }
    },
    template: `
<div>
  <button @click="increment">+</button>
</div>
`
  })

```

The event is emitted each time the `increment` method is called. We have to listen for the event in the main component and then set `count` to the value we obtained through the event that was emitted:

```

new Vue({
  el: '#app',
  data() {
    return {
      count: 0
    }
  },
  created() {
    EventBus.$on('count-incremented', (count) => {
      this.count = count
    })
  }
})

```

Note that we're making use of Vue's `created` lifecycle method to hook into the component before it's mounted and to set up the event bus.

Using an event bus is good if your application isn't complex, but please remember that, as your application grows, you may need to make use of Vuex instead.

Live Code

See the [Pen Vue Components - Event Bus](#)

Nesting Content in Components Using Slots

In all the examples we've seen so far, the components have been self-closing elements. However, in order to make components that can be composed together in useful ways, we need to be able to nest them inside one another as we do with HTML elements.

If you try using a component with a closing tag and putting some content inside, you'll see that Vue just swallows this up. Anything within the component's opening and closing tags is replaced with the rendered output from the component itself:

```

<div id="app">
  <author-detail :owner="author">
    <p>This will be replaced</p>
  </author-detail>
</div>

```

Luckily, Vue's slots make it possible to pass an arbitrary value to a component. This can be anything from DOM elements

from a parent component to a child component. Let's see how they work.

The script part of our components will look like this:

```
Vue.component('list', {
  template: '#list'
})

new Vue({
  el: "#app"
})
```

Then the templates will look like this:

```
<div id="app">
  <h2>Slots</h2>
  <list>
    <h4>I am the first slot</h4>
  </list>
  <list>
    <h4>I am the second slot</h4>
  </list>
</div>

<script type="text/x-template" id="list">
  <div>
    <h3>Child Component</h3>
    <slot></slot>
  </div>
</script>
```

The content inside our `<list>` component gets rendered between the `<slot></slot>` element tag. We can also make use of fallback content, for cases where the parent doesn't inject any.

```
<div id="app">
  <h2>Slots</h2>
  <list>
    <h4>I am the first slot</h4>
  </list>
  <list></list>
</div>

<script type="text/x-template" id="list">
  <div>
    <h3>Child Component</h3>
    <slot>This is fallback content</slot>
  </div>
</script>
```

The fallback content will render in cases where there's no content from the parent component.

Live Code

See the Pen [Vue Components - Slots](#).

Conclusion

This has been a high-level introduction to working with components in Vue. We looked at how to create components in

Vue, how to communicate from a parent to a child component via props and from a child to a parent via an event bus. We then finished off by looking at slots, a handy method for composing components in useful ways. I hope you've found the tutorial useful.

Chapter 5: A Beginner's Guide to Working with Forms in Vue

by Kingsley Silas

Forms are an integral part of modern web applications. They allow us to receive input from users which we can then validate and act upon.

In this tutorial, I'm going to demonstrate how to work with forms in Vue. We'll start off by creating a simple form and look at how to use two-way data binding to keep user input in sync with our data model. We'll then take a look at modifiers, which allow us to change the browser's default form-handling behavior, and filters, which allow us to format the output the user sees. We'll finish off by implementing form validation using the VeeValidate plugin. Each section will be accompanied by a runnable CodePen demo.

By the time you've finished reading, you'll have seen how easy Vue makes it to work with forms, and you'll be able to implement all of these techniques in your own apps.

Let's get to it.

Two-way Data Binding with v-model

Vue provides a two-way binding feature that keeps the inputs entered by your user in sync with the data model. This is made possible by using the `v-model` directive like so: `v-model="entry"`.

In the data model, you'll need to have the following:

```
data() {
  return {
    entry: 3
  }
}
```

Then our template will look like this:

```
<div id="app">
  <h2>Vue Form</h2>
  <h2>{{ entry }}</h2>
  <input type="text" v-model="entry" />
</div>
```

By default, the input field will have a value of 3, which is the initial state for our entry. However, that doesn't stop us from changing the state by entering another value in our input field. The value automatically changes as we do this. Underneath the hood, the `v-model` directive is shorthand for this:

```
<div id="app">
  <h2>Vue Form</h2>
  <h2>Input: {{ entry }}</h2>
  <input type="text"
    v-bind:value="entry"
    v-on:input="entry=$event.target.value"
  />
</div>
```

The `v-model` directive combines the power of `v-bind` and `v-on:input` to provide you with the two-way data binding for form inputs. The above snippet can also be written like this:

```
<div id="app">
  <h2>Vue Form</h2>
  <h2>Input: {{ entry }}</h2>
  <input type="text"
    :value="entry"
    @input="entry=$event.target.value"
  />
</div>
```

Now let's take a look at an example. If you'd like to follow along at home, do one of two things. You could create a new app using Vue CLI

(which you can read about in “A Beginner’s Guide to Vue CLI” in this Vue series). The other option is to use the following template, which includes Vue from a CDN, and which you can open directly in your browser:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.1.3/css/boot
  <style>
    /* Styles here */
  </style>
</head>
<body>
  <!-- Template here -->

  <script src="https://cdnjs.cloudflare.com/ajax/libs/vue/2.5.17/vue.min.js"></script>
  <script>
    // Script here
  </script>
</body>
</html>
```

We’ll start off by adding a template that includes four types of form inputs—namely a text input, a select element, radio buttons and check boxes. You can use `v-model` with all of these inputs to keep their values in sync with a Vue instance’s data model:

```
<div id="app" class="container">
  <h2>Vue Form</h2>

  <div class="output">
    <p>Name: {{ name }}</p>
    <p>Genre: {{ genre}}</p>
    <p>Gender: {{ gender }}</p>
    <p>
      Languages:
      <span v-for="(language, index) in languages" :key="index">
        {{ language }}
        <span v-if="index + 1 < languages.length">, </span>
      </span>
    </p>
    <p>Terms Agreement: {{ agreeToTerms }}</p>
  </div>

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

    <div class="gender">
      <input type="radio" id="male" value="Male" v-model="gender">
      <label for="male">Male</label>
      <br>
      <input type="radio" id="female" value="Female" v-model="gender">
      <label for="female">Female</label>
    </div>

    <div>
      <select v-model="genre">
        <option disabled value="">Please select a genre</option>
        <option>Rap</option>
        <option>Indie Pop</option>
        <option>Soul</option>
```

```

    </select>
  </div>

  <div class="languages">
    <input type="checkbox" id="javascript" value="JavaScript" v-model="languages">
    <label for="javascript">Javascript</label>
    <input type="checkbox" id="ruby" value="Ruby" v-model="languages">
    <label for="ruby">Ruby</label>
    <input type="checkbox" id="python" value="Python" v-model="languages">
    <label for="python">Python</label>
  </div>

  <div>
    <label>
      <input type="checkbox" v-model="agreeToTerms"/>
      Yes, I agree to everything
    </label>
  </div>
</form>
</div>

```

Notice that, at the top of the template, we're doing some basic interpolation to output the values of our form fields. We'll need to declare these as data properties on our Vue instance:

```

new Vue({
  el: '#app',
  data() {
    return {
      name: "John Doe",
      agreeToTerms: false,
      genre: '',
      gender: '',
      languages: []
    }
  }
})

```

We can finish off with some basic styling:

```

h2 {
  margin: 30px 0;
  text-align: center;
  font-size: 3em;
  color: tomato;
}

.output {
  text-align: center;
  margin: 30px;
}

form {
  width: 40%;
  max-width: 350px;
  margin: 0 auto 30px;
  border-radius: 4px;
  border: none;
  border-top: 2px solid tomato;
}

input[type="text"], select {
  background: rgba(255, 255, 255, 0.1);
  border: none;

```

```

font-size: 16px;
height: auto;
margin: 0;
outline: 0;
padding: 10px;
width: 100%;
background-color: #e8eeef;
color: #2E4049;
box-shadow: 0 1px 0 rgba(0, 0, 0, 0.03) inset;
margin-bottom: 20px;
}

label {
  display: inline-block;
  margin: 5px 0 10px 0;
}

input[type="checkbox"] { margin: 0 2px 8px 0; }
input[type="radio"] { margin: 0 2px 0 15px; }
.gender label { margin: 0 0 10px 0; }
.languages label { margin: 0 15px 0 0; }
p { margin-bottom: 3px; }

```

[Live Code](#)

Vue Form

Name: John Doe

Genre:

Gender:

Languages:

Terms Agreement: false

Name:

John Doe

Male

Female

Please select a genre

Javascript

Ruby

Python

Yes, I agree to everything

And here's the demo running on CodePen. Try altering any of the fields. You should see the updated value reflected in the UI.

Altering the Browser's Default Behavior with Modifiers

Events are handled in Vue using the `v-on` directive. However, there are modifiers that allow us to change the behaviors of these events.

Let's say we have a counter that can be increased or decreased depending on the button that's clicked. We can modify the click event to ensure that it's only clicked once, like so:

```
<button class="incrementButton" @click.once="handleIncrement">
```

```
    Increment
  </button>
```

[Live Code](#)

Counter

1



It becomes impossible to increment the counter multiple times. Here's a demo that demonstrates this.

Another modifier you'll use frequently is the `.prevent` modifier—which prevents the default behavior of an event. Let's redo the demo above to include a form.

The template part will look like this:

```
<div id="app">
  <h2>Counter</h2>
  <form>
    <h2>{{ count }}</h2>
    <div class="div_buttons">
      <button class="incrementButton" type="submit">
        Increment
      </button>
    </div>
  </form>
</div>
```

The form will submit each time you click the button, causing the value of `count` to change to 0. To avoid that, we have to add the `.prevent` modifier, like so:

```
<div id="app">
  <h2>Counter</h2>
  <form @submit.prevent="handleIncrement">
    ...
  </form>
</div>
```

The script section should look like this:

```
new Vue({
  el: '#app',
```

```
data() {
  return {
    count: 0
  }
},
methods: {
  handleIncrement() {
    this.count += 1;
  }
}
})
```

That will alter the default behavior of the button. You can also use this if on anchor and form elements.

[Live Code](#)

Counter

4

Increment

Here's a working demo of the technique

Other event modifiers include these:

- `.capture`: This is used for handling an event.
- `.self`: This will be triggered if the event's target is the element itself.
- `.stop`: This will stop the propagating of the event up the DOM tree.

And there are some modifiers you can apply to `v-model` itself:

- `.lazy`: This syncs the input with the data model after a change event. By default, `v-model` syncs after each input event.
- `.number`: This typecasts user input as a number.
- `.trim`: This automatically trims the user input.

Formatting Output with Filters

Filters make it possible for you to transform data. They only change the displayed data, but don't actually transform the data itself. You apply a filter to an expression using the pipe `|` symbol.

To see filters at work, let's create a filter to capitalize text entered through an input:

```
new Vue({
  el: '#app',
  data() {
    return {
      entry: "John"
    }
  }
})
```



```

    },
    filters: {
      capitalize(value) {
        return value.toUpperCase()
      }
    }
  })

```

Our filter takes a parameter and returns the result of calling `.toUpperCase()` on the parameter. This is how we make use of it on our template:

```

<div id="app">
  <h2>Filters</h2>
  <p>Input: {{ entry }}</p>
  <p>Filtered Text: {{ entry | capitalize }}</p>
  <input type="text" v-model="entry" />
</div>

```

Note that the fact that the filter is a function doesn't mean we call it with `capitalize(entry)`. In the template, we display two versions of the input—the transformed text and the original text.

Let's say you're working with an API that returns some posts and displays the date the posts were created. You may want to display a user-friendly date to the users. This is one of the scenarios where filter comes in handy. Using Moment.js, you can create a filter for the date.

Your filter will look like this:

```

date(value) {
  return moment(value).fromNow()
}

```

Nothing special is happening here: we only call moment's `.fromNow()` method on the parameter, which is then returned. To display the filtered date, we use `{{ created_at | date }}`.

[Live Code](#)

Filters

Input: John

Filtered Text: JOHN

Filtered date: 6 months ago

John

See the [Pen Formatting Data With Filters](#).

VeeValidate

VeeValidate simplifies validation for forms in Vue. In this final section, we'll build a registration form and use VeeValidate to implement some basic validation checks.

If you're following along using the CLI, you'll need to install VeeValidate using npm:

```
npm install vee-validate --save
```

You'll also need to register the plugin in main.js:

```
import VeeValidate from "vee-validate";
Vue.use(VeeValidate);
```

Alternatively, you can pull it into your application using a CDN:

```
<script src="https://unpkg.com/vee-validate@latest"></script>
```

You'll then need to amend your template like so:

```
<!DOCTYPE html>
<html lang="en">
<head> ... </head>
<body>
  <!-- Template here -->

  <script src="https://cdnjs.cloudflare.com/ajax/libs/vue/2.5.17/vue.min.js"></script>
  <script src="https://unpkg.com/vee-validate@latest"></script>
  <script>
    Vue.use(VeeValidate);
    new Vue({ .. })
  </script>
</body>
</html>
```

Next up, let's create our form. We're going to have name, gender, phone, email and password fields, which we'll bind to our data model using v-model:

```
<div id="app">
  <h2>Register</h2>

  <form @submit.prevent="register" method="post" class="form">
    <div>
      <label for="name">Name:</label>
      <input id="name" name="name" v-validate="'required'" type="text" v-model="user.name" class="form-control" />
      <span v-show="errors.has('name')" class="text-danger">{{ errors.first('name') }}</span>
    </div>
    <div>
      <label for="gender">Gender:</label>
      <select id="gender" name="gender" v-model="user.gender" class="form-control" v-validate="'required'">
        <option>Female</option>
        <option>Male</option>
      </select>
      <span v-show="errors.has('gender')" class="text-danger">
        {{ errors.first('gender') }}
      </span>
    </div>
    <div>
      <label for="phone">Phone:</label>
      <input id="phone" name="phone" v-validate="'required'" type="text" v-model="user.phone" class="form-control" />
      <span v-show="errors.has('phone')" class="text-danger">
        {{ errors.first('phone') }}
      </span>
    </div>
    <div>
      <label for="email">Email:</label>
      <input id="email" name="email" v-validate="'required|email'" type="text" v-model="user.email" class="form-control" />
      <span v-show="errors.has('email')" class="text-danger">
        {{ errors.first('email') }}
      </span>
    </div>
    <div>
      <label for="password">Password:</label>
```

```

    <input id="password" name="password" v-validate="'required'" type="password" v-model="user.password"
    <span v-show="errors.has('password')" class="text-danger">
      {{ errors.first('password') }}
    </span>
  </div>
</div>
<div>
  <button class="btn btn-lg btn-primary btn-block" type="submit">Submit</button>
</div>
</form>
</div>

```

To ensure that the required fields are filled, we pass the value required to a `v-validate` attribute in our template. This signifies that a value must be provided for the input. We won't do much more than that in our example, but please be aware that VeeValidate comes with a bunch of validation rules out of the box, which are all localized and cover most validation needs. And if you don't find what you're looking for there, it's possible to create custom rules, too.

Once the form is submitted, we prevent the browser's default submit action using the `.prevent` modifier. Then we call a custom register method. Any errors from the form will be in the `errors` object, and we can use one of the built-in helper methods to display each error for a particular field.

Next, let's implement the form's logic:

```
Vue.use(VeeValidate);
```

```

new Vue({
  el: '#app',
  data() {
    return {
      user: {
        email: '',
        password: '',
        name: '',
        phone: '',
        gender: ''
      }
    }
  },
  methods: {
    register() {
      this.$validator.validateAll().then((result) => {
        if (result) {
          alert('Form Submitted!');
          return;
        }

        alert('Form is invalid!');
      });
    }
  }
})

```

Within our `register` method, the `validateAll` method is used to validate each value against the corresponding field validations. We're also displaying two different alerts depending on the validity of the form. In your application, you'll want to replace the first alert with your POST request to the server.

And finally, the styles:

```

h2 {
  margin: 20px 0;
  text-align: center;
}
.form {
  max-width: 380px;
  padding: 15px 35px 45px;
}

```

```
margin: 0 auto;
background-color: #fff;
border: 1px solid rgba(0,0,0,0.1);
}
label {
margin: 12px 0 2px 0;
}
button {
margin: 15px 0 0 0;
}
```

[Live Code](#)

Register

Name:



Gender:



Phone:

Email:

Password:



Submit

See the Pen vue form - VeeValidate.

Conclusion

In this guide, I've demonstrated how easy it can be to work with forms in Vue. We looked at how to use two-way data binding to keep user input in sync with a data model, how to change the browser's default form-handling behavior, and how to implement form validation using the VeeValidate plugin.

At this point, you can go on to build simple to complicated forms in Vue, without the need to write long lines of code to handle validation.

Chapter 6: How to Conditionally Apply a CSS Class in Vue.js

by Chad Campbell

There are times you need to change an element's CSS classes at runtime. But when changing classes, it's sometimes best to apply style details conditionally. For example, imagine your view has a pager. Pagers are often used to navigate larger sets of items. When navigating, it can be helpful to show the user the page they're currently on. The style of the item is conditionally set, based on the current page that's being viewed.

A pager in this case may look something like this:



In this example, there are five pages. Only one of these pages is selected at a time. If you built this pager with Bootstrap, the selected page would have a CSS class named `active` applied. You'd want this class applied only if the page was the currently viewed page. In other words, you'd want to *conditionally* apply the `active` CSS class. Luckily, Vue provides a way to conditionally apply a CSS class to an element, which I'm going to demonstrate in this article.

To conditionally apply a CSS class at runtime, you can bind to a JavaScript object. To successfully complete this task, you must complete two steps. First, you must ensure that your CSS class is defined. Then, you create the class bindings in your template. I'm going to explain each of these steps in detail in the rest of this article.

Step 1: Define Your CSS Classes

Imagine, for a moment, that the five page items shown in the image above were defined using the following HTML:

```
<div id="myApp">
  <nav aria-label="Page navigation example">
    <ul class="pagination">
      <li class="page-item"><a class="page-link" href="#">1</a></li>
      <li class="page-item"><a class="page-link" href="#">2</a></li>
      <li class="page-item active"><a class="page-link" href="#">3</a></li>
      <li class="page-item"><a class="page-link" href="#">4</a></li>
      <li class="page-item"><a class="page-link" href="#">5</a></li>
    </ul>
  </nav>
</div>
```

Notice that each page in this code snippet has a list-item element (`<li ...>`). That element references the `page-item` CSS class. In the code for this article, this class is defined in the Bootstrap CSS framework. However, if it weren't defined there, it would be your responsibility to ensure that it was defined somewhere. The second CSS class is the one that's most relevant to this article, though.

The `active` CSS class is used to identify the currently selected page. For this article, this CSS class is also defined in the Bootstrap CSS. As shown in the snippet above, the `active` class is only used in the *third* list item element. As you can probably guess, this is the CSS class that you want to apply conditionally. To do that, you need to add a JavaScript object.

Step 2: Create Your Class Bindings

Let's build on the code snippet shown in step 1. When creating class bindings in your template, there are two primary choices: using the object syntax or using the array syntax. I'm going to show you how to use both approaches in the remainder of this article.

Binding using object syntax

To create a class binding using the object syntax, you have to use a JavaScript **expression**. The expression we'll be using can be seen in the code associated with this article here. That relevant code looks like this:

```
<div id="myApp">
  <nav aria-label="An example with pagination">
    <ul class="pagination">
      <li v-for="page in totalPages" v-bind:class="{ 'page-item': true, 'active': (page === currentPage) }">
```

```

        <a class="page-link" href="#">{{ page }}</a>
    </li>
</ul>
</nav>
</div>

```

I've reduced the amount of code by using Vue's baked-in `v-for` directive. This directive is used to render items in a loop. The items in this example are the pages themselves. Beyond the use of the `v-for` directive, notice the use of the `v-bind` directive.

The `v-bind` directive connects the element's `class` attribute to the Vue instance. That instance of Vue is defined like this:

```

var app = new Vue({
  el: '#myApp',
  data: {
    totalPages: 5,
    currentPage: 3
  }
});

```

This Vue instance is straight to the point. The `data` object above includes a property named `currentPage`. If you revisit the HTML template defined above, you'll notice that this property is being referenced. In fact, the JavaScript object associated with each class binding looks something like this:

```
{'page-item':true, 'active':(page === currentPage)}
```

This object defines two properties: `page-item` and `active`. Notably, these are the names of the two CSS classes discussed in Step 1. In Step 2, these two class references have become property names in a JavaScript object. The values associated with these property names are JavaScript expressions. If the expression evaluates as truthy, the CSS class will be included. If the expression evaluates to `false`, the CSS class will *not* be included. With these rules in mind, let's look at each property.

The first property, `page-item`, has a value of `true`. This hard-coded value is used because we always want to include the `page-item` class. The second property, `active`, uses a JavaScript expression. When this expression is `true`, the `active` class will be applied. This empowers us to conditionally apply the `active` class based on the value of `currentPage`. Another way to conditionally apply the `active` class is by binding to an `Array`.

Binding using array syntax

Vue lets you apply a list of CSS classes by binding to an `Array`. If you wanted to use the `Array` syntax, the HTML shown in Step 1 would become this:

```

<div id="myApp">
  <nav aria-label="An example with pagination">
    <ul class="pagination">
      <li v-for="page in totalPages" v-bind:class="[pageItemClass, (page === currentPage) ? activeClass]">
        <a class="page-link" href="#">{{ page }}</a>
      </li>
    </ul>
  </nav>
</div>

```

A running version with the `Array` syntax can be seen [here](#). The only difference is the use of an `Array` on the class binding. This alternative approach expects two additional properties to exist in your Vue's `data` object. Those two properties are: `pageItemClass` and `activeClass`. The updated Vue initialization code with these properties looks like this:

```

var app = new Vue({
  el: '#myApp',
  data: {
    totalPages: 5,
    currentPage: 3,
    pageItemClass: 'page-item',
    activeClass: 'active'
  }
});

```

As you can see, the `data` object has grown in size, but the code in the template is slightly cleaner when using the `Array` syntax. The

object syntax is a little bit more compact. The choice between the object syntax and the `Array` syntax comes down to personal preference.

Both approaches may seem to make your HTML template more complicated. However, there's actually more going on here. In reality, we're separating concerns. We're creating a template that is driven by data. This makes the view easier to test and easier to maintain as the app grows.

Chapter 7: How to Replace jQuery with Vue

by Nilson Jacques

I'm willing to bet that there are a lot of developers out there who still reach for jQuery when tasked with building simple apps. There are often times when we need to add some interactivity to a page, but reaching for a JavaScript framework seems like overkill – with all the extra kilobytes, the boilerplate, the build tools and module bundlers. Including jQuery from a CDN seems like a no-brainer.

In this article, I'd like to take a shot at convincing you that using Vue.js (referred to as Vue from here on), even for relatively basic projects, doesn't have to be a headache, and will help you write better code faster. We'll take a simple example, code it up in jQuery, and then recreate it in Vue step by step.

What We're Building

For this article, we're going to be building a basic online invoice, using this open-source template from Sparksuite. Hopefully, this should make a refreshing change from yet another to-do list, and provide enough complexity to demonstrate the advantages of using something like Vue while still being easy to follow.



Invoice #: 123
Created: January 1, 2015
Due: February 1, 2015

Sparksuite, Inc.
12345 Sunny Road
Sunnyville, TX 12345

Acme Corp.
John Doe
john@example.com

Payment Method	Check #
Check	1000
Item	Price
Website design	\$300.00
Hosting (3 months)	\$75.00
Domain name (1 year)	\$10.00
Total: \$385.00	

We're going to make this interactive by providing item, unit price, and quantity inputs, and having the *Price* column automatically recalculated when one of the values changes. We'll also add a button, to insert new empty rows into the invoice, and a *Total* field that will automatically update as we edit the data.

I've modified the template so that the HTML for a single (empty) row now looks like this:

```
<tr class="item">
  <td><input value="" /></td>
  <td><input type="number" value="0" /></td>
  <td><input type="number" value="1" /></td>
  <td>$0.00</td>
</tr>
```


jQuery

So, first of all, let's take a look at how we might do this with jQuery.

```
$('#table').on('mouseup keyup', 'input[type=number]', calculateTotals);
```

We're attaching a listener to the table itself, which will execute the `calculateTotals` function when either the *Unit Cost* or *Quantity* values are changed:

```
function calculateTotals() {
  const subtotals = $('.item').map((idx, val) => calculateSubtotal(val)).get();
  const total = subtotals.reduce((a, v) => a + Number(v), 0);
  $('.total td:eq(1)').text(formatAsCurrency(total));
}
```

This function looks for all item rows in the table and loops over them, passing each row to a `calculateSubtotal` function, and then summing the results. This total is then inserted into the relevant spot on the invoice.

```
function calculateSubtotal(row) {
  const $row = $(row);
  const inputs = $row.find('input');
  const subtotal = inputs[1].value * inputs[2].value;

  $row.find('td:last').text(formatAsCurrency(subtotal));

  return subtotal;
}
```

In the code above, we're grabbing a reference to all the `<input>`s in the row and multiplying the second and third together to get the subtotal. This value is then inserted into the last cell in the row.

```
function formatAsCurrency(amount) {
  return `$$${Number(amount).toFixed(2)}`;
}
```

We've also got a little helper function that we use to make sure both the subtotals and the total are formatted to two decimal places and prefixed with a currency symbol.

```
$('#btn-add-row').on('click', () => {
  const $lastRow = $('.item:last');
  const $newRow = $lastRow.clone();

  $newRow.find('input').val('');
  $newRow.find('td:last').text('$0.00');
  $newRow.insertAfter($lastRow);

  $newRow.find('input:first').focus();
});
```

Lastly, we have a click handler for our *Add row* button. What we're doing here is selecting the last item row and creating a duplicate. The inputs of the cloned row are set to default values, and it's inserted as the new last row. We can also be nice to our users and set the focus to the first input, ready for them to start typing.

[Live Code](#)

See the Pen jQuery Invoice.

Downsides

So what's wrong with this code as it stands, or rather, what could be better?

You may have heard some of these newer libraries, like Vue and React, claim to be declarative rather than imperative. Certainly looking at this jQuery code, the majority of it reads as a list of instructions on how to manipulate the DOM. The purpose of each section of code — the “what” — is often hard to make out through the details of “how” it's being done. Sure, we can clarify the intent of the code by breaking it up into well-named functions, but this code is still going to take some effort to mentally parse if you come back to it after a while.

The other issue with code like this is that we're keeping our application state in the DOM itself. Information about the items ordered exists only as part of the HTML making up the UI. This might not seem like a big problem when we're only displaying the information in a single location, but as soon as we start needing to display the same data in multiple places in our app, it becomes increasingly complex to ensure that each piece is kept in sync. There's no single source of truth.

Although nothing about jQuery prevents us from keeping our state outside the DOM and avoiding these problems, libraries such as Vue provide functionality and structure that facilitate creating a good architecture and writing cleaner, more modular code.

Converting to Vue

So how would we go about recreating this functionality using Vue?

As I mentioned earlier, Vue doesn't require us to use a module bundler, or a transpiler, or to opt in to their single file components (.vue files) in order to get started. Like jQuery, we can simply include the library from a CDN. Let's start by swapping out the script tag:

```
<script src="https://cdn.jsdelivr.net/npm/vue@2.5.17/dist/vue.js"></script>
```

The next thing we need to do is create a new Vue instance:

```
const app = new Vue({
  el: 'table'
});
```

The only option we need to provide here is `el`, which is a selector (like we would use with jQuery) identifying which part of the document we want Vue to manage.

We can put Vue in charge of anything from the entire page (for a single page application, for example) or a single `<div>`. For our invoice example, we'll give Vue control of the HTML table.

Data

Let's also add the data for the three example rows to our Vue instance:

```
const app = new Vue({
  el: 'table',
  data: {
    items: [
      { description: 'Website design', quantity: 1, price: 300 },
      { description: 'Hosting (3 months)', quantity: 1, price: 75 },
      { description: 'Domain name (1 year)', quantity: 1, price: 10 },
    ]
  }
});
```

```
    }  
  });
```

The `data` property is where we store the state of our application. This includes not only any data we want our app to work with, but also information about the state of the UI (for example, which section is currently active in a tab group, or whether an accordion is expanded or contracted).

Vue encourages us to keep our app's state separate from its presentation (that is, the DOM) and centralized in one place — a single source of truth.

Modifying the template

Now let's set up our template to display the items from our data object. As we've told Vue we want it to control the table, we can use its template syntax in the HTML to tell Vue how to render and manipulate it.

Using the `v-for` attribute, we can render a block of HTML for each item in our `items` array:

```
<tr class="item" v-for="item in items">  
  
</tr>
```

Vue will repeat this markup for each element of the array (or object) that you pass to the `v-for` construct, allowing you to reference each element inside the loop — in this case, as `item`. As Vue is observing all the properties of the data object, it will dynamically re-render the markup as the contents of `items` change. All we have to do is add or remove items to our app state, and Vue takes care of updating the UI.

We'll also need to add `<input>`s for the user to fill out the description, unit price, and quantity of the item:

```
<td><input v-model="item.description" /></td>  
<td>${<input type="number" v-model="item.price" /></td>  
<td><input type="number" v-model="item.quantity" /></td>  
<td>${{ item.price * item.quantity }}</td>
```

Here we're using the `v-model` attribute to set up a two-way binding between the inputs and properties on our data model. This means any change to the inputs will update the corresponding properties on the item model, and vice versa.

In the last cell, we're using double curly braces `{{ }}` to output some text. We can use any valid JavaScript expression within the braces, so we're multiplying two of our item properties together and outputting the result. Again, as Vue is observing our data model, a change to either property will cause the expression to be re-evaluated automatically.

Events and methods

Now we have our template set up to render out our `items` collection, but how do we go about adding new rows? As Vue will render whatever is in `items`, to render an empty row we just need to push an object with whatever default values we want into the `items` array.

To create functions that we can access from within our template, we need to pass them to our Vue instance as properties of a `methods` object:

```
const app = new Vue({  
  // ...  
  methods: {  
    myMethod() {}  
  }  
})
```

```
  },  
  // ...  
})
```

Let's define an `addRow` method that we can call to add a new item to our `items` array:

```
methods: {  
  addRow() {  
    this.items.push({ description: '', quantity: 1, price: 0 });  
  },  
},
```

Note that any methods we create are automatically bound to the Vue instance itself, so we can access properties from our `data` object, and other methods, as properties of `this`.

So, now that we have our method, how do we call it when the *Add row* button is clicked? The syntax for adding event listeners to an element in the template is `v-on:event-name`:

```
<button class="btn-add-row" @click="addRow">Add row</button>
```

Vue also provides a shortcut for us so we can use `@` in place of `v-on:`, as I've shown in the code above. For the handler, we can specify any method from our Vue instance.

Computed properties

Now all we need to do is display the grand total at the bottom of the invoice. Potentially we could do this within the template itself: as I mentioned earlier, Vue allows us to put any JavaScript statement between curly brackets. However, it's much better to keep anything more than very basic logic out of our templates; it's cleaner and easier to test if we keep that logic separate.

We could use another method for this, but I think a computed property is a better fit. Similar to creating methods, we pass our Vue instance a `computed` object containing functions whose results we want to use in our template:

```
const app = new Vue({  
  // ...  
  computed: {  
    total() {  
      return this.items.reduce((acc, item) => acc + (item.price * item.quantity), 0);  
    }  
  }  
});
```

Now we can reference this computed property within our template:

```
<tr class="total">  
  <td colspan="3"></td>  
  <td>Total: ${{ total }}</td>  
</tr>
```

As you might already have noticed, computed properties can be treated as if they were data; we don't have to call them with parentheses. But using computed properties has another benefit: Vue is smart enough to cache the returned value and only re-evaluate the function if one of the data properties it depends upon changes.

If we were using a method to sum up the grand total, the calculation would be performed each and every time the template was re-rendered. Because we're using a computed property, the total is only recalculated if one of the

item's quantity or price fields are changed.

Filters

You might have spotted we have a small bug in our implementation. While the unit costs are whole numbers, our total and subtotals are displayed without the cents. What we really want is for these figures to always be displayed to two decimal places.

Rather than modify both the code that calculates the subtotals and the code that calculates the grand total, Vue provides us with a nice way to deal with common formatting tasks like this: filters.

As you might have already guessed, to create a filter we just pass an object with that key to our Vue instance:

```
const app = new Vue({
  // ...
  filters: {
    currency(value) {
      return value.toFixed(2);
    }
  }
});
```

Here we've created a very simple filter called `currency`, which calls `toFixed(2)` on the value it receives and returns the result. We can apply it to any output in our template like so:

```
<td>Total: ${{ total | currency }}</td>
```

Live Code

See the Pen Vue Invoice.

Summing Up

Comparing the two versions of the code side by side, a couple things stand out about the Vue app:

- **The clear separation between the UI, and the logic/data that drives it:** the code is much easier to understand, and lends itself to easier testing
- **The UI is declarative:** you only need concern yourself with what you want to see, not how to manipulate the DOM to achieve it.

The size (in KB) of both libraries is almost identical. Sure, you could slim down jQuery a bit with a custom build, but even with a relatively simple project such as our invoice example, I think the ease of development and the readability of the code justifies the difference.

Vue can also do a lot more than we've covered here. Its strength lies in allowing you to create modular, reusable UI components that can be composed into sophisticated front-end applications. If you're interested in delving deeper into Vue, I'd recommend checking out *Getting Up and Running with the Vue.js 2.0 Framework*.

Chapter 8: Nuxt.js: a Minimalist Framework for Creating Universal Vue.js Apps

by Olayinka Omole

Universal (or Isomorphic) JavaScript is a term that has become very common in the JavaScript community. It's used to describe JavaScript code that can execute both on the client and the server.

Many modern JavaScript frameworks, like Vue.js, are aimed at building single-page applications (SPAs). This is done to improve the user experience and make the app seem faster, since users can see updates to pages instantaneously. While this has a lot of advantages, it also has a couple of disadvantages, such as long "time to content" when initially loading the app as the browser retrieves the JavaScript bundle, and some search engine web crawlers or social network robots won't see the entire loaded app when they crawl your web pages.

Server-side rendering of JavaScript is about preloading JavaScript applications on a web server and sending rendered HTML as the response to a browser request for a page.

Building server-side rendered JavaScript apps can be a bit tedious, as a lot of configuration needs to be done before you even start coding. This is the problem Nuxt.js aims to solve for Vue.js applications.

What Nuxt.js Is

Simply put, Nuxt.js is a framework that helps you build server-rendered Vue.js applications easily. It abstracts most of the complex configuration involved in managing things like asynchronous data, middleware, and routing. It's similar to Angular Universal for Angular, and Next.js for React.

According to the Nuxt.js docs, "its main scope is UI rendering while abstracting away the client/server distribution."

Static Generation

Another great feature of Nuxt.js is its ability to generate static websites with the `generate` command. It's pretty cool, and provides features similar to popular static generation tools like Jekyll.

Under the Hood of Nuxt.js

In addition to Vue.js 2.0, Nuxt.js includes the following: Vue-Router, Vuex (only included when using the store option), Vue Server Renderer and vue-meta. This is great, as it takes away the burden of manually including and configuring different libraries needed for developing a server-rendered Vue.js application. Nuxt.js does all this out of the box, while still maintaining a total size of **57kB min+gzip** (60KB with vuex).

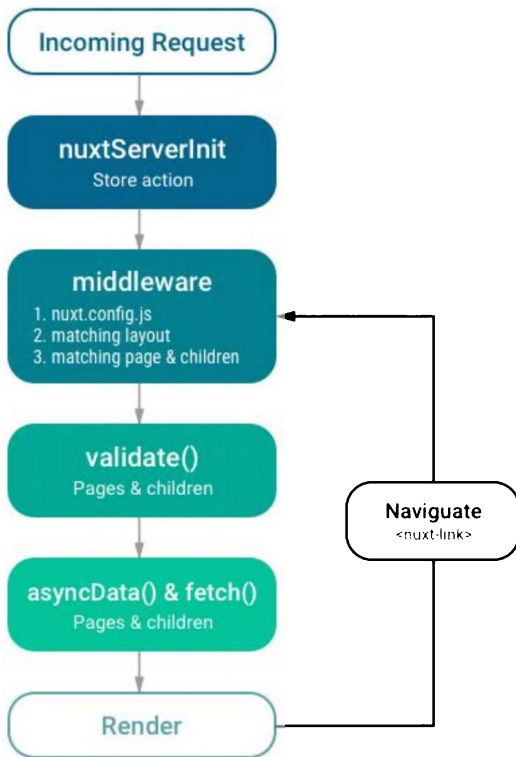
Nuxt.js also uses webpack with vue-loader and babel-loader to bundle, code-split and minify code.

How it works

This is what happens when a user visits a Nuxt.js app or navigates to one of its pages via `<nuxt-link>`:

1. When the user initially visits the app, if the `nuxtServerInit` action is defined in the store, Nuxt.js will call it and update the store.
2. Next, it executes any existing middleware for the page being visited. Nuxt checks the `nuxt.config.js` file first for global middleware, then checks the matching layout file (for the requested page), and finally checks the page and its children for middleware. Middleware are prioritized in that order.
3. If the route being visited is a dynamic route, and a `validate()` method exists for it, the route is validated.
4. Then, Nuxt.js calls the `asyncData()` and `fetch()` methods to load data before rendering the page. The `asyncData()` method is used for fetching data and rendering it on the server-side, while the `fetch()` method is used to fill the store before rendering the page.
5. At the final step, the page (containing all the proper data) is rendered.

These actions are portrayed properly in this schema, gotten from the Nuxt docs:

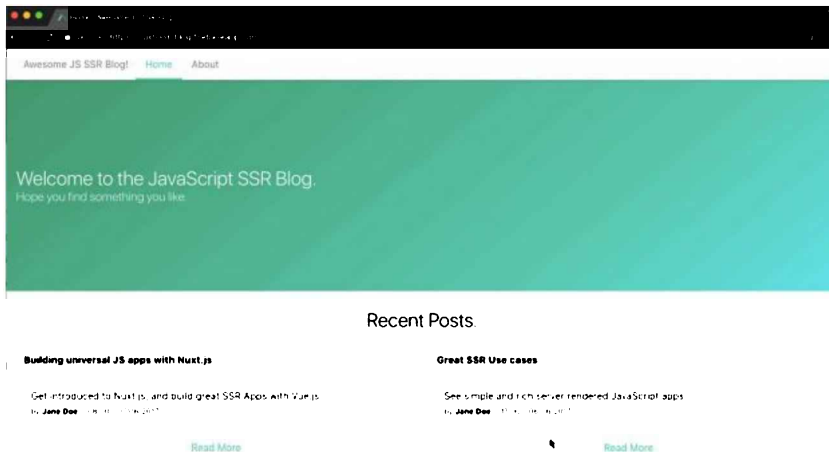


Creating A Serverless Static Site With Nuxt.js

Let's get our hands dirty with some code and create a simple static generated blog with Nuxt.js. We'll assume our posts are fetched from an API and will mock the response with a static JSON file.

To follow along properly, a working knowledge of Vue.js is needed. You can check out Jack Franklin's great getting started guide for Vue.js 2.0 if you're new to the framework. I'll also be using ES6 Syntax, and you can get a refresher on that here: sitepoint.com/tag/es6/.

Our final app will look like this:



Github

The entire code for this tutorial can be seen here on [GitHub](#), and you can check out the demo here.

Application Setup and Configuration

The easiest way to get started with Nuxt.js is to use the template created by the Nuxt team. We can install it to our project (`ssr-blog`) quickly using the `vue-cli`:

```
vue init nuxt/starter ssr-blog
```

Once you've run this command, a prompt will open and ask you a couple of questions. You can press Return to accept the default answers, or enter values of your own.

Install vue-cli

If you don't have vue-cli installed, you have to run `npm install -g @vue/cli` first, to install it.

Next, we install the project's dependencies:

```
cd ssr-blog
npm install
```

Now we can launch the app:

```
npm run dev
```

If all goes well, you should be able to visit <http://localhost:3000> to see the Nuxt.js template starter page. You can even view the page's source, to see that all content generated on the page was rendered on the server and sent as HTML to the browser.

Next, we can make some simple configurations in the `nuxt.config.js` file. We'll add a few options:

```
// ./nuxt.config.js

module.exports = {
  /*
   * Headers of the page
   */
  head: {
    titleTemplate: '%s | Awesome JS SSR Blog',
    // ...
    link: [
      // ...
      {
        rel: 'stylesheet',
        href: 'https://cdnjs.cloudflare.com/ajax/libs/bulma/0.7.2/css/bulma.min.css'
      }
    ]
  },
  // ...
}
```

In the above config file, we simply specify the title template to be used for the application via the `titleTemplate` option. Setting the `title` option in the individual pages or layouts will inject the `title` value into the `%s` placeholder in `titleTemplate` before being rendered.

We also pulled in my current CSS framework of choice, Bulma, to take advantage of some preset styling. This was done via the `link` option.

Understanding the Headers

Nuxt.js uses `vue-meta` to update the headers and HTML attributes of our apps. So you can take a look at it for a better understanding of how the headers are being set.

Now we can take the next couple of steps by adding our blog's pages and functionalities.

Working with Page Layouts

First, we'll define a custom base layout for all our pages. We can extend the main Nuxt.js layout by updating the `layouts/default.vue` file:

```
<!-- ./layouts/default.vue -->

<template>
  <div>
```



```

<!-- navigation -->
<nav class="navbar has-shadow" role="navigation" aria-label="main navigation">
  <div class="container">
    <div class="navbar-start">
      <nuxt-link to="/" class="navbar-item">
        Awesome JS SSR Blog!
      </nuxt-link>
      <nuxt-link active-class="is-active" to="/" class="navbar-item is-tab" exact>Home</nuxt-link>
      <nuxt-link active-class="is-active" to="/about" class="navbar-item is-tab" exact>About</nuxt-link>
    </div>
  </div>
</nav>
<!-- /navigation -->

<!-- displays the page component -->
<nuxt/>

</div>
</template>

<style>
  .main-content {
    margin: 30px 0;
  }
</style>

```

In our custom base layout, we add the site's navigation header. We use the `<nuxt-link>` component to generate links to the routes we want to have on our blog. You can check out the docs on `<nuxt-link>` to see how it works.

The `<nuxt>` component is really important when creating a layout, as it displays the page component.

It's also possible to do a couple of more things — like define custom document templates and error layouts — but we don't need those for our simple blog. I urge you to check out the Nuxt.js documentation on views to see all the possibilities.

Simple Pages and Routes

Pages in Nuxt.js are created as single file components in the `pages` directory. Nuxt.js automatically transforms every `.vue` file in this directory into an application route.

Building the Blog Homepage

We can add our blog homepage by updating the `index.vue` file generated by the Nuxt.js template in the `pages` directory:

```

<!-- ./pages/index.vue -->
<template>
  <div>
    <section class="hero is-medium is-primary is-bold">
      <div class="hero-body">
        <div class="container">
          <h1 class="title">
            Welcome to the JavaScript SSR Blog.
          </h1>
          <h2 class="subtitle">
            Hope you find something you like.
          </h2>
        </div>
      </div>
    </section>
  </div>
</template>

<script>
  export default {
    head: {

```

```

    title: 'Home'
  }
}
</script>

```

```
<!-- Remove the CSS styles -->
```

As stated earlier, specifying the `title` option here automatically injects its value into the `titleTemplate` value before rendering the page.

We can now reload our app to see the changes to the homepage.

Building the About page

Another great thing about Nuxt.js is that it will listen to file changes inside the `pages` directory, so there's no need to restart the application when adding new pages.

We can test this, by adding a simple `about.vue` page:

```

<!-- ./pages/about.vue -->
<template>
  <div class="main-content">
    <div class="container">
      <h2 class="title is-2">About this website.</h2>
      <p>Curabitur accumsan turpis pharetra <strong>augue tincidunt</strong> blandit. Quisque condimentum
      <br>
      <h4 class="title is-4">What we hope to achieve:</h4>
      <ul>
        <li>In fermentum leo eu lectus mollis, quis dictum mi aliquet.</li>
        <li>Morbi eu nulla lobortis, lobortis est in, fringilla felis.</li>
        <li>Aliquam nec felis in sapien venenatis viverra fermentum nec lectus.</li>
        <li>Ut non enim metus.</li>
      </ul>
    </div>
  </div>
</template>

<script>
export default {
  head: {
    title: 'About'
  }
}
</script>

```

Now, we can visit <http://localhost:3000/about> to see the about page, without having to restart the app, which is awesome.

Showing Blog Posts on the Homepage

Our current homepage is pretty bare as it is, so we can make it better by showing the recent blog posts from the blog. We'll do this by creating a `<posts>` component and displaying it in the `index.vue` page.

But first, we have to get our saved JSON blog posts and place them in a file in the app root folder. The file can be downloaded from here, or you can just copy the JSON below and save in the root folder as `posts.json`:

```

[
  {
    "id": 4,
    "title": "Building universal JS apps with Nuxt.js",
    "summary": "Get introduced to Nuxt.js, and build great SSR Apps with Vue.js.",
    "content": "<p>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.</p>",
    "author": "Jane Doe",
    "published": "08:00 - 07/06/2017"
  },
  {

```

```

    "id": 3,
    "title": "Great SSR Use cases",
    "summary": "See simple and rich server-rendered JavaScript apps.",
    "content": "<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod aliquip",
    "author": "Jane Doe",
    "published": "17:00 - 06/06/2017"
  },
  {
    "id": 2,
    "title": "SSR in Vue.js",
    "summary": "Learn about SSR in Vue.js, and where Nuxt.js can make it all faster.",
    "content": "<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor i",
    "author": "Jane Doe",
    "published": "13:00 - 06/06/2017"
  },
  {
    "id": 1,
    "title": "Introduction to SSR",
    "summary": "Learn about SSR in JavaScript and how it can be super cool.",
    "content": "<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor i",
    "author": "John Doe",
    "published": "11:00 - 06/06/2017"
  }
]

```

Where to Retrieve the Content

Ideally the posts should be retrieved from an API or resource. For example, Contentful is a service that can be used for this.

Components live in the components directory. We'll create the <posts> single file component in there:

```

<!-- ./components/Posts.vue -->
<template>
  <section class="main-content">
    <div class="container">
      <h1 class="title has-text-centered">
        Recent Posts.
      </h1>
      <div class="columns is-multiline">
        <div class="column is-half" v-for="post in posts" :key="post.id">
          <div class="card">
            <header class="card-header">
              <p class="card-header-title">
                {{ post.title }}
              </p>
            </header>
            <div class="card-content">
              <div class="content">
                {{ post.summary }}
                <br>
                <small>
                  by <strong>{{ post.author}}</strong>
                  \ \ {{ post.published }}
                </small>
              </div>
            </div>
            <footer class="card-footer">
              <nuxt-link :to="`/post/${post.id}`"
                class="card-footer-item">
                Read More
              </nuxt-link>
            </footer>
          </div>
        </div>
      </div>
    </div>
  </section>
</template>

```

```

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

<script>
  import posts from '~/posts.json'

  export default {
    name: 'posts',
    data () {
      return { posts }
    }
  }
</script>

```

We import the posts data from the saved JSON file and assign it to the `posts` value in our component. We then loop through all the posts in the component template with the `v-for` directive and display the post attributes we want.

The ~ Symbol

The `~` symbol is an alias for the `/` directory. You can check out the docs here to see the different aliases Nuxt.js provides, and what directories they're linked to.

Next, we add the `<posts>` component to the homepage:

```

<!-- ./pages/index.vue -->
<template>
<div>
  <!-- ... -->
  <posts />
</div>
</template>

<script>
import Posts from '~/components/Posts.vue'

export default {
  components: {
    Posts
  },
  // ...
}
</script>

```

Adding Dynamic Routes

Now we'll add dynamic routes for the posts, so we can access a post for example with this URL: `/post/1`.

To achieve this, we add the `post` directory to the `pages` directory and structure it like this:

```

pages
├── post
│   ├── _id
│   └── index.vue

```

This generates the corresponding dynamic routes for the application like this:

```

router: {
  routes: [
    // ...
    {
      name: 'post-id',
      path: '/post/:id',

```

```

      component: 'pages/post/_id/index.vue'
    }
  ]
}

```

Updating the single post file:

```

<!-- ./pages/post/_id/index.vue -->
<template>
  <div class="main-content">
    <div class="container">
      <h2 class="title is-2">{{ post.title }}</h2>
      <div v-html="post.content"></div>
      <br>
      <h4 class="title is-5 is-marginless">by <strong>{{ post.author }}</strong> at <strong>{{ post.published }}</strong>
    </div>
  </div>
</template>

<script>
  // import posts saved JSON data
  import posts from '~/posts.json'

  export default {
    validate ({ params }) {
      return /^\d+$/.test(params.id)
    },
    asyncData ({ params }, callback) {
      let post = posts.find(post => post.id === parseInt(params.id))
      if (post) {
        callback(null, { post })
      } else {
        callback({ statusCode: 404, message: 'Post not found' })
      }
    },
    head () {
      return {
        title: this.post.title,
        meta: [
          {
            hid: 'description',
            name: 'description',
            content: this.post.summary
          }
        ]
      }
    }
  }
</script>

```

Nuxt.js adds some custom methods to our page components to help make the development process easier. See how we use some of them on the single post page:

- Validate the route parameter with the `validate` method. Our `validate` method checks if the route parameter passed is a number. If it returns `false`, Nuxt.js will automatically load the 404 error page. You can read more on it [here](#).
- The `asyncData` method is used to fetch data and render it on the server side before sending a response to the browser. It can return data via different methods. In our case, we use a callback function to return the post that has the same `id` attribute as the route `id` parameter. You can see the various ways of using this function [here](#).
- As we've seen before, we use the `head` method to set the page's headers. In this case, we're changing the page title to the title of the post, and adding the post summary as a meta description for the page.

Great, now we can visit our blog again to see all routes and pages working properly, and also view the page source to see the HTML being generated. We have a functional server-rendered JavaScript application.

Generating Static Files

Next, we can generate the static HTML files for our pages.

We'll need to make a minor tweak though, as by default Nuxt.js ignores dynamic routes. To generate the static files for dynamic routes, we need to specify them explicitly in the `./nuxt.config.js` file.

We'll use a callback function to return the list of our dynamic routes:

```
// ./nuxt.config.js

module.exports = {
  // ...
  generate: {
    routes: (callback) {
      const posts = require('./posts.json')
      let routes = posts.map(post => `/post/${post.id}`)
      callback(null, routes)
    }
  }
}
```

You can check here for the full documentation on using the `generate` property.

To generate all the routes, we can now run this command:

```
npm run generate
```

Nuxt saves all generated static files to a `dist` folder.

Deployment on Firebase Hosting

As a final step, we can take advantage of hosting by Firebase to make our static website live in a couple of minutes. This step assumes that you have a Google account.

First, install the Firebase CLI, if you don't already have it:

```
npm install -g firebase-tools
```

To connect your local machine to your Firebase account and obtain access to your Firebase projects, run the following command:

```
firebase login
```

This should open a browser window and prompt you to sign in. Once you're signed in, visit <https://console.firebase.google.com> and click *Add project*. Make the relevant choices in the wizard that opens.

Once the project is created, go to the project's hosting page at <https://console.firebase.google.com/project/<project name>/hosting> and complete the *Get started* wizard.

Then, on your PC, from the root of your project directory, run the following command:

```
firebase init
```

In the wizard that appears, select "Hosting". Then select your newly created project from the list of options. Next choose the `dist` directory as the public directory. Select to configure the page as a single-page app and finally select "No" when asked if you want to overwrite `dist/index.html`.

Firebase will write a couple of configuration files to your project, then put the website live at <https://<project name>.firebaseapp.com>. The demo app for this article can be seen at nuxt-ssr-blog.firebaseio.com.

If you run into problems, you can find full instructions on Firebase's quickstart page.

Conclusion

In this article, we've learned how we can take advantage of Nuxt.js to build server-rendered JavaScript applications with Vue.js. We also learned how to use its `generate` command to generate static files for our pages, and deploy them quickly via a service like Firebase Hosting.

The Nuxt.js framework is really great. It's even recommended in the official Vue.js SSR GitBook. I really look forward to using it in more SSR projects and exploring all of its capabilities.

Chapter 9: Optimize the Performance of a Vue App with Async Components

by Michiel Mulders

Single-page applications sometimes cop a little flack for their slow initial load. This is because traditionally, the server will send a large bundle of JavaScript to the client, which must be downloaded and parsed before anything is displayed on the screen. As you can imagine, as your app grows in size, this can become more and more problematic.

Luckily, when building a Vue application using Vue CLI (which uses webpack under the hood), there are a number of measures one can take to counteract this. In this article, I'll demonstrate how make use of both asynchronous components and webpack's code-splitting functionality to load in parts of the page after the app's initial render. This will keep the initial load time to a minimum and give your app a snappier feel.

To follow this tutorial, you need a basic understanding of Vue.js and optionally Node.js.

Async Components

Before we dive into creating asynchronous components, let's take a look at how we normally load a component. To do so, we'll use a very simple message component:

```
<!-- Message.vue -->
<template>
  <h1>New message!</h1>
</template>
```

Now that we've created our component, let's load it into our `App.vue` file and display it. We can just import the component and add it to the `components` option so we can use it in our template:

```
<!-- App.vue -->
<template>
  <div>
    <message></message>
  </div>
</template>

<script>
import Message from "./Message";
export default {
  components: {
    Message
  }
};
</script>
```

But what happens now? The `Message` component will be loaded whenever the application is loaded, so it's included in the initial load.

This might not sound like a huge problem for a simple app, but consider something more complex like a web store. Imagine that a user adds items to a basket, then wants to check out, so clicks the checkout button which renders a box with all details of the selected items. Using the above method, this checkout box will be included in the initial bundle, although we only need the component when the user clicks the checkout button. It's even possible that the user navigates through the website without ever clicking the checkout button, meaning that it doesn't make sense to waste resources on loading this potentially unused component.

To improve the efficiency of the application, we can combine both lazy loading and code splitting techniques.

Lazy loading is all about delaying the initial load of a component. You can see lazy loading in action on sites like medium.com,

where the images are loaded in just before they're required. This is useful, as we don't have to waste resources loading all the images for a particular post up front, as the reader might skip the article halfway down.

The code splitting feature webpack provides allows you to split your code into various bundles that can then be loaded on demand or in parallel at a later point in time. It can be used to load specific pieces of code only when they're required or used.

Dynamic Imports

Luckily, Vue caters for this scenario using something called **dynamic imports**. This feature introduces a new function-like form of import that will return a Promise containing the requested (Vue) component. As the import is a function receiving a string, we can do powerful things like loading modules using expressions. Dynamic imports have been available in Chrome since version 61. More information about them can be found on the Google Developers website.

The code splitting is taken care of by bundlers like webpack, Rollup or Parcel, which understand the dynamic import syntax and create a separate file for each dynamically imported module. We'll see this later on in our console's network tab. But first, let's take a look at the difference between a static and dynamic import:

```
// static import
import Message from "./Message";

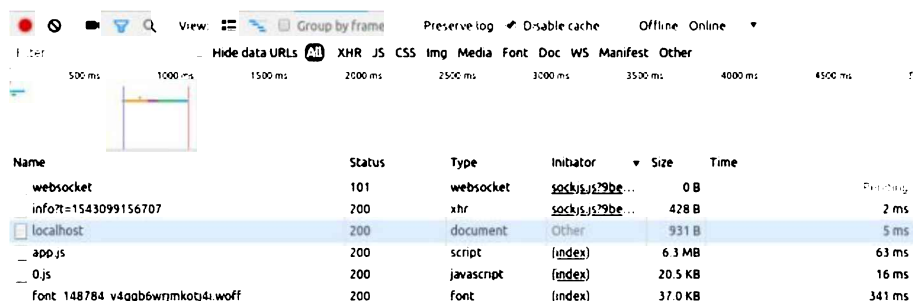
// dynamic import
import("./Message").then(Message => {
  // Message module is available here...
});
```

Now, let's apply this knowledge to our Message component, and we'll get an App.vue component that looks like this:

```
<!-- App.vue -->
<template>
  <div>
    <message></message>
  </div>
</template>

<script>
import Message from "./Message";
export default {
  components: {
    Message: () => import("./Message")
  }
};
</script>
```

As you can see, the `import()` function will resolve a Promise that returns the component, meaning that we've successfully loaded our component asynchronously. If you take a look in your devtools' network tab, you'll notice a file called `0.js` that contains your asynchronous component.



Name	Status	Type	Initiator	Size	Time
websocket	101	websocket	sockjs.js?9be...	0 B	Pending
info?n=1543099156707	200	xhr	sockjs.js?9be...	428 B	2 ms
localhost	200	document	Other	931 B	5 ms
app.js	200	script	(index)	6.3 MB	63 ms
0.js	200	javascrip	(index)	20.5 KB	16 ms
font_148784_v4ggb6w7jmkot4i.woff	200	font	(index)	37.0 KB	341 ms

Conditionally Loading Async Components

Now that we have a handle on asynchronous components, let's truly harvest their power by only loading them when they're really needed. In the previous section of this article, I explained the use case of a checkout box that's only loaded when the user hits the checkout button. Let's build that out.

Project Setup

If you don't have Vue CLI installed, you should grab that now:

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

Next, use the CLI to create a new project, selecting the default preset when prompted:

```
vue create my-store
```

Change into the project directory, then install the ant-design-vue library, which we'll be using for styling:

```
cd my-store
npm i ant-design-vue
```

Next, import the Ant Design library in `src/main.js`:

```
import 'ant-design-vue/dist/antd.css'
```

Finally, create two new components in `src/comonents`, `Checkout.vue` and `Items.vue`:

```
touch src/comonents/{Checkout.vue,Items.vue}
```

Making the Store View

Open up `src/App.vue` and replace the code there with the following:

```
<template>
  <div id="app">
    <h1>{{ msg }}</h1>
    <items></items>
  </div>
</template>

<script>
import items from "../comonents/Items"

export default {
  components: {
    items
  },
  name: 'app',
  data () {
    return {
      msg: 'My Fancy T-Shirt Store'
    }
  }
}
</script>

<style>
#app {
  font-family: 'Avenir', Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
}
```

```

    margin-top: 60px;
  }

h1, h2 {
  font-weight: normal;
}

ul {
  list-style-type: none;
  padding: 0;
}

li {
  display: inline-block;
  margin: 0 10px;
}

a {
  color: #42b983;
}
</style>

```

There's nothing fancy going on here. All we're doing is displaying a message and rendering an `<items>` component.

Next, open up `src/components/Items.vue` and add the following code:

```

<template>
  <div>
    <div style="padding: 20px;">
      <Row :gutter="16">
        <Col :span="24" style="padding:5px">
          <Icon type="shopping-cart" style="margin-right:5px"/>{{shoppingList.length}} item(s)
          <Button @click="show = true" id="checkout">Checkout</Button>
        </Col>
      </Row>
    </div>
    <div v-if="show">
      <Row :gutter="16" style="margin:0 400px 50px 400px">
        <checkout v-bind:shoppingList="shoppingList"></checkout>
      </Row>
    </div>
    <div style="background-color: #ecec; padding: 20px;">
      <Row :gutter="16">
        <Col :span="6" v-for="(item, key) in items" v-bind:key="key" style="padding:5px">
          <Card v-bind:title="item.msg" v-bind:key="key">
            <Button type="primary" @click="addItem(key)">Buy ${{item.price}}</Button>
          </Card>
        </Col>
      </Row>
    </div>
  </div>
</template>

<script>
import { Card, Col, Row, Button, Icon } from 'ant-design-vue';

export default {
  methods: {

```

```

    addItem (key) {
      if(!this.shoppingList.includes(key)) {
        this.shoppingList.push(key);
      }
    }
  },
  components: {
    Card, Col, Row, Button, Icon,
    checkout: () => import('./Checkout')
  },
  data: () => ({
    items: [
      { msg: 'First Product', price: 9.99 },
      { msg: 'Second Product', price: 19.99 },
      { msg: 'Third Product', price: 15.00 },
      { msg: 'Fancy Shirt', price: 137.00 },
      { msg: 'More Fancy', price: 109.99 },
      { msg: 'Extreme', price: 3.00 },
      { msg: 'Super Shirt', price: 109.99 },
      { msg: 'Epic Shirt', price: 3.00 },
    ],
    shoppingList: [],
    show: false
  })
}
</script>
<style>
#checkout {
  background-color:#e55242;
  color:white;
  margin-left: 10px;
}
</style>

```

In this file, we're displaying a shopping cart icon with the current amount of purchased items. The items themselves are pulled from an `items` array, declared as a `data` property. If you click on an item's `Buy` button, the `addItem` method is called, which will push the item in question to a `shoppingList` array. In turn, this will increment the cart's total.

We've also added a `Checkout` button to the page, and this is where things start to get interesting:

```
<Button @click="show = true" id="checkout">Checkout</Button>
```

When a user clicks on this button, we're setting a parameter `show` to be `true`. This `true` value is very important for the purpose of conditionally loading our `async` component.

A few lines below, you can find a `v-if` statement, which only displays the content of the `<div>` when `show` is set to `true`. This `<div>` tag contains the `checkout` component, which we only want to load when the user has hit the `checkout` button.

The `checkout` component is loaded asynchronously in the `components` option in the `<script>` section. The cool thing here is that we can even pass arguments to the component via the `v-bind` statement. As you can see, it's relatively easy to create conditional asynchronous components:

```
<div v-if="show">
  <checkout v-bind:shoppingList="shoppingList"></checkout>
</div>
```

Let's quickly add the code for the `Checkout` component in `src/components/Checkout.vue`:

```
<template>
  <Card title="Checkout Items" key="checkout">
```

```

<p v-for="(k, i) in this.shoppingList" :key="i">
  Item: {{items[Number(k)].msg}} for ${{items[Number(k)].price}}
</p>
</Card>
</template>

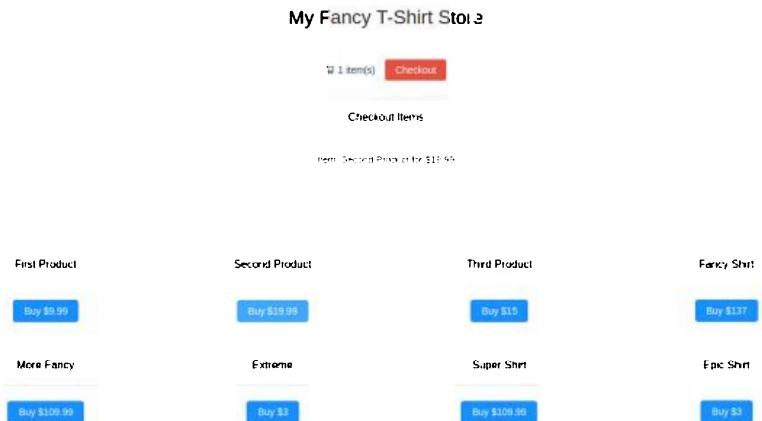
<script>
import { Card } from 'ant-design-vue';

export default {
  props: ['shoppingList'],
  components: {
    Card
  },
  data: () => ({
    items: [
      { msg: 'First Product', price: 9.99 },
      { msg: 'Second Product', price: 19.99 },
      { msg: 'Third Product', price: 15.00 },
      { msg: 'Fancy Shirt', price: 137.00 },
      { msg: 'More Fancy', price: 109.99 },
      { msg: 'Extreme', price: 3.00 },
      { msg: 'Super Shirt', price: 109.99 },
      { msg: 'Epic Shirt', price: 3.00 },
    ]
  })
}
</script>

```

Here we're looping over the props we receive as shoppingList and outputting them to the screen.

You can run the app using the `npm run serve` command. Then navigate to `http://localhost:8080/`. If all has gone according to plan, you should see something like what's shown in the image below.



Try clicking around the store with your network tab open to assure yourself that the Checkout component is only loaded when you click the Checkout button.

You can also find the code for this demo on [GitHub](#).

Async with Loading and Error Component

It's even possible to define a loading and/or error component for when the async component takes some time to load or fails

to load. It can be useful to show a loading animation, but bear in mind this again slows down your application. An asynchronous component should be small and fast to load. Here's an example:

```
const Message = () => ({
  component: import("./Message"),
  loading: LoadingAnimation,
  error: ErrorComponent
});
```

Conclusion

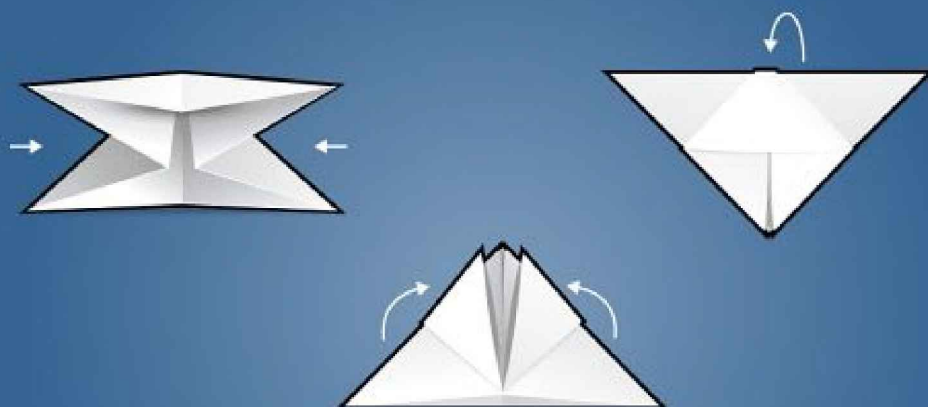
Creating and implementing asynchronous components is very easy and should be part of your standard development routine. From a UX perspective, it's important to reduce the initial load time as much as possible to maintain the user's attention. Hopefully this tutorial has assisted you with loading your own components asynchronously and applying conditions to them to delay (lazy load) the load of the component.

Book 2: Vue.js: 11 Practical Projects



VUE.JS

11 PRACTICAL PROJECTS



BUILD YOUR OWN SOPHISTICATED WEB APPS

Chapter 1: Build a Basic CRUD App with Vue.js, Node and MongoDB

by James Hibbard

Most web applications need to persist data in one form or other. When working with a server-side language, this is normally a straightforward task. However, when you add a front-end JavaScript framework to the mix, things start to get a bit trickier.

In this tutorial, I'm going to show you how to build a simple CRUD app (Create, Read, Update and Delete) using Node, MongoDB and Vue. I'll walk you through getting all of the pieces set up and demo each of the CRUD operations. After reading, you'll be left with a fully functional app that you can adapt to your own purposes. You'll also be able to use this approach in projects of your own.

As the world really doesn't need another to-do list, we'll be building an app to help students of a foreign language learn vocabulary. As you might notice from my author bio, I'm a Brit living in Germany and this is something I wish I'd had 20 years ago.

You can check out the finished product in this [GitHub repo](#). And if you're stuck for German words at any point, try out some of these.

Install the Tooling

In this section, we'll install the tools we'll need to create our app. These are Node, npm, MongoDB, MongoDB Compass (optional) and Postman. I won't go too into depth on the various installation instructions, but if you have any trouble getting set up, please visit our [forums](#) and ask for help there.

Node.js

Many websites will recommend that you head to the official Node download page and grab the Node binaries for your system. While that works, I'd suggest using a version manager (such as `nvm`) instead. This is a program which allows you to install multiple versions of Node and switch between them at will. If you'd like to find out more about this, please consult our quick tip, [Install Multiple Versions of Node.js Using nvm](#).

npm

npm is a JavaScript package manager which comes bundled with Node, so no extra installation is necessary here. We'll be making quite extensive use of npm throughout this tutorial, so if you're in need of a refresher, please consult [A Beginner's Guide to npm — the Node Package Manager](#).

MongoDB

MongoDB is a document database which stores data in flexible, JSON-like documents.

The quickest way to get up and running with Mongo is to use a service such as [mLabs](#). They have a free sandbox plan which provides a single database with 496MB of storage running on a shared virtual machine. This is more than adequate for a simple app with a handful of users. If this sounds like the best option for you, please consult their [quick start guide](#).

You can also install Mongo locally. To do this, please visit the [official download page](#) and download the correct version of the community server for your operating system. There's a link to detailed, OS-specific installation instructions next to the download link.

A MongoDB GUI

Although not strictly necessary for following along with this tutorial, you might also like to install Compass, the official GUI for MongoDB. This tool helps you visualize and manipulate your data, allowing you to interact with documents with full CRUD functionality.

At the time of writing, you'll need to fill out your details to download Compass, but you won't need to create an account.

Postman

This is an extremely useful tool for working with and testing APIs. You can download the correct version for your OS from the project's [home page](#). You can find OS-specific installation instructions [here](#).

Check That Everything Is Installed Correctly

To check that Node and npm are installed correctly, open your terminal and type this:

```
node -v
```


Follow that with this:

```
npm -v
```

This will output the version number of each program (11.1.0 and 6.4.1 respectively at the time of writing).

If you installed Mongo locally, you can check the version number using this:

```
mongo --version
```

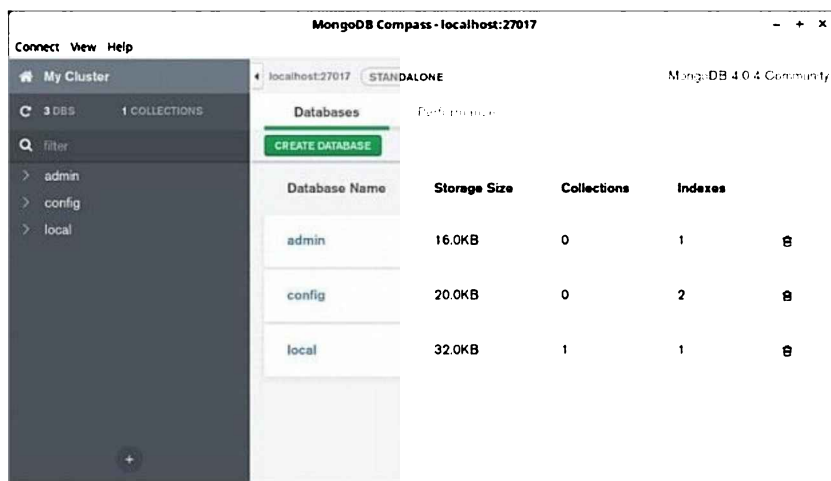
This should output a bunch of information, including the version number (4.0.4 at the time of writing).

Check the Database Connection Using Compass

If you have installed Mongo locally, you start the server by typing the following command into a terminal:

```
mongod
```

Next, open Compass. You should be able to accept the defaults (Hostname: localhost, Port: 27017), press the **CONNECT** button, and establish a connection to the database server.



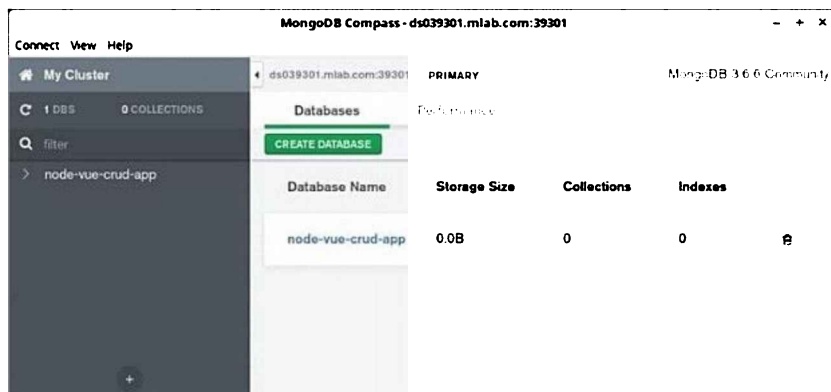
Note that the databases `admin`, `config` and `local` are created automatically.

Using a Cloud-hosted Solution

If you're using mLabs, create a database subscription (as described in their quick-start guide), then copy the connection details to the clipboard. This should be in the following form:

```
mongodb://<dbuser>:<dbpassword>@<instance>.mlab.com:<port>/<dbname>
```

When you open Compass, it will inform you that it has detected a MongoDB connection string and asks if you'd like to use it to fill out the form. Click **Yes**, then click **CONNECT** and you should be off to the races.



Note that I called my database `node-vue-crud-app`. You can call yours what you like.

Creating the Node Back End

In this section, we'll create a RESTful API for our Vue front end to consume and test it using Postman. I don't want to go into too much detail on how a RESTful API works, as the focus of this guide should be the Vue front end. If you'd like a more in-depth tutorial, there are plenty on the Internet. I found that [Build Node.js RESTful APIs in 10 Minutes](#) was quite easy to follow, and it served as inspiration for the code in this section.

Basic Setup

First, let's create the files and directories we'll need and initialize the project:

```
mkdir -p vocab-builder/server/api/{controllers,models,routes}
cd vocab-builder/server
touch server.js
touch api/controllers/vocabController.js
touch api/models/vocabModel.js
touch api/routes/vocabRoutes.js
npm init -y
```

This should give us the following folder structure:

```
.
├── server
│   ├── api
│   │   ├── controllers
│   │   │   └── vocabController.js
│   │   ├── models
│   │   │   └── vocabModel.js
│   │   └── routes
│   │       └── vocabRoutes.js
│   ├── package.json
│   └── server.js
```

Install the Dependencies

For our API, we'll be using the following libraries:

- **express**—a small framework for Node that provides many useful features, such as routing and middleware.
- **cors**—Express middleware that can be used to enable CORS in our project.
- **body-parser**—Express middleware to parse the body of incoming requests.
- **mongoose**—a MongoDB ODM (the NoSQL equivalent of an ORM) for Node. It provides a simple validation and query API to help you interact with your MongoDB database.
- **nodemon**—a simple monitor script which will automatically restart a Node application when file changes are detected.

Let's get them installed:

```
npm i express cors body-parser mongoose
npm i nodemon --save-dev
```

Next, open up `package.json` and alter the `scripts` section to read as follows:

```
"scripts": {
  "start": "nodemon server.js"
},
```

And that's the setup done.

Create the Server

Open up `server.js` and add the following code:

```
const express = require('express');
const port = process.env.PORT || 3000;
const app = express();

app.listen(port);
```

```
app.get('/', (req, res) => {
  res.send('Hello, World!');
});

console.log(`Server started on port ${port}`);
```

Then hop into the terminal and run `npm run start`. You should see a message that the server has started on port 3000. If you visit `http://localhost:3000/`, you should see “Hello, World!” displayed.

Define a Schema

Next, we need to define what the data should look like in our database. Essentially I wish to have two fields per document, `english` and `german`, to hold the English and German translations of a word. I’d also like to apply some simple validation and ensure that neither of these fields can be empty.

Open up `api/models/vocabModel.js` and add the following code:

```
const mongoose = require('mongoose');

const { Schema } = mongoose;

const VocabSchema = new Schema(
  {
    english: {
      type: String,
      required: 'English word cannot be blank'
    },
    german: {
      type: String,
      required: 'German word cannot be blank'
    }
  },
  { collection: 'vocab' }
);

module.exports = mongoose.model('Vocab', VocabSchema);
```

Note that I’m also specifying the name of the collection I wish to create, as otherwise Mongoose will attempt to name it “vocabs”, which is silly.

Setting Up the Routes

Next we have to specify how our API should respond to requests from the outside world. We’re going to create the following endpoints:

- GET `/words`—return a list of all words
- POST `/words`—create a new word
- GET `/words/:wordId`—get a single word
- PUT `/words/:wordId`—update a single word
- DELETE `/words/:wordId`—delete a single word

To do this, open up `api/routes/vocabRoutes.js` and add the following:

```
const vocabBuilder = require('../controllers/vocabController');

module.exports = app => {
  app
    .route('/words')
    .get(vocabBuilder.list_all_words)
    .post(vocabBuilder.create_a_word);

  app
    .route('/words/:wordId')
    .get(vocabBuilder.read_a_word)
    .put(vocabBuilder.update_a_word)
```

```
    .delete(vocabBuilder.delete_a_word);
};
```

You'll notice that we're requiring the `vocabController` at the top of the file. This will contain the actual logic to handle the requests. Let's implement that now.

Creating the Controller

Open up `api/controllers/vocabController.js` and add the following code:

```
const mongoose = require('mongoose');
const Vocab = mongoose.model('Vocab');

exports.list_all_words = (req, res) => {
  Vocab.find({}, (err, words) => {
    if (err) res.send(err);
    res.json(words);
  });
};

exports.create_a_word = (req, res) => {
  const newWord = new Vocab(req.body);
  newWord.save((err, word) => {
    if (err) res.send(err);
    res.json(word);
  });
};

exports.read_a_word = (req, res) => {
  Vocab.findById(req.params.wordId, (err, word) => {
    if (err) res.send(err);
    res.json(word);
  });
};

exports.update_a_word = (req, res) => {
  Vocab.findOneAndUpdate(
    { _id: req.params.wordId },
    req.body,
    { new: true },
    (err, word) => {
      if (err) res.send(err);
      res.json(word);
    }
  );
};

exports.delete_a_word = (req, res) => {
  Vocab.deleteOne({ _id: req.params.wordId }, err => {
    if (err) res.send(err);
    res.json({
      message: 'Word successfully deleted',
      _id: req.params.wordId
    });
  });
};
```

Here we're importing our model, then defining five functions corresponding to the desired CRUD functionality. If you're unsure as to what any of these functions do or how they work, please consult the [Mongoose documentation](#).

Wiring Everything Together

Now all of the pieces are in place and the time has come to wire them together. Hop back into `server.js` and replace the existing code with the following:

```

const express = require('express');
const cors = require('cors');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');

global.Vocab = require('./api/models/vocabModel');
const routes = require('./api/routes/vocabRoutes');

mongoose.Promise = global.Promise;
mongoose.set('useFindAndModify', false);
mongoose.connect(
  'mongodb://localhost/vocab-builder',
  { useNewUrlParser: true }
);

const port = process.env.PORT || 3000;
const app = express();

app.use(cors());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

routes(app);
app.listen(port);

app.use((req, res) => {
  res.status(404).send({ url: `${req.originalUrl} not found` });
});

console.log(`Server started on port ${port}`);

```

Here we start off by requiring the necessary libraries, as well as our model and routes files. We then use Mongoose's `connect` method to connect to our database (don't forget to make sure MongoDB is running if you installed it locally), before creating a new Express app and telling it to use the `bodyParser` and `cors` middleware. We then tell it to use the routes we defined in `api/routes/vocabRoutes.js` and to listen for connections on port 3000. Finally, we define a function to deal with nonexistent routes.

You can test this is working by hitting `http://localhost:3000/` in your browser. You should no longer see a "Hello, World!" message, but rather `{"url":"/ not found"}`.

Also, please note that if you're using `mLabs`, you'll need to replace:

```

mongoose.connect(
  'mongodb://localhost/vocab-builder',
  { useNewUrlParser: true }
);

```

with the details you received when creating your database:

```

mongoose.connect(
  'mongodb://<dbuser>:<dbpassword>@<instance>.mlab.com:<port>/<dbname>',
  { useNewUrlParser: true }
);

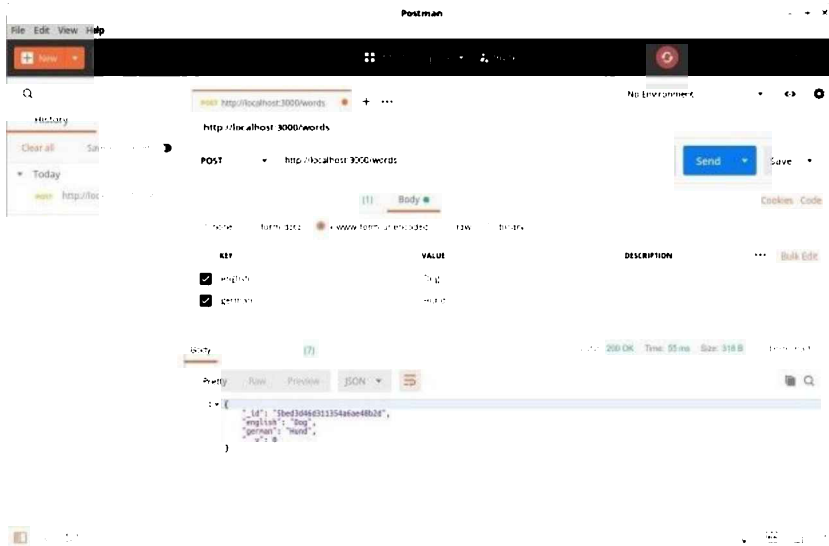
```

Testing the API with Postman

To test the API, start up Postman. How you do this will depend on how you installed it.

Create

We'll start off by creating a new word pair. Select `POST` as the method and enter `http://localhost:3000/words` as the URL. Select the `Body` tab and the radio button `x-www-form-urlencoded`, then enter two `key/value` pairs into the fields below. Press `Send` and you should receive the newly created object from the server by way of a response.



At this point, you can also view the database using Compass, to assure yourself that the collection was created with the correct entry.

Read

To test the read functionality of our API, we'll first attempt to get all words in our collection, then an individual word.

To get all of the words, change the method to `GET` and `Body` to `none`. Then hit `Send`. The response should be similar to this:

```
[
  {
    "_id": "5bed3d46d311354a6ae48b2d",
    "english": "Dog",
    "german": "Hund",
    "__v": 0
  }
]
```

Copy the `_id` value of the first word returned and change the URL to `http://localhost:3000/words/<_id>`. For me this would be:

```
http://localhost:3000/words/5bed3d46d311354a6ae48b2d
```

Hit `Send` again to retrieve just that word from our API. The response should be similar to this:

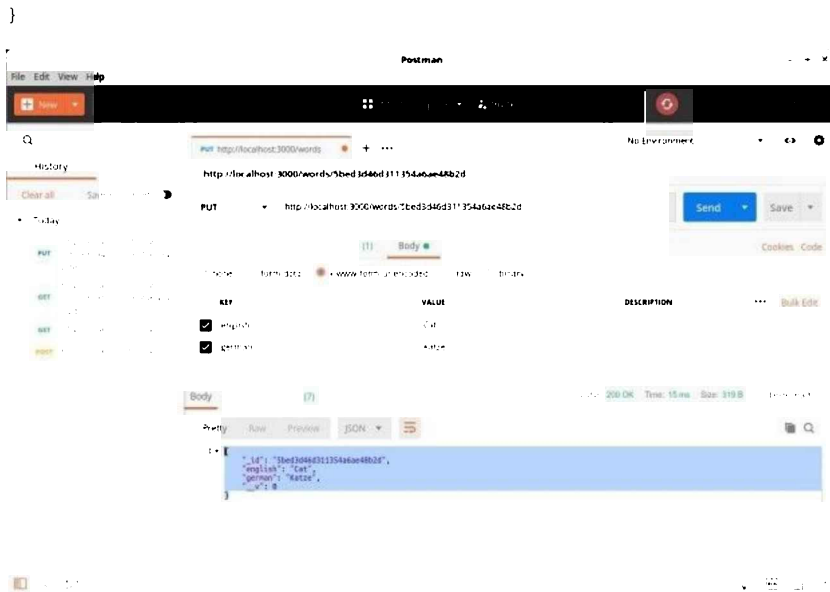
```
{
  "_id": "5bed3d46d311354a6ae48b2d",
  "english": "Dog",
  "german": "Hund",
  "__v": 0
}
```

Notice that now just the object is returned, not an array of objects.

Update

Finally, let's update a word, then delete it. To update the word, select `PUT` as the method and enter `http://localhost:3000/words/<word _id>` as the URL (where the ID of the word corresponds to `_id` as described above). Select the `Body` tab and the radio button `x-www-form-urlencoded`, then enter two key/value pairs into the fields below different than before. Press `Send` and you should receive the newly updated object from the server by way of a response:

```
{
  "_id": "5bed3d46d311354a6ae48b2d",
  "english": "Cat",
  "german": "Katze",
  "__v": 0
}
```



Destroy

To delete a word, just change the method to *DELETE* and press *Send*. Zap! The word should be gone and your collection should be empty. You can test that this is indeed the case, either using Compass, or by selecting the original request to get all words from the left hand pane in Postman and firing that off again. This time it should return an empty array.

And that's it. We're done with our back end. Don't forget that you can find all of the code for this guide on GitHub, so if something didn't work as expected, try cloning the repo and running that.

Creating the Vue Front End

Previously we created our back end in a `server` folder. We'll create the front-end part of the app in a `front-end` folder. This separation of concerns means that it would be easier to swap out the front end if we ever decided we wanted to use a framework other than Vue.

Let's get to it.

Basic Setup

We'll use Vue CLI to scaffold the project:

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

Make sure you're in the root folder of the project (`vocab-builder`), then run the following:

```
vue create front-end
```

This will open a wizard to walk you through creating a new Vue app. Answer the questions as follows:

```
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, Router, Linter
? Use history mode for router? Yes
? Pick a linter / formatter config: ESLint with error prevention only
? Pick additional lint features: Lint on save
? Where do you prefer placing config for Babel etc.? In dedicated config files
? Save this as a preset for future projects? No
```

The main thing to note here is that we're installing the Vue router, which we'll use to display the different views in our front end.

Next, change into our new directory:

```
cd front-end
```

There are a few files created by the CLI that we don't need. We can delete these now:

```
rm src/assets/logo.png src/components/HelloWorld.vue src/views/About.vue src/views/Home.vue
```

There are also a bunch of files we'll need to create:

```
cd src
touch components/{VocabTest.vue,WordForm.vue}
touch views/{Edit.vue,New.vue,Show.vue,Test.vue,Words.vue}
mkdir helpers
touch helpers/helpers.js
```

When you're finished, the contents of your `vocab-builder/front-end/src` folder should look like this:

```
.
├── App.vue
├── assets
├── components
│   ├── VocabTest.vue
│   └── WordForm.vue
├── helpers
│   └── helpers.js
├── main.js
├── router.js
└── views
    ├── Edit.vue
    ├── New.vue
    ├── Show.vue
    ├── Test.vue
    └── Words.vue
```

Install the Dependencies

For the front end, we'll be using the following libraries:

- `axios`—a Promise-based HTTP client which will allow us to make Ajax requests from our front end to our back end.
- `semantic-ui-css`—a lightweight, CSS-only version of Semantic UI.
- `vue-flash-message`—a Vue flash message component which we will use to display information to the user.

Let's get them installed:

```
npm i axios semantic-ui-css vue-flash-message
```

And with that, we're ready to start coding.

Creating the Routes

Firstly, we need to consider which routes our application will have. We'll need to perform all of the CRUD options we discussed previously, and it would also be nice if the user could test themselves against whatever words have been added to the database.

To this end, we'll create the following:

- `/words`—display all the words in the database
- `/words/new`—create a new word
- `/words/:id`—display a word
- `/words/:id/edit`—edit a word
- `/test`—test your knowledge

Deleting a word doesn't require its own route. We'll just fire off the appropriate request and redirect to `/words`.

To put this into action, open up `src/router.js` and replace the existing code with the following:

```
import Vue from 'vue';
import Router from 'vue-router';
import Words from './views/Words.vue';
import New from './views/New.vue';
import Show from './views/Show.vue';
import Edit from './views/Edit.vue';
import Test from './views/Test.vue';
```



```

Vue.use(Router);

export default new Router({
  mode: 'history',
  base: process.env.BASE_URL,
  linkActiveClass: 'active',
  routes: [
    {
      path: '/',
      redirect: '/words'
    },
    {
      path: '/words',
      name: 'words',
      component: Words
    },
    {
      path: '/words/new',
      name: 'new-word',
      component: New
    },
    {
      path: '/words/:id',
      name: 'show',
      component: Show
    },
    {
      path: '/words/:id/edit',
      name: 'edit',
      component: Edit
    },
    {
      path: '/test',
      name: 'test',
      component: Test
    }
  ]
});

```

This creates the routes we discussed above. It also redirects the root URL / to /words and tells the router to apply a class of active to the currently activated navigation link (the default would be router-link-active, which wouldn't play nicely with semantic-ui-css).

Talking of which, next we need to open up src/main.js and tell Vue to use semantic-ui-css:

```

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

import 'semantic-ui-css/semantic.css';

Vue.config.productionTip = false;

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

```

And finally, we need to replace the code in src/App.vue with the following:

```

<template>
  <div id="app">
    <div class="ui inverted segment navbar">
      <div class="ui center aligned container">

```

```

    <div class="ui large secondary inverted pointing menu compact">
      <router-link to="/words" exact class="item">
        <i class="comment outline icon"></i> Words
      </router-link>
      <router-link to="/words/new" class="item">
        <i class="plus circle icon"></i> New
      </router-link>
      <router-link to="/test" class="item">
        <i class="graduation cap icon"></i> Test
      </router-link>
    </div>
  </div>
</div>

<div class="ui text container">
  <div class="ui one column grid">
    <div class="column">
      <router-view />
    </div>
  </div>
</div>
</div>
</template>

<script>
export default {
  name: 'app'
};
</script>

<style>
#app > div.navbar {
  margin-bottom: 1.5em;
}
.myFlash {
  width: 250px;
  margin: 10px;
  position: absolute;
  top: 50;
  right: 0;
}
input {
  width: 300px;
}
div.label {
  width: 120px;
}
div.input {
  margin-bottom: 10px;
}
button.ui.button {
  margin-top: 15px;
  display: block;
}
</style>

```

This creates a nav bar at the top of the screen, then declares an outlet (the `<router-view />` component) for our views.

Start the application from within the front-end folder with the command `npm run serve`.

You should now be able to go to `http://localhost:8080` and navigate around the shell of our app.

Exciting times, huh?

Communicating with the Back End

The next thing we need to do is establish a method of communicating with our Node back end. This is where the axios library comes in that we installed previously. To make our app flexible, we can add any axios-related code to a helper file which can be included in any component that needs it. Setting things up this way has the advantage of keeping all of our Ajax logic in one place, and also means that if we ever wanted to swap out axios for a different library, it would be pretty easy.

So open up `src/helpers/helpers.js` and add the following code:

```
import axios from 'axios';

const baseUrl = 'http://localhost:3000/words/';

const handleError = fn => (...params) =>
  fn(...params).catch(error => {
    console.log(error);
  });

export const api = {
  getWord: handleError(async id => {
    const res = await axios.get(baseUrl + id);
    return res.data;
  }),
  getWords: handleError(async () => {
    const res = await axios.get(baseUrl);
    return res.data;
  }),
  deleteWord: handleError(async id => {
    const res = await axios.delete(baseUrl + id);
    return res.data;
  }),
  createWord: handleError(async payload => {
    const res = await axios.post(baseUrl, payload);
    return res.data;
  }),
  updateWord: handleError(async payload => {
    const res = await axios.put(baseUrl + payload._id, payload);
    return res.data;
  })
};
```

There are a couple of things to be aware of here. Firstly, we're exporting an `api` object that exposes methods corresponding to the endpoints we created in the previous section. These methods will make Ajax calls to our back end, which will carry out the various CRUD operations.

Secondly, due to the asynchronous nature of Ajax, we're using `async/await`, which allows us to wait for the result of an asynchronous operation (such as writing to the database) before continuing with the rest of the code. However, there's a slight caveat here. The normal way of dealing with errors within `async/await` is to wrap everything in a `try/catch` block:

```
export const api = {
  getWord: async id => {
    try {
      await axios.get(baseUrl + id);
      return res.data;
    } catch(error) {
      console.log(error);
    }
  }
  ...
}
```

However, this is very verbose and would make for some bloated code, as we would need to do it for each Ajax call. An alternative approach is to use a higher-order function to handle the error for us.

This higher-order function takes a function as an argument (function a) and returns a function (function b), which, when called, will call function a, passing along any arguments it received. Function b will also chain a `catch` block to the end of function a, meaning that if an error occurs in function a, it will be caught and dealt with.

```
const handleError = fn => (...params) =>
  fn(...params).catch(error => {
    console.log(error);
  });

export const api = {
  getWord: handleError(async id => {
    const res = await axios.get(baseUrl + id);
    return res.data;
  })
  ...
}
```

If you'd like to dive into this some more, check out our article [Flow Control in Modern JS: Callbacks to Promises to Async/Await](#).

Fetching Words from the Database

So let's get some words to display to the user. Open up the `src/views/Words.vue` component and add the following code:

```
<template>
  <div>
    <h1>Words</h1>
    <table id="words" class="ui celled compact table">
      <thead>
        <tr>
          <th>English</th>
          <th>German</th>
          <th colspan="3"></th>
        </tr>
      </thead>
      <tr v-for="(word, i) in words" :key="i">
        <td>{{ word.english }}</td>
        <td>{{ word.german }}</td>
        <td width="75" class="center aligned">Show</td>
        <td width="75" class="center aligned">Edit</td>
        <td width="75" class="center aligned">Destroy</td>
      </tr>
    </table>
  </div>
</template>

<script>
import { api } from '../helpers/helpers';

export default {
  name: 'words',
  data() {
    return {
      words: []
    };
  },
  async mounted() {
    this.words = await api.getWords();
  }
};
</script>
```

As you can see, we're importing our API to perform operations against the back end, then in the component's `mounted` lifecycle hook we're calling its `getWords` method to fetch all of the words from the database. As the component has a `words` property on its data object, we can await the answer from the back end, then insert the response into `this.words` and the component will update.

Give it a try. Use Compass or Postman to insert a word into the database, then observe that it's displayed on the page at `/words`.

Displaying a Single Word

Let's now add the functionality to display a single word from the list of words. In `src/views/Words.vue`, replace:

```
<td width="75" class="center aligned">Show</td>
```

with:

```
<td width="75" class="center aligned">
  <router-link :to="{ name: 'show', params: { id: word._id }}">Show</router-link>
</td>
```

This creates a link to the show route which we defined in our `src/router.js` file. It uses the current word's `_id` property to build the URL, resulting in something like `http://localhost:8081/words/5bed4462bbd0ff553fb008d4`.

Now open up `src/views/Show.vue` and add the following code:

```
<template>
  <div>
    <h1>Show Word</h1>

    <div class="ui labeled input fluid">
      <div class="ui label">
        <i class="germany flag"></i> German
      </div>
      <input type="text" readonly :value="word.german"/>
    </div>
    <div class="ui labeled input fluid">
      <div class="ui label">
        <i class="united kingdom flag"></i> English
      </div>
      <input type="text" readonly :value="word.english"/>
    </div>
    <div class="actions">
      <router-link :to="{ name: 'edit', params: { id: this.$route.params.id }}">
        Edit word
      </router-link>
    </div>
  </div>
</template>

<script>
import { api } from '../helpers/helpers';

export default {
  name: 'show',
  data() {
    return {
      word: ''
    };
  },
  async mounted() {
    this.word = await api.getWord(this.$route.params.id);
  }
};
</script>

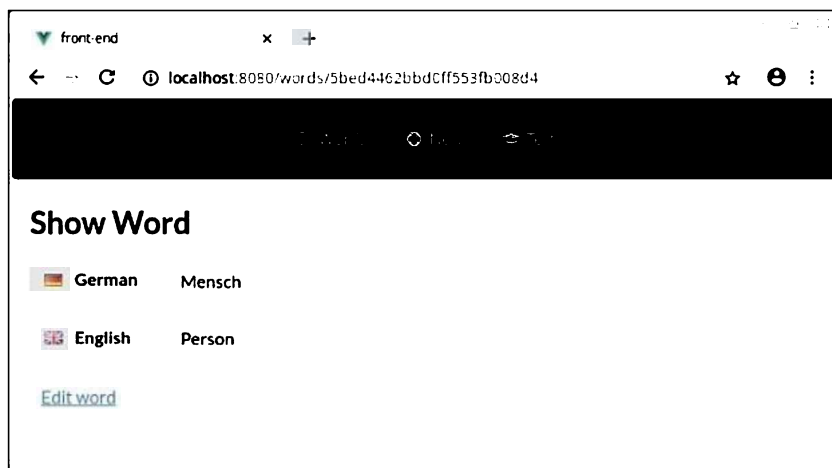
<style scoped>
.actions a {
  display: block;
  text-decoration: underline;
}
```

```
margin: 20px 10px;
}
</style>
```

When mounted, this component will use the same technique as before to get a single word from the database. It identifies the word in question by extracting its ID from the URL using `this.$route.params.id`. This is possible because of how we declared the route in `src/router.js`. Here we used path: `"/words/:id"` which means that anything after `/words/` should be available to us as `$route.params.id`.

Also take note of the `scoped` attribute on the `<style>` tag. This means that any styles declared here are applied to this component only.

If you run the app at this point, you should be able to see a list of words contained in the database under `/words` and view individual words by clicking on `Show`.



You'll notice there's an `Edit word` link in the screenshot above. Let's implement that now.

Editing a Word

Open `src/views/Edit.vue` and add the following:

```
<template>
  <div>
    <h1>Edit Word</h1>
    <word-form :word=this.word></word-form>
  </div>
</template>

<script>
import WordForm from '../components/WordForm.vue';
import { api } from '../helpers/helpers';

export default {
  name: 'edit',
  components: {
    'word-form': WordForm
  },
  data: function() {
    return {
      word: {}
    };
  },
  async mounted() {
    this.word = await api.getWord(this.$route.params.id);
  }
};
</script>
```

There's something slightly different going on in this component. If you look within the `<template>` tag, you'll notice that we're including another component, `WordForm`, and passing it the value of `this.word` as a prop. The value of `this.word` is being retrieved within the mounted lifecycle hook, just as we've done in the other components. Also notice that we're declaring the child component that the `Edit` component requires, like so:

```
components: {
  "word-form": WordForm
}
```

So why are we including a `WordForm` component and handing the word to be edited off to that? Well, the reason is reusability. By structuring things this way, we'll be able to use the same `WordForm` component in a little while when we come to implement the functionality to create a new word.

Let's have a look at the `WordForm` component. Open up `src/components/WordForm.vue` and add the following code:

```
<template>
  <form action="#" @submit.prevent="onSubmit">
    <p v-if="errorsPresent" class="error">Please fill out both fields!</p>

    <div class="ui labeled input fluid">
      <div class="ui label">
        <i class="germany flag"></i> German
      </div>
      <input type="text" placeholder="Enter word..." v-model="word.german" />
    </div>

    <div class="ui labeled input fluid">
      <div class="ui label">
        <i class="united kingdom flag"></i> English
      </div>
      <input type="text" placeholder="Enter word..." v-model="word.english" />
    </div>

    <button class="positive ui button">Submit</button>
  </form>
</template>

<script>
export default {
  name: 'word-form',
  props: {
    word: {
      type: Object,
      required: false
    }
  },
  data() {
    return {
      errorsPresent: false
    };
  },
  methods: {
    onSubmit: function() {
      console.log(`English: ${this.word.english}`);
      console.log(`German: ${this.word.german}`);
    }
  }
};
</script>

<style scoped>
.error {
  color: red;
}
```

```
}  
</style>
```

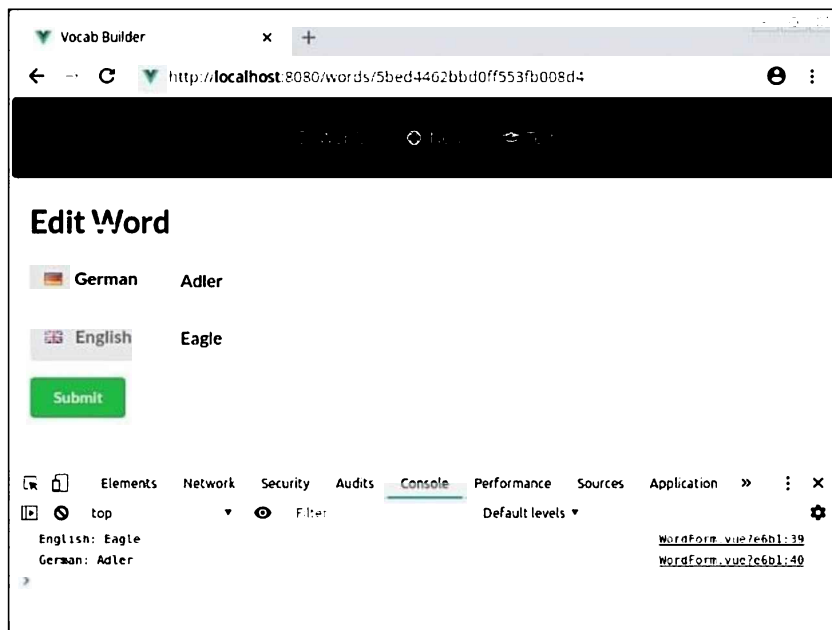
There's quite a bit going on here. Firstly, we have a form tag and we've bound the submit event to the `onSubmit` method we've defined in our methods object. We're also using the `prevent` modifier to prevent the browser's default action—in this case, submitting the form.

Next comes a paragraph tag to display an error message if either of the input fields is empty. We use `v-if` to only display this if the `errorsPresent` property evaluates to something truthy.

Then we have our form with two input fields, one for the English word and one for its German equivalent. We're using `v-model` to set up two-way data binding between the `english` and `german` properties of the prop we're passed and the input fields.

In the `<script>` section, we then declare that the `word` prop is of type `Object`, and that it isn't required (because we'll also use this form to create a new word). The data object declares the aforementioned `errorsPresent` property and our method object, the `onSubmit` method. All we're doing in this method is logging out the value of whatever has been entered into the inputs.

At this point, if you visit the *Edit word* view, you should see the correct word displayed in the fields. And when you click *Submit*, you should see the value of both input fields logged to the console.



Finally, we need to expand our `onSubmit` method to do something other than log the values of the inputs to the console. Alter it like so:

```
onSubmit: function() {  
  if (this.word.english === '' || this.word.german === '') {  
    this.errorsPresent = true;  
  } else {  
    this.$emit('createOrUpdate', this.word);  
  }  
}
```

This checks that both fields have a value. If not, it flags the error, otherwise it emits an event `createOrUpdate` which we can listen for in the parent component (in this case, `Edit`). I've called it `createOrUpdate`, as the corresponding functionality in the parent component will be doing one of these two things.

Back in `src/views/Edit.vue`, we need to listen for and respond to the `createOrUpdate` event. Change:

```
<word-form :word=this.word></word-form>
```

to:

```
<word-form @createOrUpdate="createOrUpdate" :word=this.word></word-form>
```

And implement the update functionality in the methods object:


```

methods: {
  createOrUpdate: async function(word) {
    await api.updateWord(word);
    alert('Word updated successfully!');
    this.$router.push(`/words/${word._id}`);
  }
}

```

Now when you try to update a word, it will check that neither field is blank, and if not, update the word, before alerting a success message and redirecting you to the newly updated word.

As a final touch to this section, hop back into `src/views/Words.vue` and change:

```
<td width="75" class="center aligned">Edit</td>
```

to:

```

<td width="75" class="center aligned">
  <router-link :to="{ name: 'edit', params: { id: word._id }}">Edit</router-link>
</td>

```

Adding Flash Messages

Okay, so our app is looking pretty good, but using alerts to inform the user that something happened is a jarring experience. Let's use flash messages instead. These are nicely styled, configurable messages which won't block the execution of JavaScript in the browser.

The first thing we'll need to do is to tell our app to use them. To this end, hop back into `src/helpers/helpers.js` and make sure the top of the file reads like so:

```

import axios from 'axios';
import Vue from 'vue';
import VueFlashMessage from 'vue-flash-message';
import 'vue-flash-message/dist/vue-flash-message.min.css';

Vue.use(VueFlashMessage, {
  messageOptions: {
    timeout: 3000,
    pauseOnInteract: true
  }
});

const vm = new Vue();
const baseUrl = 'http://localhost:3000/words/';

const handleError = fn => (...params) =>
  fn(...params).catch(error => {
    vm.flash(`${error.response.status}: ${error.response.statusText}`, 'error');
  });

export const api = { ... };

```

There are a few things going on here. We start by importing `Vue` and `VueFlashMessages`. We then tell `Vue` to use the `VueFlashMessages` plugin and pass it a couple of options. Finally, we create a new instance of `Vue`, as we also want to use the flash messages from within the helper file. You can see this happening within our `handleError` function in the call to `vm.flash()`.

So that we can actually display the messages, let's add a `FlashMessage` component to `src/App.vue`:

```

<div class="ui text container">
  <flash-message class="myFlash"></flash-message>

  <div class="ui one column grid">
    <div class="column">
      <router-view />
    </div>
  </div>
</div>

```

```
</div>
```

To test out whether it's working, try changing the value of the `baseUrl` variable in `src/helpers/helpers.js` to something non-existent. If all has gone according to plan, you should see a nicely formatted error message on the right hand of the screen when you visit any of our routes.

Finally, to make a flash message appear when a word was successfully edited, go back to `src/views/Edit.vue` and change:

```
alert("Word updated successfully!");
```

to:

```
this.flash('Word updated successfully!', 'success');
```

Creating a New Word

We've almost implemented all of our CRUD functionality. Next up is giving users the opportunity to add words of their own to our database. Open up `src/views/New.vue` and add the following:

```
<template>
  <div>
    <h1>New Word</h1>
    <word-form @createOrUpdate="createOrUpdate"></word-form>
  </div>
</template>
```

```
<script>
import WordForm from '../components/WordForm.vue';
import { api } from '../helpers/helpers';
```

```
export default {
  name: 'new-word',
  components: {
    'word-form': WordForm
  },
  methods: {
    createOrUpdate: async function(word) {
      const res = await api.createWord(word);
      this.flash('Word created', 'success');
      this.$router.push(`/words/${res._id}`);
    }
  }
};
</script>
```

This should hopefully look familiar to you, as it's quite similar to the `Edit` component. It also uses the `WordForm` child component, but doesn't pass it a prop. And, like the `Edit` component, it also listens for the `createOrUpdate` event. When it receives this event, it creates a new word with the given input and redirects to the newly created word.

Before we can use it, however, there are a couple of changes that need to be made to the `WordForm` component. Head back to `src/components/WordForm.vue` and expand the `props` object like so:

```
props: {
  word: {
    type: Object,
    required: false,
    default: () => {
      return {
        english: '',
        german: ''
      };
    }
  }
},
```

Here, we're initializing both values as empty strings.

Now when you head to <http://localhost:8080/words/new> you should be able to create new words.

Destroying a Word

We won't need a separate component to destroy a word, as there's no view involved. The action takes place in `src/views/Words.vue`:

First change:

```
<td width="75" class="center aligned">Destroy</td>
```

to:

```
<td width="75" class="center aligned" @click.prevent="onDestroy(word._id)">
  <a :href="`/words/${word._id}`">Destroy</a>
</td>
```

This will intercept the click on the table cell, prevent the browser's default action, then call the `onDestroy` method. Let's implement that now:

```
methods: {
  async onDestroy(id) {
    const sure = window.confirm('Are you sure?');
    if (!sure) return;

    await api.deleteWord(id);
    this.flash('Word deleted successfully!', 'success');
    const newWords = this.words.filter(word => word._id !== id);
    this.words = newWords;
  }
},
```

To make sure the user really intended to delete a word, we use a dialogue to ask them to confirm the delete. If the user clicks *OK*, a request is sent to the API to delete the word. We then flash a message to let them know that the delete was successful before removing the word from the list of words in the `data` object. This will ensure that the UI updates.

And with that, our CRUD app is fully functional. Yay!

Taking it Further

By way of a final touch, let's add the possibility for users to test themselves on the words stored in the database.

Open up `src/views/Test.vue` and add the following code:

```
<template>
  <div>
    <h1>Test</h1>

    <div v-if="words.length < 5">
      <p>You need to enter at least five words to begin the test</p>
    </div>
    <div v-else>
      <vocab-test :words="words"></vocab-test>
    </div>
  </div>
</template>

<script>
import { api } from '../helpers/helpers';
import VocabTest from '../components/VocabTest.vue';

export default {
  name: 'test',
  components: {
    'vocab-test': VocabTest
  }
}
```

```

    },
    data() {
      return {
        words: []
      };
    },
    async mounted() {
      this.words = await api.getWords();
    }
  };
</script>

```

Here, we're waiting until the component has been mounted, then fetching all of our words from the database. If there are less than five, we display a message that there aren't enough words to begin the test. Otherwise, we load the `VocabTest` component and pass it our list of words as a prop.

Now all that remains is to code the test itself. Open up `src/components/VocabTest.vue` and add the following:

```

<template>
  <div>
    <h2>Score: {{ score }} out of {{ this.words.length }}</h2>

    <form action="#" @submit.prevent="onSubmit">
      <div class="ui labeled input fluid">
        <div class="ui label">
          <i class="germany flag"></i> German
        </div>
        <input type="text" readonly :disabled="testOver" :value="currWord.german"/>
      </div>
      <div class="ui labeled input fluid">
        <div class="ui label">
          <i class="united kingdom flag"></i> English
        </div>
        <input type="text" placeholder="Enter word..." v-model="english" :disabled="testOver" autocomplete="off"/>
      </div>

      <button class="positive ui button" :disabled="testOver">Submit</button>
    </form>

    <p :class="['results', resultClass]">
      <span v-html="result"></span>
    </p>
  </div>
</template>

<script>
export default {
  name: 'vocab-test',
  props: {
    words: {
      type: Array,
      required: true
    }
  },
  data() {
    return {
      randWords: [...this.words.sort(() => 0.5 - Math.random())],
      incorrectGuesses: [],
      result: '',
      resultClass: '',
      english: '',
      score: 0,
      testOver: false
    }
  }
}

```

```

    });
  },
  computed: {
    currWord: function() {
      return this.randWords.length ? this.randWords[0] : '';
    }
  },
  methods: {
    onSubmit: function() {
      if (this.english === this.currWord.english) {
        this.flash('Correct!', 'success', { timeout: 1000 });
        this.score += 1;
      } else {
        this.flash('Wrong!', 'error', { timeout: 1000 });
        this.incorrectGuesses.push(this.currWord.german);
      }

      this.english = '';
      this.randWords.shift();

      if (this.randWords.length === 0) {
        this.testOver = true;
        this.displayResults();
      }
    },
    displayResults: function() {
      if (this.incorrectGuesses.length === 0) {
        this.result = 'You got everything correct. Well done!';
        this.resultClass = 'success';
      } else {
        const incorrect = this.incorrectGuesses.join(', ');
        this.result = `<strong>You got the following words wrong:</strong> ${incorrect}`;
        this.resultClass = 'error';
      }
    }
  }
};
</script>

<style scoped>
.results {
  margin: 25px auto;
  padding: 15px;
  border-radius: 5px;
}

.error {
  border: 1px solid #ebccd1;
  color: #a94442;
  background-color: #f2dede;
}

.success {
  border: 1px solid #d6e9c6;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

```

This guide is already very long, so I won't explain what every line of the code does. In a nutshell, however, we're receiving the word list, words, as a prop (this is an array of objects). We then declare some data properties, mostly related to displaying the test. The `randWords` property makes use of the spread operator and an arrow function to sort the words into a random order.

We're declaring `currWord` as a computed property, which will return the next word in the list of random words. I've used a computed property, as in the `onSubmit` method I'm removing the first item of the `randWords` array using `shift()` every time the user makes a guess. Mutating the array like this will cause `currWord` to be recalculated and the test will update.

Finally, the `onSubmit` method and the `displayResults` method handle form submission (when the user makes a guess) and, as you might have guessed, displaying the results.

Conclusion

This has been a long tutorial, so congratulations if you made it this far. I hope you now have a better understanding of how to set up a Vue.js front end to consume a Node API which can then persist data to a database.

And please don't forget, the complete code is available on [GitHub](#).

Chapter 2: Creating Beautiful Charts Using Vue.js Wrappers for Chart.js

by Yomi Eluwande

Charts are an important part of modern websites and applications. They help to present information that can't be easily represented in text. Charts also help to make sense of data that would ordinarily not make sense in a textual format by presenting them in a view that's easy to read and understand.

In this guide, I'll show you how to represent data in the form of various types of chart with the help of Chart.js and Vue.js. Chart.js is a simple yet flexible JavaScript charting library for developers and designers that allows drawing of different kinds of charts by using the HTML5 canvas element. A good refresher on Chart.js can be read [here](#).

Vue.js is a progressive JavaScript framework, which we'll use alongside Chart.js to demonstrate the chart examples. If you're new to Vue.js, there's an awesome primer on using Vue.js on [SitePoint](#). We'll also be using Vue CLI to scaffold a Vue.js project for the demo we're going to build.

Charts, Charts, Charts

There's an awesome collection of Vue wrappers for various charting libraries on the [awesome-vue](#) repo on GitHub. However, as we're concentrating on Chart.js, we'll be looking at the following wrappers for that library:

- vue-charts
- vue-chartjs
- vue-chartkick

We'll be using these wrappers to demonstrate how to create different types of charts and also touch on the unique features that each of them offers.

Source Code

The source code for this tutorial is available on [GitHub](#).

Scaffolding the Project with Vue CLI

Let's get started by installing Vue CLI with the following command:

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

Once that's done, we can then get started with scaffolding a project by typing:

```
vue create chart-js-demo
```

This will open a wizard to walk you through creating a new Vue app. Answer the questions as follows:

```
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, Router, Linter
? Use history mode for router? Yes
? Pick a linter / formatter config: ESLint with error prevention only
? Pick additional lint features: Lint on save
? Where do you prefer placing config for Babel etc.? In dedicated config files
? Save this as a preset for future projects? No
```

Now let's install Chart.js, as well as the various wrappers we need for our app:

```
cd chart-js-demo
npm install chart.js chartkick hchs-vue-charts vue-chartjs vue-chartkick
```

The --save Flag

If you use npm 5, there's no need for the `--save` flag anymore, as all packages are automatically saved now. [Read more about that here](#).

Let's test what we have so far and run our application:

```
npm run serve
```

If all has gone well, you should be able to open localhost:8080 and see the standard welcome page.

Basic Setup

Next, we'll need to add a view for each wrapper. Create the following files in `src/views`:

- `VueChartJS.vue`
- `VueChartKick.vue`
- `VueCharts.vue`

```
touch src/views/{VueChartJS.vue,VueChartKick.vue,VueCharts.vue}
```

Then create the following files in `src/components`:

- `LineChart.vue`
- `BarChart.vue`
- `BubbleChart.vue`
- `Reactive.vue`

```
touch src/components/{LineChart.vue,BarChart.vue,BubbleChart.vue,Reactive.vue}
```

These correspond to the type of charts we'll be using.

Finally, you can delete the `About.vue` file in `src/views`, the `HelloWorld.vue` file in `src/components`, as well as the `assets` folder in `src`. We'll not be needing those.

The contents of your `src` directory should look like this:

```
.
├── App.vue
├── components
│   ├── BarChart.vue
│   ├── BubbleChart.vue
│   ├── LineChart.vue
│   └── Reactive.vue
├── main.js
├── router.js
└── views
    ├── Home.vue
    ├── VueChartJS.vue
    ├── VueChartKick.vue
    └── VueCharts.vue
```

Adding Routes

The next thing we want to do is create the different routes in which we can view the charts for each of the wrappers we created above. We would like to have a `/charts` route to display charts made with the `vue-charts` wrapper, `/chartjs` to display charts made with the `vue-chartjs` wrapper, and lastly `/chartkick` to display charts made with the `vue-chartkick` wrapper. We'll also leave the `/` route in place to display the `Home` view.

Navigate to the `src` folder of the app and open up the `router.js` file. Replace the contents of that file with this:

```
import Vue from 'vue'
import Router from 'vue-router'
import Home from '@/views/Home'
import VueChartJS from '@/views/VueChartJS'
import VueChartKick from '@/views/VueChartKick'
import VueCharts from '@/views/VueCharts'
```

```
Vue.use(Router)
```

```
export default new Router({
  routes: [
    {
```



```

    path: '/',
    name: 'Home',
    component: Home
  },
  {
    path: '/chartjs',
    name: 'VueChartJS',
    component: VueChartJS
  },
  {
    path: '/chartkick',
    name: 'VueChartKick',
    component: VueChartKick
  },
  {
    path: '/charts',
    name: 'VueCharts',
    component: VueCharts
  }
]
}))

```

Here we imported the Vue components that we created above. Components are one of the most powerful features of Vue. They help us extend basic HTML elements to encapsulate reusable code. At a high level, components are custom elements that Vue's compiler attaches behavior to.

Lastly, we defined the routes and components which will serve the different pages we need to display the different charts.

Adding Navigation

We can define our navigation to be used across components in `src/App.vue`. We'll also add some basic styling:

```

<template>
  <div id="app">
    <div id="nav">
      <ul>
        <li><router-link to="/">Home</router-link></li>
        <li><router-link to="/chartjs">vue-chartjs</router-link></li>
        <li><router-link to="/charts">vue-charts</router-link></li>
        <li><router-link to="/chartkick">vue-chartkick</router-link></li>
      </ul>
    </div>
    <router-view />
  </div>
</template>

<style>
#app {
  font-family: 'Avenir', Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
}

#nav {
  padding: 30px;
  height: 90px;
}

#nav a {
  font-weight: bold;
  color: #2c3e50;
  text-decoration: underline;
}

```

```

}

#nav a.router-link-exact-active {
  color: #42b983;
  text-decoration: none;
}

#nav ul {
  list-style-type: none;
  padding: 0;
}

#nav ul li {
  display: inline-block;
  margin: 0 10px;
}

h1 {
  font-size: 1.75em;
}

h2 {
  line-height: 2.5em;
  font-size: 1.25em;
}
</style>

```

Nothing too revolutionary here. Note the use of the `<router-view />` component, which is where we'll display our views.

Home Component

As mentioned above, the Home component serves as the default (`/`) route. Open it up and replace the content with the code block below:

```

<template>
  <section class="hero">
    <div class="hero-body">
      <div class="container">
        <h1>Creating Beautiful Charts Using Vue.js Wrappers For Chart.js</h1>
        <h2>
          Read the article on SitePoint:
          <a href="https://www.sitepoint.com/creating-beautiful-charts-vue-chart-js/">
            https://www.sitepoint.com/creating-beautiful-charts-vue-chart-js/
          </a>
        </h2>
        <h3>
          Download the repo from GitHub:
          <a href="https://github.com/sitepoint-editors/vue-charts">
            https://github.com/sitepoint-editors/vue-charts
          </a>
        </h3>
      </div>
    </div>
  </section>
</template>

<script>
export default {
  name: 'home'
}
</script>

```

Adding Bulma

One more thing before we start adding charts. Let's add the Bulma CSS framework to the app. This should make things easier when it

comes to CSS.

Open the `public/index.html` file and add the following inside the head tag:

```
<link href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.7.2/css/bulma.min.css" rel="stylesheet">
```

Now if you visit `localhost:8080`, you should be able to navigate around the shell of the app.

Finally we can move on to creating charts!

Making Charts with vue-chartjs

`vue-chartjs` is a `Chart.js` wrapper that allows us to easily create reusable chart components. Reusability means we can import the base chart class and extend it to create custom components.

With that being said, we'll be demonstrating four types of charts that can be built using `vue-chartjs`: a line chart, a bar chart, a bubble chart, and a bar chart that demonstrates reactivity (the chart updates whenever there's a change in the data set).

Line Chart

To create a line chart, we'll create a component to render this type of chart only. Open the `LineChart.vue` component file inside the `src/components` folder and add the following code:

```
<script>
  //Importing Line class from the vue-chartjs wrapper
  import { Line } from 'vue-chartjs'

  //Exporting this so it can be used in other components
  export default {
    extends: Line,
    data () {
      return {
        datacollection: {
          //Data to be represented on x-axis
          labels: ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September'],
          datasets: [
            {
              label: 'Data One',
              backgroundColor: '#f87979',
              pointBackgroundColor: 'white',
              borderWidth: 1,
              pointBorderColor: '#249EBF',
              //Data to be represented on y-axis
              data: [40, 20, 30, 50, 90, 10, 20, 40, 50, 70, 90, 100]
            }
          ]
        }
      },
    },
    //Chart.js options that controls the appearance of the chart
    options: {
      scales: {
        yAxes: [{
          ticks: {
            beginAtZero: true
          },
          gridLines: {
            display: true
          }
        }],
        xAxes: [ {
          gridLines: {
            display: false
          }
        } ]
      }
    }
  },

```

```

        legend: {
          display: true
        },
        responsive: true,
        maintainAspectRatio: false
      }
    },
    mounted () {
      //renderChart function renders the chart with the datacollection and options object.
      this.renderChart(this.datacollection, this.options)
    }
  }
}
</script>

```

Let's discuss what the code above is doing. The first thing we did was import the chart class we needed (in this case, `Line`) from the `vue-chartjs` and exported it.

The `data` property contains a `datacollection` object which contains all the information we'll need to build the line chart. This includes the `Chart.js` configuration, like labels, which will be represented on the x-axis, the datasets, which will be represented on the y-axis, and the `options` object, which controls the appearance of the chart.

The `mounted` function calls `renderChart()` which renders the chart with the `datacollection` and `options` objects passed in as parameters.

Now, let's open the `views/VueChartJS.vue` file and add in the following code:

```

<template>
  <section class="container">
    <h1>Demo examples of vue-chartjs</h1>
    <div class="columns">
      <div class="column">
        <h3>Line Chart</h3>
        <line-chart></line-chart>
      </div>
      <div class="column">
        <h3>Bar Chart</h3>
        <!--Bar Chart example-->
      </div>
    </div>
    <div class="columns">
      <div class="column">
        <h3>Bubble Chart</h3>
        <!--Bubble Chart example-->
      </div>
      <div class="column">
        <h3>Reactivity - Live update upon change in datasets</h3>
        <!--Reactivity Line Chart example-->
      </div>
    </div>
  </section>
</template>

<script>
  import LineChart from '@/components/LineChart'
  import BarChart from '@/components/BarChart'
  import BubbleChart from '@/components/BubbleChart'
  import Reactive from '@/components/Reactive'

  export default {
    name: 'VueChartJS',
    components: {
      LineChart,

```

```

    BarChart,
    BubbleChart,
    Reactive
  }
}
</script>

```

The `VueChartJS.vue` file contains the `template` section which holds the HTML code, a `script` section which holds the JavaScript code, and a `style` section which holds the CSS code.

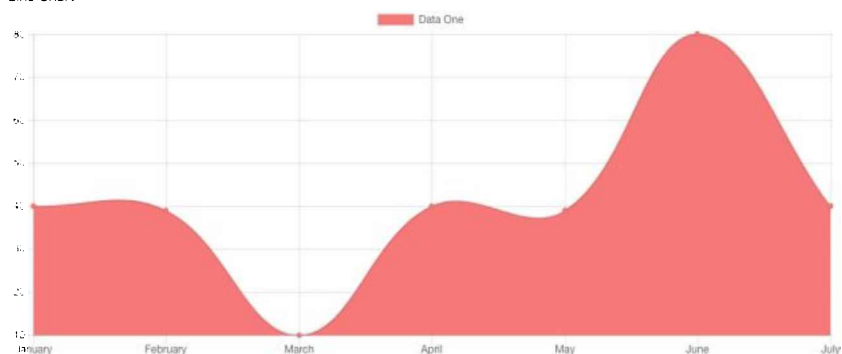
Inside the `template` section, we used the `columns` class from `Bulma` to create a layout that has two columns and two rows. We also added a `line-chart` component which will be created in the `script` section.

Inside the `script` section, we imported the `.vue` files we created earlier and referenced them in the `components` object. This means we can then use them in the HTML code like this: `line-chart`, `bar-chart`, `bubble-chart` and `reactive`.

Now if you navigate to `/chartjs`, the line chart should display on that page.

Live Code

Line Chart



See the [Pen vue-chart.js - line chart](#)

Bar Chart

For the next chart, we'll create a custom component that helps to render bar charts only. Open the `BarChart.vue` component file inside the `src/components` folder and add the following code:

```

<script>
  //Importing Bar class from the vue-chartjs wrapper
  import { Bar } from 'vue-chartjs'
  //Exporting this so it can be used in other components
  export default {
    extends: Bar,
    data () {
      return {
        datacollection: {
          //Data to be represented on x-axis
          labels: ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September'],
          datasets: [
            {
              label: 'Data One',
              backgroundColor: '#f87979',
              pointBackgroundColor: 'white',
              borderWidth: 1,
              pointBorderColor: '#249EBF',
              //Data to be represented on y-axis
              data: [40, 20, 30, 50, 90, 10, 20, 40, 50, 70, 90, 100]
            }
          ]
        }
      }
    }
  },

```

```

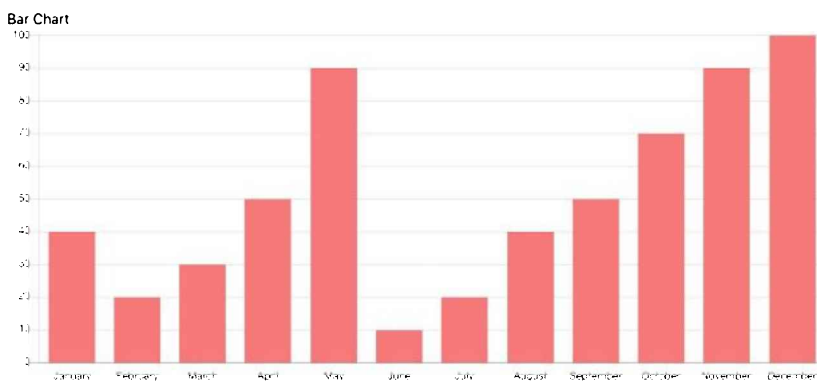
//Chart.js options that controls the appearance of the chart
options: {
  scales: {
    yAxes: [{
      ticks: {
        beginAtZero: true
      },
      gridLines: {
        display: true
      }
    }],
    xAxes: [ {
      gridLines: {
        display: false
      }
    } ]
  },
  legend: {
    display: true
  },
  responsive: true,
  maintainAspectRatio: false
}
},
mounted () {
  //renderChart function renders the chart with the datacollection and options object.
  this.renderChart(this.datacollection, this.options)
}
}
</script>

```

The code above works similarly to the one in the `LineChart.vue` file. The only difference here is that the `Bar` class is imported and extended instead of `Line`.

One more thing: before we can see our bar chart, we need to edit the `VueChartJS.vue` file and replace `<!--Bar Chart example-->` with `<bar-chart></bar-chart>`. We're simply using the `Bar` component we created inside in our HTML template.

Live Code



See the [Pen vuechart-js - bar chart](#).

Bubble Chart

For the bubble chart, we'll create a component that helps to render bubble charts only.

Bubble Charts

Bubble charts use bubbles/circles to display data in a three dimension method (x, y, r). x is used to display the horizontal axis data, y is

used to display the vertical data, and `r` is used to display the size of the individual bubbles.

Open the `BubbleChart.vue` component file inside the `src/components` folder and add the following code:

```
<script>
//Importing Bubble class from the vue-chartjs wrapper
import { Bubble } from 'vue-chartjs'
//Exporting this so it can be used in other components
export default {
  extends: Bubble,
  data () {
    return {
      datacollection: {
        //Data to be represented on x-axis
        labels: ['Data'],
        datasets: [
          {
            label: 'Data One',
            backgroundColor: '#f87979',
            pointBackgroundColor: 'white',
            borderWidth: 1,
            pointBorderColor: '#249EBF',
            //Data to be represented on y-axis
            data: [
              {
                x: 100,
                y: 0,
                r: 10
              },
              {
                x: 60,
                y: 30,
                r: 20
              },
              {
                x: 40,
                y: 60,
                r: 25
              },
              {
                x: 80,
                y: 80,
                r: 50
              },
              {
                x: 20,
                y: 30,
                r: 25
              },
              {
                x: 0,
                y: 100,
                r: 5
              }
            ]
          }
        ]
      },
      //Chart.js options that controls the appearance of the chart
      options: {
        scales: {
          yAxes: [{
            ticks: {
```

```

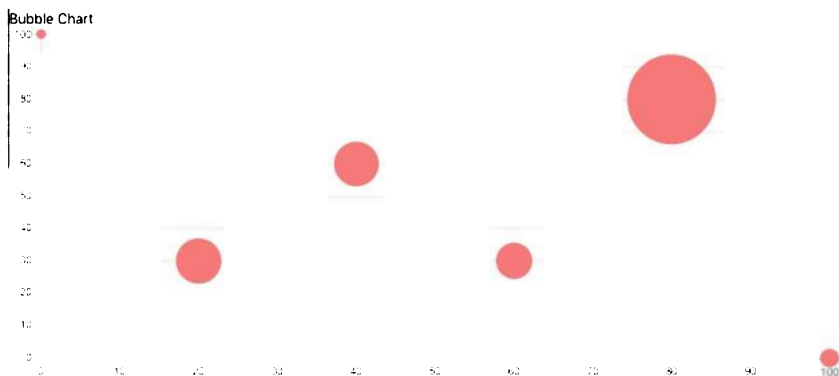
        beginAtZero: true
      },
      gridLines: {
        display: true
      }
    }],
    xAxes: [ {
      gridLines: {
        display: false
      }
    }
  ]
},
legend: {
  display: true
},
responsive: true,
maintainAspectRatio: false
}
}
},
mounted () {
  //renderChart function renders the chart with the datacollection and options object.
  this.renderChart(this.datacollection, this.options)
}
}
</script>

```

The code above works similarly to the code in the `LineChart.vue` and `BarChart.vue` files. The difference here is that the `Bubble` class is imported and extended instead of `Line` or `Bar` and the `datasets` object takes in different values for `x`, `y` and `z`.

We need to do one more thing before we can see our bubble chart—the same thing we did for other charts: edit the `VueChartJS.vue` file and replace `<!--Bubble Chart example-->` with `<bubble-chart></bubble-chart>`.

Live Code



See the Pen [vue-chartjs - bubble chart](#).

Bar Chart That Demonstrates Reactivity

The last example we'll be demonstrating with `vue-chartjs` is how it can be used to make a chart that automatically updates whenever there's a change in the data set. It's important to note that `Chart.js` ordinarily doesn't offer this feature, but `vue-chartjs` does—with the help of either of these two mixins:

- `reactiveProp`
- `reactiveData`

Open the `Reactive.vue` component file inside the `src/components` folder and type in the following code:

```

<script>
  //Importing Bar and mixins class from the vue-chartjs wrapper

```



```

import { Bar, mixins } from 'vue-chartjs'
//Getting the reactiveProp mixin from the mixins module.
const { reactiveProp } = mixins
export default Bar.extend({
  mixins: [ reactiveProp ],
  data () {
    return {
      //Chart.js options that control the appearance of the chart
      options: {
        scales: {
          yAxes: [{
            ticks: {
              beginAtZero: true
            },
            gridLines: {
              display: true
            }
          }
        ],
        xAxes: [ {
          gridLines: {
            display: false
          }
        }
      ],
      legend: {
        display: true
      },
      responsive: true,
      maintainAspectRatio: false
    }
  },
  mounted () {
    // this.chartData is created in the mixin and contains all the data needed to build the chart.
    this.renderChart(this.chartData, this.options)
  }
})
</script>

```

The first thing we did in the code above was to import the Bar and mixins classed from the vue-chartjs wrapper and extend Bar. We also extracted the reactiveProp mixin from the mixins module to be used for reactivity.

The reactiveProp mixin extends the logic of your chart component, automatically creates a prop named chartData, and adds a Vue watcher to this prop. We didn't create a dataset object this time, as all the data needed will be inside the chartData prop.

For this to work, we'll need to edit our VueChartJS.vue even further. Open VueChartJS.vue, and after the components property, add the following code:

```

data () {
  return {
    // instantiating datacollection with null
    datacollection: null
  }
},
created () {
  //anytime the vue instance is created, call the fillData() function.
  this.fillData()
},
methods: {
  fillData () {
    this.datacollection = {
      // Data for the y-axis of the chart
      labels: ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September',

```



```

    dataset: [65, 59, 80, 81, 56, 55, 40]
  }
}

```

And we can easily display a line chart by using `<chartjs-line></chartjs-line>` in our template, a bar chart by using `<chartjs-bar></chartjs-bar>`, and a radar chart by using `<chartjs-radar></chartjs-radar>`.

Before we go on, we need to register `vue-charts` globally. Open the `main.js` inside the `src` folder and add the following lines of code under the existing `import` statements:

```

import 'chart.js'
import 'hchs-vue-charts'

```

```
Vue.use(window.VueCharts)
```

Here we import the `Chart.js` library and `hchs-vue-charts` wrapper from the `node_modules` folder and register it globally with `Vue.use(window.VueCharts)`.

Now let's get started with `vue-charts`. Open up the `VueCharts.vue` file we created earlier and type in the following code:

```

<template>
  <section class="container">
    <h1>Demo examples of vue-charts</h1>
    <div class="columns">
      <div class="column">
        <h3>Line Chart</h3>
        <!--Line Chart Example-->
      </div>
      <div class="column">
        <h3>Bar Chart</h3>
        <!--Bar Chart Example-->
      </div>
    </div>
    <div class="columns">
      <div class="column">
        <h3>Radar Chart</h3>
        <!--Radar Chart Example-->
      </div>
      <div class="column">
        <h3>Data Binding Line Chart</h3>
        <!--Data Binding Line Chart Example-->
      </div>
    </div>
  </section>
</template>

<script>
export default {
  name: 'VueCharts',
  data () {
    return {
      labels: ['January', 'February', 'March', 'April', 'May', 'June', 'July'],
      dataset: [65, 59, 80, 81, 56, 55, 40]
    }
  }
}
</script>

```

In the code block above we created placeholders for charts, and all we have to do is start putting them in.

We then added the labels and data we'll be using for the various charts in the `data` object inside the `script` section.

Let's begin to create the charts now.

Line Chart

To add the line chart, all we have to do is add `<chartjs-line></chartjs-line>` in place of the comment `<!--Line Chart Example-->` in the template section of the `VueCharts.vue` file. We don't need to import any component because `vue-charts` has been declared globally in `main.js`.

Bar Chart

We do the same for bar charts. Add `<chartjs-bar></chartjs-bar>` in place of the comment `<!--Bar Chart Example-->` in the template section of the `VueCharts.vue` file.

Radar Chart

We do the same for radar charts. Add `<chartjs-radar></chartjs-radar>` in place of the comment `<!--Radar Chart Example-->` in the template section of the `VueCharts.vue` file.

Once that's done, you can navigate to `/charts` and check out the charts we just created.

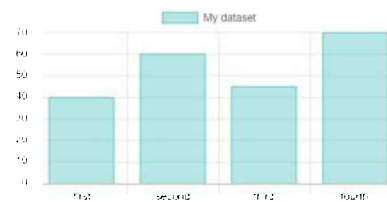
Live Code

Demo examples of vue-charts

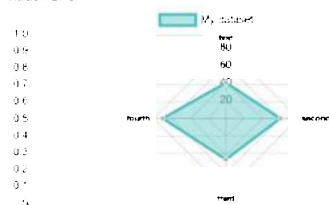
Line Chart



Bar Chart



Radar Chart



Data Binding Line Chart

See the Pen [vue-charts demo](#).

A Data Binding Example with a Line Chart

Data binding in `vue-charts` is similar to that of `vue-chartjs`, although `vue-charts` automatically updates the chart whenever the data set changes.

To test this, we'll create a function named `addData` and add it to the `methods` object of the component:

```
methods: {
  addData () {
    this.dataset.push(this.dataentry)
    this.labels.push(this.datalabel)
    this.datalabel = ''
    this.dataentry = ''
  }
}
```

The `addData` method pushes the current value of the data form input and label form input (which we'll be adding soon) to the current data set. Now let's add the component as well as the form to add new data and a label.

Add the following piece of code in place of the comment `<!--Data Binding Line Chart Example-->`:

```
<form @submit.prevent="addData">
  <input placeholder="Add a Data" v-model="dataentry">
  <input placeholder="Add a Label" v-model="datalabel">
  <button type="submit">Submit</button>
</form>
```

```
<chartjs-line :labels="labels" :data="dataset" :bind="true"></chartjs-line>
```

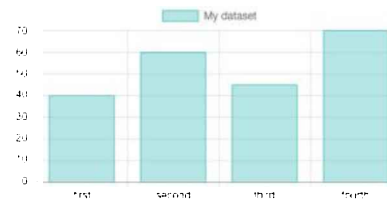
The code above is a form that allows you to type in values of data and label and then submit the form. The chart will render automatically with the latest entry.

Live Code

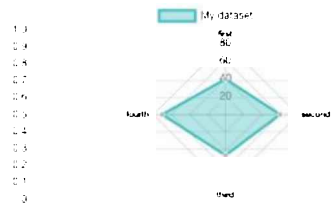
Line Chart



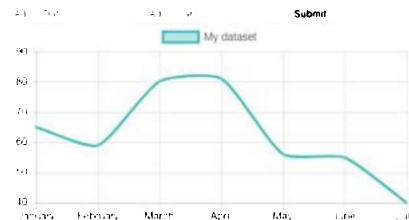
Bar Chart



Radar Chart



Data Binding Line Chart



See the Pen vue-charts data binding.

Making Charts with vue-chartkick

vue-chartkick is a Vue wrapper that allows you to create beautiful charts with one line. We'll be demonstrating the use of this library with four examples. We'll be creating a line chart, bar chart, scatter chart and also use the download feature (downloadable charts).

In terms of handling data needed for the chart data set, vue-chartkick gives us two options. We can do something like `<bar-chart :data="[['Work', 1322], ['Play', 1492]]"></bar-chart>`, in which the data is added inline, or `<line-chart :data="chartData"></line-chart>`, and we declare the chartData array in the Vue data object like this:

```
data () {
  return {
    chartData: [['Jan', 44], ['Feb', 27], ['Mar', 60], ['Apr', 55], ['May', 37], ['Jun', 40], ['Jul', 60]]
  }
}
```

Let's get started with using vue-chartkick. Before we go on, open the main.js file inside the src folder and add the following code:

```
import Chartkick from 'chartkick'
import VueChartkick from 'vue-chartkick'
```

```
Vue.use(VueChartkick, { Chartkick })
```

Here we import the Chartkick library and the vue-chartkick wrapper from node_modules and register it globally with Vue.use(VueChartkick, { Chartkick }).

To start creating charts with vue-chartkick, open up the VueChartKick.vue file we created earlier and add the following code:

```
<template>
  <section class="container">
    <h1>Demo examples of vue-chartkick</h1>
    <div class="columns">
      <div class="column">
        <h3>Line Chart</h3>
        <!--Line Chart example-->
      </div>
      <div class="column">
        <h3>Bar Chart</h3>
      </div>
    </div>
  </section>
</template>
```

```

        <!--Bar Chart example-->
    </div>
</div>
<div class="columns">
    <div class="column">
        <h3>Scatter Chart</h3>
        <!--Scatter chart example-->
    </div>
    <div class="column">
        <h3>Downloadable Line Chart</h3>
        <!--Downloadable line chart-->
    </div>
</div>
</section>
</template>

<script>
    export default {
        name: 'VueChartKick',
        data () {
            return {
                chartData: [['Jan', 44], ['Feb', 27], ['Mar', 60], ['Apr', 55], ['May', 37], ['Jun', 40], ['Jul', 45]]
            }
        }
    }
</script>

```

In the code block above, we again created placeholders for the different types of charts we'll be creating, and all we have to do is start putting them in.

We then created the `chartData` array that holds data we'll be using for the various charts in the data object inside the script section. Let's begin to create the charts now.

Line Chart

Creating a line chart with `vue-chartkick` is very simple and straightforward. All you have to do is bring in the `vue-chartkick` `line-chart` component and also determine what dataset you want to use—something like this: `<line-chart :data="chartData"></line-chart>`. In this case, the `data` prop is set to the `chartData` array in the Vue data object.

Therefore, replace the comment `<!--Line Chart example-->` inside the `VueChartKick.vue` file with `<line-chart :data="chartData"></line-chart>` and the line chart should display nicely now.

Bar Chart

We can create a bar chart in the same way we created the line chart above. But in this case, we'll try something different. Instead of using the `chartData` array as the data set, we'll add the data set inside the props, like this: `<bar-chart :data="[['Work', 1322], ['Play', 1492]]"></bar-chart>`.

Replace the comment `<!--Bar Chart example-->` inside the `VueChartKick.vue` file with `<bar-chart :data="[['Work', 1322], ['Play', 1492]]"></bar-chart>`, and your bar chart should display now.

Scatter Chart

Scatter charts also work like the line and bar charts above. Replace the comment `<!--Scatter Chart example-->` inside the `VueChartKick.vue` file with `<scatter-chart :data="[[174.0, 80.0], [176.5, 82.3], [180.3, 73.6]]"></scatter-chart>` and your scatter chart should display now.

Downloading a Chart

`vue-chartkick` has a pretty neat feature that `vue-chartjs` and `vue-charts` lack: the ability to create downloadable charts. Once a chart has been set to downloadable, users can easily save the chart as an image. So how does this work? Let's check it out with a line chart:

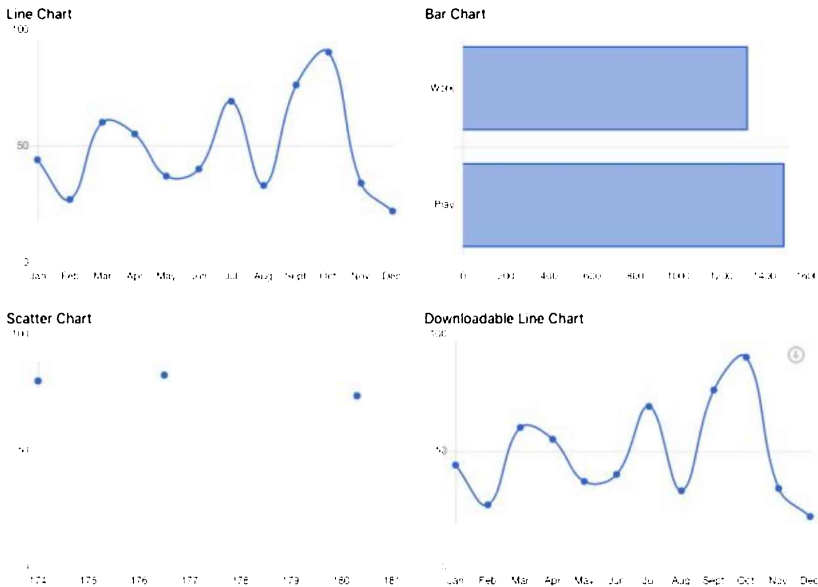
```
<line-chart :data="chartData" :download="true"></line-chart>
```

We set the `:download` prop to `true`, and that means the chart is downloadable. That's all!

Replace the comment `<!--Downloadable line chart-->` inside the `VueChartKick.vue` file with `<line-chart :data="chartData" :download="true"></line-chart>` to see and test the download feature on your app. You can hover over the new chart to see the download button.

Live Code

Demo examples of vue-chartkick



See the Pen vue-chartkick demos.

Comparisons

So now that we've gone through the three Vue.js wrappers for Chart.js, what are the pros and cons of each?

vue-chartjs

`vue-chartjs` is perhaps the most powerful of the three wrappers. It gives so much room for flexibility and also ships with a reactivity feature (the ability to re-render a chart automatically if there's been a change in the data).

A con for the wrapper would be that configuring the chart can be somewhat complex at the beginning, due to the various options available, and also because you have to create an external file to hold the configuration for the wrapper.

vue-charts

`vue-chartjs` is a pretty good Chart.js wrapper. It's very easy to set up, unlike `vue-chartjs`. It doesn't require you to create separate components to process data and render charts. All the data required for the chart can be instantiated in the Vue instance, as seen in the examples above. It also ships with a reactivity feature (the ability to render a chart automatically if there's been a change in the data set) without a need for mixins.

vue-chartkick

`vue-chartkick` is also a pretty good Chart.js wrapper. It may be the wrapper with the easiest path to usage among the three Chart.js wrappers.

The data set for the chart can be added in two different ways in `vue-chartkick`. We can do something like `<bar-chart :data="[['Work', 1322], ['Play', 1492]]"></bar-chart>`, in which the data is added inline, or `<line-chart :data="chartData"></line-chart>`, and we declare the `chartData` array in the Vue data object as seen above in the examples. It also ships with an awesome feature that allows you to download your charts as an image—something that the other wrappers lack.

The only disadvantage of `vue-chartkick` is that it doesn't support reactivity out of the box.

Conclusion

In this guide, I introduced various Vue wrappers for Chart.js, and also demonstrated examples of how to use each of them to create beautiful charts for your next web project.

The source code for this tutorial is available on GitHub, and if you'd like to see a live demo of the tutorial, you can also see that [here](#).

Chapter 3: Build a Real-time Chat App with Pusher and Vue.js

by Michael Wanyoike

Apps that communicate in real time are becoming more and more popular nowadays, as they make for a smoother, more natural user experience.

In this tutorial, we're going to build a real-time chat application using Vue.js powered by ChatKit, a service provided by Pusher. The ChatKit service will provide us with a complete back end necessary for building a chat application on any device, leaving us to focus on building a front-end user interface that connects to the ChatKit service via the ChatKit client package.

Prerequisites

This is an intermediate- to advanced-level tutorial. You'll need to be familiar with the following concepts to follow along:

- Vue.js basics
- Vuex fundamentals
- employing a CSS framework

You'll also need Node installed on your machine. You can do this by downloading the binaries from the official website, or by using a version manager. This is probably the easiest way, as it allows you to manage multiple versions of Node on the same machine.

Finally, you'll need to install Vue CLI globally with the following command:

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

At the time of writing, Node 10.14.1 and Vue CLI 3.2.1 are the latest versions.

About the Project

We're going to build a rudimentary chat application similar to Slack or Discord. The app will do the following:

- have multiple channels and rooms
- list room members and detect presence status
- detect when other users start typing

As mentioned earlier, we're just building the front end. The ChatKit service has a back-end interface that will allow us to manage users, permissions and rooms.

You can find the complete code for this project on GitHub.

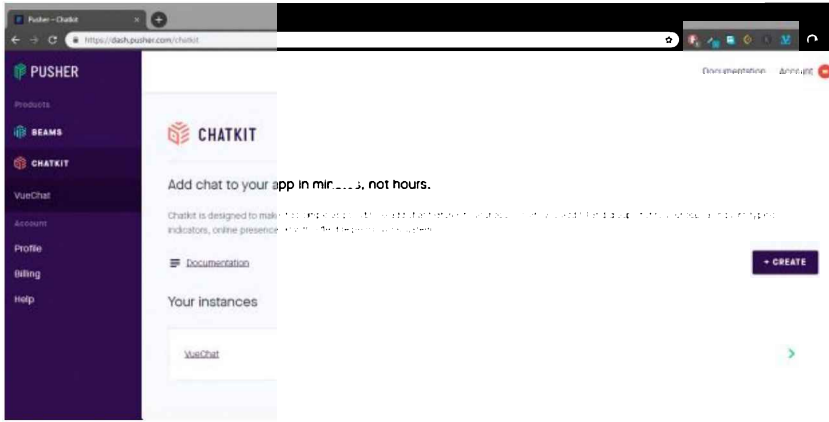
Setting up a ChatKit Instance

Let's create our ChatKit instance, which is similar to a server instance if you're familiar with Discord.

Go to the ChatKit page on Pusher's website and click the *Sign Up* button. You'll be prompted for an email address and password, as well as the option to sign in with GitHub or Google.

Select which option suits you best, then on the next screen fill out some details such as *Name*, *Account type*, *User role* etc.

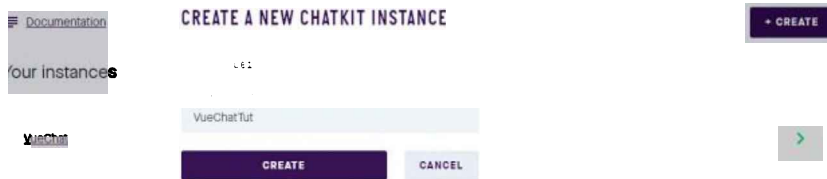
Click *Complete Onboarding* and you'll be taken to the main Pusher dashboard. Here, you should click the ChatKit Product.



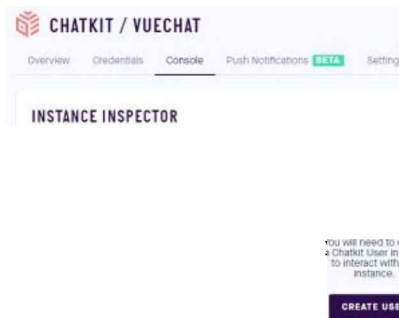
Click the Create button to create a new ChatKit Instance. I'm going to call mine VueChatTut.

Add chat to your app in minutes, not hours.

Chatkit is designed to make it as simple as possible to add chat features to your apps. It lets you add 1-1 and group chat to your app, along with typing indicators, online presence, and a flexible permissions system.



We'll be using the free plan for this tutorial. It supports up to 1,000 unique users, which is more than sufficient for our needs. Head over to the *Console* tab. You'll need to create a new user to get started. Go ahead and click the *Create User* button.



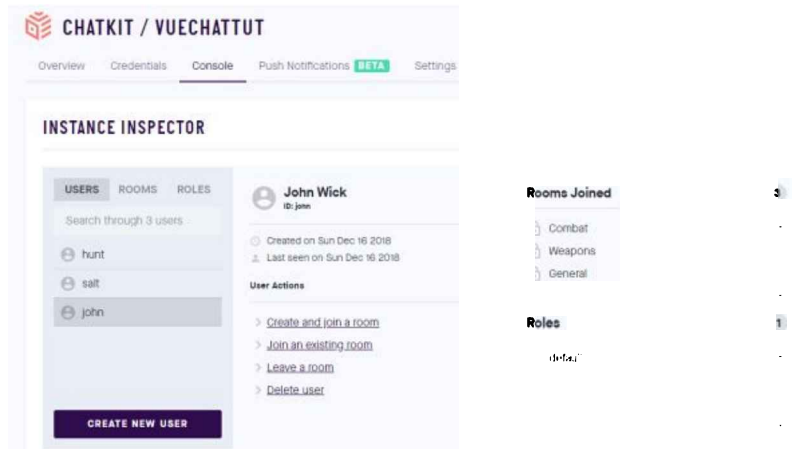
I'm going to call mine "john" (*User Identifier*) and "John Wick" (*Display Name*), but you can name yours however you want. The next part is easy: create the two or more users. For example:

- salt, Evelyn Salt
- hunt, Ethan Hunt

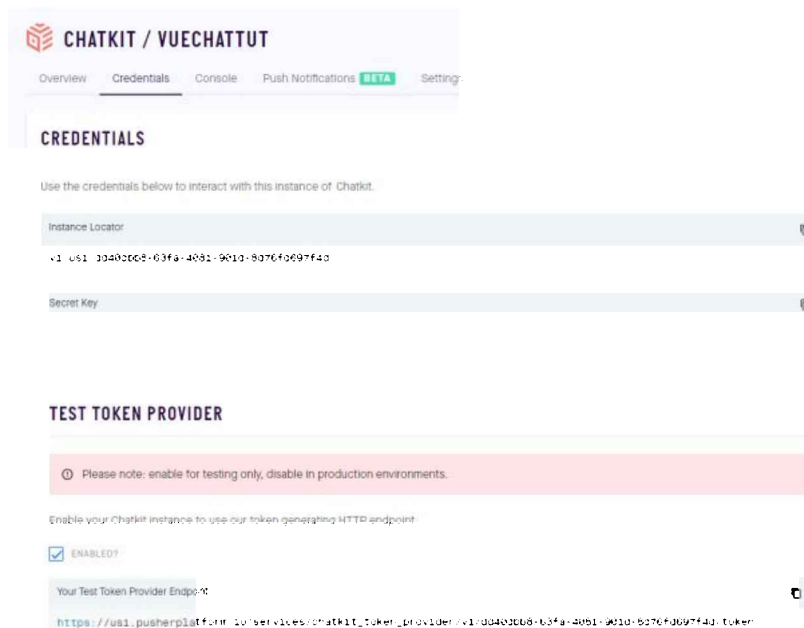
Create three or more rooms and assign users. For example:

- General (john, salt, hunt)
- Weapons (john, salt)
- Combat (john, hunt)

Here's a snapshot of what your *Console* interface should like.



Next, you can go to the *Rooms* tab and create a message using a selected user for each room. This is for testing purposes. Then go to the *Credentials* tab and take note of the *Instance Locator*. We'll need to activate the *Test Token Provider*, which is used for generating our HTTP endpoint, and take a note of that, too.



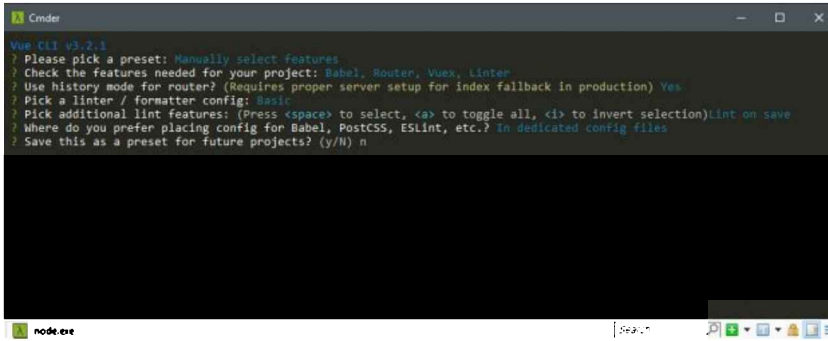
Our ChatKit back end is now ready. Let's start building our Vue.js front end.

Scaffolding the Vue.js Project

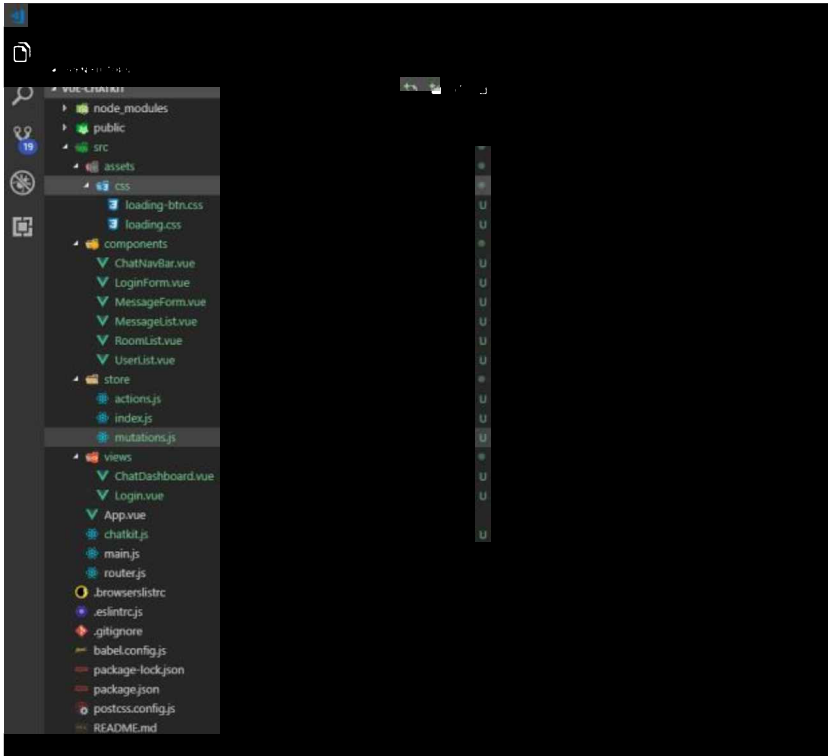
Open your terminal and create the project as follows:

```
vue create vue-chatkit
```

Select *Manually select features* and answer the questions as shown below.



Make doubly sure you've selected Babel, Vuex and Vue Router as additional features. Next, create the following folders and files as follows:



Make sure to create all the folders and files as demonstrated. Delete any unnecessary files that don't appear in the above illustration.

For those of you that are at home in the console, here are the commands to do all that:

```
mkdir src/assets/css
mkdir src/store
```

```
touch src/assets/css/{loading.css,loading-btn.css}
touch src/components/{ChatNavBar.vue,LoginForm.vue,MessageForm.vue,MessageList.vue,RoomList.vue,UserList.vue}
touch src/store/{actions.js,index.js,mutations.js}
touch src/views/{ChatDashboard.vue,Login.vue}
touch src/chatkit.js
```

```
rm src/components/HelloWorld.vue
rm src/views/{About.vue,Home.vue}
rm src/store.js
```

When you're finished, the contents of the `src` folder should look like so:

```
.
├── App.vue
├── assets
│   └── css
```

```

|   |   |   | loading-btn.css
|   |   |   | loading.css
|   |   |   | logo.png
|--- chatkit.js
|--- components
|   |--- ChatNavBar.vue
|   |--- LoginForm.vue
|   |--- MessageForm.vue
|   |--- MessageList.vue
|   |--- RoomList.vue
|   |--- UserList.vue
|--- main.js
|--- router.js
|--- store
|   |--- actions.js
|   |--- index.js
|   |--- mutations.js
|--- views
|   |--- ChatDashboard.vue
|   |--- Login.vue

```

For the `loading-btn.css` and the `loading.css` files, you can find them on the loading.io website. These files are not available in the npm repository, so you need to manually download them and place them in your project. Do make sure to read the documentation to get an idea on what they are and how to use the customizable loaders.

Next, we're going to install the following dependencies:

- `@pusher/chatkit-client`, a real-time client interface for the ChatKit service
- `bootstrap-vue`, a CSS framework
- `moment`, a date and time formatting utility
- `vue-chat-scroll`, which scrolls to the bottom automatically when new content is added
- `vuex-persist`, which saves Vuex state in browser's local storage

```
npm i @pusher/chatkit-client bootstrap-vue moment vue-chat-scroll vuex-persist
```

Do check out the links to learn more about what each package does, and how it can be configured.

Now, let's configure our Vue.js project. Open `src/main.js` and update the code as follows:

```

import Vue from 'vue'
import BootstrapVue from 'bootstrap-vue'
import VueChatScroll from 'vue-chat-scroll'

import App from './App.vue'
import router from './router'
import store from './store/index'

import 'bootstrap/dist/css/bootstrap.css'
import 'bootstrap-vue/dist/bootstrap-vue.css'
import './assets/css/loading.css'
import './assets/css/loading-btn.css'

Vue.config.productionTip = false
Vue.use(BootstrapVue)
Vue.use(VueChatScroll)

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

```

Update `src/router.js` as follows:

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

```
import Router from 'vue-router'
import Login from './views/Login.vue'
import ChatDashboard from './views/ChatDashboard.vue'
```

```
Vue.use(Router)
```

```
export default new Router({
  mode: 'history',
  base: process.env.BASE_URL,
  routes: [
    {
      path: '/',
      name: 'login',
      component: Login
    },
    {
      path: '/chat',
      name: 'chat',
      component: ChatDashboard,
    }
  ]
})
```

Update `src/store/index.js`:

```
import Vue from 'vue'
import Vuex from 'vuex'
import VuexPersistence from 'vuex-persist'
import mutations from './mutations'
import actions from './actions'
```

```
Vue.use(Vuex)
```

```
const debug = process.env.NODE_ENV !== 'production'
```

```
const vuexLocal = new VuexPersistence({
  storage: window.localStorage
})
```

```
export default new Vuex.Store({
  state: {
  },
  mutations,
  actions,
  getters: {
  },
  plugins: [vuexLocal.plugin],
  strict: debug
})
```

The `vuex-persist` package ensures that our Vuex state is saved between page reloads or refreshes.

Our project should be able to compile now without errors. However, don't run it just yet, as we need to build the user interface.

Building the UI Interface

Let's start by updating `src/App.vue` as follows:

```
<template>
  <div id="app">
    <router-view/>
  </div>
</template>
```

Next, we need to define our Vuex store states as they're required by our UI components to work. We'll do this by going to our Vuex store in `src/store/index.js`. Just update the state and getters sections as follows:

```
state: {
  loading: false,
  sending: false,
  error: null,
  user: [],
  reconnect: false,
  activeRoom: null,
  rooms: [],
  users: [],
  messages: [],
  userTyping: null
},
getters: {
  hasError: state => state.error ? true : false
},
```

These are all the state variables that we'll need for our chat application. The `loading` state is used by the UI to determine whether it should run the CSS loader. The `error` state is used to store information of an error that has just occurred. We'll discuss the rest of the state variables when we cross their bridges.

Next open `src/view/Login.vue` and update as follows:

```
<template>
  <div class="login">
    <b-jumbotron header="Vue.js Chat"
      lead="Powered by Chatkit SDK and Bootstrap-Vue"
      bg-variant="info"
      text-variant="white">
      <p>For more information visit website</p>
      <b-btn target="_blank" href="https://pusher.com/chatkit">More Info</b-btn>
    </b-jumbotron>
    <b-container>
      <b-row>
        <b-col lg="4" md="3"></b-col>
        <b-col lg="4" md="6">
          <LoginForm />
        </b-col>
        <b-col lg="4" md="3"></b-col>
      </b-row>
    </b-container>
  </div>
</template>

<script>
import LoginForm from '@components/LoginForm.vue'

export default {
  name: 'login',
  components: {
    LoginForm
  }
}
</script>
```

Next, insert code for `src/components/LoginForm.vue` as follows:

```
<template>
  <div class="login-form">
    <h5 class="text-center">Chat Login</h5>
    <hr>
    <b-form @submit.prevent="onSubmit">
```

```

    <b-alert variant="danger" :show="hasError">{{ error }} </b-alert>

    <b-form-group id="userInputGroup"
      label="User Name"
      label-for="userInput">
      <b-form-input id="userInput"
        type="text"
        placeholder="Enter user name"
        v-model="userId"
        autocomplete="off"
        :disabled="loading"
        required>
      </b-form-input>
    </b-form-group>

    <b-button type="submit"
      variant="primary"
      class="ld-ext-right"
      v-bind:class="{ running: loading }"
      :disabled="isValid">
      Login <div class="ld ld-ring ld-spin"></div>
    </b-button>
  </b-form>
</div>
</template>

<script>
import { mapState, mapGetters } from 'vuex'

export default {
  name: 'login-form',
  data() {
    return {
      userId: '',
    }
  },
  computed: {
    isValid: function() {
      const result = this.userId.length < 3;
      return result ? result : this.loading
    },
    ...mapState([
      'loading',
      'error'
    ]),
    ...mapGetters([
      'hasError'
    ])
  }
}
</script>

```

As mentioned earlier, this is an advanced tutorial. If you have trouble understanding any of the code here, please go to the prerequisites or the project dependencies for information.

We can now start the Vue dev server via `npm run serve` to ensure our application is running without any compilation issues.



Chat Login

User Name

You can confirm the validation is working by entering a username. You should see the *Login* button activate after entering three characters. The *Login* button doesn't work for now, as we haven't coded that part. We'll look into it later. For now, let's continue building our chat user interface.

Go to `src/view/ChatDashboard.vue` and insert the code as follows:

```
<template>
  <div class="chat-dashboard">
    <ChatNavBar />
    <b-container fluid class="ld-over" v-bind:class="{ running: loading }">
      <div class="ld ld-ring ld-spin"></div>
      <b-row>
        <b-col cols="2">
          <RoomList />
        </b-col>

        <b-col cols="8">
          <b-row>
            <b-col id="chat-content">
              <MessageList />
            </b-col>
          </b-row>
          <b-row>
            <b-col>
              <MessageForm />
            </b-col>
          </b-row>
        </b-col>

        <b-col cols="2">
          <UserList />
        </b-col>
      </b-row>
    </b-container>
  </div>
</template>

<script>
import ChatNavBar from '@components/ChatNavBar.vue'
import RoomList from '@components/RoomList.vue'
import MessageList from '@components/MessageList.vue'
import MessageForm from '@components/MessageForm.vue'
```

```

import UserList from '@components/UserList.vue'
import { mapState } from 'vuex';

export default {
  name: 'Chat',
  components: {
    ChatNavBar,
    RoomList,
    UserList,
    MessageList,
    MessageForm
  },
  computed: {
    ...mapState([
      'loading'
    ])
  }
}
</script>

```

The ChatDashboard will act as a layout parent for the following child components:

- ChatNavBar, a basic navigation bar
- RoomList, which lists rooms that the logged in user has access to, and which is also a room selector
- UserList, which lists members of a selected room
- MessageList, which displays messages posted in a selected room
- MessageForm, a form for sending messages to the selected room

Let's put some boilerplate code in each component to ensure everything gets displayed.

Insert boilerplate code for src/components/ChatNavBar.vue as follows:

```

<template>
  <b-navbar id="chat-navbar" toggleable="md" type="dark" variant="info">
    <b-navbar-brand href="#">
      Vue Chat
    </b-navbar-brand>
    <b-navbar-nav class="ml-auto">
      <b-nav-text>{{ user.name }} | </b-nav-text>
      <b-nav-item href="#" active>Logout</b-nav-item>
    </b-navbar-nav>
  </b-navbar>
</template>

<script>
import { mapState } from 'vuex'

export default {
  name: 'ChatNavBar',
  computed: {
    ...mapState([
      'user',
    ])
  },
}
</script>

<style>
#chat-navbar {
  margin-bottom: 15px;
}
</style>

```

Insert boilerplate code for src/components/RoomList.vue as follows:

```

<template>
  <div class="room-list">
    <h4>Channels</h4>
    <hr>
    <b-list-group v-if="activeRoom">
      <b-list-group-item v-for="room in rooms"
        :key="room.name"
        :active="activeRoom.id === room.id"
        href="#"
        @click="onChange(room)">
        # {{ room.name }}
      </b-list-group-item>
    </b-list-group>
  </div>
</template>

```

```

<script>
import { mapState } from 'vuex'

export default {
  name: 'RoomList',
  computed: {
    ...mapState([
      'rooms',
      'activeRoom'
    ]),
  }
}
</script>

```

Insert boilerplate code for `src/components/UserList.vue` as follows:

```

<template>
  <div class="user-list">
    <h4>Members</h4>
    <hr>
    <b-list-group>
      <b-list-group-item v-for="user in users" :key="user.username">
        {{ user.name }}
        <b-badge v-if="user.presence"
          :variant="statusColor(user.presence)"
          pill>
          {{ user.presence }}</b-badge>
      </b-list-group-item>
    </b-list-group>
  </div>
</template>

```

```

<script>
import { mapState } from 'vuex'

export default {
  name: 'user-list',
  computed: {
    ...mapState([
      'loading',
      'users'
    ])
  },
  methods: {
    statusColor(status) {
      return status === 'online' ? 'success' : 'warning'
    }
  }
}

```

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

Insert boilerplate code for `src/components/MessageList.vue` as follows:

```
<template>  
  <div class="message-list">  
    <h4>Messages</h4>  
    <hr>  
    <div id="chat-messages" class="message-group" v-chat-scroll="{smooth: true}">  
      <div class="message" v-for="(message, index) in messages" :key="index">  
        <div class="clearfix">  
          <h4 class="message-title">{{ message.name }}</h4>  
          <small class="text-muted float-right">@{{ message.username }}</small>  
        </div>  
        <p class="message-text">  
          {{ message.text }}  
        </p>  
        <div class="clearfix">  
          <small class="text-muted float-right">{{ message.date }}</small>  
        </div>  
      </div>  
    </div>  
  </div>  
</template>
```

```
<script>  
import { mapState } from 'vuex'
```

```
export default {  
  name: 'message-list',  
  computed: {  
    ...mapState([  
      'messages',  
    ])  
  }  
}
```

```
</script>
```

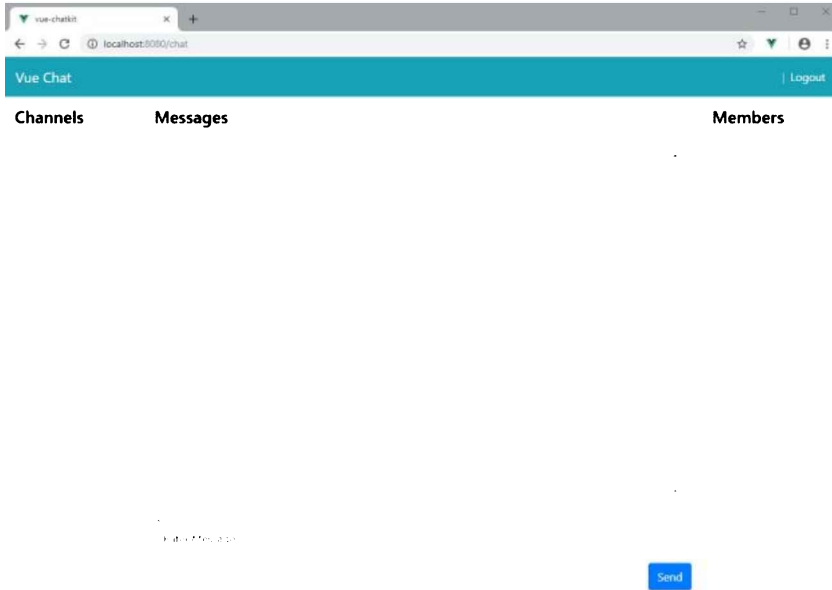
```
<style>  
.message-list {  
  margin-bottom: 15px;  
  padding-right: 15px;  
}  
.message-group {  
  height: 65vh !important;  
  overflow-y: scroll;  
}  
.message {  
  border: 1px solid lightblue;  
  border-radius: 4px;  
  padding: 10px;  
  margin-bottom: 15px;  
}  
.message-title {  
  font-size: 1rem;  
  display: inline;  
}  
.message-text {  
  color: gray;  
  margin-bottom: 0;  
}
```

```
.user-typing {  
  height: 1rem;  
}  
</style>
```

Insert boilerplate code for `src/components/MessageForm.vue` as follows:

```
<template>  
  <div class="message-form ld-over">  
    <small class="text-muted">@{{ user.username }}</small>  
    <b-form @submit.prevent="onSubmit" class="ld-over" v-bind:class="{ running: sending }">  
      <div class="ld ld-ring ld-spin"></div>  
      <b-alert variant="danger" :show="hasError">{{ error }} </b-alert>  
      <b-form-group>  
        <b-form-input id="message-input"  
          type="text"  
          v-model="message"  
          placeholder="Enter Message"  
          autocomplete="off"  
          required>  
        </b-form-input>  
      </b-form-group>  
      <div class="clearfix">  
        <b-button type="submit" variant="primary" class="float-right">  
          Send  
        </b-button>  
      </div>  
    </b-form>  
  </div>  
</template>  
  
<script>  
import { mapState, mapGetters } from 'vuex'  
  
export default {  
  name: 'message-form',  
  data() {  
    return {  
      message: ''  
    }  
  },  
  computed: {  
    ...mapState([  
      'user',  
      'sending',  
      'error',  
      'activeRoom'  
    ]),  
    ...mapGetters([  
      'hasError'  
    ])  
  }  
}  
</script>
```

Go over the code to ensure nothing is a mystery to you. Navigate to <http://localhost:8080/chat> to check if everything is running. Check the terminal and browser consoles to ensure there are no errors at this point. You should now have the following view.



Pretty empty, right? Let's go to `src/store/index.js` and insert some mock data in state:

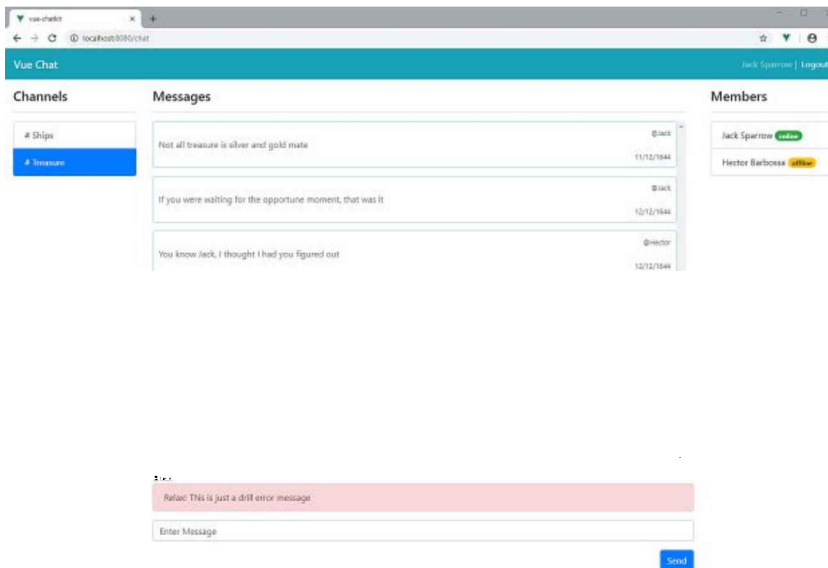
```
state: {
  loading: false,
  sending: false,
  error: 'Relax! This is just a drill error message',
  user: {
    username: 'Jack',
    name: 'Jack Sparrow'
  },
  reconnect: false,
  activeRoom: {
    id: '124'
  },
  rooms: [
    {
      id: '123',
      name: 'Ships'
    },
    {
      id: '124',
      name: 'Treasure'
    }
  ],
  users: [
    {
      username: 'Jack',
      name: 'Jack Sparrow',
      presence: 'online'
    },
    {
      username: 'Barbossa',
      name: 'Hector Barbossa',
      presence: 'offline'
    }
  ],
  messages: [
    {
      username: 'Jack',
      date: '11/12/1644',
      text: 'Not all treasure is silver and gold mate'
    },
  ],
}
```

```

    {
      username: 'Jack',
      date: '12/12/1644',
      text: 'If you were waiting for the opportune moment, that was it'
    },
    {
      username: 'Hector',
      date: '12/12/1644',
      text: 'You know Jack, I thought I had you figured out'
    }
  ],
  userTyping: null
},

```

After saving the file, your view should match the image below.



This simple test ensures that all components and states are all tied up together nicely. You can now revert the state code back to its original form:

```

state: {
  loading: false,
  sending: false,
  error: null,
  user: null,
  reconnect: false,
  activeRoom: null,
  rooms: [],
  users: [],
  messages: [],
  userTyping: null
}

```

Let's start implementing concrete features, starting with the login form.

Password-less Authentication

For this tutorial, we'll employ a password-less non-secure authentication system. A proper, secure authentication system is outside the scope of this tutorial. To start with, we need to start building our own interface that will interact with ChatKit service via the `@pusher/chatkit-client` package.

Go back to the ChatKit dashboard and copy the *instance* and *test token* parameters. Save them in the file `.env.local` at the root of your project like this:

```
VUE_APP_INSTANCE_LOCATOR=
```

```
VUE_APP_TOKEN_URL=  
VUE_APP_MESSAGE_LIMIT=10
```

I've also added a `MESSAGE_LIMIT` parameter. This value simply restricts the number of messages our chat application can fetch. Make sure to fill in the other parameters from the credentials tab.

Next, go to `src/chatkit.js` to start building our chat application foundation:

```
import { ChatManager, TokenProvider } from '@pusher/chatkit-client'  
  
const INSTANCE_LOCATOR = process.env.VUE_APP_INSTANCE_LOCATOR;  
const TOKEN_URL = process.env.VUE_APP_TOKEN_URL;  
const MESSAGE_LIMIT = Number(process.env.VUE_APP_MESSAGE_LIMIT) || 10;  
  
let currentUser = null;  
let activeRoom = null;  
  
async function connectUser(userId) {  
  const chatManager = new ChatManager({  
    instanceLocator: INSTANCE_LOCATOR,  
    tokenProvider: new TokenProvider({ url: TOKEN_URL }),  
    userId  
  });  
  currentUser = await chatManager.connect();  
  return currentUser;  
}  
  
export default {  
  connectUser  
}
```

Notice that we're casting the `MESSAGE_LIMIT` constant to a number, as by default the `process.env` object forces all of its properties to be of type string.

Insert the following code for `src/store/mutations`:

```
export default {  
  setError(state, error) {  
    state.error = error;  
  },  
  setLoading(state, loading) {  
    state.loading = loading;  
  },  
  setUser(state, user) {  
    state.user = user;  
  },  
  setReconnect(state, reconnect) {  
    state.reconnect = reconnect;  
  },  
  setActiveRoom(state, roomId) {  
    state.activeRoom = roomId;  
  },  
  setRooms(state, rooms) {  
    state.rooms = rooms  
  },  
  setUsers(state, users) {  
    state.users = users  
  },  
  clearChatRoom(state) {  
    state.users = [];  
    state.messages = [];  
  },  
  setMessages(state, messages) {  
    state.messages = messages
```



```

    },
    addMessage(state, message) {
      state.messages.push(message)
    },
    setSending(state, status) {
      state.sending = status
    },
    setUserTyping(state, userId) {
      state.userTyping = userId
    },
    reset(state) {
      state.error = null;
      state.users = [];
      state.messages = [];
      state.rooms = [];
      state.user = null
    }
  }
}

```

The code for mutations is really simple—just a bunch of setters. You'll soon understand what each mutation function is for in the later sections. Next, update `src/store/actions.js` with this code:

```

import chatkit from '../chatkit';

// Helper function for displaying error messages
function handleError(commit, error) {
  const message = error.message || error.info.error_description;
  commit('setError', message);
}

export default {
  async login({ commit, state }, userId) {
    try {
      commit('setError', '');
      commit('setLoading', true);
      // Connect user to ChatKit service
      const currentUser = await chatkit.connectUser(userId);
      commit('setUser', {
        username: currentUser.id,
        name: currentUser.name
      });
      commit('setReconnect', false);

      // Test state.user
      console.log(state.user);
    } catch (error) {
      handleError(commit, error)
    } finally {
      commit('setLoading', false);
    }
  }
}

```

Next, update `src/components/LoginForm.vue` as follows:

```

import { mapState, mapGetters, mapActions } from 'vuex'

//...
export default {
  //...
  methods: {
    ...mapActions([
      'login'
    ])
  }
}

```

```

    }},
    async onSubmit() {
      const result = await this.login(this.userId);
      if(result) {
        this.$router.push('chat');
      }
    }
  }
}
}

```

You'll have to restart the Vue.js server in order to load `env.local` data. If you see any errors regarding unused variables, ignore them for now. Once you've done that, navigate to `http://localhost:8080/` and test the login feature:



Chat Login

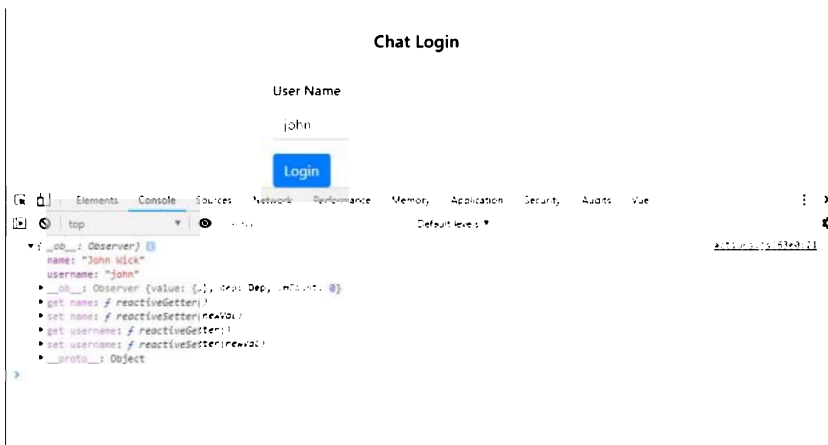
The requested user does not exist

User Name

peter

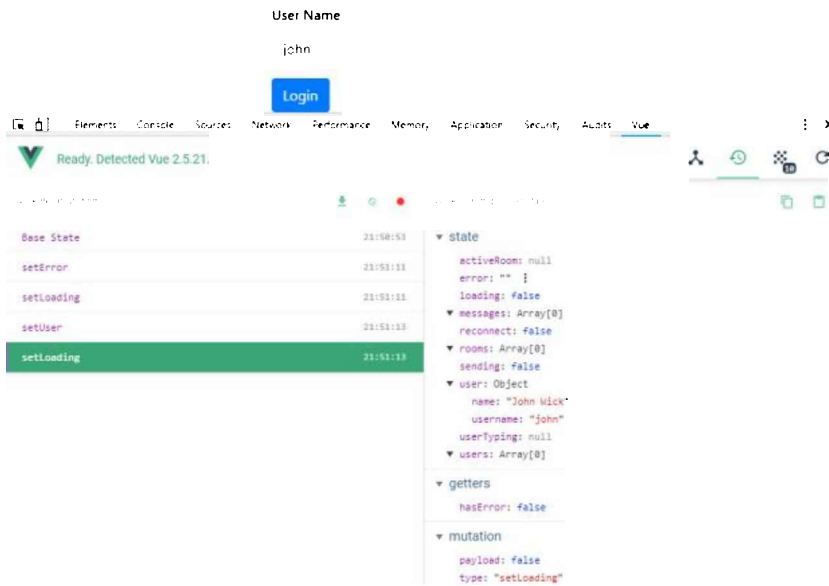
Login

In the above example, I've used an incorrect username just to make sure the error handling feature is working correctly.



In this screenshot, I've used the correct username. I've also opened up the browser console tab to ensure that the `user` object has been populated. Better yet, if you've installed Vue.js Dev Tools in Chrome or Firefox, you should be able to see more detailed information.

Chat Login



If everything's working correctly for you at this point, move on to the next step.

Subscribing to a Room

Now that we've successfully verified that the login feature works, we need to redirect users to the ChatDashboard view. The code `this.$router.push('chat');` does this for us. However, our action login needs to return a Boolean to determine when it's okay to navigate to the ChatDashboard view. We also need to populate the RoomList and UserList components with actual data from the ChatKit service.

Update `src/chatkit.js` as follows:

```
//...
import moment from 'moment'
import store from './store/index'

//...
function setMembers() {
  const members = activeRoom.users.map(user => ({
    username: user.id,
    name: user.name,
    presence: user.presence.state
  }));
  store.commit('setUsers', members);
}

async function subscribeToRoom(roomId) {
  store.commit('clearChatRoom');
  activeRoom = await currentUser.subscribeToRoom({
    roomId,
    messageLimit: MESSAGE_LIMIT,
    hooks: {
      onMessage: message => {
        store.commit('addMessage', {
          name: message.sender.name,
          username: message.senderId,
          text: message.text,
          date: moment(message.createdAt).format('h:mm:ss a D-MM-YYYY')
        });
      },
      onPresenceChanged: () => {
```

```

        setMembers();
    },
    onUserStartedTyping: user => {
        store.commit('setUserTyping', user.id)
    },
    onUserStoppedTyping: () => {
        store.commit('setUserTyping', null)
    }
}
});
setMembers();
return activeRoom;
}

export default {
    connectUser,
    subscribeToRoom
}

```

If you look at the hooks section, we have event handlers used by the ChatKit service to communicate with our client application. You can find the full documentation [here](#). I'll quickly summarize the purpose of each hook method:

- `onMessage` receives messages
- `onPresenceChanged` receives an event when a user logs in or out
- `onUserStartedTyping` receives an event that a user is typing
- `onUserStoppedTyping` receives an event that a user has stopped typing

For the `onUserStartedTyping` to work, we need to emit a typing event from our `MessageForm` while a user is typing. We'll look into this in the next section.

Update the login function in `src/store/actions.js` with the following code:

```

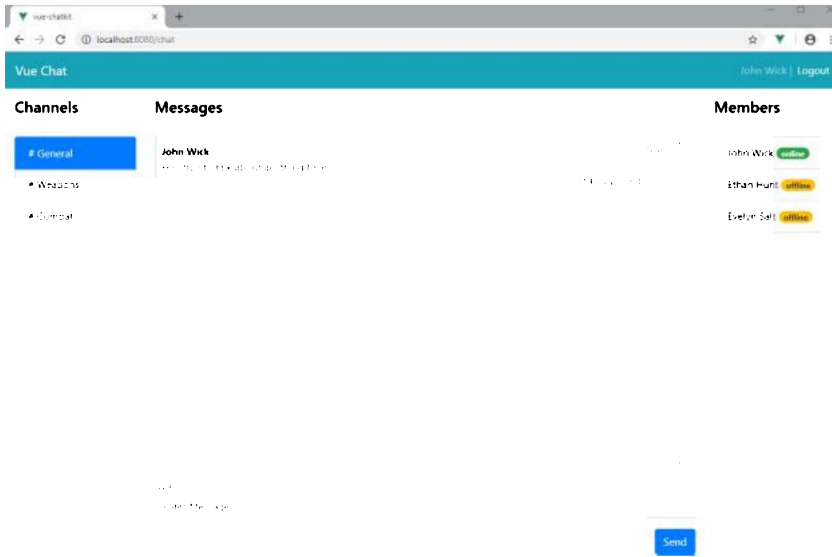
//...
try {
    //... (place right after the `setUser` commit statement)
    // Save list of user's rooms in store
    const rooms = currentUser.rooms.map(room => ({
        id: room.id,
        name: room.name
    })))
    commit('setRooms', rooms);

    // Subscribe user to a room
    const activeRoom = state.activeRoom || rooms[0]; // pick last used room, or the first one
    commit('setActiveRoom', {
        id: activeRoom.id,
        name: activeRoom.name
    });
    await chatkit.subscribeToRoom(activeRoom.id);

    return true;
} catch (error) {
    //...
}

```

After you've saved the code, go back to the login screen and enter the correct username. You should be taken to the following screen.



Nice! Almost all the components are working without additional effort since we wired them up properly to the Vuex store. Try sending a message via ChatKit's dashboard console interface. Create a message and post it to the `General` room. You should see the new messages pop up automatically in the `MessageList` component. Soon, we'll implement the logic for sending messages from our Vue.js app.

If You Experience Issues

In case you're experiencing issues, try the following:

- restart the Vue.js server
- clear your browser cache
- do a hard reset/refresh (available in Chrome if the *Console* tab is open and you hold the *Reload* button for five seconds)
- clear `localStorage` using your browser console

If everything is running okay up to this point, continue with the next section, where we implement logic for changing rooms.

Changing Rooms

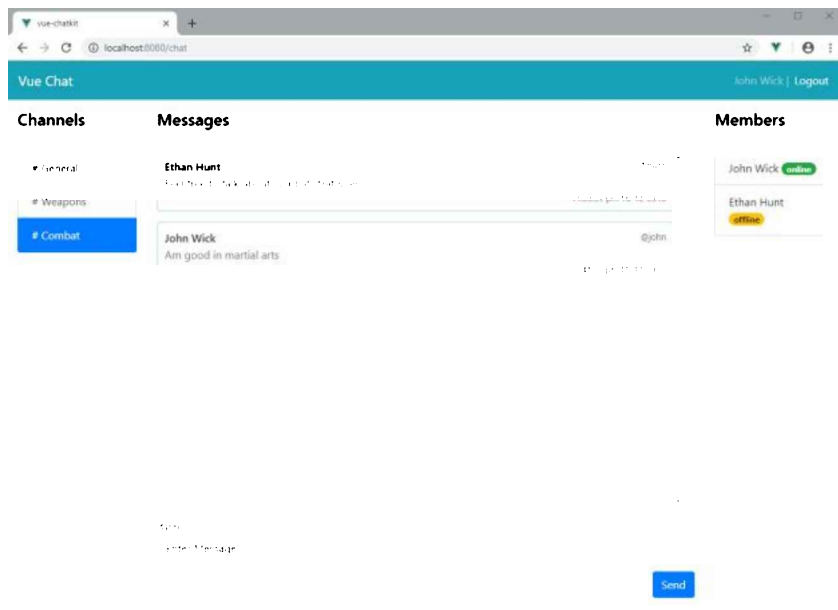
This part is quite simple, since we've already laid out the foundation. First, we'll create an `action` that will allow users to change rooms. Go to `src/store/actions.js` and add this function right after the `login` action handler:

```
async changeRoom({ commit }, roomId) {
  try {
    const { id, name } = await chatkit.subscribeToRoom(roomId);
    commit('setActiveRoom', { id, name });
  } catch (error) {
    handleError(commit, error)
  }
},
```

Next, go to `src/components/RoomList.vue` and update the script section as follows:

```
import { mapState, mapActions } from 'vuex'
//...
export default {
  //...
  methods: {
    ...mapActions([
      'changeRoom'
    ]),
    onChange(room) {
      this.changeRoom(room.id)
    }
  }
}
```

If you recall, we've already defined `@click="onChange(room)"` in the `b-list-group-item` element. Let's test out this new feature by clicking the items in the `RoomList` component.



Your UI should update with each click of the room. The `MessageList` and `UserList` component should display the correct information for the selected room. For the next section, we'll implement multiple features at once.

Reconnecting the User After a Page Refresh

You may have noticed that, when you do some changes to `store/index.js`, or you do a page refresh, you get the following error: `Cannot read property 'subscribeToRoom' of null`. This happens because the state of your application gets reset. Luckily, the `vuex-persist` package maintains our Vuex state between page reloads by saving it in the browser's local storage.

Unfortunately, the references that connect our app to the ChatKit server are reset back to null. To fix this, we need to perform a reconnect operation. We also need a way to tell our app that a page reload has just happened and that our app needs to reconnect in order to continue functioning properly. We'll implement this code in `src/components/ChatNavbar.vue`. Update the script section as follows:

```
<script>
import { mapState, mapActions, mapMutations } from 'vuex'

export default {
  name: 'ChatNavBar',
  computed: {
    ...mapState([
      'user',
      'reconnect'
    ])
  },
  methods: {
    ...mapActions([
      'logout',
      'login'
    ]),
    ...mapMutations([
      'setReconnect'
    ]),
    onLogout() {
      this.$router.push({ path: '/' });
      this.logout();
    },
    unload() {
      if(this.user.username) { // User hasn't logged out
```

```

        this.setReconnect(true);
    }
}
},
mounted() {
    window.addEventListener('beforeunload', this.unload);
    if(this.reconnect) {
        this.login(this.user.username);
    }
}
}
</script>

```

Let me break down the sequence of events so that you can understand the logic behind reconnecting to the ChatKit service:

1. **unload.** When a page refresh occurs, this method gets called. It checks first the state `user.username` has been set. If it has, it means the user has not logged out. The state `reconnect` is set to `true`.
2. **mounted.** This method gets called every time `ChatNavbar.vue` has just finished rendering. It first assigns a handler to an event listener that gets called just before the page unloads. It also does a check if state `reconnect` has been set to `true`. If so, then the login procedure is executed, thus reconnecting our chat application back to our ChatKit service.

I've also added a Logout feature, which we'll look into later.

After making these changes, try refreshing the page. You'll see the page update itself automatically as it does the reconnection process behind the scenes. When you switch rooms, it should work flawlessly.

Sending Messages, Detecting User Typing and Logging Out

Let's start with implementing these features in `src/chatkit.js` by adding the following code:

```

//...
async function sendMessage(text) {
    const messageId = await currentUser.sendMessage({
        text,
        roomId: activeRoom.id
    });
    return messageId;
}

export function isTyping(roomId) {
    currentUser.isTypingIn({ roomId });
}

function disconnectUser() {
    currentUser.disconnect();
}

export default {
    connectUser,
    subscribeToRoom,
    sendMessage,
    disconnectUser
}

```

While the functions `sendMessage` and `disconnectUser` will be bundled in ChatKit's module export, `isTyping` function will be exported separately. This is to allow `MessageForm` to directly send typing events without involving the Vuex store.

For `sendMessage` and `disconnectUser`, we'll need to update the store in order to cater for things like error handling and loading status notifications. Go to `src/store/actions.js` and insert the following code right after the `changeRoom` function:

```

async sendMessage({ commit }, message) {
    try {
        commit('setError', '');
        commit('setSending', true);
    }
}

```

```

    const messageId = await chatkit.sendMessage(message);
    return messageId;
  } catch (error) {
    handleError(commit, error)
  } finally {
    commit('setSending', false);
  }
},
async logout({ commit }) {
  commit('reset');
  chatkit.disconnectUser();
  window.localStorage.clear();
}

```

For the logout function, we call `commit('reset')` to reset our store back to its original state. It's a basic security feature to remove user information and messages from the browser cache.

Let's start by updating the form input in `src/components/MessageForm.vue` to emit typing events by adding the `@input` directive:

```

<b-form-input id="message-input"
  type="text"
  v-model="message"
  @input="isTyping"
  placeholder="Enter Message"
  autocomplete="off"
  required>
</b-form-input>

```

Let's now update the script section for `src/components/MessageForm.vue` to handle message sending and emitting of typing events. Update as follows:

```

<script>
import { mapActions, mapState, mapGetters } from 'vuex'
import { isTyping } from '../chatkit.js'

export default {
  name: 'message-form',
  data() {
    return {
      message: ''
    }
  },
  computed: {
    ...mapState([
      'user',
      'sending',
      'error',
      'activeRoom'
    ]),
    ...mapGetters([
      'hasError'
    ])
  },
  methods: {
    ...mapActions([
      'sendMessage',
    ]),
    async onSubmit() {
      const result = await this.sendMessage(this.message);
      if(result) {
        this.message = '';
      }
    },
  },

```



```

    async isTyping() {
      await isTyping(this.activeRoom.id);
    }
  }
}
</script>

```

And in `src/MessageList.vue`:

```

import { mapState } from 'vuex'

export default {
  name: 'message-list',
  computed: {
    ...mapState([
      'messages',
      'userTyping'
    ])
  }
}

```

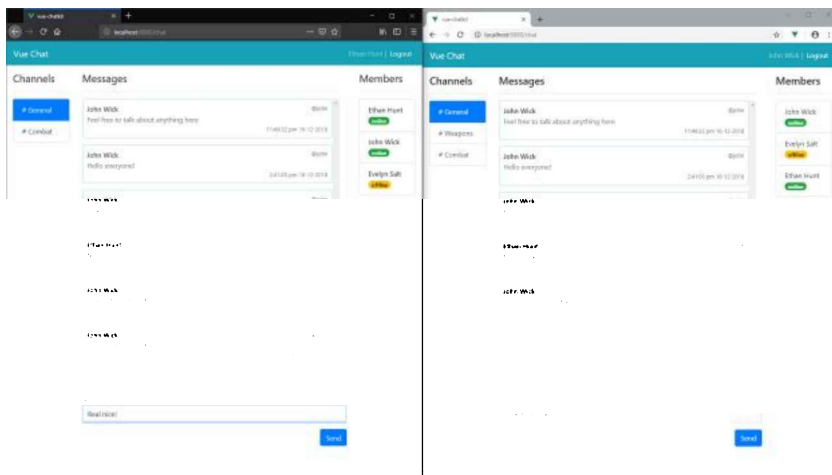
The send message feature should now work. In order to display a notification that another user is typing, we need to add an element for displaying this information. Add the following snippet in the template section of `src/components/MessageList.vue`, right after the `message-group` div:

```

<div class="user-typing">
  <small class="text-muted" v-if="userTyping">@{{ userTyping }} is typing...</small>
</div>

```

To test out this feature, simply log in as another user using a different browser and start typing. You should see a notification appear on the other user's chat window.



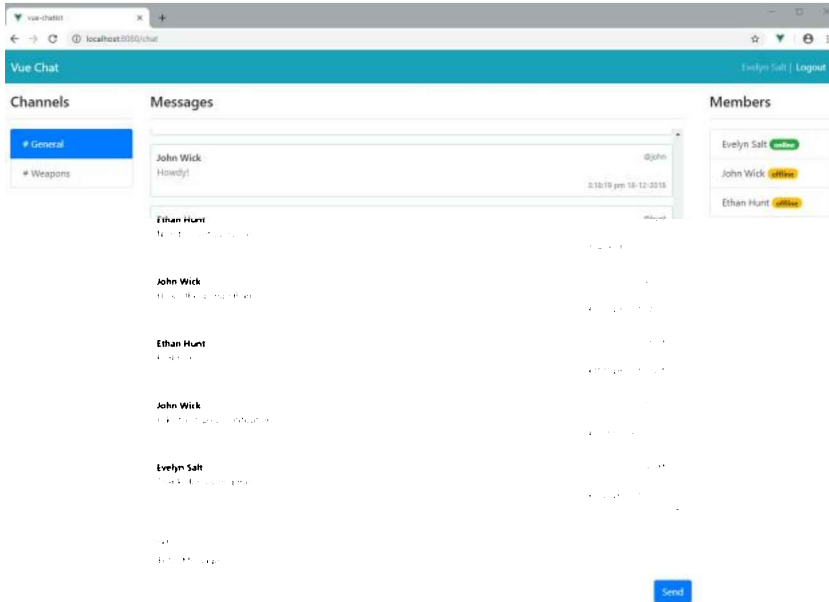
Let's finish this tutorial by implementing the last feature, logout. Our Vuex store already has the necessary code to handle the logout process. We just need to update `src/components/ChatNavBar.vue`. Simply link the Logout button with function handler `onLogout` that we had specified earlier:

```

<b-nav-item href="#" @click="onLogout" active>Logout</b-nav-item>

```

That's it. You can now log out and log in again as another user.



Summary

We've now come to the end of the tutorial. The ChatKit API has enabled us to rapidly build a chat application in a short time. If we were to build a similar application from scratch, it would take us several weeks, since we'd have to flesh out the back end as well. The great thing about this solution is that we don't have to deal with hosting, managing databases and other infrastructure issues. We can simply build and deploy the front-end code to client devices on web, Android and IOS platforms.

Please do take a look at the documentation, as there's a ton of back-end features I wasn't able to show you in this tutorial. Given time, you can easily build a full-featured chat application that can rival popular chat products like Slack and Discord.

Chapter 4: Building a Vue Front End for a Headless CMS

by Michael Wanyoike

In this guide, we'll learn how to build a modern blog website using Vue.js and GraphCMS, a headless CMS platform.

If you're looking to start a quick blog today, my recommendation is to go straight to WordPress.

But what if you're a media powerhouse and you want to deliver your content as fast as possible to multiple devices? You'll probably also need to integrate your content with ads and other third-party services. Well, you could do that with WordPress, but you'll come across a few problems with that platform.

1. You'll need to install a plugin to implement additional features. The more plugins you install, the slower your website becomes.
2. PHP is quite slow compared to most JavaScript web frameworks. From a developer's perspective, it's much easier and faster to implement custom features on a JavaScript-powered front end.

JavaScript offers superior performance to PHP in browser loading tests. In addition, modern JavaScript and its ecosystem provides a far more pleasant development experience when it comes to building new web experiences fast.

So there's been a growth of **headless CMS** solutions—which are simply back ends for managing content. With this approach, developers can focus on building fast and interactive front ends using a JavaScript framework of their choice. Customizing a JavaScript-powered front end is much easier than making changes on a WordPress site.

GraphCMS differs from most Headless CMS platforms in that, instead of delivering content via REST, it does so via GraphQL. This new technology is superior to REST, as it allows us to construct queries that touch on data belonging to multiple models in a single request.

Consider the following model schema:

Post

- id: Number
- title: String
- content : String
- comments : array of Comments

Comment

- id: Number
- name: String
- message: String

The above models have a one(Post)-to-many(Comments) relationship. Let's see how we can fetch a single Post record attached with all linked Comment records.

If the data is in a relational database, you have to construct either one inefficient SQL statement, or two SQL statements for fetching the data cleanly. If the data is stored in a NoSQL database, you can use a modern ORM like Vuex ORM to fetch the data easily for you, like this:

```
const post = Post.query()  
  .with('comments')  
  .find(1);
```

Quite simple! You can easily pass this data via REST to the intended client. But here's the problem: whenever the data requirement changes at the client end, you'll be forced to go back to your back-end code to either update your existing API endpoint, or create a new one that provides the required data set. This back and forth process is tiring and repetitive.

What if, at the client level, you could just ask for the data you need and the back end will provide it for you, without you doing extra work? Well, that's what GraphQL is for.

Prerequisites

Before we begin, I'd like to note that this is a guide for intermediate to advanced users. I won't be going over the basics, but rather will show you how to quickly build a Vue.js blog using GraphCMS as the back end. You'll need to be proficient in the following areas:

- ES6 and ES7 JavaScript
- Vue.js (using CLI version 3)
- GraphQL

That's all you need to know to get started with this tutorial. Also, a background in using REST will be great, as I'll be referencing this a lot. If you'd like a refresher, this article might help: "REST 2.0 Is Here and Its Name Is GraphQL".

About the Project

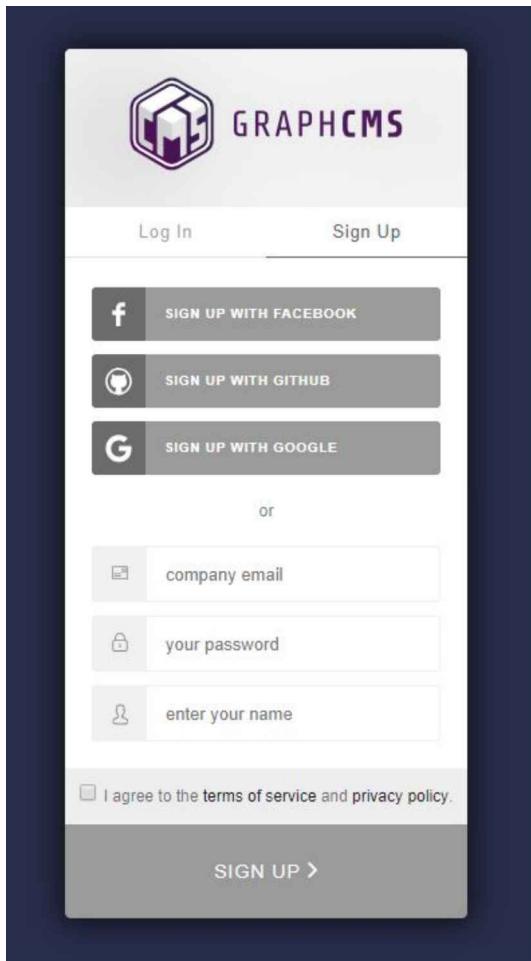
We'll build a very simple blog application with a basic comment system. Below are the links you can visit to check out the completed project:

- CodeSandbox.io demo
- GitHub repo

Please note that a READ-ONLY token has been used in the demo and consequently the comments system won't work. You'll need to supply your OPEN permission token and endpoint as per the instructions in this tutorial for it to work.

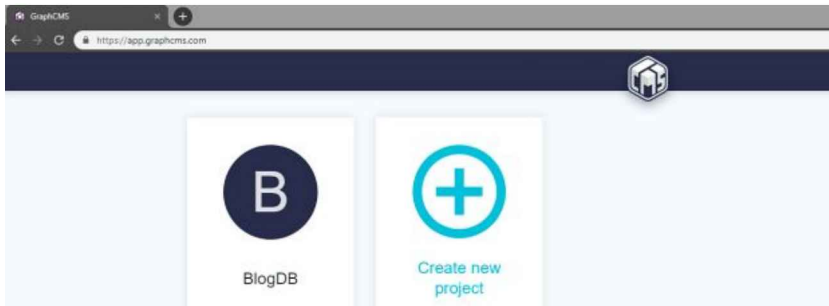
Create GraphCMS Project Database

Head over to the GraphCMS website and click the "Start Building for Free" button. You'll be taken to their signup page.

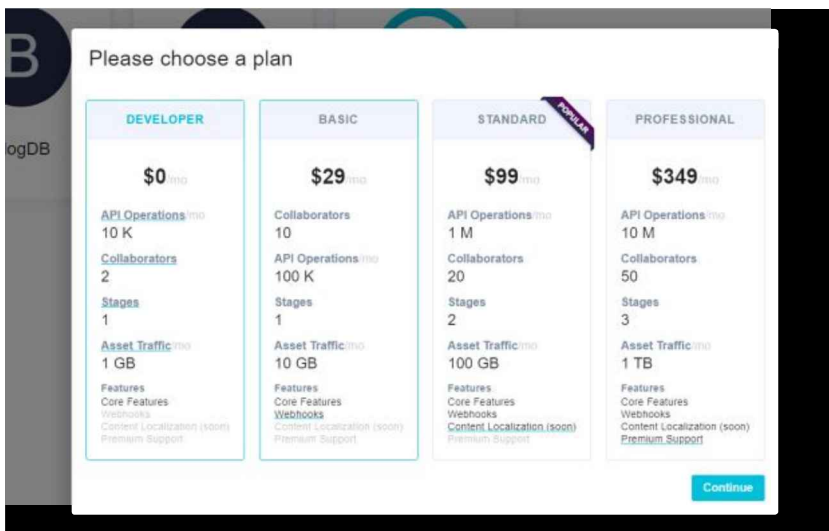


The image shows a mobile-style sign-up form for GraphCMS. At the top left is the GraphCMS logo, a purple hexagon with a white cube-like shape inside. To the right of the logo is the text 'GRAPHCMS'. Below the logo and text are two links: 'Log In' and 'Sign Up'. Underneath these links are three large, dark grey buttons with white text and icons: 'SIGN UP WITH FACEBOOK' (with a Facebook 'f' icon), 'SIGN UP WITH GITHUB' (with a GitHub octocat icon), and 'SIGN UP WITH GOOGLE' (with a Google 'G' icon). Below these buttons is the word 'or' in a small font. There are three input fields with light grey borders and icons: 'company email' (with an envelope icon), 'your password' (with a lock icon), and 'enter your name' (with a person icon). At the bottom of the form is a checkbox with the text 'I agree to the terms of service and privacy policy.' and a large, dark grey button with the text 'SIGN UP >'.

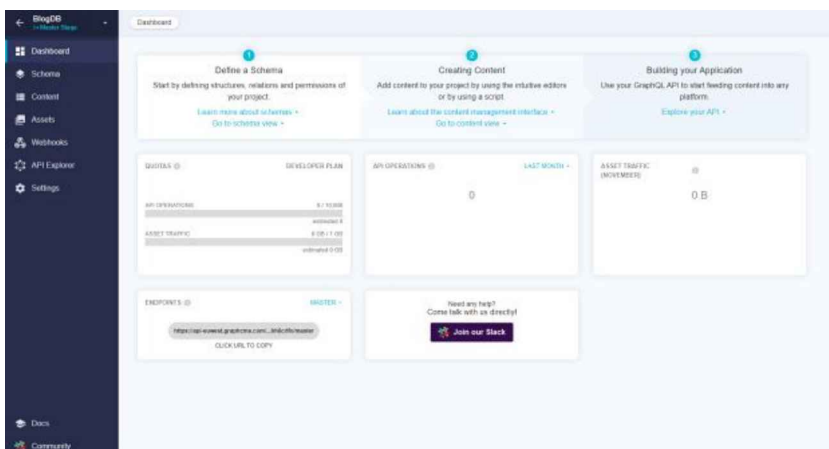
Sign up using your preferred method. Once you've completed the account authentication and verification process, you should be able to access the main dashboard.



In the above example, I've already created a project called "BlogDB". Go ahead and create a new one, and call it whatever you want. After you've entered the name, you can leave the rest of the fields in their defaults. Click *Create* and you'll be taken to their project plan.



For the purposes of this tutorial, select the free Developer plan then click *Continue*. You'll be taken to the project's dashboard, which looks something like this:



Go to the *Schema* tab. We're going to create the following models, each with the following fields:

Category

- name: Single line text, required, unique

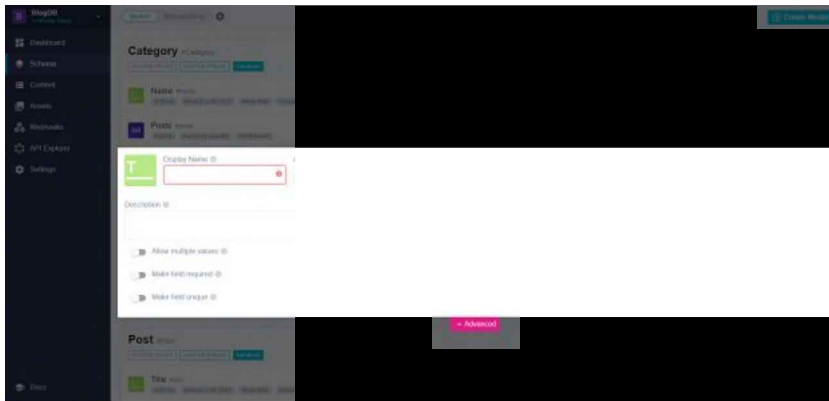
Post

- slug: Single line text, required, unique
- title: Single line text, required, unique
- content: Multi line text

Comment

- name: Single line text, required
- message: Multi line text, required

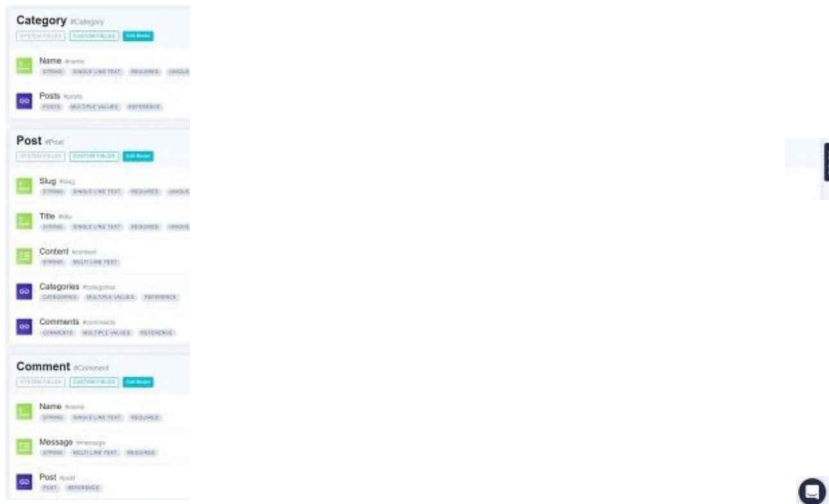
Use the *Create Model* button to create models. On the right side, you should find a hidden panel for Fields, which is activated by clicking the *Fields* button. Drag the appropriate field type onto the model's panel. You will be presented with a form to fill in your field's attributes. Do note at the bottom there's a pink button labeled *Advanced*. Clicking it will expand the panel to give you more field attributes you can enable.



Next, you'll need to add the relationship between models as follows:

- Post > Categories (many-to-many)
- Post > Comments (one-to-many)

Use the *Reference* field to define this relationship. You can add this field to any side; GraphCMS will automatically create the opposite relation field in the referenced model. When you've completed defining the models, you should have something like this:



You've now completed the first part. Let's now provide some data to our models.

GraphQL Data Migration

To add content to your models, you can simply click the *Content* tab in the project dashboard where you can create new records for each of your models. However, if you find this to be a slow method, you'll be happy to know that I've created a GraphCMS migration tool that copies data from CSV files and uploads them to your GraphCMS database. You can find the project here in this [GitHub repository](#). To start using the project, simply download it into your workspace like this:

```
git clone git@github.com:sitepoint-editors/graphcsms-data-migration.git
```

```
cd graphcms-data-migration
npm install
```

Next, you'll need to grab your GraphCMS project's API endpoint and token from the dashboard's *Settings* page. You'll need to create a new token. For the permission level, use `OPEN`, as this will allow the tool to perform `READ` and `WRITE` operations on your GraphCMS database. Create a file called `.env` and put it at the root of the project:

```
ENDPOINT=<Put api endpoint here>
TOKEN=<Put token with OPEN permission here>
```

Next, you may need to populate the CSV files in the `data` folder with your own. Here's some sample data that has been used:

```
// Categories.csv
name
Featured
Food
Fashion
Beauty

// Posts.csv
title,slug,content,categories
Food Post 1,food-post-1,Breeze through Thanksgiving by making this Instant Pot orange cranberry sauce,Food
Food Post 2,food-post-2,This is my second food post,Food
Food Post 3,food-post-3,This is my last and final food post,Food
Fashion Post 1,fashion-post-1,This is truly my very first fashion post,Fashion|Featured
Fashion Post 2,fashion-post-2,This is my second fashion post,Fashion
Fashion Post 3,fashion-post-3,This is my last and final fashion post,Fashion
Beauty Post 1,Beauty-post-1,This is truly my very first Beauty post,Beauty|Featured
Beauty Post 2,Beauty-post-2,This is my second beauty post,Beauty
```

You can change the content if you want. Make sure not to touch the top row, as otherwise you'll change the field names. Please note, for the column `categories`, I've used the pipe `|` character as a delimiter.

To upload the CSV data to your GraphCMS database, execute the following commands in this order:

```
npm run categories
npm run posts
```

Each script will print out records that have uploaded successfully. The reason we uploaded `categories` first is so that the `posts` records can link successfully to existing `category` records.

If you want to clean out your database, you can run the following command:

```
npm run reset
```

This script will delete all your model's contents. You'll get a report indicating how many records were deleted for each model.

I hope you find the tool handy. Go back to the dashboard to confirm that data for the `Posts` and `Categories` have successfully been uploaded.

With the back end taken care of, let's start building our front-end blog interface.

Building the Blog's Front End Using Vue.js

As mentioned earlier, we are going to build a very simple blog application powered by a GraphCMS database back end. Launch a terminal and navigate to your workspace.

If you haven't got Vue CLI installed, do that now:

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

Then create a new project:

```
vue create vue-graphcms
```

Choose to manually select features, then select the following options:

- Features: Babel, Router
- Router History Mode: Y
- ESLint with error prevention only
- Lint on save
- Config file placement: Dedicated Config Files
- Save preset: your choice

Once the project creation process is complete, change into the project directory and install the following dependencies:

```
npm install bootstrap-vue axios
```

To set up Bootstrap-Vue in our project, simply open `src/main.js` and add the following code:

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

```
Vue.config.productionTip = false;
Vue.use(BootstrapVue);
```

Next, we need to start laying down our project structure. In the `src/components` folder, delete the existing files and create these new ones:

- `CommentForm.vue`
- `CommentList.vue`
- `Post.vue`
- `PostList.vue`

In the `src/views` folder, delete `About.vue` and create a new file called `PostView.vue`. As seen from the demo, we'll have several category pages each displaying a list of posts filtered by category. Technically, there will only be one page that will display a different list of posts based on an active route name. The `PostList` component will filter posts based on the current route.

Let's first set up the routes. Open `src/router.js` and replace the existing code with this:

```
import Vue from "vue";
import Router from "vue-router";
import Home from "../views/Home.vue";
import Post from "../views/PostView.vue";
```

```
Vue.use(Router);
```

```
export default new Router({
  mode: "history",
  base: process.env.BASE_URL,
  linkActiveClass: "active",
  routes: [
    {
      path: "/",
      name: "Featured",
      component: Home
    },
    {
      path: "/food",
      name: "Food",
      component: Home
    },
    {
      path: "/fashion",
      name: "Fashion",
      component: Home
    },
    {
      path: "/beauty",
      name: "Beauty",
```



```

    component: Home
  },
  {
    path: "/post/:slug",
    name: "Post",
    component: Post
  }
]
});

```

Now that we have our routes, let's set up our navigation menu. Open `src/App.vue` and replace the existing code with this:

```

<template>
  <div id="app">
    <b-navbar toggleable="md" type="dark" variant="info">
      <b-navbar-toggle target="nav_collapse"></b-navbar-toggle>
      <b-navbar-brand href="#">GraphCMS Vue</b-navbar-brand>
      <b-collapse is-nav id="nav_collapse">
        <b-navbar-nav>
          <router-link class="nav-link" to="/" exact>Home</router-link>
          <router-link class="nav-link" to="/food">Food</router-link>
          <router-link class="nav-link" to="/fashion">Fashion</router-link>
          <router-link class="nav-link" to="/beauty">Beauty</router-link>
        </b-navbar-nav>
      </b-collapse>
    </b-navbar>

    <b-container>
      <router-view/>
    </b-container>
  </div>
</template>

```

This will add a nav bar to the top of our site with links to our different categories.

Save the file and update the following files accordingly:

src/views/Home.vue

```

<template>
  <div class="home">
    <PostList />
  </div>
</template>

<script>
import PostList from "@components/PostList.vue";

export default {
  name: "home",
  components: {
    PostList
  }
};
</script>

```

src/components/PostList.vue

```

<template>
  <section class="post-list">
    <h1>{{ category }} Articles</h1>
    <hr/>
    <p>Put list of posts here!</p>
  </section>
</template>

```

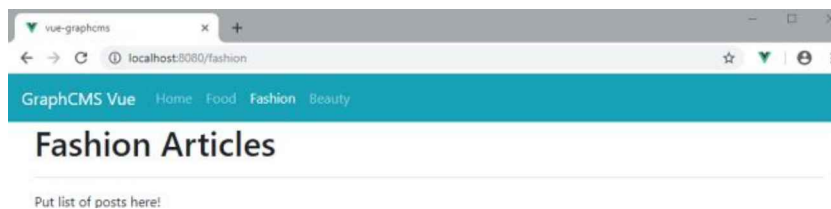
```

<script>
export default {
  name: "PostList",
  data() {
    return {
      category: ""
    };
  },
  created() {
    this.category = this.$route.name;
  },
  watch: {
    $route() {
      this.category = this.$route.name;
    }
  }
};
</script>

```

Notice that, in the `PostList` component, we're using a custom watcher to update our `category` data property, based on our current URL.

Now we're ready to perform a quick test to confirm the routes are working. Spin up the Vue.js server using the command `npm run serve`. Open a browser at `localhost:8080` and test each navigation link. The `category` property should output the same value we defined in route name's attribute.



Pulling in Data From GraphCMS

Now that we have our routing code working, let's see how we can pull information from our GraphCMS back end. At the root of your project, create an `env.local` file and populate it with values for the following fields:

```

VUE_APP_ENDPOINT=
VUE_APP_TOKEN=

```

Do note that Vue.js single-page applications only load custom environment variables starting with `VUE_APP`. You can find the API endpoint and token from your GraphCMS dashboard settings page. For the token, make sure to create one with `OPEN` permission, as that will allow both `READ` and `WRITE` operations. Next, create the file `src/graphcms.js` and copy the following code:

```

import axios from "axios";

export const ENDPOINT = process.env.VUE_APP_ENDPOINT;
const TOKEN = process.env.VUE_APP_TOKEN;

const headers = {
  "Content-Type": "application/json",
  Authorization: `Bearer ${TOKEN}`
};

export const apiClient = axios.create({
  headers
});

```

```

export const POSTS_BY_CATEGORY_QUERY = `
  query PostsByCategory($category: String!){
    category(where: {
      name: $category
    })
  }
`;

export const POST_BY_SLUG_QUERY = `
  query PostBySlug($slug: String!){
    post(where: {
      slug: $slug
    })
    {
      id
      title
      content
      categories {
        name
      }
      comments {
        name
        message
      }
    }
  }
`;

export const CREATE_COMMENT_MUTATION = `
  mutation CreateComment($post: PostWhereUniqueInput!, $name: String!, $message: String!){
    createComment(data: {
      name: $name,
      message: $message,
      post: {
        connect: $post
      },
      status: PUBLISHED
    })
    {
      id
      name
      message
    }
  }
`;

```

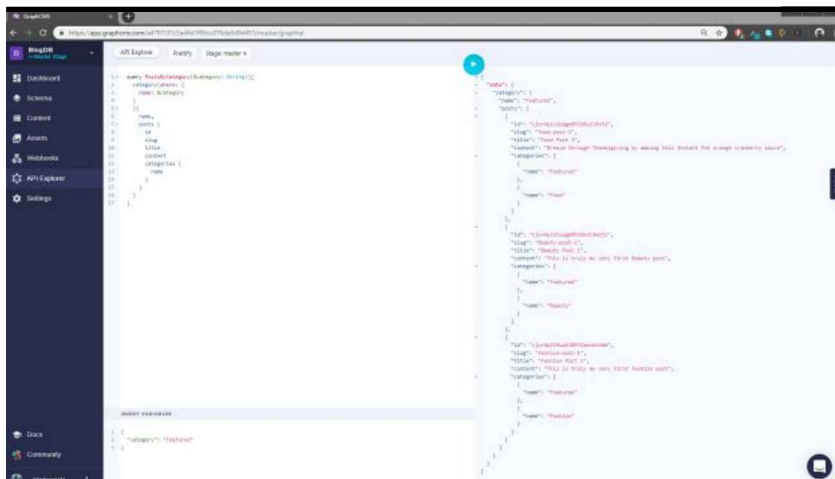
This helper file we just created provides two main functions:

- It creates an instance of axios that's configured to perform authorized requests to your GraphCMS back end.
- It contains GraphQL queries and mutations used in this project. These are responsible for fetching posts (either by category or by

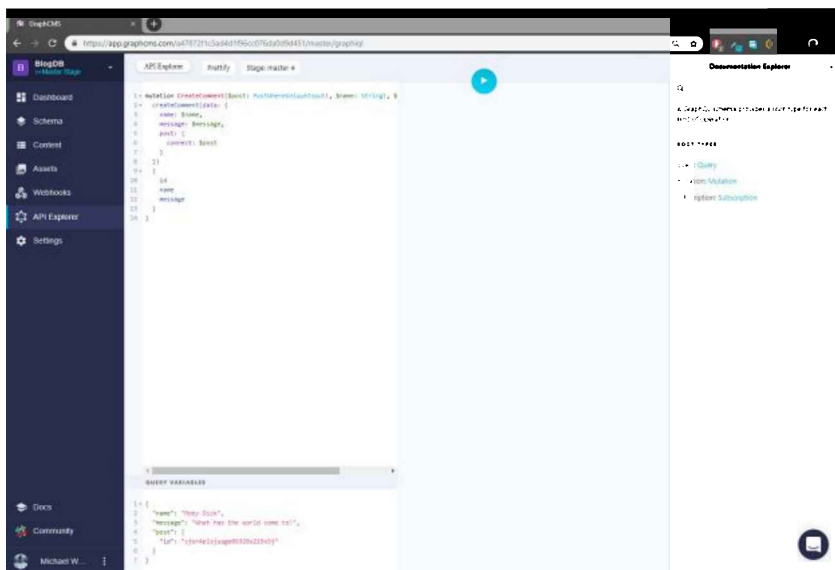
slug), as well as for creating new comments. If you'd like to find out more about GraphQL queries and mutations, please consult the GraphQL docs.

You can also use the API explorer in your project dashboard to test out these queries and mutations. To do this, copy the query or mutation from the code above and paste it into the top window of the API explorer. Enter any query variables in the window below that, then hit the *Play* button. You should see the results in a new pane on the right.

Here's a query example:



Here's a mutation example:



Displaying the Data in a Template

Now, let's create our HTML template in our `src/components/PostList.vue` that will display a list of posts in a neat way. We'll also add the axios code that will pull in posts data from our GraphCMS database:

```
<template>
  <section class="post-list">
    <h1>{{ category }} Articles</h1>
    <hr/>
    <b-row v-if="loading">
      <b-col class="text-center">
        <div class="lds-dual-ring"></div>
      </b-col>
    </b-row>
    <div v-if="!loading" >
```

```

    <b-card tag="article" v-for="post in posts" :key="post.id" :title="post.title" :sub-title="post.c
      <p class="card-text">
        {{ post.content }}
      </p>
      <router-link class="btn btn-primary" :to="'post/' + post.slug">
        Read Post
      </router-link>
    </b-card>
  </div>
</section>
</template>

```

```

<script>
import { ENDPOINT, apiClient, POSTS_BY_CATEGORY_QUERY } from "../graphcms.js";

```

```

export default {
  name: "PostList",
  data() {
    return {
      category: "",
      loading: false,
      posts: []
    };
  },
  methods: {
    async fetchPosts() {
      try {
        this.loading = true;
        const response = await apiClient.post(ENDPOINT, {
          query: POSTS_BY_CATEGORY_QUERY,
          variables: {
            category: this.category
          }
        });
      } catch (error) {
        console.log(error);
      }
    }
  },
  created() {
    this.category = this.$route.name;
    this.fetchPosts();
  },
  watch: {
    $route() {
      this.category = this.$route.name;
      this.posts = [];
      this.fetchPosts();
    }
  }
};
</script>

```

```

<style>
h1{
  margin-top: 25px !important;
}
.lds-dual-ring {

```

```

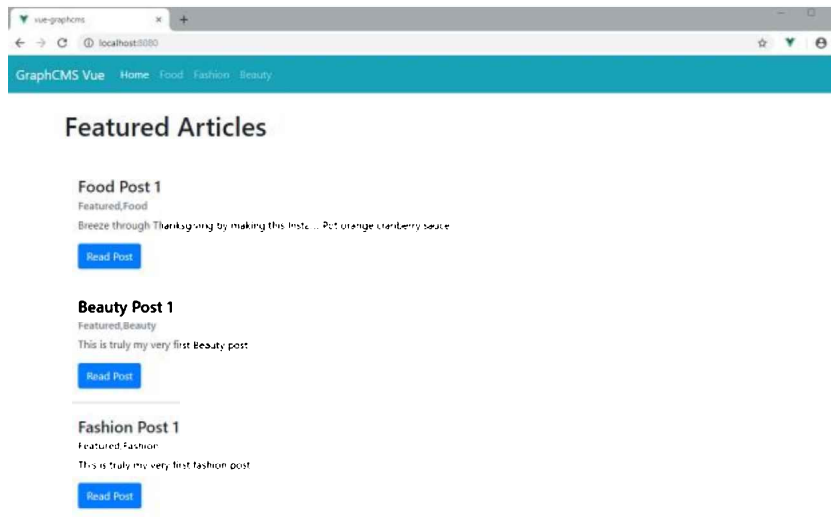
display: inline-block;
width: 64px;
height: 64px;
}
.lds-dual-ring:after {
  content: " ";
  display: block;
  width: 46px;
  height: 46px;
  margin: 1px;
  border-radius: 50%;
  border: 5px solid #ccc;
  border-color: #ccc transparent #ccc transparent;
  animation: lds-dual-ring 1.2s linear infinite;
}
@keyframes lds-dual-ring {
  0% {
    transform: rotate(0deg);
  }
  100% {
    transform: rotate(360deg);
  }
}
</style>

```

Let's quickly go through the code's main features:

- **Loading.** When a request is made, a loading spinner is displayed to indicate to the user there's something in progress. When the request is fulfilled, the loading spinner is replaced with a list of posts.
- **Query.** In order to get a list of posts by category, I found it easier to query for the category, then use the category-to-posts relationship to get access to the filtered posts.
- **Created.** When the page is loaded for the first time, the `fetchPosts()` function is called from within the `created` lifecycle hook.
- **Watch.** When the route URL changes, the `fetchPosts()` function is called each time.

After making those changes, you should now have the following view:



Displaying an Individual Post

Make sure the top main navigation is working as expected. Let's now work on the `Post` component. It will have its own `fetchPost()` function, where it will query by `slug`. If you're wondering where the `slug` parameter is coming from, let me remind you of this bit of code we put in `router.js`:

```

//...
{

```

```

    path: '/post/:slug',
    name: 'Post',
    component: Post
  },
  //...

```

This states that anything that comes after `/post/` in the URL is available to us in the component as `this.$route.params.slug`.

The `post` component is a parent of the `CommentForm` and `CommentList` components. The `comments` data will be passed as props to the `CommentList` component from the `Posts` record. Let's insert code for `src/components/CommentList.vue` now:

```

<template>
  <section class="comment-list">
    <hr/>
    <h4 class="text-muted">Comments</h4>
    <b-card v-for="comment in comments" :title="comment.name" title-tag="h5" :key="comment.id">
      <p class="card-text text-muted">{{ comment.message }} </p>
    </b-card>
    <p v-if="comments.length === 0" class="text-center text-muted">No comments posted yet!</p>
  </section>
</template>

<script>
export default {
  name: "CommentsList",
  props: ["comments"]
};
</script>

```

Unless you've manually entered comments via the GraphCMS dashboard, don't expect to see any results just yet. Let's add code to `src/components/CommentForm.vue` that will enable users to add comments to a blog post:

```

<template>
  <section class="comment-form">
    <h4 class="text-muted">Comment Form</h4>
    <b-form @submit.prevent="onSubmit">
      <b-form-group label="Name">
        <b-form-input id="input-name" type="text" v-model="name" placeholder="Enter your name" required="" />
      </b-form-group>
      <b-form-group label="Message">
        <b-form-textarea id="input-message" v-model="message" placeholder="Enter your comment" :rows="4" />
      </b-form-group>
      <b-button type="submit" variant="primary">Submit</b-button>
    </b-form>
  </section>
</template>

<script>
import { apiClient, ENDPOINT, CREATE_COMMENT_MUTATION } from "../graphcms.js";

export default {
  name: "CommentForm",
  props: ["post"],
  data() {
    return {
      name: "",
      message: ""
    };
  },
  methods: {
    async onSubmit() {
      const formattedComment = {
        name: this.name,

```

```

    message: this.message,
    post: {
      id: this.post.id
    }
  };
  try {
    const response = await apiClient.post(ENDPOINT, {
      query: CREATE_COMMENT_MUTATION,
      variables: formattedComment
    });

    const body = await response.data.data;
    const newComment = body.createComment;
    this.post.comments.push(newComment);
    this.name = "";
    this.message = "";
  } catch (error) {
    console.log(error);
  }
}
};
</script>

```

```

<style>
  .comment-form {
    margin-top: 35px;
  }
</style>

```

We now have a basic comment form capable of submitting a new comment to our GraphQL back-end system. Once the new comment is saved, we'll take the returned object and add it to the `post.comments` array. This should trigger the `CommentList` component to display the newly added `Comment`.

Let's now build the `src/components/Post.vue` component:

```

<template>
  <section class="post">
    <b-row v-if="loading">
      <b-col>
        <div class="lds-dual-ring text-center"></div>
      </b-col>
    </b-row>
    <b-row v-if="!loading">
      <b-col>
        <h1>{{post.title}}</h1>
        <h4 class="text-muted">{{post.categories.map(cat => cat.name).toString()}}</h4>
        <hr>
        <p>{{ post.content }}</p>
      </b-col>
    </b-row>
    <!-- List of comments -->
    <b-row v-if="!loading">
      <b-col>
        <CommentList :comments="post.comments" />
      </b-col>
    </b-row>
    <!-- Comment form -->
    <b-row v-if="!loading">
      <b-col>
        <CommentForm :post="post" />
      </b-col>
    </b-row>
  </section>

```



```

</section>
</template>

<script>
import { ENDPOINT, apiClient, POST_BY_SLUG_QUERY } from "../graphcms.js";
import CommentList from "@/components/CommentList";
import CommentForm from "@/components/CommentForm";

export default {
  name: "Post",
  components: {
    CommentList,
    CommentForm
  },
  data() {
    return {
      loading: false,
      slug: "",
      post: {}
    };
  },
  methods: {
    async fetchPost() {
      try {
        this.loading = true;
        const response = await apiClient.post(ENDPOINT, {
          query: POST_BY_SLUG_QUERY,
          variables: {
            slug: this.slug
          }
        });
      } catch (error) {
        console.log(error);
      }
    }
  },
  created() {
    this.slug = this.$route.params.slug;
    this.fetchPost();
  }
};
</script>

```

Finally, here's the code for `src/views/PostView.vue` to tie everything together:

```

<template>
  <div class="post-view">
    <Post/>
  </div>
</template>

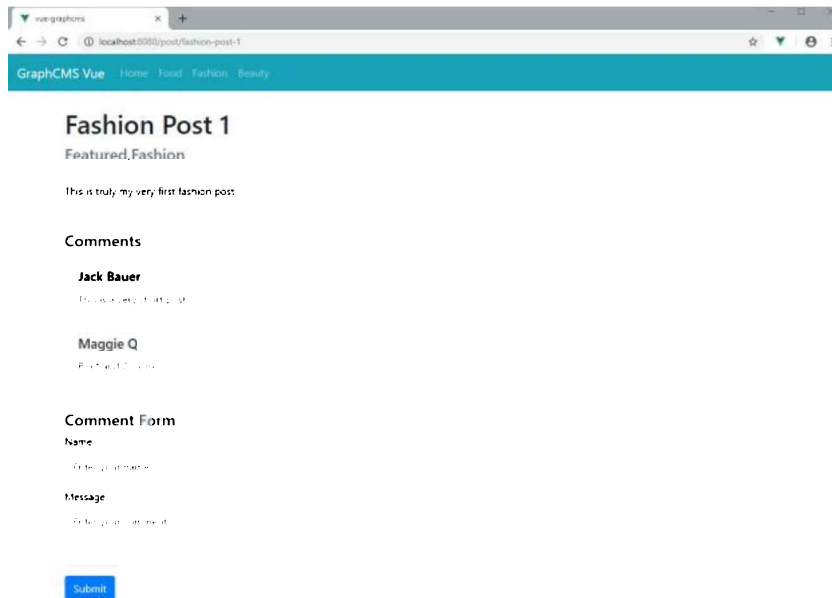
<script>
import Post from "@/components/Post.vue";

export default {
  name: "PostView",
  components: {
    Post
  }
}

```

```
};  
</script>
```

You should now have the following view for Posts. Take note of the `:slug` at the end of the URL `localhost:8080/post/fashion-post-1`:



In the above example, I've added a couple of comments to test out the new feature. Make sure you do the same on yours.

Summary

I hope you've seen how easy it is to build a blog website using Vue.js and GraphQL. If you'd been using plain PHP and MySQL, you'd have written much more code. Even with a PHP framework, you still would have written more code for a simple blog application.

For the sake of this tutorial, I had to keep things as simple as possible. You may note that this blog project is far from even meeting a minimalistic blog setup. There are several things we haven't tackled, such as error handling, form validation and caching. For the last bit, I recommend Apollo Client, as it has mechanisms for caching GraphQL query results. Then of course there needs to be an author model, and a proper comments system that supports authentication and message approval.

If you're up to it, please go ahead and take this simple Vue.js GraphCMS blog even further.

Chapter 5: How to Build a Chrome Extension with Vue

by James Hibbard

Browser extensions are small programs that can modify and enhance the functionality of a web browser. They can be used for a variety of tasks, such as blocking ads, managing passwords, organizing tabs, altering the look and behavior of web pages, and much more.

The good news is that browser extensions aren't difficult to write. They can be created using the web technologies you're already familiar with—HTML, CSS and JavaScript—just like a regular web page. However, unlike regular web pages, extensions have access to a number of browser-specific APIs, and this is where the fun begins.

In this tutorial, I'm going to show you how to build a simple extension for Chrome, which alters the behavior of the new tab page. For the JavaScript part of the extension, I'll be using the Vue.js framework, as it will allow us to get up and running quickly and is a lot of fun to work with.

Tutorial Code

The code for this tutorial can be found on [GitHub](#).

The Basics of a Chrome Extension

The core part of any Chrome extension is a manifest file and a background script. The manifest file is in a JSON format and provides important information about an extension, such as its version, resources, or the permissions it requires. A background script allows the extension to react to specific browser events, such as the creation of a new tab.

To demonstrate these concepts, let's start by writing a "Hello, World!" Chrome extension.

Make a new folder called `hello-world-chrome` and two files: `manifest.json` and `background.js`:

```
mkdir hello-world-chrome
cd hello-world-chrome
touch manifest.json background.js
```

Open up `manifest.json` and add the following code:

```
{
  "name": "Hello World Extension",
  "version": "0.0.1",
  "manifest_version": 2,
  "background": {
    "scripts": ["background.js"],
    "persistent": false
  }
}
```

The `name`, `version` and `manifest_version` are all required fields. The `name` and `version` fields can be whatever you want; the `manifest_version` should be set to 2 (as of Chrome 18).

The `background` key allows us to register a background script, listed in an array after the `scripts` key. The `persistent` key should be set to `false` unless the extension uses `chrome.webRequest` API to block or modify network requests.

Now let's add the following code to `background.js` to make the browser say hello when the extension is installed:

```
chrome.runtime.onInstalled.addListener(() => {
  alert('Hello, World!');
});
```

Finally, let's install the extension. Open Chrome and enter `chrome://extensions/` in the address bar. You should see a page displaying the extensions you've installed.

As we want to install our extension from a file (and not the Chrome Web Store) we need to activate *Developer mode* using the toggle in the top right-hand corner of the page. This should add an extra menu bar with the option *Load unpacked*. Click this button and select the `hello-world-chrome` folder you created previously. Click *Open* and you should see the extension installed and a "Hello, World!" popup appear.



Congratulations! You just made a Chrome extension.

Overriding Chrome's New Tab Page

The next step will be to have our extension greet us when we open up a new tab. We can do this by making use of the `Override Pages` API.

Disable Other Extensions

Before you progress, please make sure to disable any other extensions which override Chrome's new tab page. Only one extension at a time may alter this behavior.

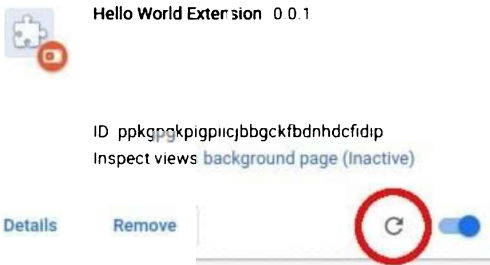
We'll start off by creating a page to display instead of the new tab page. Let's call it `tab.html`. This should reside in the same folder as your manifest file and background script:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>My New Tab Page!</title>
</head>
<body>
  <h1>My New Tab Page!</h1>
  <p>You can put any content here you like</p>
</body>
</html>
```

Next we need to tell the extension about this page. We can do so by specifying a `chrome_url_overrides` key in our manifest file, like so:

```
"chrome_url_overrides": {
  "newtab": "tab.html"
}
```

Finally, you need to reload the extension for the changes to take effect. You can do this by clicking the *reload* icon for the Hello World extension on Chrome's extensions page.



Now, when you open a new tab, you should be greeted by your custom message.

Adding Vue to the Extension

Now we have a very basic implementation of our extension up and running, the time has come to think about what the rest of the desired functionality will look like. When a user opens a new tab, I would like the extension to:

- Fetch a joke from the wonderful icanhazdadjoke.com.
- Display that joke in a nicely formatted manner to the user.
- Display a button for the user to favorite the joke. This will save the joke to `chrome.storage`.
- Display a button for the user to list favorited jokes.

You could, of course, do all of this with plain JavaScript, or a library like jQuery—and if that’s your thing, feel free!

For the purposes of this tutorial, however, I’m going to implement this functionality using Vue and the awesome `vue-web-extension-boilerplate`.

Using Vue allows me to write better, more organized code faster. And as we’ll see, the boilerplate provides several scripts that take the pain out of some of the common tasks when building a Chrome extension (such as having to reload the extension whenever you make changes).

vue-web-extension-boilerplate

This section assumes that you have Node and npm installed on your computer. If this isn’t the case, you can either head to the project’s home page and grab the relevant binaries for your system, or you can use a version manager. I would recommend using a version manager.

We’ll also need Vue CLI installed and the `@vue/cli-init` package:

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

With that done, let’s grab a copy of the boilerplate:

```
vue init kocal/vue-web-extension new-tab-page
```

This will open a wizard which asks you a bunch of questions. To keep this tutorial focused, I answered as follows:

```
? Project name new-tab-page
? Project description A Vue.js web extension
? Author James Hibbard <jim@example.com>
? License MIT
? Use Mozilla's web-extension polyfill? No
? Provide an options page? No
? Install vue-router? No
? Install vuex? No
? Install axios? Yes
? Install ESLint? No
? Install Prettier? No
? Automatically install dependencies? npm
```

You can adapt your answers to suit your preferences, but the main thing to be certain of is that you choose to install `axios`. We’ll be using this to fetch the jokes.

Next, change into the project directory and install the dependencies:

```
cd new-tab-page
npm install
```

And then we can build our new extension using one of the scripts the boilerplate provides:

```
npm run watch:dev
```

This will build the extension into a `dist` folder in the project root for development and watch for changes.

To add the extension to Chrome, go through the same process as outlined above, making sure to select the `dist` folder as the extension directory. If all goes according to plan, you should see a “Hello world!” message when the extension initializes.

Project Setup

Let’s take a minute to look around our new project and see what the boilerplate has given us. The current folder structure should look like this:

```
.
├── dist
│   └── <the built extension>
├── node_modules
│   └── <one or two files and folders>
├── package.json
├── package-lock.json
├── scripts
│   ├── build-zip.js
│   └── remove-evals.js
├── src
│   ├── background.js
│   ├── icons
│   │   ├── icon_128.png
│   │   ├── icon_48.png
│   │   └── icon.xcf
│   ├── manifest.json
│   └── popup
│       ├── App.vue
│       ├── popup.html
│       └── popup.js
└── webpack.config.js
```

As you can see, from the config file in the project root, the boilerplate is using webpack under the hood. This is awesome, as this gives us Hot Module Reloading for our background script.

The `src` folder contains all of the files we’ll be using for the extension. The manifest file and `background.js` should be familiar, but also notice a `popup` folder containing a Vue component. When the boilerplate builds the extension into the `dist` folder, it will pipe any `.vue` files through the `vue-loader` and output a JavaScript bundle which the browser can understand.

Also in the `src` folder is an `icons` folder. If you look in Chrome’s toolbar, you should see a new icon for our extension (also known as the browser action). This is being pulled from this folder. If you click it, you should see a popup open which displays “Hello world!” This is created by `popup/App.vue`.

Finally, note a `scripts` folder containing two scripts—one to remove `eval` usages to comply with the Content Security Policy of Chrome Web Store and one to package your extension into a `.zip` file, which is necessary when uploading it to the Chrome Web Store.

There are also various scripts declared in the `package.json` file. We’ll be using `npm run watch:dev` for developing the extension and later on `npm run build-zip` to generate a ZIP file to upload to the Chrome Web Store.

Using a Vue Component for the New Tab Page

Start off by removing the annoying `alert` statement from `background.js`.

Now let’s make a new `tab` folder in the `src` folder to house the code for our new tab page. We’ll add three files to this new folder—`App.vue`, `tab.html`, `tab.js`:

```
mkdir src/tab
```

```
touch src/tab/{App.vue,tab.html,tab.js}
```

Open up `tab.html` and add the following:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>New Tab Page</title>
  <link rel="stylesheet" href="tab.css">
</head>
<body>
  <div id="app"></div>
  <script src="tab.js"></script>
</body>
</html>
```

Nothing special going on here. This is a simple HTML page which will hold our Vue instance.

Next, in `tab.js` add:

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

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

Here we import `Vue`, pass a selector for the element that we want it to replace with our application, then tell it to render our `App` component.

Finally, in `App.vue`:

```
<template>
  <p>{{ message }}</p>
</template>

<script>
export default {
  data () {
    return {
      message: "My new tab page"
    }
  }
}
</script>

<style scoped>
p {
  font-size: 20px;
}
</style>
```

Before we can use this new tab page, we need to update the manifest file:

```
{
  "name": "new-tab-page",
  ...
  "chrome_url_overrides": {
    "newtab": "tab/tab.html"
  }
}
```

And we also need to have the boilerplate compile our files and copy them over to the `dist` folder, so that they're available to the

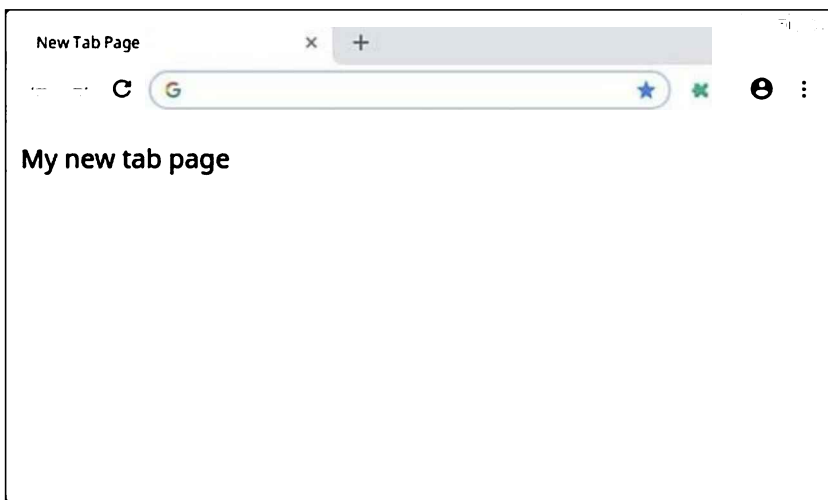
extension.

Alter `webpack.config.js` like so, updating both the `entry` and `plugins` keys:

```
entry: {
  'background': './background.js',
  'popup/popup': './popup/popup.js',
  'tab/tab': './tab/tab.js'
},

plugins: [
  ...
  new CopyWebpackPlugin([
    { from: 'icons', to: 'icons', ignore: ['icon.xcf'] },
    { from: 'popup/popup.html', to: 'popup/popup.html', transform: transformHtml },
    { from: 'tab/tab.html', to: 'tab/tab.html', transform: transformHtml },
    ...
  ])
]
```

You'll need to restart the `npm run watch:dev` task for these changes to take effect. Once you've done this, reload the extension and open a new tab. You should see "My new tab page" displayed.



Fetching and Displaying Jokes

Okay, so we've overridden Chrome's new tab page and we've replaced it with a mini Vue app. Now let's make it do more than display a message.

Alter the template section in `src/tab/App.vue` as follows:

```
<template>
  <div>
    <div v-if="loading">
      <p>Loading...</p>
    </div>
    <div v-else>
      <p class="joke">{{ joke }}</p>
    </div>
  </div>
</template>
```

Change the `<script>` section to read as follows:

```
<script>
import axios from 'axios';

export default {
  data () {
```



```

    return {
      loading: true,
      joke: "",
    }
  },
  mounted() {
    axios.get(
      "https://icanhazdadjoke.com/",
      { 'headers': { 'Accept': 'application/json' } }
    )
      .then(res => {
        this.joke = res.data.joke;
        this.loading = false;
      });
  }
}
</script>

```

And finally, change the `<style>` section to read as follows:

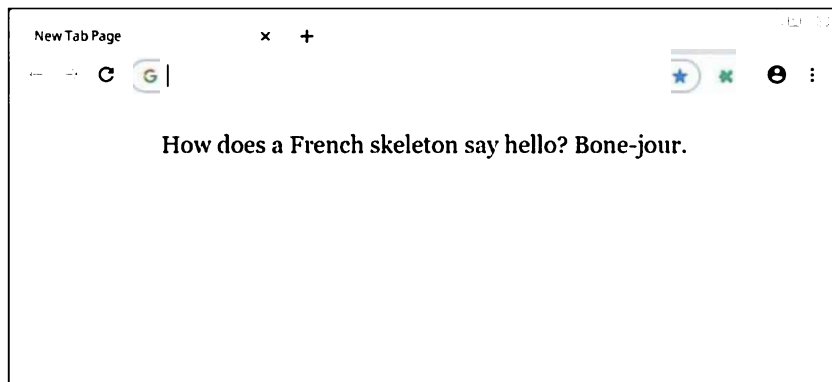
```

<style>
body {
  height: 98vh;
  text-align: center;
  color: #353638;
  font-size: 22px;
  line-height: 30px;
  font-family: Merriweather, Georgia, serif;
  background-size: 200px;
  display: flex;
  align-items: center;
  justify-content: center;
}

.joke {
  max-width: 800px;
}
</style>

```

If you're running the `npm run watch:dev` task, the extension should automatically reload and you should see a joke displayed whenever you open a new tab page.



Once you've verified it's working, let's take a minute to understand what we've done.

In the template, we're using a v-if block to either display a loading message or a joke, depending on the state of the `loading` property. Initially, this will be set to `true` (displaying the loading message), then our script will fire off an Ajax request to retrieve the joke. Once the Ajax request completes, the `loading` property will be set to `false`, causing the component to be re-rendered and our joke to be displayed.

In the `<script>` section, we're importing `axios`, then declaring a couple of data properties—the aforementioned `loading` property and

a `joke` property to hold the joke. We're then making use of the `mounted` lifecycle hook, which fires once our Vue instance has been mounted, to make an Ajax request to the joke API. Once the request completes, we update both of our data properties to cause the component to re-render.

So far, so good.

Persisting Jokes to Chrome's Storage

Next, let's add some buttons to allow the user to favorite a joke and to list out favorited jokes. As we'll be using Chrome's storage API to persist the jokes, it might be worth adding a third button to delete all of the favorited jokes from storage.

First, add the following to the bottom of `manifest.json`:

```
"permissions": [ "storage" ]
```

This gives the extension access to the `chrome.storage` API.

Reload the extension, then add the buttons to the `v-else` block:

```
<div v-else>
  <p class="joke">{{ joke }}</p>

  <button @click="likeJoke" :disabled="likeButtonDisabled">Like Joke</button>
  <button @click="logJokes" class="btn">Log Jokes</button>
  <button @click="clearStorage" class="btn">Clear Storage</button>
</div>
```

Nothing too exciting here. Note the way that we are binding the like button's `disabled` property to a data property on our Vue instance to determine its state. This is because a user shouldn't be able to like a joke more than once.

Next, add the click handlers and the `likeButtonDisabled` to our script section:

```
export default {
  data () {
    return {
      loading: true,
      joke: "",
      likeButtonDisabled: false
    }
  },
  methods: {
    likeJoke(){
      chrome.storage.local.get("jokes", (res) => {
        if(!res.jokes) res.jokes = [];
        res.jokes.push(this.joke);
        chrome.storage.local.set(res);
        this.likeButtonDisabled = true;
      });
    },
    logJokes(){
      chrome.storage.local.get("jokes", (res) => {
        if(res.jokes) res.jokes.map(joke => console.log(joke));
      });
    },
    clearStorage(){
      chrome.storage.local.clear();
    }
  },
  mounted() { ... }
}
```

Here we've declared three new methods to deal with the three new buttons.

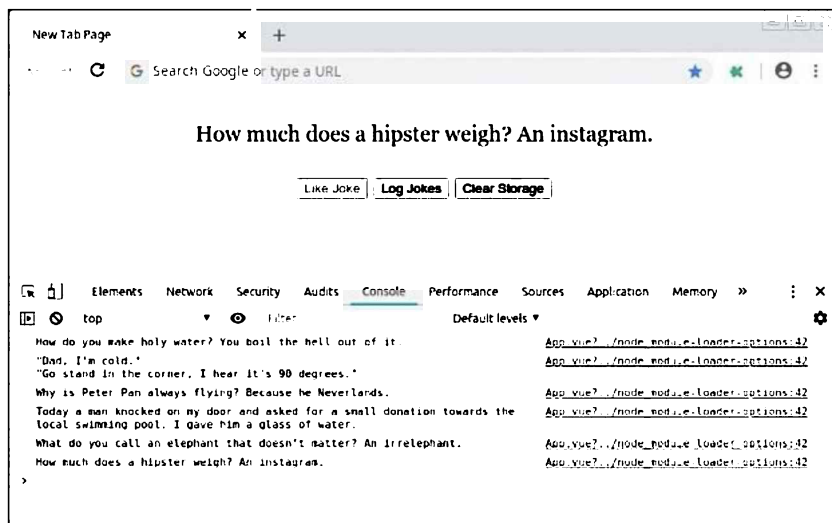
The `likeJoke` method looks for a `jokes` property in Chrome's storage. If it's missing (that is, the user has yet to like a joke), it initializes it to an empty array. Then it pushes the current joke onto this array and saves it back to storage. Finally it sets the

likeButtonDisabled data property to true, disabling the like button.

The logJokes method also looks for a jokes property in Chrome's storage. If it finds one, it loops over all of its entries and logs them to the console.

Hopefully what the clearStorage method does is clear.

Go ahead and try this new functionality in the extension and satisfy yourself that it works.



Adding Some Polish to the Extension

Okay, so that seems to work, but the buttons are ugly and the page is a little plain. Let's finish off this section by adding some polish to the extension.

As a next step, install the vue-awesome library. This will allow us to use some Font Awesome icons in our page and make those buttons look a bit nicer:

```
npm install vue-awesome
```

Register the library with our Vue app in src/tab/tab.js:

```
import Vue from 'vue';
import App from './App';
import "vue-awesome/icons";
import Icon from "vue-awesome/components/Icon";
```

```
Vue.component("icon", Icon);
```

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

Now alter the template like so:

```
<template>
  <div>
    <div v-if="loading" class="centered">
      <p>Loading...</p>
    </div>
    <div v-else>
      <p class="joke">{{ joke }}</p>

      <div class="button-container">
        <button @click="likeJoke" :disabled="likeButtonDisabled" class="btn"><icon name="thumbs-up"></i>
        <button @click="logJokes" class="btn"><icon name="list"></icon></button>
```

```
        <button @click="clearStorage" class="btn"><icon name="trash"></icon></button>
    </div>
</div>
</div>
</template>
```

Finally, let's add some more styling to the buttons and include a picture of everyone's favorite dad:

```
<style>
body {
  height: 98vh;
  text-align: center;
  color: #353638;
  font-size: 22px;
  line-height: 30px;
  font-family: Merriweather, Georgia, serif;
  background: url("https://dablnmslvvntp.cloudfront.net/wp-content/uploads/2018/12/1544189726troll-dad.");
  background-size: 200px;
  display: flex;
  align-items: center;
  justify-content: center;
}

.joke {
  max-width: 800px;
}

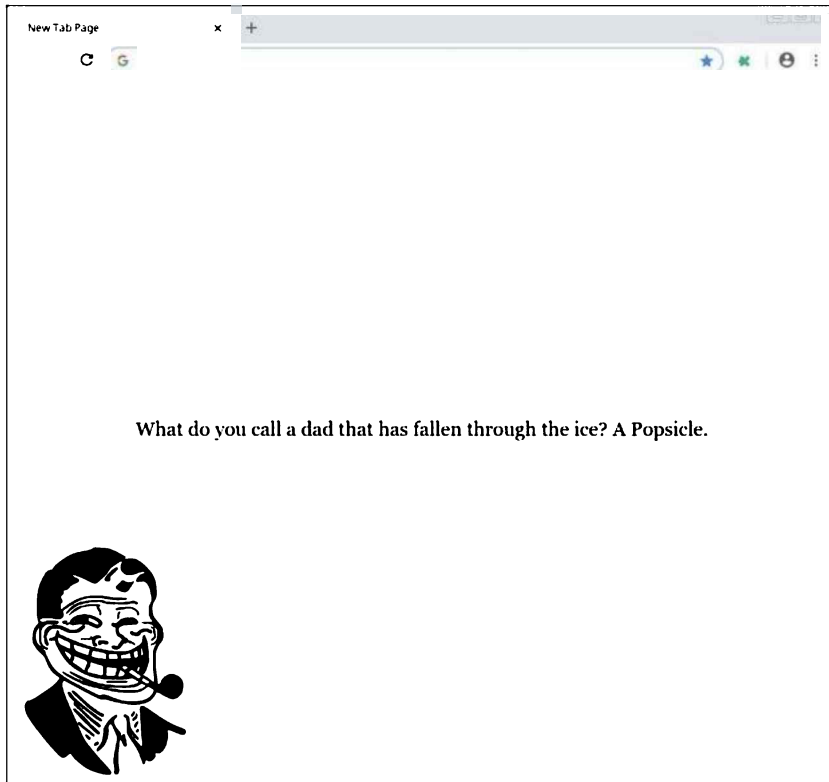
.button-container {
  position: absolute;
  right: 0px;
  top: calc(50% - 74px);
}

.btn {
  background-color: #D8D8D8;
  border: none;
  color: white;
  padding: 12px 16px;
  font-size: 16px;
  cursor: pointer;
  display: block;
  margin-bottom: 5px;
  width: 50px;
}

.btn:hover {
  background-color: #C8C8C8;
}

.btn.disabled {
  background-color: #909090;
}
</style>
```

The extension should reload. Try opening a new tab and you should see something like this.



Uploading the Extension to the Chrome Web Store

Should you want to make your extension available for others to download, you'd do this via the Chrome Web Store.

The first thing you'll need in order to do this is a Google account, which you can use to sign in to the Developer Dashboard. You'll be prompted for your developer details, and before you publish your first app, you must pay a one-time \$5 developer signup fee (via credit card).

Next, you need to create a ZIP file for your app. You can do this locally by running the `npm run build-zip`. This will create a `dist-zip` folder in your project root, containing a ZIP file ready to upload to the Web Store.

For a minimal extension, this is all you'd really need to do. However, before you upload anything, it's worth reading the official Publish in the Chrome Web Store guide.

Conclusion

And with that, we're done. In this tutorial, I've highlighted the main parts of a Chrome extension and shown how to use the `vue-web-extension` boilerplate to build an extension using `Vue.js`. We finished off by looking at how to upload an extension to the Web Store and everything that involves.

I hope you enjoyed this tutorial and can use it to get started building Chrome extensions of your own. Let me know if you make anything cool.

Chapter 6: Build Your Own Link-sharing Site with Nuxt.js and vue-kindergarten

by Nilson Jacques

In this tutorial, we're going to create a our own link-sharing news site, along the lines of Echo JS or Hacker News, complete with comments and upvoting. The tech stack we'll be using consists of Vue.js, the Nuxt.js Vue framework, and an access-control/authorization library called vue-kindergarten.

This guide is not intended as a "deep dive" into any of the libraries used. Rather, the aim is to give you some exposure to these tools and how they can be used together to create a real-world application.

Node and npm

To follow along, you'll need to have a recent version of Node.js installed, and your copy of npm needs to be $\geq 5.2.0$.

The GitHub Repo

There's a GitHub repo available with the finished code.

Installing Nuxt.js

We're going to use Nuxt.js for this project. Although it's well known for its server-side rendering (SSR) functionality, we won't actually be using this part of Nuxt. Instead, we'll take advantage of Nuxt's convention-over-configuration approach to reduce the amount of boilerplate we have to write to get a single page application up and running.

Open up a terminal/command prompt and create a new folder. I've called mine "yans" for "Yet Another News Site", but feel free to name yours as you like:

```
mkdir yans && cd yans
```

Commands

These commands are for Linux, and will probably work on macOS too. Windows users will have to translate accordingly!

To bootstrap the project, launch the Nuxt.js installer via npm:

```
npx create-nuxt-app client
```

The npx tool will download and run the installer, and you'll be presented with a series of questions. This isn't an in-depth tutorial on Nuxt, so I'm not going to go into detail on these.

Here are the selections you'll need to select to follow along with this guide:

1. *Project name* (your choice)
2. *Project description* (your choice)
3. *Use a custom framework* ("none")
4. *Use a custom UI framework* ("bulma")
5. *Choose rendering mode* ("Single Page App")
6. *Use axios module* ("yes")
7. *Use eslint* (your choice)
8. *Use prettier* (your choice)
9. *Author name* (your choice)
10. *Choose a package manager* (your choice)

Linting and Coding Styles

If you install ESLint and Prettier (as I did for this tutorial) you'll need to run `npm run lint -- --fix` after installation, or your Nuxt project won't compile.

The code formatting used in this guide reflects the default configuration. If this bothers you, feel free to change the settings to your liking (Prettier will even add those semicolons back in for you!) or opt out altogether (**recommended**).

When the installer has finished, it's time to change into the newly created `client` folder and start building the site!

Layout and Styling

We're going to be using the Bulma CSS framework to style our news site. Nuxt helpfully gave us the option to add this from the installer, but we'll also need to pull in Font Awesome for the icons.

The most straightforward way to do this is to edit the Nuxt config file and add a link to the CDN CSS file.

client/nuxt.config.js

```
/*
** Headers of the page
*/
head: {
  title: pkg.name,
  meta: [
    { charset: 'utf-8' },
    { name: 'viewport', content: 'width=device-width, initial-scale=1' },
    { hid: 'description', name: 'description', content: pkg.description }
  ],
  link: [
    { rel: 'icon', type: 'image/x-icon', href: '/favicon.ico' },
    {
      rel: 'stylesheet',
      href: 'https://use.fontawesome.com/releases/v5.6.3/css/all.css'
    }
  ]
},
```

We're adding a second item to the link array, under the head property. This will add the CDN link to our SPA's template.

The next thing we're going to want to do is customize the basic layout, to add a menu bar that will appear across all the pages on the site.

Open up layouts/default.vue, and replace the contents with the following code.

client/layouts/default.vue

```
<template>
  <div>
    <nav
      class="navbar is-info"
      role="navigation"
      aria-label="main navigation"
    >
      <div class="navbar-brand">
        <nuxt-link
          class="navbar-item"
          to="/"
        >
          YANS!
        </nuxt-link>

        <a
          role="button"
          class="navbar-burger burger"
          aria-label="menu"
          aria-expanded="false"
          data-target="navbarBasicExample"
        >
          <span aria-hidden="true"/>
          <span aria-hidden="true"/>
          <span aria-hidden="true"/>
        </a>
      </div>
  </div>
```

```

<div class="navbar-menu">
  <div class="navbar-start">
    <nuxt-link
      to="/"
      class="navbar-item"
    >Top</nuxt-link>
    <nuxt-link
      to="/new"
      class="navbar-item"
    >New</nuxt-link>
    <nuxt-link
      to="/submit"
      class="navbar-item"
    >Submit</nuxt-link>
  </div>

  <div class="navbar-end">
    <div class="navbar-item">
      <div class="buttons">
        <nuxt-link
          to="/register"
          class="button is-primary"
        >
          <strong>Register</strong>
        </nuxt-link>
        <nuxt-link
          to="/login"
          class="button is-light"
        >Log in</nuxt-link>
      </div>
    </div>
  </div>
</div>
</nav>
<nuxt/>
</div>
</template>

```

The basic markup for the navigation bar comes from Bulma's own documentation, but we're replacing the anchor tags with Nuxt's own `<nuxt-link>` components, which allow us to interact with the instance of Vue Router it uses. Also, notice that, down near the bottom of the template, we have the `<nuxt/>` element, so the router knows where to display our content.

As you can see, our menu bar has links for *Top* (posts in order of the most voted), *New* (most recent posts), and *Submit* (a form for submitting new links).

Faking the Back End

Rather than cover building a back end for the site (a topic best left to a separate guide) we're going to create a fake API server. Normally for this sort of task I'd reach for `json-server`, which does a great job of turning a JSON file into a RESTful mock API, but we're also going to need to support authentication.

Fortunately, there's a module called `json-server-auth` which builds on `json-server` to add JWT authentication and access control. Let's install that and get it set up as our fake back end.

Switch from your `client` folder into the parent directory. Create a new folder called `server` and change into it, and init a new `package.json` file:

```

cd ..
mkdir server && cd server
npm init -y

```

Also install the `json-server-auth` package:

```

npm i json-server-auth

```


If you're not familiar with the original json-server package, the way it works is that you create a JSON file which is used as a data store for the mock API. The file contains an object with keys for each of the endpoints you wish your API to have (such as "users").

Our file will look as follows.

server/db.json

```
{
  "users": [],
  "posts": [],
  "comments": []
}
```

This is the basic structure we need for our news site. You can find a copy of db.json, prepopulated with sample data in the project repo.

Under normal circumstances, that would be enough to get a mock back end up and running. However, as we're going to be using the the Nuxt community's Auth module, we need to do a little customization first.

After the Auth module makes a successful authentication request, it makes a request to a "user" endpoint, which it expects to return a JSON object representing the logged-in user.

The json-server-auth module doesn't provide this functionality out of the box, so we're going to add it ourselves. Create a new file called index.js, and add the following code.

server/index.js

```
const jsonServer = require('json-server')
const auth = require('json-server-auth')

const app = jsonServer.create()
const router = jsonServer.router('db.json')

const { rewriter } = require('json-server-auth/dist/guards')

const rules = rewriter({
  user: 660
})

app.use(rules)

app.db = router.db

app.use(auth)
app.get('/user', (req, res, next) => {
  const { email } = req.claims
  const { db } = req.app
  const user = Object.assign({}, db.get('users').find({ email }).value())
  delete user.password
  res.json({ user })
})
app.use(router)

app.listen(3001)
console.log('API server started on port 3001')
```

Most of this code comes from the json-server-auth documentation, but this is the code I've added:

```
app.get('/user', (req, res, next) => {
  const { email } = req.claims
  const { db } = req.app
  const user = Object.assign({}, db.get('users').find({ email }).value())
  delete user.password
  res.json({ user })
})
```

When a request is made to the /users endpoint, this handler gets the user's email (from the JWT) and fetches the user from the JSON file

DB. It removes the password field from the object, before returning it in the response.

Now that's done, we can spin up an instance of our back end and leave it running as we develop the site itself:

```
node index.js
```

If you're following along, you'll also need to start the Nuxt server in the `client` folder with:

```
npm run dev
```

Proxying with the axios Module

We're going to take advantage of one of the features of Nuxt's `axios` module: `proxying`. This will allow us to treat our API as if it's on the same server as the site itself, making requests to `/api/users` instead of something like `http://localhost:3001/users`.

To set this up, all we have to do is add some settings to the Nuxt configuration file.

`client/nuxt.config.js`

```
/*
** axios module configuration
*/
axios: {
  proxy: true
},

proxy: {
  '/api/': { target: 'http://localhost:3001', pathRewrite: { '^/api/': '' } }
},
```

What we're doing here is telling `axios` that we want to use a `proxy`, and then defining a separate `proxy` key with details of the URL we want to map.

In this case, we're taking any requests to URLs beginning in `/api/` and redirecting them to our mock API server. The `pathRewrite` option strips off the `/api/` segment from the path, as we don't need it as part of the server URL.

Now that we've got our back-end server set up and the proxy configured, let's start adding some functionality to our site.

Authentication

As I mentioned before, we're going to use the `Auth` module plugin for Nuxt. With a little configuration, this module provides us with some helpful methods for authenticating a user with our server, and some Nuxt middleware for restricting routes to logged-in users.

The first thing we need to do is install the module from `npm` (don't forget to change back to the `client` folder before this step!):

```
npm i @nuxtjs/auth
```

Once this has downloaded, we need to open the Nuxt config file again, and add some settings.

`client/nuxt.config.js`

```
/*
** Nuxt.js modules
*/
modules: [
  '@nuxtjs/auth',
  // Doc: https://github.com/nuxt-community/axios-module#usage
  '@nuxtjs/axios',
  // Doc:https://github.com/nuxt-community/modules/tree/master/packages/bulma
  '@nuxtjs/bulma'
],
```

First of all, we need to tell Nuxt about the plugin, by adding the module name to the `modules` array as above.

Directly below the `modules` section, let's add a new key for the `auth` module's settings.

`client/nuxt.config.js`

```

auth: {
  strategies: {
    local: {
      endpoints: {
        login: {
          url: '/api/login',
          method: 'post',
          propertyName: 'accessToken'
        },
        logout: false,
        user: { url: '/api/user', method: 'get' }
      }
    }
  }
},

```

The auth module allows you to configure different “strategies” (that is, different authentication methods), and we want to set up the “local” strategy which supports JWT-based authentication.

Within the configuration for the strategy, we need to specify details of “login”, “logout”, and “user” endpoints for the module to make requests to, and how it can retrieve the auth token. Note that we’re setting `logout` to `false` as our authentication API doesn’t support this method. The module will just delete the cached token and revert to a logged out state.

The other thing we need to do to get the module working is have a Vuex store, as it uses this to store the authentication state and the logged-in user. We’ll also be using a Vuex for our application, so let’s go ahead and set this up as an empty store.

client/store/index.js

```

import Vuex from 'vuex'

const createStore = () => {
  return new Vuex.Store({
    state: {},
    getters: {},
    actions: {},
    mutations: {}
  })
}

export default createStore

```

Nuxt requires our store to be exported as a factory function due to how the SSR functionality works, but other than that this is a regular Vuex store.

We’ll also need a couple of pages to handle registration and logging in. Let’s create those next.

Registration

First, create a new file called `register.vue` within the `pages` folder. Let’s look at the template section before we add the script block.

client/pages/register.vue

```

<template>
  <section class="section">
    <div class="container">

      <div class="field">
        <label class="label">Username</label>
        <div class="control">
          <input
            v-model="username"
            class="input"
            type="text"
            autofocus
          >

```

```

    </div>
  </div>

  <div class="field">
    <label class="label">Email</label>
    <div class="control">
      <input
        v-model="email"
        class="input"
        type="text"
      >
    </div>
  </div>

  <div class="field">
    <label class="label">Password</label>
    <div class="control">
      <input
        v-model="password"
        class="input"
        type="password"
      >
    </div>
  </div>

  <div class="field">
    <div class="control">
      <button
        :disabled="isSaving"
        class="button is-link"
        @click="registerUser"
      >Register</button>
    </div>
  </div>
</section>
</template>

```

Here we've just got three basic input fields, all connected via `v-model` to data properties on the component. When the submit button is clicked, it will call the `registerUser` method in the component.

And now the code:

```

<script>
import { mapActions } from 'vuex'

export default {
  data: () => ({
    isSaving: false,
    username: null,
    email: null,
    password: null
  }),
  methods: {
    ...mapActions(['saveUser']),
    async registerUser() {
      this.isSaving = true

      await this.saveUser({
        username: this.username,
        email: this.email,
        password: this.password
      })
    }
  }
}

```

```

    this.$auth.login({
      data: {
        email: this.email,
        password: this.password
      }
    })

    this.$router.push('/')
  }
}
</script>

```

The `registerUser` method is pretty straightforward. We set an `isSaving` flag to `true`, which prevents the form being resubmitted while we're processing it. We then dispatch a Vuex action, `saveUser`, passing in the three input values and waiting for it to complete. Lastly, we call the auth module's `login` method with the same values to log the newly registered user in, and redirect to the home page.

One of the benefits to using Nuxt is that we don't have to manually create routes for the pages. Any component created in the `pages` folder will automatically have a route generated for it with the same name.

We don't actually have a `saveUser` action in our store yet, so let's write that now.

client/store/index.js

```

actions: {
  async saveUser(context, { username, email, password }) {
    const data = {
      id: uuid(),
      username,
      email,
      password,
      timestamp: new Date().getTime()
    }

    try {
      await this.$axios.post('/api/register', data)
    } catch (error) {
      console.error(error)
    }
  }
}

```

This action takes the email, username, and password data passed into it and builds a new user object. Notice that we're assigning a UUID as the user ID here. Although `json-server-auth` will assign an ID for us if we leave it blank, I prefer to use a UUID.

We'll need to install the module for generating UUIDs via npm, and import it at the top of the store:

```
npm i uuid
```

client/store/index.js

```

import Vuex from 'vuex'
import uuid from 'uuid/v4'

```

The `axios` module makes `axios` available in the store as `this.$axios`, making it straightforward to send a POST request to our API server. We're just going to log any errors to the console for now, but you'd probably want to notify the user of the error somehow (for example, with the Nuxt toast module).

The finished page should look like this:

Username

Email

Password

Register

Login

Let's tackle the login page next. Create `login.vue` within the `pages` folder, and add the following template.

client/pages/login.vue

```
<template>
  <section class="hero">
    <div class="hero-body">
      <div class="container has-text-centered">
        <div class="column is-4 is-offset-4">
          <h3 class="title has-text-grey">Login</h3>
          <p class="subtitle has-text-grey">Please log in to proceed.</p>
          <div class="box">
            <form @submit.prevent="onSubmit">
              <div class="field">
                <div class="control">
                  <input
                    v-model="email"
                    class="input"
                    type="email"
                    placeholder="Your Email"
                    autofocus
                  >
                </div>
              </div>
              <div class="field">
                <div class="control">
                  <input
                    v-model="password"
                    class="input"
                    type="password"
                    placeholder="Your Password"
                  >
                </div>
              </div>
              <button class="button is-block is-info is-fullwidth">Login</button>
            </form>
          </div>
          <div
            v-if="showError"
            class="notification is-danger"
          >
            <strong>Oops - we couldn't log you in!</strong><br>
            Please check your email and password and try again.
          </div>
        </div>
      </div>
    </div>
  </section>
```

```
</template>
```

Most of the markup here is for applying styling, courtesy of Bulma. The important things to note are that we've got `v-model` properties set on the inputs, so the component can access the values, and we've got an `onSubmit` handler attached to the form (along with a `.prevent` modifier, to prevent the default action). There's also an error message that will be shown if the `showError` data property is set to `true`.

Here's the script section:

```
<script>
export default {
  data: () => ({
    email: '',
    password: '',
    showError: false
  }),
  methods: {
    async onSubmit() {
      try {
        await this.$auth.login({
          data: {
            email: this.email,
            password: this.password
          }
        })
      } catch (error) {
        this.showError = true
      }
    }
  }
}
</script>
```

This should be fairly self-explanatory, but we're calling the `auth` module's `login` method and passing the credentials from the form. If an error is thrown, we set `showError` to `true`, alerting the user that there was a problem.

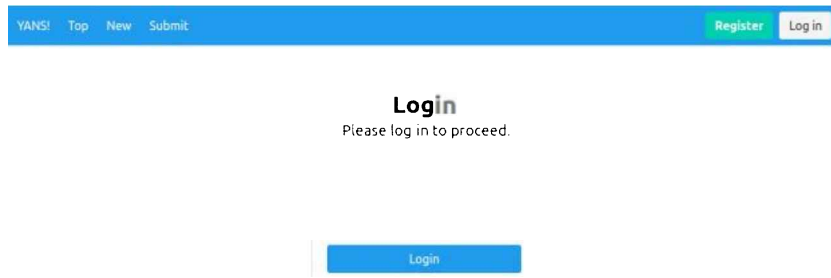
Now that we've added registration and login functionality, we should go back and tweak our layout slightly, to change the menu based on the current user's authentication state.

client/layouts/default.vue

```
<div class="navbar-end">
  <div class="navbar-item">
    <div class="buttons">
      <nuxt-link
        v-if="!$auth.loggedIn"
        to="/register"
        class="button is-primary"
      >
        <strong>Register</strong>
      </nuxt-link>
      <nuxt-link
        v-if="!$auth.loggedIn"
        to="/login"
        class="button is-light"
      >Log in</nuxt-link>
      <button
        v-else
        class="button is-light"
        @click="$auth.logout()"
      >Log out</button>
    </div>
  </div>
</div>
```

The auth module provides us with a helper (`$auth.loggedIn` from within a template) for checking the logged-in state. We can use it to hide the “register” and “log in” buttons if the current user is authenticated and display a “log out” button instead.

Here’s what the finished page looks like:



Adding New Posts

Now that we have our authentication set up, and a way for users to register and log in, let’s create a page to allow users to submit new links for our news site.

Go to the `pages` folder and create a new file called `submit.vue`.

client/pages/submit.vue

```
<template>
  <section class="section">
    <div class="container">

      <div class="field">
        <label class="label">Title</label>
        <div class="control">
          <input
            v-model="title"
            class="input"
            type="text"
            placeholder="Text input"
          >
        </div>
      </div>

      <div class="field">
        <label class="label">Full URL (inc. http(s)://)</label>
        <div class="control">
          <input
            v-model="url"
            class="input"
            type="text"
            placeholder="Text input"
          >
        </div>
      </div>

      <div class="field">
        <div class="control">
          <button
            :disabled="isSaving"
            class="button is-link"
            @click="submitPost"
          >Submit</button>
        </div>
      </div>
    </div>
  </section>
</template>
```



```
</section>
</template>
```

The template here is very similar in structure to our registration form: just a couple of input fields with associated `v-model` properties, and a submit button that triggers a handler on the component.

Here's the script section:

```
<script>
import { mapActions } from 'vuex'

export default {
  middleware: 'auth',
  data: () => ({
    isSaving: false,
    title: null,
    url: null
  }),
  methods: {
    ...mapActions(['savePost']),
    async submitPost() {
      this.isSaving = true

      await this.savePost({
        title: this.title,
        url: this.url
      })

      this.title = null
      this.url = null
      this.isSaving = false
    }
  }
}
</script>
```

This section is also very similar to what we did before, taking the details and dispatching the `savePost` action to the store. On completion, we're simply resetting the form to allow further submissions. It would be a good idea to pop up a notification at this point to inform the user their link has been saved, which I'll leave as an exercise for the reader.

Note that for this component, we're specifying a `middleware` property of "auth". This middleware is part of the `auth` module, and will restrict access to the page to logged-in users. If a guest user tries to access the page, they'll be redirected to the login form.

The form should look like this:



The screenshot shows a web interface with a blue header bar. On the left side of the header, there is a navigation menu with the text "YANSI: Top New Submit". On the right side of the header, there is a "Log out" button. Below the header, there are two input fields. The first field is labeled "Title" and the second field is labeled "URL". Below these fields is a blue "Submit" button.

Let's go ahead and write the `savePost` action.

clients/store/index.js

```
actions: {
  // ...
  async savePost(context, { title, url }) {
    const user = this.$auth.user
    const data = {
      id: uuid(),
```

```

    userId: user.id,
    author: user.username,
    title,
    url,
    votes: [],
    timestamp: new Date().getTime()
  }

  try {
    await this.$axios.post('/api/posts', data)
  } catch (error) {
    console.error(error)
  }
},
},
},

```

We start off by getting a reference to the current user from the auth module (available within our store as `this.$auth`). We're then constructing an object literal with the properties for the new post. We're also adding an empty `votes` array, as this is how we're going to keep track of the users who've upvoted a post.

Once the data object is built, we're using the `axios` module to submit it via a POST request to our back-end API.

Fetching and Displaying Posts

Now that we have a way for users to submit links, we're going to need a way to retrieve those and display them, so let's build out the "new posts" page.

Before creating the page itself, let's make the necessary changes to the store.

client/store/index.js

```

state: {
  posts: []
},

```

First of all, we need to add a `posts` array to the store state, to hold the posts once we've retrieved them from the API.

Next, we'll need an action to make the API request.

client/store/index.js

```

actions: {
  // ...
  async getPosts({ commit }) {
    try {
      const res = await this.$axios.get('/api/posts')
      commit('setPosts', { posts: res.data })
    } catch (error) {
      console.error(error)
    }
  },
},

```

Here we're just making a GET request to the `posts` endpoint of the API, and passing the result along to the `setPosts` mutation. Let's add that.

client/store/index.js

```

mutations: {
  setPosts(state, { posts }) {
    state.posts = posts
  }
},

```

This is a simple mutation that simply replaces the existing `posts` array with the one we get back from the API.

For our “new posts” view, we’re going to display the links in descending order of creation. Although this logic could go in the component itself, let’s create a Vuex getter. This will enable us to access the sorted list of posts elsewhere in the app should we want to.

client/store/index.js

```
getters: {
  newPosts: state => state.posts.sort((a, b) => a.timestamp - b.timestamp)
},
```

As with computed properties within components, if the `posts` array is modified, the `newPosts` getter will update and be re-rendered by any components using it.

Displaying the “Top Posts” Page

The page to display the top voted posts is virtually identical to this one, except that it uses a different getter to return the posts in a different order. You can have a go at building this out yourself, or take a look at this guide’s accompanying code repo.

Let’s add the listing page. Create a new file under `pages` called `new.vue`, and add the following code.

client/pages/new.vue

```
<template>
  <section class="section">
    <div class="container">
      <h1 class="title">Latest Links</h1>
      <div>
        <news-item
          v-for="post in newPosts"
          :key="post.id"
          :item="post"
        />
      </div>
    </div>
  </section>
</template>

<script>
import { mapActions, mapGetters } from 'vuex'
import NewsItem from '~/components/NewsItem'

export default {
  components: { NewsItem },
  fetch({ store }) {
    store.dispatch('getPosts')
  },
  computed: mapGetters(['newPosts'])
}
</script>
```

I’ve shown the entire file here, as it’s only a relatively small component. It contains the Nuxt-specific `fetch` method, which is called whenever the user tries to navigate to the page. The method receives a context object containing the store (among other things), allowing us to dispatch actions before the component is loaded.

Once the `getPosts` action has completed, the `newPosts` getter will be updated, and the component can render the list of posts. We’re delegating the display of each post to the `<NewsItem>` component here, which is actually a custom component. Let’s take a look at that next.

The NewsItem Component

Displaying a post seems like something we might want to do in multiple places on our news site, so it makes sense to extract this into a separate component. It also encapsulates any markup and display logic, helping to keep the parent components cleaner and focused on the code needed to do their jobs.

Let’s go to the `components` folder and create a new component file called `NewsItem.vue`. As we’ve done previously, we’ll look at the template first.

client/components/NewsItem.vue

```
<template>
  <article class="media">
    <figure class="media-left">
      <button class="button is-white">
        <span class="icon is-small">
          <i
            class="fas fa-thumbs-up"
            title="Vote up" />
          </span>
        </button>
      </figure>
      <div class="media-content">
        <div>
          <p>
            <a href="item.url"><strong>{{ item.title }}</strong></a> <small>({{ item.url | domain }})</small>
          </p>
        </div>
        <nav class="level is-mobile">
          <div class="level-left">
            <span class="level-item">
              <small>{{ votes }} · posted by {{ item.author }} · {{ item.timestamp | timeAgo }}</small>
            </span>
            <nuxt-link
              :to="`/news/${item.id}`"
              class="level-item"
            >
              <span class="icon is-small"><i class="fas fa-comments" /></span>
            </nuxt-link>
          </div>
        </nav>
      </div>
    </article>
  </template>
```

A lot of this markup is part of Bulma's media object, which provides us with a nice structure to build on. You'll notice we've got an upvote button (which we'll wire up a bit later on), and a couple of output filters: `domain` and `timeAgo`. Let's look at the JavaScript and see what they're doing.

client/components/NewsItem.vue

```
<script>
import parseDomain from 'parse-domain'
import distanceInWordsStrict from 'date-fns/distance_in_words_strict'

export default {
  filters: {
    domain(value) {
      if (!value) return ''
      const parts = parseDomain(value)
      return parts ? `${parts['domain']}.${parts['tld']}` : ''
    },
    timeAgo(value) {
      if (!value) return ''
      return distanceInWordsStrict(new Date(), new Date(value), {
        addSuffix: true
      })
    }
  },
  props: {
    item: {
      type: Object,
```

```

    required: true
  }
},
computed: {
  votes() {
    const votes = this.item.votes.length
    const suffix = votes === 1 ? 'vote' : 'votes'
    return `${votes} ${suffix}`
  }
}
}
</script>

```

As you can see from the component code, we're importing two third-party modules to help with the formatting: `parse-domain`, and the `date-fns` function `distanceInWordsStrict`.

The `parse-domain` module takes a URL and returns an array of the constituent parts. Using this, we can display just the domain of the link, so the user knows at a glance which site the content comes from. The `distanceInWordsStrict` function takes two dates and returns a human-friendly description of the time elapsed (such as "5 minutes ago").

Dependencies!

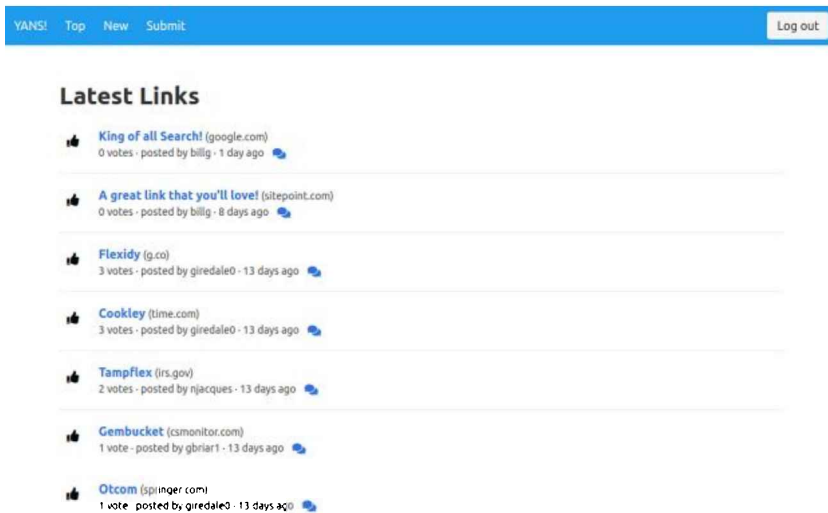
Don't forget to install the dependencies!

```
npm i parse-domain date-fns
```

Both of these could have been implemented as computed functions, but because their purpose is to *format* output for display, using filters seemed more appropriate. In addition, we could easily re-implement them as global filters, if we needed to use them in other components of the app, without having to change the template.

One place we *are* using a computed property is to display the number of votes and correctly pluralize the text when necessary.

With the `page` and `NewsItem` component complete, the screen should look something like this with the dummy data:



Handling Comments

Having a list of links is all well and good, but we could do with a way to get users interacting more with the site. Let's add a comments view to allow our users to discuss the links that get posted.

This page is a little different from the ones we've created so far. Since we're going to be displaying the comments for an individual post, we want the URL to contain the post ID. In this case, we want our URL to conform to the format `/news/6d369e63-da45-4bc3-8853-47106636e256`, where the ID is a dynamic parameter.

To do this in Nuxt, we name our page component `pages/news/_id.vue`. The underscore prefix lets Nuxt know it needs to make this part of the route dynamic, and make the value available to us via a router param.

Let's create the file, and add in the following code.

client/pages/news/_id.vue

```
<template>
  <section class="section">
    <div class="container">
      <news-item :item="selectedPost"/>
      <comment-form
        v-if="$auth.loggedIn"
        @post="saveComment"
      />
      <p v-else>
        Please <nuxt-link to="/register">register</nuxt-link> or
        <nuxt-link to="/login">log in</nuxt-link> to comment.
      </p>
    </div>
  </section>
</template>

<script>
import { mapActions, mapState } from 'vuex'
import NewsItem from '~/components/NewsItem'
import CommentForm from '~/components/CommentForm'

export default {
  fetch({ store, params }) {
    const postId = params.id
    return store.dispatch('getComments', { postId })
  },
  components: { CommentForm, NewsItem },
  computed: mapState(['selectedPost']),
  methods: mapActions(['saveComment'])
}
</script>

<style scoped>
p {
  text-align: center;
  margin: 2em;
}
</style>
```

The `fetch` method here is retrieving the dynamic route parameter `id` from the context object and dispatching the `getComments` action to the store, passing the parameter.

We're making use of the `NewsItem` component again to display the selected post, and we're also rendering a `CommentForm` component (that we'll create in a moment). We're only going to display the form if the user is authenticated. Otherwise, we'll prompt them to either log in or register. The form component emits a `post` event, which we're using to directly trigger the store action `saveComment`.

client/components/CommentForm.vue

```
<template>
  <article class="media">
    <div class="media-content">
      <div class="field">
        <p class="control">
          <textarea
            v-model="body"
            class="textarea"
            placeholder="Add a comment..."
          />
        </p>
      </div>
    </div>
  </article>
```

```

<div class="field">
  <p class="control">
    <button
      :disabled="!body"
      class="button"
      @click="commentPosted"
    >Post comment</button>
  </p>
</div>
</div>
</article>
</template>

```

```

<script>
export default {
  data: () => ({
    body: null
  }),
  methods: {
    commentPosted() {
      this.$emit('post', { body: this.body })
      this.body = null
    }
  }
}
</script>

```

The `CommentForm` component is pretty straightforward, as we're only dealing with a single input. When the user clicks the button, the component emits a `post` event with the comment text as the payload.

Let's create the two store actions we need for this page to work.

client/store/index.js

```

state: {
  posts: [],
  selectedPost: null
},

```

Firstly, we add a `selectedPost` property to hold the post the user is viewing.

client/store/index.js

```

actions: {
  // ...
  async getComments({ commit }, { postId }) {
    try {
      const res = await this.$axios.get(
        `/api/posts/${postId}?_embed=comments`
      )
      commit('setSelectedPost', { post: res.data })
    } catch (error) {
      console.error(error)
    }
  },
  async saveComment({ commit, state }, { body }) {
    const user = this.$auth.user
    const data = {
      id: uuid(),
      postId: state.selectedPost.id,
      userId: user.id,
      author: user.username,
      body,
      timestamp: new Date().getTime()
    }
  }
}

```

```

    }

    try {
      const res = await this.$axios.post('/api/comments', data)
      commit('addComment', { comment: res.data })
    } catch (error) {
      console.error(error)
    }
  }
},

```

Both of these actions are similar to the ones we created previously, so there's not much to point out about this code. One thing that's worth mentioning, though, is that we're re-requesting the selected post from the API, but appending `?_embed=comments` to the URL. This will tell `json-server-auth` to return the post with all the associated comments attached as an array.

We also need a couple of additional mutations.

client/store/index.js

```

mutations: {
  // ...
  addComment(state, { comment }) {
    state.selectedPost.comments.push(comment)
  },
  setSelectedPost(state, { post }) {
    state.selectedPost = post
  },
},

```

Committing `setSelectPost` assigns the passed `post` object to the `selectedPost` key in the store's state. The `addComment` mutation takes the comment (having already been persisted by our API) and pushes it into the `comments` array on our current post, allowing the view to update.

Displaying Posted Comments

You might have noticed we aren't displaying the comments anywhere in our page! Let's rectify that by updating the `NewsItem` to display comments if they're present on the post object. Open up the component and add the following markup to the template, below the closing `</nav>` tag.

client/components/NewsItem.vue

```

<article
  v-for="comment in comments"
  :key="comment.id"
  class="media"
>
  <figure class="media-left">
    <p class="image is-48x48">
      
    </p>
  </figure>
  <div class="media-content">
    <div class="content">
      <p>
        {{ comment.body }}
        <br>
        <small>{{ comment.author }} · {{ comment.timestamp | timeAgo }}</small>
      </p>
    </div>
  </div>
</article>

```

Also add a new computed property to the component object:

```

computed: {

```

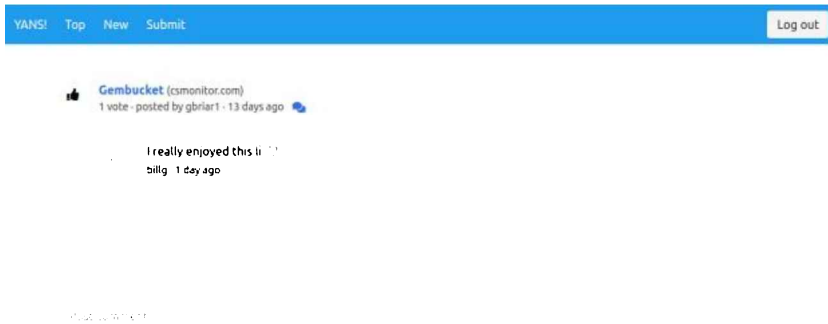


```

comments() {
  return this.item.comments || []
},
votes() {
  // ...
}
}

```

And voila! You should now see a nicely formatted list of comments appear beneath the post as you add them!



Upvoting Posts

The only thing that's left to add to our news site is the ability for users to upvote quality posts. Let's start by adding the necessary action and mutation to the store.

client/store/index.js

```

actions: {
  // ...
  async upvotePost({ commit }, { postId }) {
    const postUrl = `/api/posts/${postId}`
    try {
      const { data } = await this.$axios.get(postUrl)
      data.votes.push(this.$auth.user.id)
      const res = await this.$axios.put(postUrl, data)
      commit('updatePost', { post: res.data })
    } catch (error) {
      console.error(error)
    }
  }
},

```

Our `upvotePost` action receives the ID of the post to upvote. Firstly, we re-fetch the post from the API, and push the current user's ID into the `votes` array. After saving the post back to the server, we commit the `updatePost` mutation, which we'll write next.

client/store/index.js

```

mutations: {
  // ...
  updatePost(state, { post }) {
    const index = state.posts.findIndex(p => p.id === post.id)
    if (index >= 0) {
      state.posts.splice(index, 1, post)
    }
    if (state.selectedPost && state.selectedPost.id === post.id) {
      state.selectedPost.votes = post.votes
    }
  }
},

```

This mutation looks up the index of the post that's been upvoted, and replaces it with the copy returned from the server. If the user is

voting from one of the list views, they'll see the vote count update dynamically. We also check if `selectedPost` is set and is the same as the upvoted post and updates the `votes` property.

Let's wire up these new actions to our existing list and comments views, so users can start voting on their favorite links.

First of all, we need to make a small change to the `NewsItem` component, so it can emit an event when the upvote button is clicked.

client/components/NewsItem.vue

```
<button
  class="button is-white"
  @click="$emit('upvote')"
>
  <span class="icon is-small">
    <i
      class="fas fa-thumbs-up"
      title="Vote up" />
    </span>
  </button>
```

Next, let's amend the "New Posts" page.

client/pages/new.vue

```
<news-item
  v-for="post in newPosts"
  :key="post.id"
  :item="post"
  @upvote="upvotePost(post)"
/>
```

In the template, all that's required is to add a listener for the `upvote` event. There's no payload emitted with the event, we're simply passing the current post in the loop to the component's `upvotePost` method.

client/pages/new.vue

```
methods: {
  ...mapActions(['upvotePost']),
  onUpvote(post) {
    this.upvotePost({ postId: post.id })
  }
},
```

In the component object, we've added the `onUpvote` method, which just extracts the ID from the `post` object and dispatches it to the store's `upvotePost` action.

The comments page is similar.

client/pages/news/_id.vue

```
<news-item
  :item="selectedPost"
  @upvote="onUpvote(selectedPost)"
/>
```

The component's `onUpvote` method is slightly different, as we have to remember to set the `updateSelectedPost` flag.

client/pages/news/_id.vue

```
methods: {
  ...mapActions(['saveComment', 'upvotePost']),
  onUpvote(post) {
    this.upvotePost({ postId: post.id })
  }
},
```

Authorization

One thing you may have noticed while working on the last section is that there's nothing to stop a user from upvoting a post more than once. In fact, there's nothing to stop them from upvoting their own posts, either, which seems a little unfair!

Rather than put the logic to check for this into our components, we're going to make use of an authorization library called `vue-kindergarten`. The benefit of using a library like this is that it allows us to separate out the authorization logic from our components, preventing it from bloating them with extra responsibilities, and making that logic easy to re-use across our application.

So, let's install the package from npm:

```
npm i vue-kindergarten
```

Once it's downloaded, we need to register it with Vue.js. Normally we'd do this in the file where we create our main Vue instance, but with Nuxt we do this by creating a plugin file.

client/plugins/vue-kindergarten.js

```
import Vue from 'vue'
import VueKindergarten from 'vue-kindergarten'

Vue.use(VueKindergarten, {
  child: store => store && store.state.auth.user
})
```

The code is pretty straightforward: we import Vue and the `vue-kindergarten` library, and register it as we would in a normal Vue app. The plugin takes an `options` object with a `child` property. This should be a function that will be passed an instance of the Vuex store and return the current user.

vue-kindergarten Terminology

The library's terminology revolves around pre-school, with concepts like sandboxes, perimeters, and governesses, so wherever you see "child", read "user".

With the plugin file created, we just need to add a reference to it to the `plugins` array in the Nuxt config file.

client/nuxt.config.js

```
plugins: ['~/plugins/vue-kindergarten'],
```

Our next step is to create a **perimeter**. This is a file that contains all the authorization logic for a particular entity or situation. Make a new folder within the Nuxt project called `perimeters`, and add a new file called `postsPerimeter.js` with the following content.

client/perimeters/postsPerimeter.js

```
import { createPerimeter } from 'vue-kindergarten'

export default createPerimeter({
  purpose: 'post',

  can: {
    upvote(post) {
      return (
        this.isAuthenticated() &&
        this.isAuthor(post) === false &&
        post.votes.includes(this.child.id) === false
      )
    }
  },

  isAuthenticated() {
    return this.child !== null
  },

  isAuthor(post) {
    return this.child.id === post.userId
  }
})
```

```
  }  
  })
```

Here we're importing the `createPerimeter` function from the `vue-kindergarten` module, and using it to define a new perimeter, or set of rules. The `purpose` property lets us define the name of the entity that the rules apply to. When we add the perimeter to a component, its rules become available under the `$post` variable within the template.

The perimeter contains a couple of helper methods for checking if the user is logged in, and if they're the author of a given post. We also have a `can` property that lets us define actions the current user is allowed to take. Here, we've added an `upvote` method, which will return `true` only if the user is authenticated, is *not* the author of the post, and hasn't already upvoted the post.

Let's see how we can use this perimeter within our "New Posts" page to control who is allowed to vote on any given post.

client/pages/new.vue

```
<script>  
import { mapActions, mapGetters } from 'vuex'  
import postsPerimeter from '~/perimeters/postsPerimeter'  
import NewsItem from '~/components/NewsItem'  
  
export default {  
  components: { NewsItem },  
  fetch({ store }) {  
    store.dispatch('getPosts')  
  },  
  computed: mapGetters(['newPosts']),  
  methods: {  
    ...mapActions(['upvotePost']),  
    onUpvote(post) {  
      if (this.$isAllowed('upvote', post)) {  
        this.upvotePost({ postId: post.id })  
      }  
    }  
  },  
  perimeters: [postsPerimeter]  
}
```

The sharp-eyed among you will have spotted that we're importing our new perimeter, and adding it to a new `perimeters` array on the component. The only other change here is to the `onUpvote` method, where we're using the `this.$isAllowed` helper to check if the vote should be submitted.

As you can see, all the checking logic is neatly encapsulated in the perimeter. If we decide to add an additional requirement in future (for example, that the user must have confirmed their email address, or must have a minimum ranking on the site) we can do that without requiring any changes to our components.

Summary

And there we have it: you've now built your own news site that you can use to start your own version of Reddit! In all seriousness though, I do hope you've enjoyed following along and have picked up some useful knowledge that you can apply to future projects of your own.

If you'd like to take a more in-depth look at Nuxt.js, and Vue in general, check out my new SitePoint book called *Jump Start Vue.js*.

Chapter 7: An Introduction to Data Visualization with Vue and D3.js

by Christopher Vundi

Web applications are normally data-driven and oftentimes the need arises to visualize this data. That's where charts and graphs come in. They make it easier to convey information, as well as demonstrate correlations or statistical relationships. Information presented in the form of a chart or a graph is also easier for a non-native speaker to understand.

In this tutorial, we'll learn how to visualize data in a Vue project. For this, we'll be using the popular D3.js library, which combines powerful visualization components and a data-driven approach to DOM manipulation.

Let's get started.

The GitHub Repo

The code for this tutorial can be found on GitHub.

What is D3?

As you can read on the project's home page, D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG, and CSS. Its emphasis on web standards gives you the full capabilities of modern browsers without tying yourself to a proprietary framework.

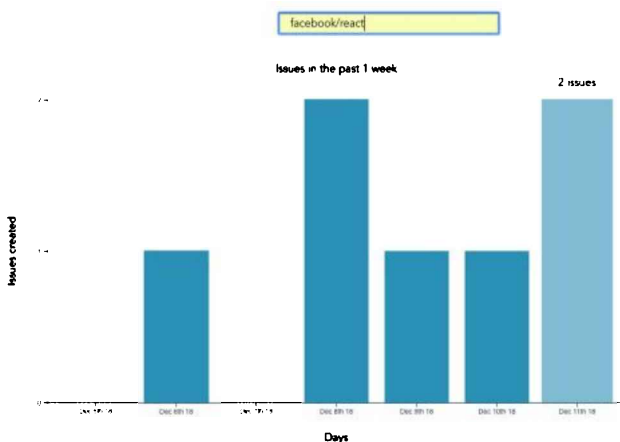
Whereas most people will refer to D3.js as a data visualization library, it's *not*. D3 is more of a framework comprising different parts—such as jQuery parts (which help us select and manipulate DOM elements), Lodash parts, animation parts, data analysis parts, and data visualization parts.

In this tutorial, we'll be working with the visualization aspect of D3. The real meat of D3 when visualizing data is:

- the availability of functions for decorating data with drawing instructions
- creating new drawable data from source data
- generating SVG paths
- creating data visualization elements (like an axis) in the DOM from your data and methods

What We'll Be Building

We want to create an app that lets users search for a repo on GitHub, then get a visual representation of issues opened in the past week that are still open. The end result will look like this:



Prerequisites

This tutorial assumes you have a working knowledge of Vue. Previous knowledge of D3.js isn't required, but if you'd like to get up to speed

quickly, you might want to read our [D3 by example tutorial](#).

You'll also 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.

Finally, we'll be using the following packages to build our app:

- [Vue CLI](#)—to scaffold out the project
- [D3.js](#)—to visualize our data
- [Lodash](#)—which provides a handful of utility methods
- [Moment JS](#)—for date and time formatting
- [axios](#)—an HTTP client to help us make requests to an external API

New Vue Project

I prefer creating new Vue projects using Vue CLI. (If you're not familiar with Vue CLI, our [beginner's guide](#) in this Vue series gives a full introduction.) Vue CLI provides a nice folder structure for placing different sections of the code, such as styles, components, and so on.

Make sure that the CLI is installed on your machine:

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

Then create a new project with the following command:

```
vue create issues-visualization
```

Picking Presets

While creating a new project using Vue CLI, you'll be prompted to pick a preset. For this particular project, we'll just stick with the default (Babel + ESLint).

Once our new Vue project has been created, we `cd` into the project folder and add the various node modules we'll need:

```
npm install lodash d3 axios moment
```

Even though this is a simple app that doesn't have many running parts, we'll still take the components approach instead of dumping all the code inside the `App.vue` file. We're going to have two components, the `App` component and a `Chart` component that we're yet to create.

The `App` component will handle fetching data from GitHub, then pass this data to the `Chart` component as props. The actual drawing of the chart will happen inside the `Chart` component. Structuring things this way has the advantage that, if you want to use a library other than `axios` to fetch the data, it'll be easier to swap it out. Also, if you want to swap `D3` for a different charting library, that'll be easier too.

Building the Search Interface

We'll start by building a search interface that lets users enter the name of the repo they want to see visualized.

In `src/App.vue`, get rid of everything inside the `<template>` tag and replace the content with this:

```
<template>
  <div id="app">
    <form action="#" @submit.prevent="getIssues">
      <div class="form-group">
        <input
          type="text"
          placeholder="owner/repo Name"
          v-model="repository"
          class="col-md-2 col-md-offset-5"
        >
      </div>
    </form>
  </div>
</template>
```

Here we have a form which, upon submission, prevents the browser's default submission action, then calls a `getIssues` method that we're yet to define. We're also using a `v-model` directive to bind the input from the form to a `repository` property inside the `data`

model of our Vue instance. Let's declare that `repository` property as an empty string. We'll also add a `startDate` property, which we'll later use as the first date in our time range:

```
import moment from "moment";
import axios from "axios";

export default {
  name: "app",
  data() {
    return {
      issues: [],
      repository: "",
      startDate: null
    };
  },
  methods: {
    getIssues() {
      // code goes in here
    }
  }
};
```

Now on to creating the `getIssues` method:

```
getIssues() {
  this.startDate = moment()
    .subtract(6, "days")
    .format("YYYY-MM-DD");

  axios
    .get(
      `https://api.github.com/search/issues?q=repo:${this.repository}+is:issue+is:open+created:>=${this.startDate}`
      { params: { per_page: 100 } }
    )
    .then(response => {
      const payload = this.getDateRange();

      response.data.items.forEach(item => {
        const key = moment(item.created_at).format("MMM Do YY");
        const obj = payload.filter(o => o.day === key)[0];
        obj.issues += 1;
      });

      this.issues = payload;
      console.log(this.issues);
    });
}
```

In the above block of code, we start by setting the `startDate` data property to six days ago and formatting it for use with the GitHub API.

We then use `axios` to make an API request to GitHub to get all issues for a particular repository that were opened in the past week and that are still open. You can refer to GitHub's search API if you need more examples on how to come up with query string parameters.

When making the HTTP request, we set the results count to 100 per page (the max possible). There are hardly any repositories with over 100 new issues per week, so this should be fine for our purposes. By default, the `per_page` value is 30.

If the request completes successfully, we use a custom `getDateRange` method to initialize a `payload` variable that we will be able to pass to the `Chart` component. This payload is an array of objects that will look like so:

```
[
  {day: "Dec 7th 18", issues: 0},
  {day: "Dec 8th 18", issues: 0},
  {day: "Dec 9th 18", issues: 0},
  {day: "Dec 10th 18", issues: 0},
```

```

{day: "Dec 11th 18", issues: 0},
{day: "Dec 12th 18", issues: 0},
{day: "Dec 13th 18", issues: 0}
]

```

After that, we iterate over the API's response. The data we're interested in is in an `items` key on a `data` property on the response object. From this, we take the `created_at` key (which is a timestamp) and format it as the `day` property in our objects above. From there, we then look up the corresponding date in the `payload` array and increment the `issues` count for that date by one.

Finally, we assign the `payload` array to our `issues` data property and log the response.

Next, let's add in the `getDateRange` method:

```

methods: {
  getDateRange() {
    const startDate = moment().subtract(6, 'days');
    const endDate = moment();
    const dates = [];

    while (startDate.isSameOrBefore(endDate)) {
      dates.push({
        day: startDate.format('MMM Do YY'),
        issues: 0
      });

      startDate.add(1, 'days');
    }

    return dates;
  },
  getIssues() { ... }
}

```

Before we get to the visualization bit, let's also log any errors we might encounter when making our request to the console (for debugging purposes):

```

axios
  .get(...)
  .then(response => {
    ...
  })
  .catch(error => {
    console.error(error);
  });

```

We'll add some UX for informing the user in the case that something went wrong later.

So far, we have an input field that lets the user enter the organization/repository name they wish to search issues for. Upon form submission, all issues opened in the past one week are logged to the console.

Below is an example of what was logged on the console for the `facebook/react` repo:

```

▼ (7) [{...}, {...}, {...}, {...}, {...}, {...}, {...}] ⓘ
  ▶ 0: {day: "Dec 6th 18", issues: 1}
  ▶ 1: {day: "Dec 7th 18", issues: 0}
  ▶ 2: {day: "Dec 8th 18", issues: 2}
  ▶ 3: {day: "Dec 9th 18", issues: 1}
  ▶ 4: {day: "Dec 10th 18", issues: 1}
  ▶ 5: {day: "Dec 11th 18", issues: 1}
  ▶ 6: {day: "Dec 12th 18", issues: 1}
    length: 7
  ▶ __proto__: Array(0)

```

If you start up the Vue dev server using `npm run serve` and enter some different repos, you should see something similar. If you're stuck for inspiration, check out GitHub's Trending page.

Next comes the fun bit—visualizing this data.

Drawing a Bar Chart Using D3

Earlier on, we mentioned that all the drawing will be handled inside a `Chart` component. Let's create the component:

```
touch src/components/Chart.vue
```

D3 works on SVG elements, and for us to draw anything with D3, we need to have an SVG element on the page. In our newly created component (`src/components/Chart.vue`), let's create an SVG tag:

```
<template>
  <div>
    <svg></svg>
  </div>
</template>
```

For this particular tutorial, we'll visualize our data using a bar chart. I picked a bar chart because it represents a low complexity visual element while it teaches the basic application of D3.js itself. The bar chart is also a good intro to the most important D3 concepts, while still having fun!

Before proceeding, let's update our `App` component to include the newly created `Chart` component below the form:

```
<template>
  <div id="app">
    <form action="#" @submit.prevent="getIssues">
      ...
    </form>

    <chart :issues="issues"></chart>
  </div>
</template>
```

Let's also register it as a component:

```
import Chart from './components/Chart.vue';

export default {
  name: "app",
  components: {
    Chart
  },
  ...
}
```

Notice how we're passing the value of the `issues` data property to the `Chart` component as a prop:

```
<chart :issues="issues"></chart>
```

Let's now update our `Chart` component to make use of that data:

```
<script>
import * as d3 from "d3";
import _ from "lodash";

export default {
  props: ["issues"],
  data() {
    return {
      chart: null
    };
  },
  watch: {
    issues(val) {
      if (this.chart !== null) this.chart.remove();
    }
  }
}
```

```

        this.renderChart(val);
    }
},
methods: {
    renderChart(issues_val) {
        // Chart will be drawn here
    }
}
};
</script>

```

In the above code block, we're importing D3 and Lodash. We then instantiate a `chart` data property as `null`. We'll assign a value to this when we start drawing later on.

Since we want to draw the chart every time the value of `issues` changes, we've created a watcher for `issues`. Each time this value changes, we'll destroy the old chart and then draw a new chart.

Drawing will happen inside the `renderChart` method. Let's start fleshing that out:

```

renderChart(issues_val) {
    const margin = 60;
    const svg_width = 1000;
    const svg_height = 600;
    const chart_width = 1000 - 2 * margin;
    const chart_height = 600 - 2 * margin;

    const svg = d3
        .select("svg")
        .attr("width", svg_width)
        .attr("height", svg_height);
}

```

Here, we set the height and width of the SVG element we just created. The `margin` attribute is what we'll use to give our chart some padding.

D3 comes with DOM selection and manipulation capabilities. Throughout the tutorial, you'll see lots of `d3.select` and `d3.selectAll` statements. The difference is that `select` will return the first matching element while `selectAll` returns all matching elements.

The Axes

For bar charts, data can either be represented in a vertical or horizontal format. D3 comes with axis methods, which let us define our axes the way we want:

- `axisLeft`
- `axisTop`
- `axisBottom`
- `axisRight`

Today, we'll be creating a vertical bar chart. For vertical bar charts, we'll only need the `axisLeft` and `axisBottom` methods:

```

renderChart(issues_val) {
    ...

    this.chart = svg
        .append("g")
        .attr("transform", `translate(${margin}, ${margin})`);

    const yScale = d3
        .scaleLinear()
        .range([chart_height, 0])
        .domain([0, _.maxBy(issues_val, "issues").issues]);

    this.chart
        .append("g")
        .call(d3.axisLeft(yScale).ticks(_.maxBy(issues_val, "issues").issues));
}

```

```

const xScale = d3
  .scaleBand()
  .range([0, chart_width])
  .domain(issues_val.map(s => s.day))
  .padding(0.2);

this.chart
  .append("g")
  .attr("transform", `translate(0, ${chart_height})`)
  .call(d3.axisBottom(xScale));
}

```

The above block of code draws axes on the SVG element. Let's go through it step by step:

```

this.chart = svg.append('g')
  .attr('transform', `translate(${margin}, ${margin})`);

```

We first specify where we want our chart to start within the SVG element. When working with D3, for any element we want to be added to the SVG, we usually call the `append` method, then define attributes for this new element.

To add attributes to an element, we usually call the `attr` method, which takes in two parameters. The first parameter is the attribute we want to apply to the selected DOM element, and the second parameter is the value we want, or a callback function that returns the desired value. Here we're moving the start of the chart to the 60, 60 position of the SVG:

```

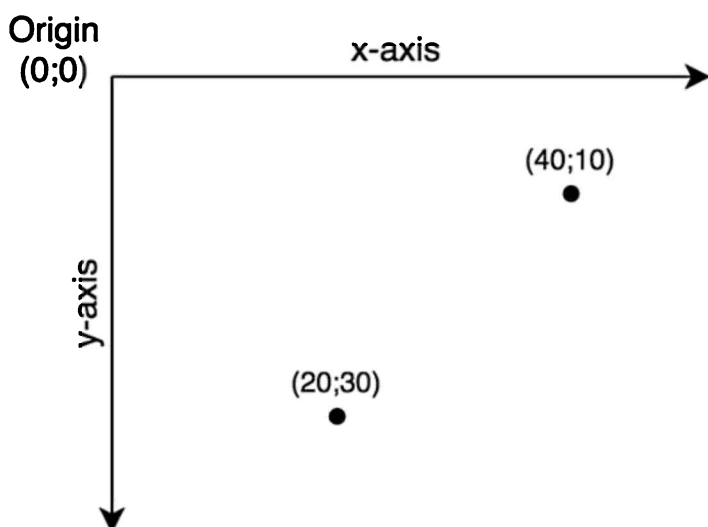
const yScale = d3.scaleLinear()
  .range([chart_height, 0])
  .domain([0, _.maxBy(issues_val, 'issues').issues]);

this.chart.append('g')
  .call(d3.axisLeft(yScale)
  .ticks(_.maxBy(issues_val, 'issues').issues));

```

This codeblock draws the y-axis while making use of D3 scales to come up with a y-scale. Scales are functions that will transform our data by either increasing or decreasing its value for better visualization.

The `range` function specifies the length that should be divided between the limits of the input data. You may have noticed I used `height` as the first parameter and not zero when calling `range`. This is because the SVG coordinate system starts from the top left corner. You'll get to understand this better when we get to draw the bar heights.



On the other hand, the `domain` function denotes minimum and maximum values of the input data. For this particular data set, we want to start from zero to the highest value in our data set. Think of the domain as the input and range as the output.

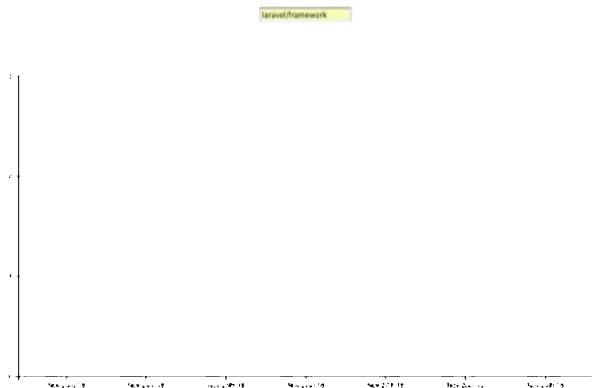
After defining a y-scale, we make use of this scale to draw the axis by calling the `axisLeft` method, which takes in the y-scale as the argument.

The snippet below draws the x-axis:

```
const xScale = d3.scaleBand()
  .range([0, chart_width])
  .domain(issues_val.map((s) => s.day))
  .padding(0.2)

this.chart.append('g')
  .attr('transform', `translate(0, ${chart_height})`)
  .call(d3.axisBottom(xScale));
```

For the `xScale` we use the `scaleBand` function, which helps to split the range into bands and compute the coordinates and widths of the bars with additional padding. Assuming 3 is the maximum number of issues raised across all dates, the chart output should look like this:



If you test this in your browser with the code we've covered so far, you should see something similar.

Now enough with the axes. Let's spit some bars!

Drawing Bars

For our bar chart, the bar width will be fixed and the bar height will vary depending on the dataset size:

```
renderChart(issues_val) {
  ...

  const barGroups = this.chart
    .selectAll("rect")
    .data(issues_val)
    .enter();

  barGroups
    .append("rect")
    .attr("class", "bar")
    .attr("x", g => xScale(g.day))
    .attr("y", g => yScale(g.issues))
    .attr("height", g => chart_height - yScale(g.issues))
    .attr("width", xScale.bandwidth());
}
```

Let's address how we added the bars. First, we created a `barGroups` element:

```
const barGroups = this.chart
  .selectAll('rect')
```

```
.data(issues_val)
.enter()
```

Calling the `selectAll` method on our chart returns an empty selection/array, since we don't have any rectangles in our chart so far. We then chain the `data` method, passing in the dataset we want visualized. This puts the data in a waiting state for further processing.

The next step is to chain the `enter` method. The `enter` method looks both at the data set we passed into `data()` and at the selection we get after calling `selectAll()`, and then tries to look for “matches”—more like mapping between our sample data and elements already present in the DOM. In this particular case, no matches were found.

Select, Enter and Append

This article is an excellent guide for understanding the `select`, `enter` and `append` sequence when working with D3 to manipulate the DOM.

Since `selectAll('rect')` returned an empty array, the `enter` method will return a new selection representing the elements in our data set.

Note that, after chaining `enter()`, every item in the returned array is acted upon individually. This means that any method chained onto `barGroups` will define the behavior of individual items.

```
barGroups
.append('rect')
.attr('class', 'bar')
.attr('x', (g) => xScale(g.day))
.attr('y', (g) => yScale(g.issues))
.attr('height', (g) => chart_height - yScale(g.issues))
.attr('width', xScale.bandwidth());
```

The code block above creates a rectangle for each item in our in our data set. We give each of these rectangles a class of `bar`.

To set the `x` and `y` coordinates for the rectangles, we use the scaling functions we defined earlier. So far, these rectangles are sitting on top of each other, and we need to give our rectangles some height and width.

The width of our bars would be determined by the `scaleBand` function. Chaining the `bandwidth` function to the `xScale` returns a calculated bandwidth from the range and padding provided to the `x-scale`.

To set the bar height, we subtract the computed `y`-coordinate of the bar from the height of the `SVG` to get the correct representation of the value as a column. Remember that, when working with `SVGs`, `x` and `y` coordinates are always calculated starting from the top-left corner.

Adding Labels

So far we have a bar chart. But this chart is not really helpful, as it doesn't tell the user what each axis represents. To give our chart more context, we'll have to add labels for the axes as well as a chart title.

To add labels, we append text elements to our `SVG`:

```
svg
.append('text')
.attr('class', 'label')
.attr('x', -(chart_height / 2) - margin)
.attr('y', margin / 2.4)
.attr('transform', 'rotate(-90)')
.attr('text-anchor', 'middle')
.text('Issues opened')
```

```
svg
.append('text')
.attr('class', 'label')
.attr('x', chart_width / 2 + margin)
.attr('y', chart_height + margin * 1.7)
.attr('text-anchor', 'middle')
.text('Days')
```

```
svg
```

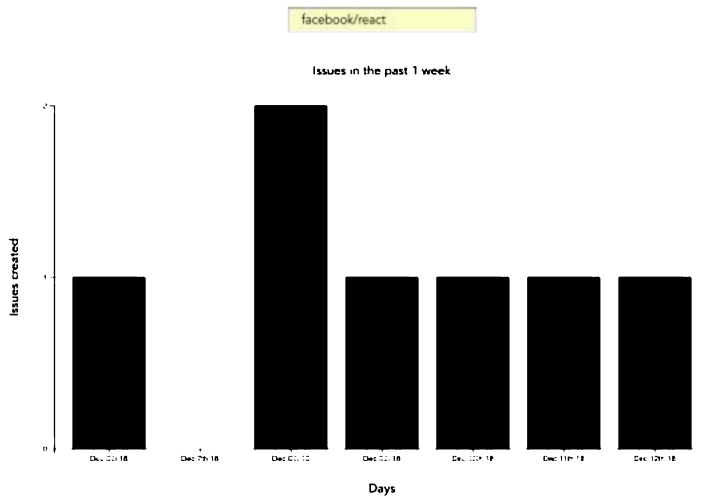
```

.append('text')
.attr('class', 'title')
.attr('x', chart_width / 2 + margin)
.attr('y', 40)
.attr('text-anchor', 'middle')
.text('Issues in the past 1 week')

```

The text elements can be positioned with x and y coordinates while text alignment is done with the `text-anchor` attribute. To add the text itself, we call the `text` method on the text element.

We can test things out by serving our app then searching for a repo. Search for any popular repo—such as `facebook/react`:



Back in our `App` component, we can now get rid of the console statement inside the `getIssues` method:

```
console.log(this.issues)
```

Even though our chart does an excellent job of visualizing our data, there's still much to be done in terms of the user experience. In the next section, we'll look at how to add transitions to D3 charts.

Adding Transitions

For this particular chart, we want to make it such that, when one hovers over a bar element, its shade changes and the number of issues represented by the bar shows up at the top of the bar.

For this to happen, we have to do some event handling on `mouseenter` and `mouseleave` for the `barGroups`.

Edit the `barGroups` code block above the three `svg` blocks:

```

barGroups
  ...
  .attr("width", xScale.bandwidth())
  .on("mouseenter", function(actual, i) {
    d3.select(this)
      .transition()
      .duration(300)
      .attr("opacity", 0.6)
      .attr("x", a => xScale(a.day) - 5)
      .attr("width", xScale.bandwidth() + 10);
  })
  barGroups
    .append("text")
    .attr("class", "value")
    .attr("x", a => xScale(a.day) + xScale.bandwidth() / 2)
    .attr("y", a => yScale(a.issues) - 20)
    .attr("text-anchor", "middle")
    .text((a, idx) => {

```

```

    return idx !== i ? "" : `${a.issues} issues`;
  });
});

```

We call the transition method to add animations to an element when working with D3.

Each time the cursor hovers over a bar, we reduce the opacity of the bar and increase the width by 10px. We also add text on top of the bar, stating the number of issues the bar represents. This transition takes a duration of 300 milliseconds.

Since we don't want to leave this bar in the new state when the mouse leaves, let's define a mouseleave event, which removes the selection features we had applied in the mouseenter event:

```

barGroups
...
.attr("width", xScale.bandwidth())
.on("mouseenter", function(actual, i) { ... })
.on("mouseleave", function() {
  d3.selectAll(".issues").attr("opacity", 1);

  d3.select(this)
    .transition()
    .duration(300)
    .attr("opacity", 1)
    .attr("x", a => xScale(a.day))
    .attr("width", xScale.bandwidth());

  svg.selectAll(".value").remove();
});

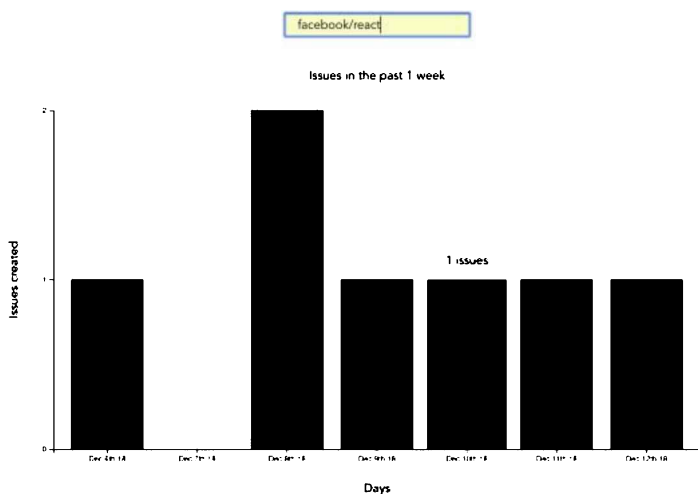
```

What we are doing above is setting the opacity of the bar to the original value and also removing the text from the bar.

```
svg.selectAll('.value').remove();
```

The code above removes any text on the DOM that has a class of value.

Here's the final result:



If you try things out in your browser now, you should see something like the above.

Some Final UI/UX Touches

When making the request to GitHub, we anticipate some loading time before getting a response back from GitHub. For UX purposes, we'll make the user aware we're still retrieving results from GitHub by flashing a loading alert on the page. In the App component, add this snippet to the HTML section of your code—right above the Chart component:

```
<div class="alert alert-info" v-show="loading">Loading...</div>
```

```
<chart :issues="issues"></chart>
```

For prototyping purposes, we'll leverage Bootstrap. Vue has an official Bootstrap package that we're going to install:

```
npm install bootstrap-vue
```

Once the package has been installed, we'll have to import Bootstrap's CSS classes to our `main.js` file to make the alert show up:

```
[...]
import "bootstrap/dist/css/bootstrap.css";
import "bootstrap-vue/dist/bootstrap-vue.css";
```

We're almost done, but we need to add a `loading` attribute to our data model—whose value will determine whether or not to show the loading alert:

```
data() {
  return {
    loading: false,
    ...
  };
},
```

Every time we make a new request, we'll set the value of `loading` to `true`, so the alert shows up, and then we'll finally set this value to `false` for successful requests or in the case the app errored:

```
getIssues() {
  this.loading = true;
  ...
  axios.get( ... )
  .then( ... )
  .catch( ... )
  .finally(() => (this.loading = false));
}
```

Error Handling

As it stands, we're just logging any errors we might run into on the console—something the user isn't aware of. To make the user aware if something goes wrong when making the request, we'll flash an error alert on the page. Add this snippet below the loading alert we just created:

```
<div class="alert alert-danger" v-show="errored">An error occurred</div>
<chart :issues="issues"></chart>
```

We also need to update our data model with the `errored` attribute, which we initialize with a `false` value:

```
data() {
  return {
    loading: false,
    errored: false,
    ...
  };
},
```

We'll set the `errored` value to `true` every time we run into an error. This should happen inside the catch block:

```
getIssues() {
  this.loading = true;
  this.errored = false;
  ...
  axios.get( ... )
  .then( ... )
  .catch(error => {
    console.error(error);
    this.errored = true;
  })
}
```



```
    .finally( ... );  
  }
```

Some Basic Styling

Eagle-eyed readers will have noticed that in the screen shot at the top of this guide, the color of the bars was blue. Let's add that as a final touch to our `App` component:

```
<style>  
.bar {  
  fill: #319bbe;  
}  
</style>
```

Conclusion

Congratulations for making it to this point. Even though we have a working bar chart, there's still more we can do to make the chart more visually appealing. We could, for example, add grid lines, or alternatively visualize this data using a line chart.

What we drew was a simple bar chart, but `D3.js` has amazing capabilities in terms of what you can do. Any data visualization that you've seen anywhere—including powerpoint presentations—can probably be handled by `D3`. This article is a good read for some of the things `D3` is capable of doing. You can also check out `D3` on [GitHub](#) to see other use cases.

Chapter 8: How to Build a Reusable Component with Vue

by Deji Atoyebi

Vue is a progressive framework that allows you to build your applications as collections of self-contained, reusable components.

Components are used to create what can be called custom HTML elements. Intrinsically, each element bears similar sets of data and functionalities. For example, say you're building a Twitter-like social network. For the newsfeed, you'll require a way to present a list of the latest activities to users. Each activity will have a similar set of data: the name of the creator of the activity, the date the activity was published, the actual content, and so on. Also, you'd want users to carry out actions such as liking, unliking and sharing these activities.

In this scenario, simply creating a card component would cut it. And although the card component will populate different activities, the underlying structure of the data as well as functionalities will remain the same. No need to repeat and modify code. After creating the card component, all we'd need to do is to insert it into the body of our HTML document, as we do every other HTML element:

```
<feed-card v-for="activity for activities"></feed-card>
```

While building a card component for a news feed would be quite an interesting topic, in this tutorial we'd be touching on something different: how to build a re-usable modal component.

Live Code

For the impatient among you, here's a demo of the code running on CodePen.

But Why a Modal?

Modals are a very important, multi-purpose feature of many websites. Their main function is to bring information to the immediate attention of users. They are particularly useful when crucial information is to be passed or an action must be carried out before users gain access to, say, a resource. A common use case of modals is to present an authentication (login or sign-up) form before a user can carry out a particular action. I assume that you'll be required to create a modal component at least once during your journey as a web developer—hence the importance of this tutorial.

What To Consider Before Building Our Component

Before we get our hands dirty with actually building a modal component, it's important to consider a couple of points.

Do We Use a Single-file Component or Not?

Don't be deceived by the name: single-file components (SFCs) are just regular components. The "single-file" part is only meant to denote that, rather than your component being defined in the HTML file in which it's used, alongside the root Vue instance, it would be defined in a file of its own. This comes with its own advantages, part of which is reusability (if you're building a non-trivial web app that uses your component in different HTML pages).

In this tutorial, we'll leverage Vue's incremental adoptability. We'll start by building everything in one file and including Vue from a CDN, then in a second step, we'll look at how to refactor things into an SFC.

Do We Use ES6 or Not?

If you're going to be using Vue at all, you need to decide whether or not your code will make use of ES6 features. Using ES6 features comes with cross-browser compatibility issues, often meaning you'll need to use Babel to transpile it to something older browsers can understand.

For the code in this guide, we'll use ES6 throughout. It's a ratified standard, after all, and we're in the favorable position of not having to support ancient browsers. Note that when we convert the modal to SFCs, the JavaScript will be bundled and transpiled via webpack.

Important Components and Behavior

What are the important components of a modal and how do we expect our modal to behave?

Every modal has two common components to it: the mask and the container (let's call them that). The mask is the transparent dark (or white) background that appears with most modals. The modal overlies the rest of the content in the page. Once you click on a mask, the modal exits. The container, on the other hand, is the part of the modal that "contains" the important stuff.

With this knowledge, creating a reusable modal component becomes super easy, as we've been able to whittle down its design into its two

major parts.

We'll use Materialize for our design, as we don't want to get too stuck with writing too much code in CSS.

Building the Modal Component

Now it's time for the fun stuff. First, create a file named `modal.html` and add the following code. This pulls in our dependencies from a CDN and defines the HTML structure for our mask and container:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>Page Title</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/css/materialize.min.css">
    <style></style>
  </head>
  <body>
    <div class="container">
      <div class="modal-mask">
        <div class="modal-container"></div>
      </div>
    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2.5.21/dist/vue.js"></script>
    <script></script>
  </body>
</html>
```

Next, let's add in some basic styles to the `<style>` block in the `<head>` section:

```
.modal-mask{
  position: fixed;
  z-index: 100;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  background-color: rgba(0, 0, 0, .3);
  overflow-y:auto;
}
.modal-container {
  width: 80%;
  height: auto;
  margin: 50px auto 0;
  position: relative;
  padding: 16px;
  background-color: #fff;
  border-radius: 2px;
  box-shadow: 0 2px 8px rgba(0, 0, 0, .33);
  box-sizing: border-box;
}
```

Open the file in your browser. You'll see that the modal mask is a transparent dark background. The height and width are set to 100%, as we want it to occupy the whole page while the modal is in focus. The modal's container is a `<div>` element that occupies 80% of the width of the mask. Feel free to customize yours however you wish.

Creating a Vue Component

The next step is to make a Vue component out of our modal. Add the following code to the final `<script>` block:

```
new Vue({
```

```
    el: ".container",
    data() {
      return {};
    }
  });
```

The code for the modal now needs to be wrapped in a `<template>` block so that it can be referenced in the script defining the modal as a component:

```
<div class="container"></div>

<template id="modal-template">
  <div class="modal-mask">
    <div class="modal-container"></div>
  </div>
</template>
```

Now, we can define our modal as a Vue component:

```
Vue.component("modal", {
  template: "#modal-template"
});

new Vue({ ... });
```

Now, when you open up your `modal.html`, you'd find that there's absolutely nothing shown in the page. This is because the `<template>` block wasn't rendered when the page was being created.

Showing and Hiding the Modal

At this point, we should touch on how to call our modal. Let's model it after a real-world situation: the modal will display when a button is clicked. Thus, insert a button to the container div, along with our modal:

```
<div class="container">
  <button> Show Modal </button>
  <modal></modal>
</div>
```

Since the button does absolutely nothing for now, we need to work on both the root Vue instance and the modal component so that the modal appears when the button is clicked. First, change the code in your root instance:

```
new Vue({
  el: ".container",
  data() {
    return {
      showModal: false
    };
  },
  methods: {
    actShowModal() {
      this.showModal = true;
    }
  }
});
```

In the above, we initialized `showModal` in our root instance to indicate when our modal is being shown to a user and when it's not. We also added a method to manipulate this data property. As you'll soon see, calling `actShowModal` will make our modal pop up.

The next task is to enable our `<button>` to call `actShowModal` when it's clicked:

```
<button v-on:click="actShowModal">Show Modal</button>
```

Unfortunately, clicking the button still makes no difference. This is because we haven't hooked up our modal component to the data in our root instance. What we want is for our modal to display when `showModal` is `true` and disappear when `showModal` is `false`. The way to achieve this is by using props. Our code should now be:

```
<modal :show_modal="showModal"></modal>
```

And:

```
Vue.component("modal", {
  template: "modal-template",
  props: ["show_modal"]
});
```

Now that we've added the `show_modal` as a prop, we need our modal to appear when `show_modal` is true. This requires a simple change in the template of our modal component:

```
<template id="modal-template">
  <div class="modal-mask" v-if='show_modal'>
    <div class="modal-container"></div>
  </div>
</template>
```

Now, open up `modal.html`. Notice that, when you click the button, the modal pops up. Great! But there's still a problem: there's no way to close the modal ... yet.

We need the modal to disappear when we click on the mask. To achieve this, `showModal` would have to be `false`, and we need to make this happen from the modal component since it houses the mask.

First, create a method called `closeModal` in the component that emits the change:

```
Vue.component("modal", {
  ...
  methods: {
    closeModal() {
      this.$emit("close_modal");
    }
  }
});
```

And then change the code of your template so it looks like this:

```
<template id="modal-template">
  <div class="modal-mask" v-if="show_modal" @click="closeModal">
    <div class="modal-container"></div>
  </div>
</template>
```

Now when the mask is clicked, the modal component emits a `close_modal` event. Let's catch that event in our root Vue instance and toggle the modal's display state to `false`:

```
<modal :show_modal="showModal" @close_modal="actCloseModal"></modal>
```

Also this:

```
methods: {
  actShowModal(){ ... },
  actCloseModal(){
    this.showModal = false;
  }
},
```

Now, our modal should close when the mask is clicked—which is super awesome. But there's yet another problem: clicking the modal's container also closes the modal.

A simple explanation for this is that the click event propagates (bubbles up). To stop that, let's use Vue's way of stopping such propagation, the `@click.stop` event modifier:

```
<template id="modal-template">
  <div class="modal-mask" v-if="show_modal" @click="closeModal">
    <div class="modal-container" @click.stop></div>
  </div>
```

```
</template>
```

There, solved! Our modal works the way it should now. Take some time to run the code we have so far and make sure it works correctly.

Creating an Authentication Modal

Currently, our modal container is empty. Let's change that and write some code to build a login form to be embedded in this container.

Let's start off by adding our login form into the mix. Remember, the Materialize framework is to be used, so don't be too bothered about the HTML class definitions:

```
<template id="modal-template">
  <div class="modal-mask" v-if="show_modal" @click="closeModal">
    <div class="modal-container clearfix" @click.stop>
      <h4 class="blue-text">Login</h4>
      <form class="col s12">
        <div class="input-field col s12">
          <input v-model="email" id="email_address" type="text">
          <label for="email_address">Email Address</label>
        </div>
        <div class="input-field col s12 ">
          <input v-model="password" id="password" type="text" >
          <label for="password">Password</label>
        </div>
        <a class="waves-effect waves-light btn form-btn teal">Sign in</a>
      </form>
    </div>
  </div>
</template>
```

And add the following to style the form's button:

```
.form-btn {
  float: right;
}
.clearfix::after {
  content: "";
  clear: both;
  display: table;
}
```

We aren't going to implement a back end for the authentication component, so it's reasonable that our button does nothing. Nonetheless, let's define the email and password data properties on our component, as we already have our inputs in the login form pointing to them through the v-model directive. This would be useful for implementing form validation. We won't do that in this tutorial, but you can read more about this in our tutorial *A Beginner's Guide to Working with Forms in Vue* in this Vue series:

```
Vue.component("modal", {
  template: "#modal-template",
  props: ["show_modal"],
  data() {
    return {
      email: "",
      password: ""
    };
  },
  methods: { ... }
});
```

Open up your modal.html, and ... boom! We've just created a login modal all by ourselves.

Since everything works nice and smoothly, we're afforded the opportunity to think of sleeker ways of doing things.

Making Our Modal Reusable with Slots

Although our modal is now great and usable, there's still a little problem: the login form is tightly knitted to the modal component. So, at

the moment, if we wanted to have different modals in our application—perhaps one for user registration and another for something else—we'd have to repeat our code by creating yet another modal component from scratch.

Is there a way to keep the skeleton of the modal component separate while we plug in whatever content we wish? You bet there is.

With what we have already, it's easy to separate our login form from the modal component so that we can use the latter for other content when we need to. It all starts with creating a separate login component that would have the data and methods that are required for a login form to operate:

```
Vue.component("login", {
  template: "#login-template",
  data() {
    return {
      email: "",
      password: ""
    };
  }
});
```

Here's the code defining login template:

```
<template id="login-template">
  <div class="clearfix">
    <h4 class="blue-text">Login</h4>
    <form class="col s12">
      <div class="input-field col s12">
        <input v-model="email" id="email_address" type="text">
        <label for="email_address">Email Address</label>
      </div>
      <div class="input-field col s12 ">
        <input v-model="password" id="password" type="text" >
        <label for="password">Password</label>
      </div>
      <a class="waves-effect waves-light btn form-btn teal">Sign in</a>
    </form>
  </div>
</template>
```

Let's not forget to rid the modal container of the login form, as we no longer need it. Right now, what we need is to define a slot in the modal container, since that's where we want any and every content in the modal to live:

```
<template id="modal-template">
  <div class="modal-mask" v-if="show_modal" @click="closeModal">
    <div class="modal-container" @click.stop>
      <slot></slot>
    </div>
  </div>
</template>
```

Also, delete the email and password keys in the modal component; those are now properties of our login component, just as they ought to be. The modal component's definition should now look something like this:

```
Vue.component("modal", {
  template: "#modal-template",
  props: ["show_modal"],
  methods: {
    closeModal() {
      this.$emit("close_modal", false);
    }
  }
});
```

Inserting our login component into the modal component as provided by our slot takes this simple step:

```
<modal :show_modal="showModal" @close_modal="actCloseModal">
```

```
</login></login>
</modal>
```

Preview your modal and you'll find that everything works fine: the login component is rendered into the modal component.

Adding a Registration Component

To go ahead to show how reusable this modal component is now, let's create a registration component that will also be rendered in the modal component. This will be similar to creating our login component: first, write code for the template and then the component definition.

```
<template id="registration-template">
  <div class="clearfix">
    <h4 class="blue-text">Register</h4>
    <form class="col s12">
      <div class="input-field col s6">
        <input v-model="firstName" placeholder="" id="first_name" type="text">
        <label for="first_name">First Name</label>
      </div>

      <div class="input-field col s6">
        <input v-model="lastName" id="last_name" type="text">
        <label for="last_name">Last Name</label>
      </div>

      <div class="input-field col s12">
        <input v-model="username" id="username" type="text" class="">
        <label for="username">Username</label>
      </div>

      <div class="input-field col s12">
        <input v-model="email" id="email_address" type="text">
        <label for="email_address">Email address</label>
      </div>

      <div class="input-field col s12">
        <input v-model="password" id="password" type="password">
        <label for="password">Password</label>
      </div>

      <a class="waves-effect waves-light btn teal form-btn">Proceed</a>
    </form>
  </div>
</template>
```

The component definition:

```
Vue.component("register", {
  template: "#registration-template",
  data() {
    return {
      firstName: "",
      lastName: "",
      username: "",
      email: "",
      password: ""
    };
  }
});
```

What we're hoping to do is to render the login or register component into the modal component separately. Say there are two buttons in a web page and while one shows the login modal when clicked, the other shows the registration modal. We should have this as our code:

```
<modal :show_modal="showModal" @close_modal="actCloseModal" choice="choice">
```



```

<login v-if="choice === 'login'"></login>
<register v-else-if="choice === 'register'"></register>
</modal>

```

As the code suggests, there's a data property on our root instance named `choice` that could either be "login" or "register". This is then used in our modal component as a prop, as can be seen in the following code:

```

Vue.component("modal", {
  template: "#modal-template",
  props: ["show_modal", "choice"],
  methods: { ... }
});

Vue.component("login", { ... });
Vue.component("register", { ... });

```

```

new Vue({
  el: ".container",
  data() {
    return {
      showModal: false,
      choice: ""
    };
  },
  methods: { ... }
});

```

Next, let's create the aforementioned two buttons—one to show the login modal, and the other to show the registration modal. So now we have these:

```

<div class="container">
  <button v-on:click="actShowModal('login')"> Login </button>
  <button v-on:click="actShowModal('register')"> Register </button>

  <modal :show_modal="showModal" @close_modal="actCloseModal" choice="choice">
    <login v-if="choice === 'login'"> </login>
    <register v-else-if="choice === 'register'"></register>
  </modal>
</div>

```

Our `actShowModal` method is defined on the root instance will then change to accept a choice as argument:

```

new Vue({
  el: ".container",
  data() { ... },
  methods: {
    actShowModal(choice) {
      this.choice = choice;
      this.showModal = true;
    },
    actCloseModal() {
      this.showModal = false;
    }
  }
});

```

Some Accessibility Improvements

One thing that we shouldn't overlook is that people may be using a keyboard to navigate our modal. Let's make life better for them:

```

<div class="container" @keyup.escape.stop="actCloseModal()">
  ...
</div>

```

This will let the modal be dismissed by pressing the `Esc` key.

When the modal opens, you might also want to focus the first element in the form. This would be done by creating a custom directive like so:

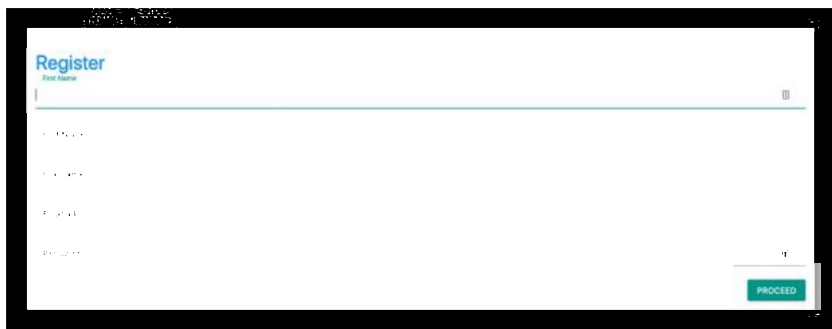
```
Vue.directive("focus", {
  inserted: el => {
    el.focus();
  }
});
```

You can apply this to any element that should receive focus. In our case, this will be the first input element in either form:

```
<input v-model="email" id="email_address" type="text" v-focus>
<input v-model="firstName" placeholder="" id="first_name" type="text" v-focus>
```

And now, we're all done! Our modal component nicely renders both the register and login components based on the button we click. Open your `modal.html` to see the outcome.

Live Code



Here's a demo of the code running on CodePen.

What we've done is quite valuable, in that it prevents unnecessary code duplication. We've ended up creating a somewhat one-size-fits-all modal component. However, to make it perfectly reusable, we need to turn it and other components to SFCs and use webpack to bundle our code as well as transpile it so that it works on all major browsers. We'll do this in the next part of this tutorial.

Single-file Components

Single-file components are a clean way to achieve modularity while working with Vue.js. They allow us to create `.vue` files which house our components, and they also come in handy when we're working on a complex application and would like to avoid the clutter that writing Vue code in an HTML file presents. In our case, turning our `Modal`, `Registration` and `Login` components into SFCs helps prevent the extremely clunky act of copying and pasting code for reuse.

Since your browser cannot read `.vue` files, we'll be using Vue CLI to scaffold a project. Under the hood, this uses webpack as well as a couple of other dependencies to bundle our code.

Let's get started.

Basic Setup

To use the CLI, you'll need a recent version of Node.js installed on your system. You can do this by downloading the binaries from the official website, or by using a version manager. This is probably the easiest way, as it allows you to manage multiple versions of Node on the same machine.

Next, proceed to installing Vue CLI globally on your machine with the following command:

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

Create a scaffold for our project with the following command:

```
vue create my-modal
```

Select the default preset, then when the CLI has finished installing all the dependencies, change into this directory:

```
cd my-modal
```

Next, let's install the Materialize dependency:

```
npm i materialize-css
```

Open up `src/main.js` so that we can include it and make it global to our application:

```
import Vue from "vue";
import App from "./App.vue";
import "materialize-css/dist/css/materialize.min.css";
import "materialize-css/dist/js/materialize.min.js";
```

```
Vue.config.productionTip = false;
Vue.directive("focus", {
  inserted: el => {
    el.focus();
  }
});
```

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

While we're at it, we can add our custom directive, too.

Creating Our Single-file Components

Open up `src/App.vue` and replace the code with the following:

```
<template>
  <div class="container" @keyup.escape.stop="actCloseModal()">
    <button v-on:click="actShowModal('login')"> Login </button>
    <button v-on:click="actShowModal('register')"> Register </button>

    <modal :show_modal="showModal" @close_modal="actCloseModal" choice="choice">
      <login v-if="choice === 'login'"></login>
      <register v-else-if="choice === 'register'"></register>
    </modal>
  </div>
</template>
```

```
<script>
import modal from "./components/modal.vue";
import login from "./components/login.vue";
import register from "./components/register.vue";
```

```
export default {
  name: "app",
  components: {
    modal,
    login,
    register
  },
  data() {
    return {
      showModal: false,
      choice: ""
    };
  },
  methods: {
    actShowModal(choice) {
      this.choice = choice;
      this.showModal = true;
    },
    actCloseModal() {
```

```

        this.showModal = false;
    }
}
};
</script>

<style>
body {
  padding: 15px;
}
</style>

```

There's nothing new going on here in terms of functionality, but notice how much more organized the code is.

Next, create the components we're going to need in the `src/components` folder:

```
touch src/components/{modal.vue,register.vue,login.vue}
```

When you're finished, the `src` folder should look like this:

```

.
├── App.vue
├── assets
│   └── logo.png
├── components
│   ├── login.vue
│   ├── modal.vue
│   └── register.vue
└── main.js

```

The final thing that remains to do is to add the code to these components.

In `modal.vue`:

```

<template id="modal-template">
  <div class="modal-mask" v-if="show_modal" @click="closeModal">
    <div class="modal-container" @click.stop>
      <slot></slot>
    </div>
  </div>
</template>

<script>
export default {
  name: "modal",
  props: ["show_modal", "choice"],
  methods: {
    closeModal() {
      this.$emit("close_modal", false);
    }
  }
};
</script>

<style>
.modal-mask{
  position: fixed;
  z-index: 100;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  background-color: rgba(0, 0, 0, .3);
  overflow-y:auto;

```

```

}
.modal-container {
  width: 80%;
  height: auto;
  margin: 50px auto 0;
  position: relative;
  padding: 16px;
  background-color: #fff;
  border-radius: 2px;
  box-shadow: 0 2px 8px rgba(0, 0, 0, .33);
  box-sizing: border-box;
}
.form-btn {
  float: right;
}
.clearfix::after {
  content: "";
  clear: both;
  display: table;
}
</style>

```

Here we're declaring our CSS, as it pertains more to the modal than the other components. Were we to add a `scoped` attribute, the definitions would only be available to the template in this file.

In `login.vue`:

```

<template id="login-template">
  <div class="clearfix">
    <h4 class="blue-text">Login</h4>
    <form class="col s12">
      <div class="input-field col s12">
        <input v-model="email" id="email_address" type="text" v-focus>
        <label for="email_address">Email Address</label>
      </div>
      <div class="input-field col s12">
        <input v-model="password" id="password" type="text">
        <label for="password">Password</label>
      </div>
      <a class="waves-effect waves-light btn form-btn teal">Sign in</a>
    </form>
  </div>
</template>

<script>
export default {
  name: "login",
  data() {
    return {
      email: "",
      password: ""
    };
  }
};
</script>

```

And in `register.vue`:

```

<template id="registration-template">
  <div class="clearfix">
    <h4 class="blue-text">Register</h4>
    <form class="col s12">
      <div class="input-field col s6">
        <input v-model="firstName" placeholder="" id="first_name" type="text" v-focus>

```

```

    <label for="first_name">First Name</label>
  </div>

  <div class="input-field col s6">
    <input v-model="lastName" id="last_name" type="text">
    <label for="last_name">Last Name</label>
  </div>

  <div class="input-field col s12">
    <input v-model="username" id="username" type="text" class="">
    <label for="username">Username</label>
  </div>

  <div class="input-field col s12">
    <input v-model="email" id="email_address" type="text">
    <label for="email_address">Email address</label>
  </div>

  <div class="input-field col s12">
    <input v-model="password" id="password" type="password">
    <label for="password">Password</label>
  </div>

  <a class="waves-effect waves-light btn teal form-btn">Proceed</a>
</form>
</div>
</template>

<script>
export default {
  name: "register",
  data() {
    return {
      firstName: "",
      lastName: "",
      username: "",
      email: "",
      password: ""
    };
  }
};
</script>

```

And that's all there is to it. If you now spin up the dev server using the command `npm run serve`, you'll be able to see your app running at `http://localhost:8080`.

Conclusion

In this tutorial, I've demonstrated how to build a simple modal component. We looked at basic accessibility features, how to design the modal so as to be reusable, and finally how to use Vue CLI to structure our app as single-file components.

I hope I've shown how easy it can be to get started building components of your own, as well as how Vue's component-based approach makes it a joy to build an interface in such a declarative way.

Chapter 9: How to Build a Game with Vue.js

by Ivaylo Gerchev

One of the reasons I like Vue.js so much is that it can be used in a wide variety of scenarios and can be integrated into many different kinds of projects. So to me, Vue is a universal, multipurpose framework. In this aspect, it reminds me of the Python programming language, which is also easy to learn and use, and can be applied in many areas.

In this tutorial, we'll explore one of the less obvious uses for Vue—game development.

This Tutorial's Code Repository

The code for this tutorial can be found on GitHub. You can also view a demo of the finished game here.

Why Vue Instead of a Real Game Development Framework

Obviously, Vue is not tailored to game development. So why we would want to choose it instead of a real game development framework/engine? Well, let's explore two good reasons:

1. If you're familiar with Vue and/or you work with it on a daily basis, you already have the necessary skills and knowledge. To create a game with Vue, you just apply what you already know while shifting your mind to a different context—instead of having to learn a whole new game framework.
2. Vue offers features that make game development easy:
 - Vue's reactivity system allows for building highly interactive interfaces with minimum effort.
 - Transitions and animations are an essential part of most games. Vue has built-in support for them.
 - Vue's ecosystem is rich enough to supply you with whatever feature you need for your game project.

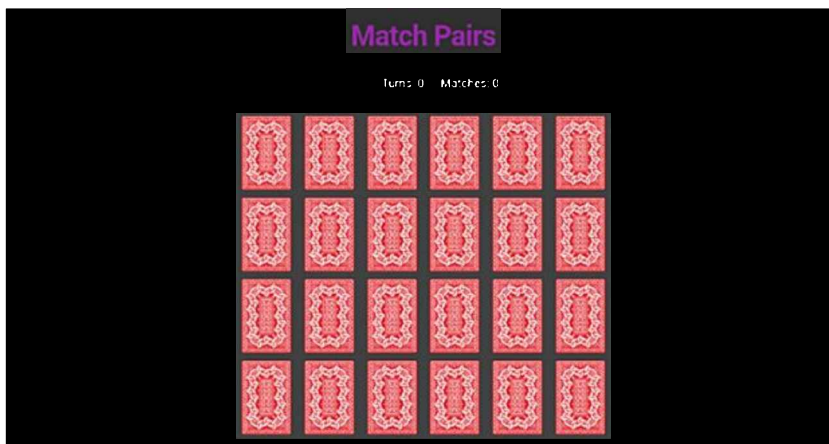
Planning the Game

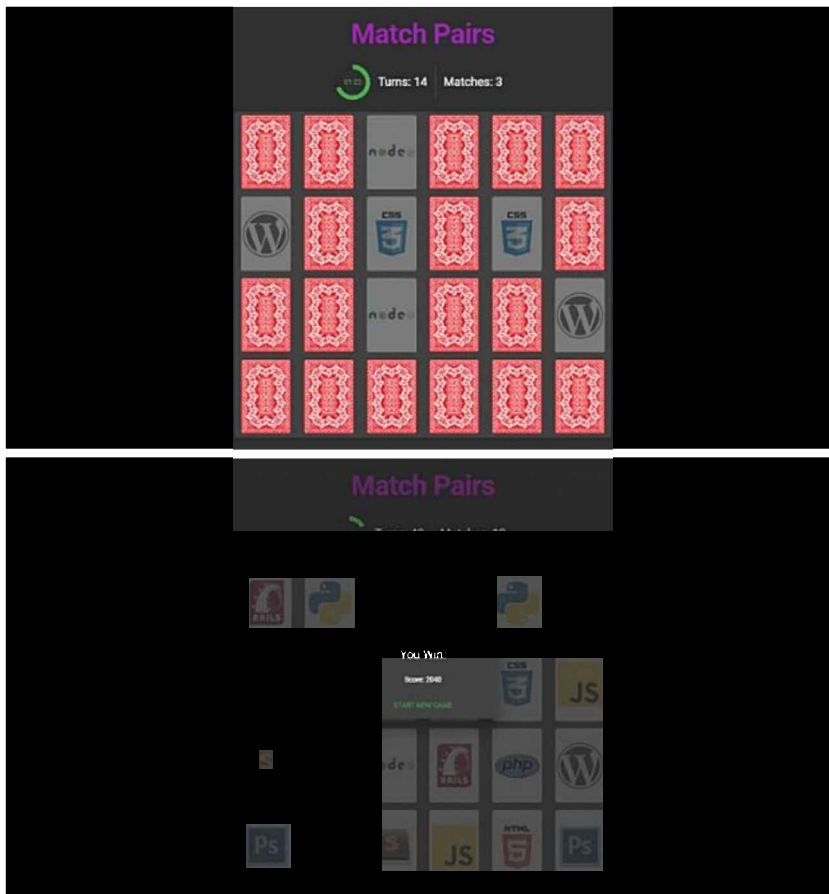
What we're going to build today is an electronic variant of the popular Match Pairs game. Let's now lay out the features we want to implement, along with the components we'll need for that.

The game UI will be composed of four components:

- A cards board, which will hold the cards.
- An animated countdown timer, which will display the time remaining for the player to match all the pairs. When the time remaining is 10 seconds or less, the color of the timer will change to red.
- A status bar, which will display the number of the turns taken and matches found.
- A splash screen, which will show up when the game is finished—either because the time ran out or because all pairs were discovered. It will display a “You Lose!” or “You Win!” message accordingly, as well as the game's score.

Here are three screenshots showing the starting, playing, and finishing phases of the game:





Getting Started

Now it's time for action. To follow along with this, you'll need to have Node, npm and Vue CLI installed on your system.

To install Node (and by default npm), 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.

To install the CLI run:

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

That done, we'll create a new `match-pairs` project, selecting the "default" preset when prompted:

```
vue create match-pairs
cd match-pairs
```

Next, we add the Vuetify plugin:

```
vue add vuetify
```

Again, choose the "default" preset, which has the a la carte system enabled. To activate it, we need to create a `webpack.config.js` file, in the project's root, with the following content:

```
const VuetifyLoaderPlugin = require('vuetify-loader/lib/plugin')
module.exports = {
  plugins: [
    new VuetifyLoaderPlugin()
  ]
}
```

Now let's add two more packages that we'll need—Lodash and MomentJS:

```
npm i lodash moment
```


Finally, we can run the app with this command:

```
npm run serve
```

Preparing the Project Files and Components

Now we need to do some preparation.

First, we need to modify the `src/main.js` file. We swap its content with the following:

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

import _ from 'lodash'

Vue.config.productionTip = false

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

Now, in the `src/components` directory, we delete the `HelloWorld.vue` example file and create the necessary components for the game: `CardsBoard.vue`, `Timer.vue`, `StatusBar.vue`, and `Splash.vue`.

Next, we open `src/App.vue`, delete the content and insert the following instead:

```
<template>
  <v-app dark>
    <v-container>
      <v-layout class="font-weight-bold display-3 purple--text" justify-center mb-4>Match Pairs</v-layout>
      <v-layout mb-4>
        <timer></timer>
        <status-bar></status-bar>
      </v-layout>
      <cards-board></cards-board>
    </v-container>
    <splash></splash>
  </v-app>
</template>
```

Here, we construct the game layout using the components we've just created.

Next, we populate the script with the following:

```
<script>
const cardsSet = [
  {
    name: "php",
    img: "https://s3-us-west-2.amazonaws.com/s.cdpn.io/74196/php-logo_1.png"
  },
  {
    name: "css3",
    img: "https://s3-us-west-2.amazonaws.com/s.cdpn.io/74196/css3-logo.png"
  },
  ...
];

let shuffleCards = () => {
  let cards = [].concat(_.cloneDeep(cardsSet), _.cloneDeep(cardsSet));
  return _.shuffle(cards);
};

import CardsBoard from "./components/CardsBoard";
import StatusBar from "./components/StatusBar";
```

```

import Timer from "../components/Timer";
import Splash from "../components/Splash";
import _ from "lodash";
import moment from "moment";

export default {
  name: "App",
  components: {
    CardsBoard,
    StatusBar,
    Timer,
    Splash
  },
  data() {
    return {
      cards: [],
      started: false,
      duration: 120000,
      progress: 0,
      time: null,
      timer: null,
      flipBackTimer: null,
      turns: 0,
      matches: 0,
      showSplash: false,
      result: "",
      score: 0
    };
  },
  created() {
    this.resetGame();
  },
  methods: {
    resetGame() {
      this.started = false;
      this.progress = 0;
      this.time = null;
      this.turns = 0;
      this.matches = 0;
      this.showSplash = false;
      this.score = 0;

      let cards = shuffleCards();

      _ .each(cards, card => {
        card.flipped = false;
        card.found = false;
      });

      this.cards = cards;
    },
  }
};
</script>

```

First, we add the necessary images for the cards. For the sake of brevity, I've left many of them out. You can get the full list from the [GitHub repo](#).

The `shuffleCards()` function doubles the images to make them pairs and shuffles them all.

Next, we import our components and register them in the `components` object. We also import `Lodash` and `Moment`.

After that, we define all data properties which we'll need for the game. We add the `created()` hook, which will invoke the

`resetGame()` method on the app's creation. The `resetGame()` turns back all data properties to their initial states.

Now we can add some helper methods used to manage flipped and found states of the cards:

```
methods: {
  ...

  flippedCards() {
    return _.filter(this.cards, card => card.flipped);
  },

  sameFlippedCard() {
    let flippedCards = this.flippedCards();
    if (flippedCards.length == 2) {
      if (flippedCards[0].name == flippedCards[1].name) return true;
    }
  },

  setCardFounds() {
    _.each(this.cards, card => {
      if (card.flipped) card.found = true;
    });
  },

  checkAllFound() {
    let foundCards = _.filter(this.cards, card => card.found);
    if (foundCards.length == this.cards.length) return true;
  },

  clearFlips() {
    _.map(this.cards, card => (card.flipped = false));
  },

  clearFlipBackTimer() {
    clearTimeout(this.flipBackTimer);
    this.flipBackTimer = null;
  },
}
```

Here is what the helpers do:

- `flippedCards()` returns all flipped cards—that is, those having `flipped` property set to `true`
- `sameFlippedCard` uses the above helper to check whether two flipped cards are same
- `setCardFounds()` sets the flipped cards as found
- `checkAllFound` checks whether all cards are found
- `clearFlips()` clears all flipped cards
- `clearFlipBackTimer` resets the timer used when we flip the cards.

Finally, we add methods for starting and finishing the game:

```
methods: {
  ...

  startGame() {
    this.started = true;
    let duration = this.duration;
    this.timer = setInterval(() => {
      if (duration == 1000) {
        this.finishGame();
      }
      duration -= 1000;
      this.time = moment(duration).format("mm:ss");
      this.progress = (duration / this.duration) * 100;
    }, 1000);
  },
}
```

```

},

finishGame() {
  this.started = false;
  clearInterval(this.timer);
  this.score = Math.round(this.progress) * 10 * this.matches;

  if (this.matches == cardsSet.length) {
    this.result = "Win";
  } else {
    this.result = "Lose";
  }

  this.showSplash = true;
}
}

```

As we'll see a bit later, `startGame()` is called after we flip the first card. It creates the game timer, which is used in the `Timer.vue` component. It also updates the time and progress props. When one second is left, the `finishGame()` is called. The latter clears the game timer, calculates the score and result props, and activates the splash screen.

Building the Game Components

Okay, it's time to build the components we created earlier.

The CardsBoard Component

We open the `CardsBoard.vue` and put the following:

```

<template>
  <v-layout>
    <v-flex xs12 sm10 offset-sm1 lg8 offset-lg2>
      <v-card>
        <div class="flex-container">
          <div
            v-for="(card, i) in cards"
            :key="i"
            class="card"
            :class="{ flipped: card.flipped, found: card.found }"
            @click="flipCard(card)"
          >
            <div class="back"></div>
            <div class="front" :style="{ backgroundImage: 'url(' + card.img + ')' }"></div>
          </div>
        </div>
      </v-card>
    </v-flex>
  </v-layout>
</template>

<script>
export default {
  props: ["cards"],
  methods: {
    flipCard(card) {
      this.$emit("flipcard", card);
    }
  }
};
</script>

```

Here, we use the `v-for` directive to populate the board with the cards by using the passed `cards` prop. For each card we bind `.flipped` and/or `.found` class(es), depending on the states of their `flipped` and `found` props. We create the back and front faces of

the cards and populate the front using the `img` prop. And of course, we add a click event listener, which will emit a `flipcard` event.

For the code above to work, we'll need some CSS. So we put this in the style section:

```
<style>
.flex-container {
  display: flex;
  flex-flow: row wrap;
  justify-content: space-around;
}
.card {
  position: relative;
  width: 100px;
  height: 150px;
  margin: 1px;
  transition: opacity 0.5s;
}
@media only screen and (max-width: 768px) {
  .card {
    width: 50px;
    height: 75px;
  }
}
.card .front,
.card .back {
  border-radius: 5px;
  position: absolute;
  left: 0;
  right: 0;
  top: 0;
  bottom: 0;
  width: 100%;
  height: 100%;
  backface-visibility: hidden;
  transition: transform 0.3s;
  transform-style: preserve-3d;
}
.card .back {
  background-image: url("https://s3-us-west-2.amazonaws.com/s.cdpn.io/102308/card_backside.jpg");
  background-size: 100%;
  background-repeat: no-repeat;
}
.card .front {
  transform: rotateY(-180deg);
  background-size: 100%;
  background-repeat: no-repeat;
  background-position: center;
  background-color: white;
}
.card.flipped .back,
.card.found .back {
  transform: rotateY(180deg);
}
.card.flipped .front,
.card.found .front {
  transform: rotateY(0deg);
}
.card.found {
  opacity: 0.3;
}
</style>
```

I'm not going into too much detail here. Basically, these styles create correct cards layout and add flipping transitions.

Now, let's move to the `App.vue` and add the following:

```
<cards-board :cards="cards" @flipcard="flipCard"></cards-board>
```

Here, we bind the `cards` prop and register the `flipcard` event listener. Now, let's add the `flipCard()` method to the methods in the script section:

```
methods: {
  ...

  flipCard(card) {
    if (card.found || card.flipped) return;

    if (!this.started) {
      this.startGame();
    }

    let flipCount = this.flippedCards().length;
    if (flipCount == 0) {
      card.flipped = !card.flipped;
    } else if (flipCount == 1) {
      card.flipped = !card.flipped;
      this.turns += 1;

      if (this.sameFlippedCard()) {
        this.matches += 1;
        this.flipBackTimer = setTimeout(() => {
          this.clearFlipBackTimer();
          this.setCardFound();
          this.clearFlips();

          if (this.checkAllFound()) {
            this.finishGame();
          }
        }, 1000);
      } else {
        this.flipBackTimer = setTimeout(() => {
          this.clearFlipBackTimer();
          this.clearFlips();
        }, 1000);
      }
    }
  }
}
```

First, if we click on a flipped or found card, the function will terminate. If `started` is false, it will start the game. Next, when we open the first card, it will change its `flipped` prop to `true`. When we open the second card, it will change its `flipped` prop to `true` and will increase the `turns` prop. Next, if the cards are the same, it will increase the `matches` prop and set a `flipBackTimer`. It will wait one second and will set the `cards found` props to `true`, and will clear the flipped cards. If all cards are found, the game will finish. If the two opened cards are not same, it will just clear the flipped cards.

The Timer Component

Open `Timer.vue` and add the following:

```
<template>
  <v-layout justify-end>
    <v-progress-circular
      :rotate="-90"
      :value="progress"
      :size="70"
      :width="9"
      :color="color"
    >{{time}}</v-progress-circular>
```

```

    </v-layout>
  </template>

  <script>
  export default {
    props: ["progress", "time"],
    data() {
      return {
        color: "success"
      };
    },
    watch: {
      progress() {
        if (this.progress <= 10) {
          this.color = "red";
        } else {
          this.color = "success";
        }
      }
    }
  };
  </script>

```

Here, we use the Vuetify Progress component. We need to change the timer's color dynamically depending on the progress value. In order to update the progress prop constantly, we need to add a watcher for it.

Now we move to App.vue and bind the passed props:

```
<timer :progress="progress" :time="time"></timer>
```

StatusBar

Open StatusBar.vue and add the following:

```

<template>
  <v-layout align-center class="headline">
    <span class="mx-3">Turns: {{ turns }}</span>
    <v-divider vertical></v-divider>
    <span class="mx-3">Matches: {{ matches }}</span>
  </v-layout>
</template>

<script>
export default {
  props: ["turns", "matches"]
};
</script>

```

Here, we just pass the turns and matches props, and use them in the template.

In App.vue, we bind the passed props:

```
<status-bar :turns="turns" :matches="matches"></status-bar>
```

Splash

Finally, open Splash.vue and add the following:

```

<template>
  <v-layout row>
    <v-dialog v-model="showSplash" persistent max-width="300">
      <v-card class="text-xs-center">
        <v-card-text>
          <div class="headline mb-4">You {{result}}!</div>
          <div>Score: {{score}}</div>
        </v-card-text>
      </v-card>
    </v-dialog>
  </v-layout>

```

```

    </v-card-text>
    <v-card-actions>
      <v-spacer></v-spacer>
      <v-btn color="green darken-1" flat @click="resetGame">Start New Game</v-btn>
      <v-spacer></v-spacer>
    </v-card-actions>
  </v-card>
</v-dialog>
</v-layout>
</template>

<script>
export default {
  props: ["showSplash", "result", "score"],
  methods: {
    resetGame() {
      this.$emit("resetgame");
    }
  }
};
</script>

```

We use the Vuetify Dialog component for the splash screen. The splash will show up when the showSplash is true.

In App.vue, we bind the passed props and register the emitted resetgame event:

```
<splash :showSplash="showSplash" :result="result" :score="score" @resetgame="resetGame"></splash>
```

And with that, our journey into the world of game development is at an end. Now, just start the game (npm run serve) and enjoy!

Conclusion

As you saw, building a game with Vue is easy and fun. In many cases, Vue can successfully replace a real game framework. With a little imagination and some effort we can create interesting and creative games with different levels of complexity. Check this collection for some examples. Happy game development!

Chapter 10: Build a Shopping List App with Vue, Vuex and Bootstrap Vue

by Michael Wanyoike

As a Vue app grows, keeping track of state throughout the app can become a tricky process. For example, a given piece of state might need to be used in multiple components. Or maybe it's needed by a component which is nested several components deeper than the one in which it's stored.

Vuex is the official state management solution for Vue and is designed to help you manage state throughout your app as your application grows. In this tutorial, I'll offer a practical example of how to get up and running with Vuex by building a shopping list app (which is a glorified to-do app, to be honest).

Demo and Code

If you're curious to see what we'll end up with, you can view the live version running on CodeSandbox. If you'd like to grab a copy of the code, please see this GitHub repo.

Prerequisites

Before we start, I'll assume that you:

- have a basic knowledge of Vue.js
- have a basic knowledge of Vuex and its core concepts (check out "Getting Started with Vuex: a Beginner's Guide" in this Vue series)
- are familiar with ES6 and ES7 language features

To follow along with this tutorial, I recommend you install code highlighting for Vue in your editor of choice. Vetur for VSCode is a popular choice for this, as is Vue Syntax Highlight for Sublime.

I also recommend that you install the Vue.js DevTools for your browser, as I'll be making use of them later on to inspect the state of our application. You can find the Chrome extension here, or the Firefox extension here.

In addition, you'll need to have a recent version of Node.js that's not older than version 6.0. At the time of writing, Node.js v10.13.0 (LTS) and npm version 6.4.1 are the most recent. If you don't have a suitable version of Node installed on your system already, I recommend using a version manager.

Finally, you should have the most recent version of Vue CLI installed:

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

At the time of writing, this was v3.1.3.

Create a New Project

Let's generate a new project using the CLI:

```
vue create vuex-todo
```

A wizard will open up to guide you through the project creation. Select *Manually select features* and ensure that you choose to install Vuex. After you've finished selecting your options, Vue CLI will scaffold your project and install the dependencies.

We're going to use the following dependencies for our Vuex project. They're not part of Vuex, but they'll make our

web interface pretty and interactive.

- Bootstrap Vue - Bootstrap CSS framework V4 re-engineered for Vue.js projects
- VeeValidate, a form validation framework for Vue.js projects

Let's change into the project directory and install them:

```
cd vuex-todo
npm install bootstrap-vue vee-validate
```

To test everything has run correctly, start up the app using `npm run serve` and visit `http://localhost:8080`. You should see a welcome screen.

Basic Project Setup

Now that we have all our dependencies installed, we can now go ahead and start building our main application structure. Under the `src/components` folder, delete the `HelloWorld.vue` file, then create two new files `—TodoList.vue` and `TodoItem.vue`:

```
rm src/components/HelloWorld.vue
touch src/components/{TodoList.vue, TodoItem.vue}
```

Next, let's inject Bootstrap Vue and VeeValidate by adding the following code to `main.js`:

```
...
import BootstrapVue from 'bootstrap-vue'
import VeeValidate from 'vee-validate';
import 'bootstrap/dist/css/bootstrap.css'
import 'bootstrap-vue/dist/bootstrap-vue.css'
...
Vue.config.productionTip = false
Vue.use(BootstrapVue)
Vue.use(VeeValidate);
```

Now open up `src/App.vue` and replace the existing code with the following:

```
<template>
  <div id="app">
    <b-jumbotron bg-variant="info" text-variant="white">
      <template slot="header">
        <b-container>Vuex Todo App</b-container>
      </template>
      <template slot="lead">
        <b-container>Built using Bootstrap-Vue.js</b-container>
      </template>
    </b-jumbotron>

    <b-container>
      <div class="todo-page">
        <h2>Shopping List</h2>
        <hr>
        <TodoList />
      </div>
    </b-container>
  </div>
```

```

</template>

<script>
  import TodoList from '@components/TodoList.vue'

  export default {
    name: 'TodoView',
    components: {
      TodoList
    }
  }
</script>

```

Here we're using Bootstrap components to lay out our application page. It's easier this way instead of defining Bootstrap class names for each HTML element. You can find the documentation for the components used on the Bootstrap Vue.js website.

For the components, we'll just insert some placeholder code. Insert the following code into components/TodoList.vue:

```

<template>
  <p>To-do List component under construction</p>
</template>

```

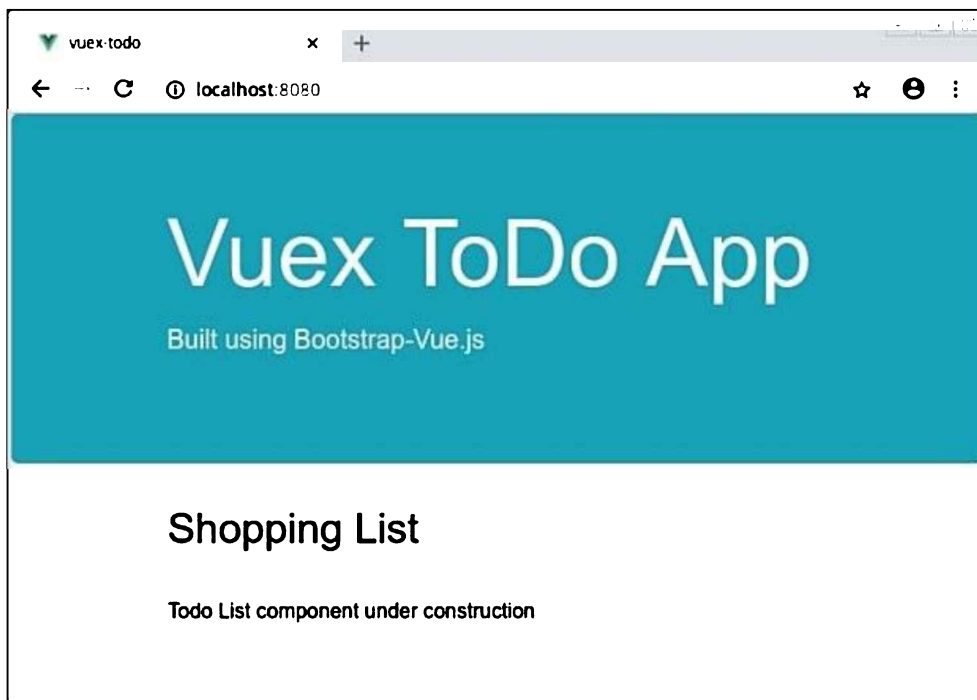
Insert the following code for components/ToDoItem.vue:

```

<template>
  <p>To-do Item component under construction</p>
</template>

```

Finally, head back to <http://localhost:8080>, where you should have the following page view:



Building the To-do List

To build our to-do list, we'll take a bottom-up approach. In other words, we'll start by building our Vuex store, then finish by creating the components. We'll use a very simple `TodoItem` model for our `TodoList` collection that only consists of two properties:

```
{
  name: "Butter",
  done: false
}
```

Creating the Store

Open `src/store.js` and replace the default export with the following code:

```
export default new Vuex.Store({
  state: {
    items: [
      {
        name: "Milk",
        done: false
      },
      {
        name: "Bread",
        done: true
      },
      {
        name: "Cake",
        done: false
      }
    ]
  },
  mutations: {
    addItem(state, item) {
      state.items.push(item)
    },
    editItem(state, { item, name = item.name, done = item.done }) {
      item.name = name;
      item.done = done;
    },
    removeItem(state, item) {
      state.items.splice(state.items.indexOf(item), 1);
    }
  },
  actions: {
    addItem({ commit }, item) {
      commit("addItem", {
        name: item,
        done: false
      })
    },
    editItem({ commit }, { item, name }) {
      commit("editItem", { item, name });
    },
    toggleItem({ commit }, item) {
      commit("editItem", { item, done: !item.done });
    }
  }
});
```

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

As you can see, we start off by declaring our items in a state object.

We then have three mutations for manipulating the state—`addItem`, `editItem` and `removeItem`. The `addItem` mutation handler takes an item and pushes it on to state, and the `removeItem` mutation handler takes an item and uses Array’s `splice` method to remove it from state. The `editItem` mutation handler perhaps looks a bit funky, but all it’s doing is accepting an object as an argument with three properties:

- `item`: the record to be updated.
- `name`: the new value for name field. If none is provided, it uses `item.name`.
- `done`: the new value for the done field. If none is provided, it uses `item.done`.

It then updates the properties accordingly.

This syntax is new to ES6. You can read more about it in our post “ES6 in Action: Destructuring Assignment”.

Finally come the actions. These will call our mutations to create a new item, to edit an item and to toggle an item’s done property. Notice how the `editItem` and `toggleItem` action handlers also take advantage of destructuring to accomplish their tasks.

If you’d like a refresher on these core concepts of Vuex, please refer to “Getting Started with Vuex: a Beginner’s Guide” in this Vue series.

Building the `TodoList` Component

Let’s now build out our `src/components/TodoList` component.

Start by adding the following code to `src/components/TodoList.vue`:

```
<template>  
  <div class="todo-list">  
    <!-- start of to-do form -->  
    <b-form class="row" >  
      <b-col cols="10">  
        <b-form-input  
          id="item"  
          name="item"  
          class="w-100"  
          placeholder="What do you want to buy?"  
        ></b-form-input>  
      </b-col>  
      <b-col cols="2">  
        <b-button type="submit" variant="primary">Add Item</b-button>  
      </b-col>  
    </b-form>  
    <!-- end of to-do form -->  
  
    <!-- start of to-do list -->  
    <b-row>  
      <b-col md="10">  
        <b-list-group>  
          <b-list-group-item v-for="(item, index) in items" :key="index" :item="item">
```

```

        {{ item.name }}
      </b-list-group-item>
    </b-list-group>
  </b-col>
</b-row>
<!-- end of to-do list -->
</div>
</template>

```

```

<script>
import { mapState } from 'vuex';

export default {
  name: 'TodoList',
  computed: {
    ...mapState([
      'items'
    ])
  },
};
</script>

```

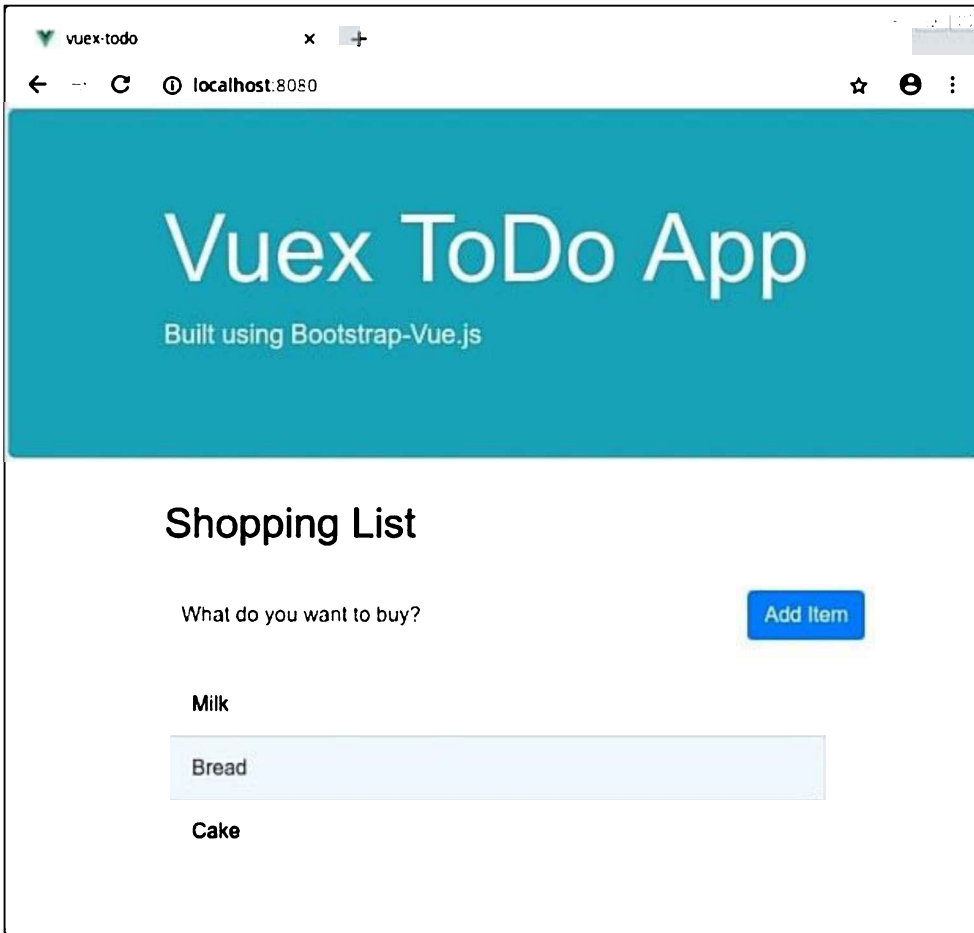
```

<style>
form {
  margin-bottom: 25px;
}
.list-group-item {
  display: flex;
}
.list-group-item:hover{
  background-color: aliceblue;
}
.checked {
  font-style: italic;
  text-decoration: line-through !important;
  color: gray;
  background-color: #eaeaea;
}
</style>

```

It's essentially made up of a form and list group for displaying the shopping list items. There's a bit of CSS and Bootstrap styling classes to make the layout look right. If there's anything here you're not sure of, please consult the Bootstrap Vue documentation.

This is what you should have on the to-do page:



Inspect the App's State Using the Vue DevTools

If you've installed the Vue.js browser extension mentioned at the beginning of this guide, you can press F12 and then click the *Vue* tab. In here, you'll see another tab called *Vuex*. Click it. You should be able to see the entire store for your application.



Once we've made the component interactive, you'll be able to add new items or delete old ones and see the Vuex store update accordingly. You'll also be able to revert mutations by clicking on the *Revert* button.

Adding an Item

Let's now make our shopping list interactive. There are several things we need to do here:

- Allow users to add items, mark them as done and delete them.
- Add validation to the input form using the VeeValidate package. The only rule we're using is `v-validate="'required'"` that will prevent the submission of a blank form.
- Display validation error messages if present using VeeValidate `errors` object.
- Create a local state variable called `item` that will bind to the input form.
- Create an `onSubmit()` function that will handle the form's POST events using Vue.js `@submit.prevent` modifier.

Let's start enabling the "Add Item" feature. Update the template section in `src/components/ToDoList.vue` as follows:

```
<!-- start of to-do form -->
<b-row>
  <b-col>
    <!-- display validation error -->
    <b-alert v-if="errors.has('item')" show dismissible variant="danger">
      {{ errors.first('item') }}
    </b-alert>
  </b-col>
</b-row>
<!-- post to onSubmit function -->
<b-form class="row" @submit.prevent="onSubmit">
  <b-col cols="10">
    <!-- bind to local `item` state -->
    <b-form-input
      id="item"
      class="w-100"
      name="item"
      type="text"
      placeholder="What do you want to buy?"
      v-model="item"
      v-validate="'required'"
      autocomplete="off"
    ></b-form-input>
  </b-col>
  <b-col cols="2">
    <b-button type="submit" variant="primary">Add Task</b-button>
  </b-col>
</b-form>
<!-- end of to-do form -->
```

Let's now define the local state `item` and `onSubmit()` function in the `<script>` section as follows:

```
import { mapState, mapActions } from 'vuex';

export default {
  name: 'ToDoList',
  data() {
    return {
      item:''
    }
  }
}
```



```

    }
  },
  computed: {
    ...mapState([
      'items'
    ])
  },
  methods: {
    ...mapActions([
      'addItem',
    ]),
    async onSubmit() {
      const isValid = await this.$validator.validateAll();
      if(isValid) {
        await this.addItem(this.item);
        this.item=''; // Clear form after successful save
        this.$validator.reset();
      }
    },
  },
};

```

In the `onSubmit` method, we first execute a validation method to ensure that the form input value is valid. If it's not valid, an error message is displayed. If it's valid, the item is passed to the `addItem` action. The local state `item` is cleared and the validator is reset to prevent incorrect error messages from showing up. After you've implemented the changes and tested the form, you should have something similar:

Shopping List

The image shows a web form for a shopping list. At the top, there is a red error message box that says "The item field is required." with a close button (X). Below the error message is a text input field with the placeholder text "What do you want to buy?". To the right of the input field is a blue button labeled "Add Task". Below the input field, there is a list of items: "Milk", "Bread", and "Cake".

Remember to check out the [Vue DevTools](#) to see how the store reacts to your input events.

The To-do Item Component

Let's now work on the `src/components/ToDoItem` component. Copy the following code:

```

<template>
  <div class="todo-item">
    <b-list-group-item class="row">
      {{ item.name }}
    </b-list-group-item>
  </div>

```

```
</template>
```

```
<script>
  export default {
    name: 'TodoItem',
    props: ['item']
  }
</script>
```

Next, go back to `src/components/ToDoList`. We need to replace the `<b-list-group-item>` code section with the new component `TodoItem`. Go to the template section and replace the code section with `<b-list-group>` with this:

```
<b-list-group>
  <TodoItem v-for="(item, index) in items" :key="index" :item="item" />
</b-list-group>
```

In order to use the `TodoItem` component in `ToDoList`, you'll need to import it in the `<script>` section like this:

```
import TodoItem from './TodoItem.vue'
//..

export default {
  name: 'ToDoList',
  components: {
    TodoItem
  },
  //..
}
```

That's it. Save your changes and the to-do shopping list should operate as before. Now that we have a dedicated `TodoItem` component, we can now easily add more features:

- toggle item state (done field)
- edit item names
- delete items

Toggle an Item's State

Let's start by toggling the item status. In the `todo` store, we already have an action that can do that for us:

```
toggleItem({ commit }, item) {
  commit("editItem", { item, done: !item.done })
}
```

By mapping the action locally using the `mapAction` helper, we can access this function directly from the `TodoItem` component like this:

```
this.toggleItem(item)
```

Let's add a checkbox input by updating the `<template>` code as follows:

```
<template>
  <div class="todo-item">
    <b-list-group-item v-bind:class="{ checked: item.done }">
```

```

<b-row>
  <b-col cols="1">
    <b-form-checkbox
      :checked="item.done"
      @change="changeItemStatus(item) "
    >
  </b-form-checkbox>
</b-col>
<b-col cols="9">
  <span>{{ item.name }}</span>
</b-col>
</b-row>
</b-list-group-item>
</div>
</template>

```

We now should have the following view:

Shopping List

What do you want to buy?

Add Task

Milk

Bread

Cake

However, do note that we run into an error if we try to toggle any of the checkboxes. This is because we haven't defined the `changeItemStatus` function.

Let's do that now:

```

import { mapActions } from 'vuex'

export default {
  name: 'TodoItem',
  props: ['item'],
  methods: {
    ...mapActions([
      'toggleItem'
    ]),
    changeItemStatus(item) {
      this.toggleItem(item);
    }
  }
};

```

Go ahead and test the checkbox. The items should now be toggleable. You can confirm via the DevTools extension that the store is actually reacting to your clicks and that the item's `done` property is updating in state.

Removing an Item

Next, let's add a *remove* feature. For this, we won't dispatch an action, but instead we'll commit the `removeItem` mutation directly from the `TodoItem` component. This is the code to accomplish that:

```
<template>
  ...
  <b-col cols="10">
    <span>{{ item.name }}</span>
  </b-col>
  <b-col cols="1">
    <b-button-close @click="removeItem(item)"></b-button-close>
  </b-col>
  ...
</template>

<script>
  ...
  methods: {
    ...mapActions([
      'toggleItem'
    ]),
    changeItemStatus(item) {
      this.toggleItem(item);
    },
    removeItem(item) {
      this.$store.commit("removeItem", item)
    },
  },
}
...
</script>
```

After you've saved your changes, go ahead and test the delete button to make sure that it's working properly.



Updating an Item

Finally, let's add the `edit` feature. This one is going to be a bit tricky. This is what we're going to do:

- We'll create a local state called `editing`, which by default will be set to `false`.
- When it's `false`, the item's name will be displayed via a `span` element.
- When it's `true`, the item's name will be displayed via an input text box.
- We'll create a custom Vue.js directive for automatically setting focus on the text box when it's visible.

- Using events triggered via `@keyup.enter` and `@blur`, we'll call a custom function `doneEdit`, for handling the edit changes.
- Using the event triggered via `@keyup.esc`, we'll call a custom function, `cancelEdit`, for cancelling the edit. This function will also reset the `item.name` state back to its original, and set the editing value to `false`.

Let's now start implementing the changes. We'll add the text box by replacing the relevant code in the `template` section in `src/components/ToDoItem.vue` with this:

```
<template>
...
<b-col cols="10">
  <span v-if="!editing" @dblclick="editing = true">{{ item.name }}</span>
  <input class="edit"
    v-show="editing"
    v-focus="editing"
    :value="item.name"
    @keyup.enter="doneEdit"
    @keyup.esc="cancelEdit"
    @blur="doneEdit"
  >
</b-col>
...
</template>
```

Next, we'll add the editing Boolean state variable, the `focus` directive, as well as the `doneEdit` and `cancelEdit` functions in the `script` section.

Update the code as follows:

```
export default {
  //...
  data () {
    return {
      editing: false
    }
  },
  directives: {
    focus (el, { value }, { context }) {
      if (value) {
        context.$nextTick(() => {
          el.focus()
        })
      }
    }
  },
  methods: {
    ...mapActions([
      'toggleItem',
      'editItem'
    ]),
    //..
    doneEdit(event) {
      const value = event.target.value.trim();
```

```

    const { item } = this;
    if (!value) {
      this.removeItem(item)
    } else if (this.editing) {
      this.editItem({ item, name:value });
      this.editing = false
    }
  },
  cancelEdit(event) {
    event.target.value = this.item.name;
    this.editing = false;
  }
}
}
}

```

Save the changes and test the new editing feature. To make a change on an item, simply double click on it. You can press Return or click away to save the changes. You can also hit ESC if you don't want to save the change. You should have a similar view as below:

Shopping List

What do you want to buy? Add Task

✕

Bread
✕

Cake
✕

Taking it Further

Well done if you made it this far! You've grasped the majority of the skills you need to build a Vuex project. I'm going to leave you with a challenge. The project we just built is using a hard-coded list of to-do items. In other words, every time we perform a full browser refresh, the `todo` store is reset back to its hard-coded values:

```

state: {
  items: [
    {
      name: "Milk",
      done: false
    },
    {
      name: "Bread",
      done: true
    },
    {
      name: "Cake",
      done: false
    }
  ]
}

```

```
},
```

I want you set the initial state for items to an empty array like this:

```
state: {
  items: []
}
```

Then, I want you to use a real database for persisting the `items` collection. You can use any database system you're comfortable using, such as MongoDB, a ready-made API interface such as JSON server, or a cloud-hosted database such as mLab. You can then use a library like `axios` to fetch data from your remote API to the application's store.

Here's some clues on how you can set it up. You'll need to create a new mutation for this task in your `todo` store that looks like this:

```
export default {
  state: {
    items: []
  },
  mutations: {
    fetchTodos(state, items) {
      state.items = items
    }
  },
  actions: {
    async fetchTodos({commit}) {
      const response = await axios.get('http://localhost:4000/api/todos');
      commit("fetchItems", response.data)
    }
  }
}
```

Then in your `TodoList` component, you can use Vue's `created` lifecycle hook to retrieve the data:

```
<script>
import { mapActions } from 'vuex'

export default {
  computed: {
    ...mapActions([
      'fetchTodos'
    ])
  }
  created: function() {
    this.fetchTodos()
  }
}
</script>
```

Note that I haven't done any error handling. I'll leave it to you to figure out how to display an appropriate error message in case the `fetchTodos` function fails.

Summary

On a personal level, I really like Vuex as someone coming from a background in Redux and MobX state management libraries. I find Vuex to be a bit leaner and cleaner. It does a lot of the heavy lifting for you, making it easy to implement your code. The Vue DevTools are a big bonus that allow you to debug your Vuex store easily in the event that something goes wrong.

Vuex is tightly integrated with Vue.js and is maintained by the Vue.js team. I would say that's a pretty good reason to rely on Vuex for a commercial project. Being open source, you can easily confirm that the project is being actively maintained on GitHub. You can also confirm that it doesn't have too many unresolved issues.

With that said, you should note that I haven't been able to cover 100% of the features available in the Vuex library. I would highly recommend you check out their current documentation to learn more about its advanced features.

Chapter 11: How to Develop and Test Vue Components with Storybook

by Ivaylo Gerchev

In today's web application development, components play a central role. The complexity in modern apps requires us to split the codebase into small and manageable pieces—components—which then we can use as Lego bricks to build the entire app. To facilitate the building process and make it more automated and interactive, some smart developers have created specialized tools for component development. One such a tool is Storybook.

What Is Storybook?

Storybook is an interactive environment for developing and testing UI components. It allows developers to build components and play with them in isolation from the actual app. A Storybook project can be exported as a separate, static app, which is ideal for making live and interactive component showcases. Storybook is easy to configure and use, and can be extended via a simple API. It has a bunch of useful add-ons, both official and community made, which enhance the core with lots of different functionality.

Getting Started

In this tutorial, we'll learn how to use Storybook to develop Vue components.

The Finished Result

If you'd like to view the finished result, you can find the Storybook deployed to GitHub Pages.

To follow along with this tutorial, you'll need Git and Node installed on your machine. You can find instructions on how to install Git [here](#). To install Node, you can download the binaries 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.

As a starting point, we'll use a `vue-pomodoro` timer app that I've already written. The Pomodoro Technique is a time-management method developed by Francesco Cirillo in the late 1980s, whereby one uses a timer to break down work into intervals, separated by short breaks. I created this app using Vue CLI. (See "A Beginner's Guide to Vue CLI" in this Vue series if you're not familiar with it.) I also chose to install the Babel, ESLint and Jest plugins. We'll need the Jest plugin for testing our components later on.

Clone the app to your local machine, install the dependencies and make sure that it works:

```
git clone git@github.com:sitepoint-editors/vue-pomodoro.git
cd vue-pomodoro
npm i
npm run serve
```

Installing Storybook

Next we need to install Storybook and some additional dependencies.

Checking Your Progress

The repo has a second branch (named `storybook`) which you can check out, should you wish to jump straight to the finished code, or simply check your progress. You can see the `storybook` branch on GitHub, or you can check it out via the terminal:

```
git clone git@github.com:sitepoint-editors/vue-pomodoro.git
git checkout storybook
```

Install the Storybook itself by running the following:

```
npm i --save-dev @storybook/vue
```

Next, we need to add some dependencies:

```
npm i --save-dev vue-loader
npm i --save-dev babel-preset-vue
```

The next step is to add the `storybook` script in our project's `package.json` file:

```
{
  "scripts": {
    ...
    "storybook": "start-storybook -p 9001 -c .storybook"
  }
}
```

The script starts the Storybook development server at port 9001 and defines `.storybook` as a config directory. We'll also need to create a `config.js` file in the `.storybook` directory, in which we import and register our components and load our stories. We'll learn what stories are and how to create them in the next section.

```
mkdir .storybook
touch .storybook/config.js
```

Add the following to `config.js`:

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

import Vue from 'vue';

// Import your custom components.
import State from '../src/components/State.vue';
import Countdown from '../src/components/Countdown.vue';
import Controls from '../src/components/Controls.vue';

// Register your custom components.
Vue.component('state', State);
Vue.component('countdown', Countdown);
Vue.component('controls', Controls);

function loadStories() {
  // You can require as many stories as you need.
  require('../src/stories');
}

configure(loadStories, module);
```

Writing Stories

Stories are the heart of Storybook. A story represents a particular state of one single component or multiple components grouped together. We can have as many stories for as many components which we need. The great thing is that when we edit the stories the changes are applied and displayed immediately. This gives us a smooth, interactive experience.

The stories are stored in their own directory, which keeps them closer to the app's components. In our case, we navigate to the `src` directory, and create a new `stories` folder inside. Then add an `index.js` file:

```
mkdir src/stories
touch src/stories/index.js
```

Writing stories is quite simple. Let's create our first story now in `index.js`:

```
import Vue from "vue";
import { storiesOf } from "@storybook/vue";

storiesOf("State", module).add("state toggle", () => ({
  template: `<state :isworking="false"></state>`
}));
```

The `storiesOf` method wraps a story or set of stories about a particular component or group of components. The first argument defines a name for the wrapper. The individual stories are just different states, which we create by using the `add` method. The first argument defines the state's name, and the second is a function in which we create the actual state. There are many ways available to render a state. To see them all, check out the [Vue real examples](#).

Let's now see our story in action by running the project:

```
npm run storybook
```

You should see the following in your browser:

```
11:05:00 AM X Rest:
- state
  state toggle
```

Installing and Using Add-ons

As we saw, Storybook's core has limited functionality. To add more useful features, we'll need to install some add-ons. There are two types of add-ons:

- **Decorators** wrap a story or stories and transform them in some way. For example, we can create a center text decorator, which will center the text in our story. We'll see a practical example of this later on.
- **Native Add-ons** allow for low-level interaction with the Storybook platform. We'll also see examples of those.

Installing the Storysource Add-on

The first add-on we'll install is Storysource. It adds a "STORY" panel to the storybook, which shows and highlights the source of the selected story. It's useful because we can see how the story is written without the need to open the story file and search for the story manually. To install the add-on, we run:

```
npm install -D @storybook/addon-storysource
```

Before we can use the add-on, we need to register it in a special `addons.js` file, which we must create in the `.storybook` directory:

```
touch .storybook/addons.js
```

Then add:

```
import '@storybook/addon-storysource/register';
```

And one more thing. The add-on needs some custom webpack configuration. So we create a `webpack.config.js` file in the same `.storybook` directory:

```
touch .storybook/webpack.config.js
```

And add the following content to it:

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

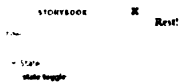
module.exports = (storybookBaseConfig, configType, defaultConfig) => {
  defaultConfig.module.rules.push({
    test: [/\.stories\.js$/, /index\.js$/],
    loaders: [require.resolve('@storybook/addon-storysource/loader')],
    include: [path.resolve(__dirname, '../src')],
    enforce: 'pre',
  });

  return defaultConfig;
};
```

Restarting the Storybook

After we've installed new add-ons, we need to restart the storybook so for the changes to take effects.

Finally, we can see the add-on in action:



Installing the Options Add-on

The next add-on we'll install is Options. It gives us the ability to use some options for configuring Storybook's UI.

Install it with this:

```
npm install -D @storybook/addon-options
```

Register it in the addons.js:

```
import '@storybook/addon-options/register';
```

Then we can use it in our config.js file like this:

```
import { configure, addDecorator } from '@storybook/vue';
import { withOptions } from '@storybook/addon-options';

addDecorator(
  withOptions({
    name: 'Vue Pomodoro',
  })
);
```

We import the `addDecorator` method and set the necessary options via `withOptions` decorator.

Now, the name in the top left corner of our storybook will be "VUE POMODORO", instead of "STORYBOOK". You can play with other options as you wish.

Installing Actions, Links, and Knobs Add-ons

In this section we'll install three more add-ons: Actions, Links, and Knobs.

You can install them like so:

```
npm install -D @storybook/addon-knobs @storybook/addon-links @storybook/addon-actions
```

And register them in addons.js:

```
import '@storybook/addon-knobs/register';
import '@storybook/addon-links/register';
import '@storybook/addon-actions/register';
```

Registration Order

The order of registration matters here, because the add-ons will be displayed in the same sequence in the Storybook's panels section.

Using the Knobs Add-on

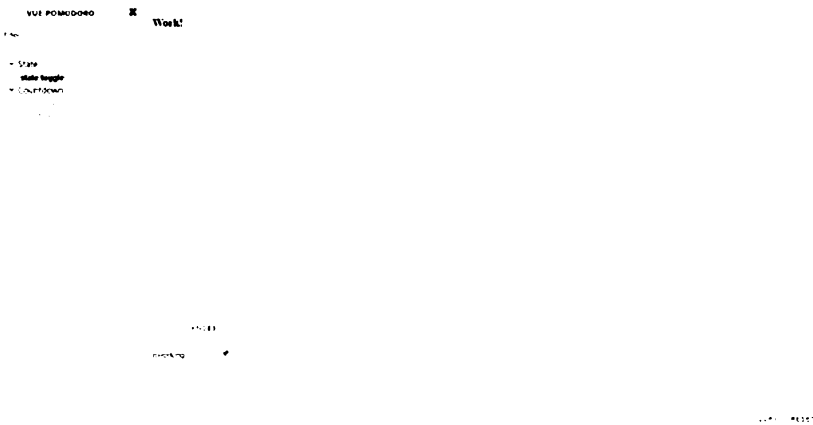
Knobs allow us to edit Vue props dynamically. Let's see this in action. In the `index.js` file add the following:

```
import { withKnobs, boolean } from '@storybook/addon-knobs';

...

storiesOf("State", module)
  .addDecorator(withKnobs)
  .add("state toggle", () => ({
    props: {
      isworking: {
        type: Boolean,
        default: boolean("isworking", true)
      }
    },
    template: `<state :isworking="isworking"></state>`
  }));
```

Here, we import the `withKnobs` decorator and `boolean` knob. Then, we add the decorator to our story. We define the `isworking` prop and use the `boolean` knob to define the default value. Now, when we go to this story, we'll see a checkbox in the "KNOBS" panel bound to the `isworking` prop.



Creating and Using a Custom Decorator

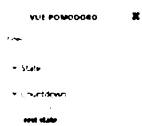
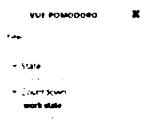
Let's now see how to create a custom decorator. In the same `index.js` file, add the following:

```
const CenterDecorator = () => ({
  template: '<div style="textAlign: center;"><story/></div>',
});

...

storiesOf("Countdown", module)
  .addDecorator(CenterDecorator)
  .add("work state", () => ({
    template: `<countdown :minute="1" :second="0" :isworking="true"></countdown>`
  }))
  .add("rest state", () => ({
    template: `<countdown :minute="5" :second="0" :isworking="false"></countdown>`
  }));
```

What we do here is define a `CenterDecorator` decorator. We wrap the `<story/>` component with our custom functionality. Then we add it to the two new states we created for the `Countdown` component.



Using the Links and Actions Add-ons

In this section, we'll create stories about the Pomodoro controls buttons, and we'll use links to simulate the real world behavior when we click them.

In the `index.js` file, add the following:

```
import { linkTo } from '@storybook/addon-links';
import { action } from '@storybook/addon-actions';

...

storiesOf("Controls", module)
  .add("started state", () => ({
    template: `<controls state="started" @paused="linkToPaused()" @stopped="linkToStopped()"></controls>`,
    methods: {
      linkToPaused: linkTo("Controls", "paused state"),
      linkToStopped: linkTo("Controls", "stopped state")
    }
  }))
  .add("paused state", () => ({
    template: `<controls state="paused" @started="linkToStarted()" @stopped="linkToStopped()"></controls>`,
    methods: {
      linkToStarted: linkTo("Controls", "started state"),
      linkToStopped: linkTo("Controls", "stopped state")
    }
  }))
  .add("stopped state", () => ({
    template: `<controls state="stopped" @started="linkToStarted()", action()></controls>`,
    methods: {
      linkToStarted: linkTo("Controls", "started state"),

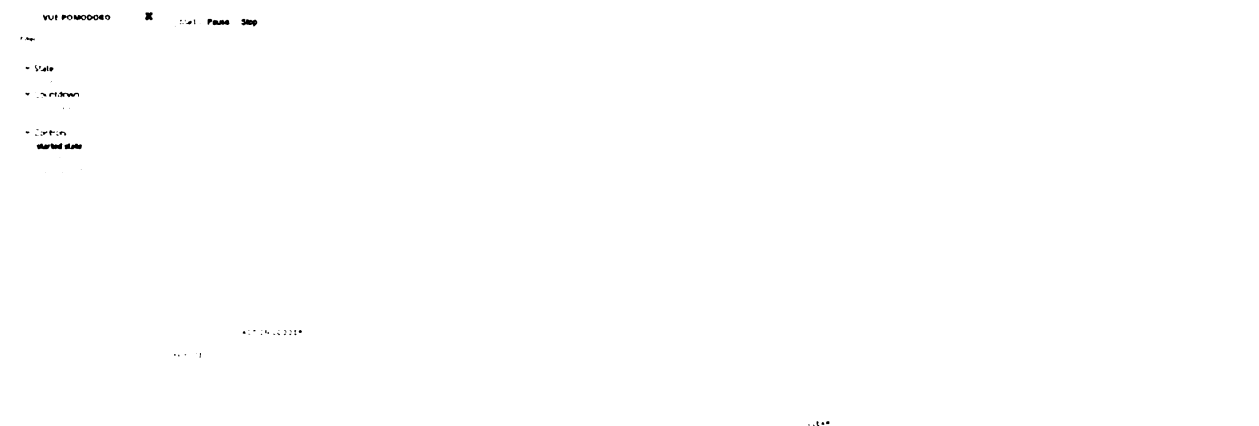
```

```

    action: action("Starting")
  }
  ));

```

Here, we import the `linkTo` and `action` methods. Then we create stories about the three possible states of the buttons. Next, we create links to navigate between the states according to the pressed button. In the stopped state we use also an action. So when we press the `Start` button, it will log the start action in the "ACTION LOGGER" panel.



Combining Multiple Components

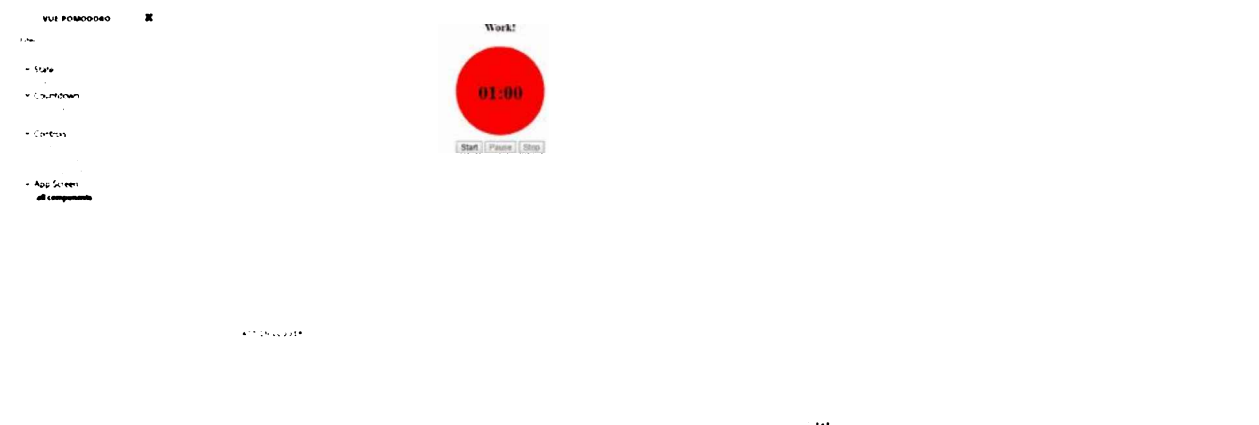
In Storybook, we can not only create a story for one single component, but we can combine multiple components into one Story screen. So when we're done with the single components, we can then combine them to represent different screens of our app. Let's try this. In `index.js`, add the following:

```

storiesOf("App Screen", module)
  .addDecorator(CenterDecorator)
  .add("all components", () => ({
    template: `
      <div>
        <state :isworking="true"></state>
        <countdown minute="1" second="0" :isworking="true"></countdown>
        <controls state="stopped"></controls>
      </div>`
  }));

```

Here, we use the `CenterDecorator` to center all components. And voilà.



Testing Components

So we have all our components ready. Now it's time to test them. Testing is too wide a topic to be discussed here entirely. There are many different way to test components. Here, we'll narrow our exploration to three of them:

- **Manual testing** relies on developers manually exploring components and making sure they work and behave correctly.
- **Structural/Snapshot testing** captures the rendered markup of components. At the next run, the new snapshots are compared to the old ones to make sure they match. If they don't match, we have two choices. We can accept the differences if they're the result of intentional changes, and update the snapshots. Or we can fix them if they're the result of incorrect code.
- **Unit testing** verifies that, with a given input, we'll get the expected output.

If you're new to JavaScript testing, you might like to have a quick read of "How to Test React Components Using Jest" to get a handle on the general concepts.

Implementing Structural Testing with the Storyshots Add-on

The Storyshots add-on adds automated snapshot testing to our storybook. Let's see what that looks like.

First, install the add-on:

```
npm install --save-dev @storybook/addon-storyshots
```

Now, we create a `storyshots.spec.js` file in the `tests/unit` directory:

```
touch tests/unit/storyshots.spec.js
```

We then put the following in it:

```
import initStoryshots from '@storybook/addon-storyshots';

initStoryshots();
```

And that's it. Now we run the test:

```
npm run test:unit
```

And here's what we should see in the terminal:

```
test:unit Run unit tests with Jest vue-cli-service test:unit
Run task Parameters
Output
$ vue-cli-service test:unit
PASS tests/unit/storyshots.spec.js (12.256s)

Storyshots
  State
    ↓ state toggle (46ms)
  Countdown
    ↓ work state (20ms)
    ↓ rest state (7ms)
  Controls
    ↓ started state (10ms)
    ↓ paused state (6ms)
    ↓ stopped state (6ms)
  App Screen
    ↓ all components (13ms)

> 7 snapshots written.
Snapshot Summary
> 7 snapshots written from 1 test suite.

Test Suites: 1 passed, 1 total
Tests:       7 passed, 7 total
Snapshots:  7 written, 7 total
Time:        24.628s
Ran all test suites.
```

All snapshots will be placed in a `__snapshots__` directory, which is automatically created in the `tests/unit` directory.

Now, to demonstrate the snapshots diffing, let's change the `isworking` prop, in our very first story, to `false`, and re-run the tests. As we can see in the screenshot below, the change is caught by the add-on, and it tells us that we can update the snapshots by using the `-u` flag.


```
test:unit Run unit tests with Jest vue-cli-service test:unit
Run task Parameters
Output
$ vue-cli-service test:unit
FAIL tests/unit/storyshots.spec.js

Storyshots
State
  * state toggle (57ms)
Countdown
  ↓ work state (19ms)
  ↓ rest state (8ms)
Controls
  ↓ started state (10ms)
  ↓ paused state (6ms)
  ↓ stopped state (6ms)
App Screen
  ↓ all components (12ms)

* Storyshots › State › state toggle

expect(value).toMatchSnapshot()

Received value does not match stored snapshot "Storyshots State state toggle 1".

- Snapshot
+ Received

<h3>
- Work!
+ Rest!
</h3>

at match (node_modules/@storybook/addon-storyshots/dist/test-bodies.js:27:20)
at node_modules/@storybook/addon-storyshots/dist/test-bodies.js:39:10
at Object.<anonymous> (node_modules/@storybook/addon-storyshots/dist/api/snapshotsTestsTemplate.js:46:33)

> 1 snapshot failed.
Snapshot Summary
> 1 snapshot failed from 1 test suite. Inspect your code changes or re-run jest with '-u' to update them.

Test Suites: 1 failed, 1 total
Tests: 1 failed, 6 passed, 7 total
Snapshots: 1 failed, 6 passed, 7 total
Time: 6.739s, estimated 13s
Ran all test suites.

Total task duration: 8.99s
```

To make this easier, we create a `test:update` script in the `package.json`:

```
{
  "scripts": {
    ...
    "test:update": "vue-cli-service test:unit -u"
  }
}
```

Now, if we want to accept the changes, we can run this script and the snapshots will be updated to their new version.

Unit Testing

As I mentioned at the beginning of this guide, the project has the Jest plugin installed. So we also have the Vue Test Utils at our disposal. Given that, let's explore a practical example of unit testing.

In the `tests/unit` directory we create a `components.spec.js` file:

```
touch tests/unit/components.spec.js
```

And add the following content:

```
import { mount } from "@vue/test-utils";
import State from "../../src/components/State.vue";

describe("State", () => {
  it("has title Work!", () => {
    const wrapper = mount(State, {
      propsData: {
```

```

    isworking: true
  }
});
const title = wrapper.find("h3");
expect(title.text()).toBe("Work!");
});
it("has title Rest!", () => {
  const wrapper = mount(State, {
    propsData: {
      isworking: false
    }
  });
  const title = wrapper.find("h3");
  expect(title.text()).toBe("Rest!");
});
});

```

We use the `mount` method to create wrappers for the `State.vue` component with the two possible states of the prop `isworking`. Then we get the title where the prop is used and test whether it has the proper string, depending on the `isworking` state.

Now, let's run the tests again:

```
npm run test:unit
```

And everything should be correct.

Exporting Storybook as a Static App

After the hard work is done, we may want to show our components to others. Doing this is easy. We just need to export our storybook as a static app and deploy it. Let's do it.

First, we need to add a build script to `package.json`:

```

{
  "scripts": {
    ...
    "storybook-static": "build-storybook -c .storybook -o .storybook-static"
  }
}

```

Now, when we run the script, it will output a static Storybook in the `.storybook-static` directory.

Publishing the Storybook to GitHub

As the storybook is a static app, it's easy to publish it anywhere on the Web. In this final section, I'll demonstrate how to add the project to GitHub.

First, we create a new repo `vue-pomodoro-storybook`.

Now, we copy the origin URL of the repo and add it to our local Git project with this command:

```
git remote add origin https://github.com/<USERNAME>/vue-pomodoro-storybook.git
```

Finally, we push the repo to GitHub:

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

Now, let's deploy our static app to GitHub Pages.

We navigate to `.storybook-static` folder and run:

```

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

```

Then, we deploy to GitHub Pages with this command:

```
git push -f https://github.com/<USERNAME>/vue-pomodoro-storybook.git master:gh-pages
```

This will create a new branch `gh-pages` and will upload the static app to it.

And that's it.

Github

Don't forget, you can find the deployed storybook here.

Conclusion

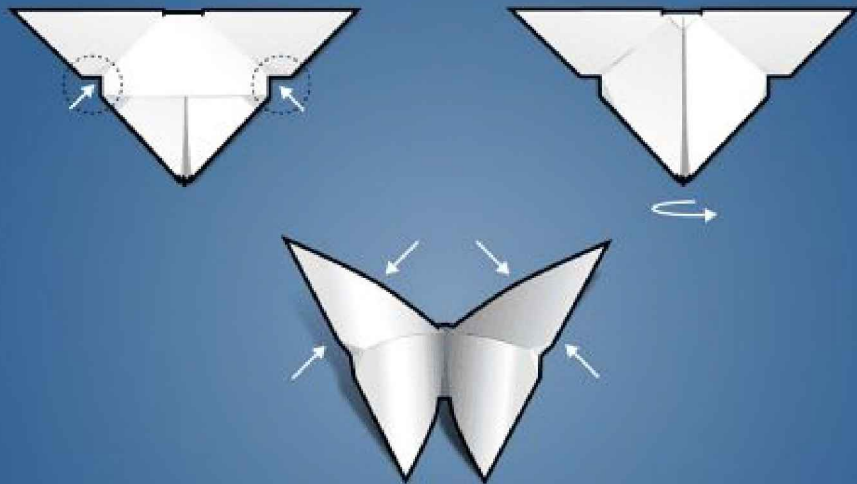
I hope you enjoyed learning Storybook. As you saw, it's a powerful playground, which we can use to interactively develop and test self-contained UI components. It's also great for making a live components showcase or documentation. I encourage you to check it out.

Book 3: 11 Vue.js: Tools & Skills



VUE.JS

TOOLS & SKILLS



BUILD YOUR OWN SOPHISTICATED WEB APPS

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.

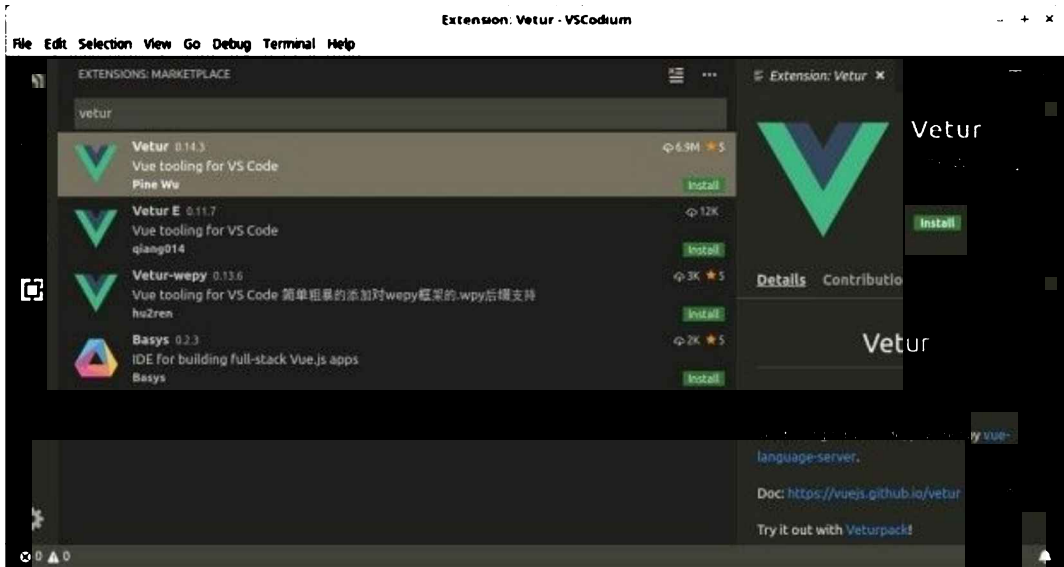
Throughout the rest of this guide, for the sake of consistency, I'll refer to the editor as VS Code.

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:

```
App.vue - VSCodium
File Edit Selection View Go Debug Terminal Help
App.vue x
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>
```

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

However, installing Vetur will make it look like so:

```
App.vue - VSCodium
File Edit Selection View Go Debug Terminal Help
App.vue x
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>
```

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>
```

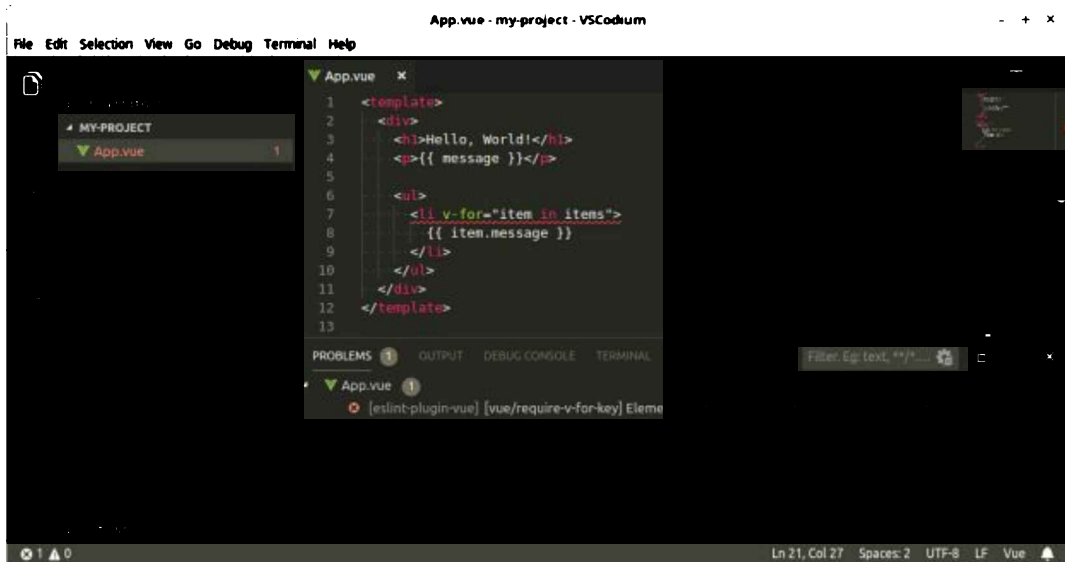
Error Checking/Linting

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


```
    </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:

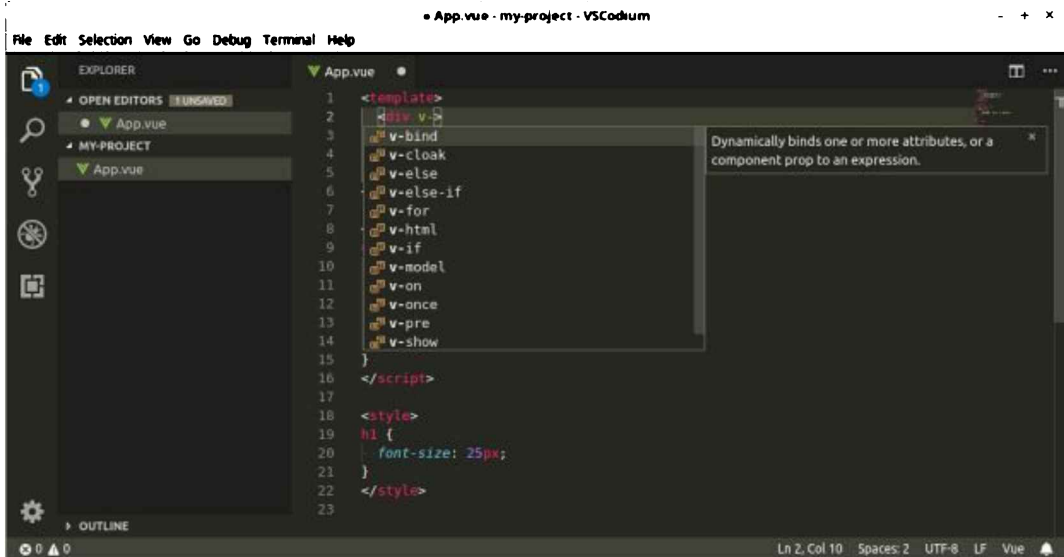
```
<li v-for="(item, i) in items" :key="i">
  {{ item.message }}
</li>
```

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.

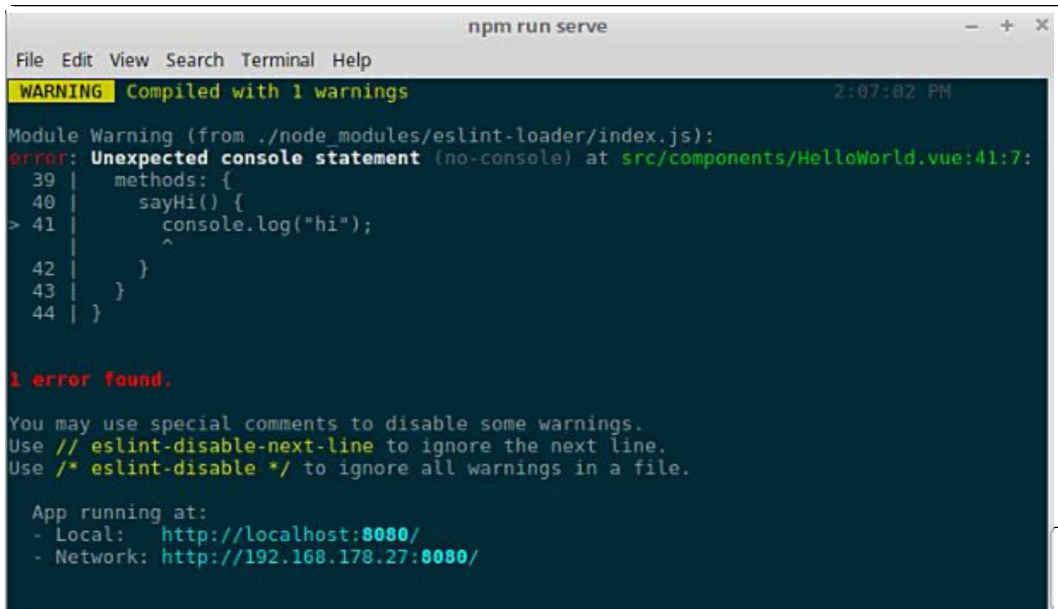
Let's put that to the test.

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.

A terminal window titled "npm run serve" with a menu bar (File, Edit, View, Search, Terminal, Help) and window controls. The terminal output shows a warning: "WARNING Compiled with 1 warnings" at 2:07:02 PM. Below this, it shows a module warning from ./node_modules/eslint-loader/index.js: "error: Unexpected console statement (no-console) at src/components/HelloWorld.vue:41:7". The code snippet shows a method sayHi() with a console.log("hi"); statement on line 41. The error message indicates that the no-console rule is violated. Below the code, it says "1 error found." and provides instructions on how to disable warnings using // eslint-disable-next-line or /* eslint-disable */. At the bottom, it shows the app running at http://localhost:8080/ and http://192.168.178.27:8080/.

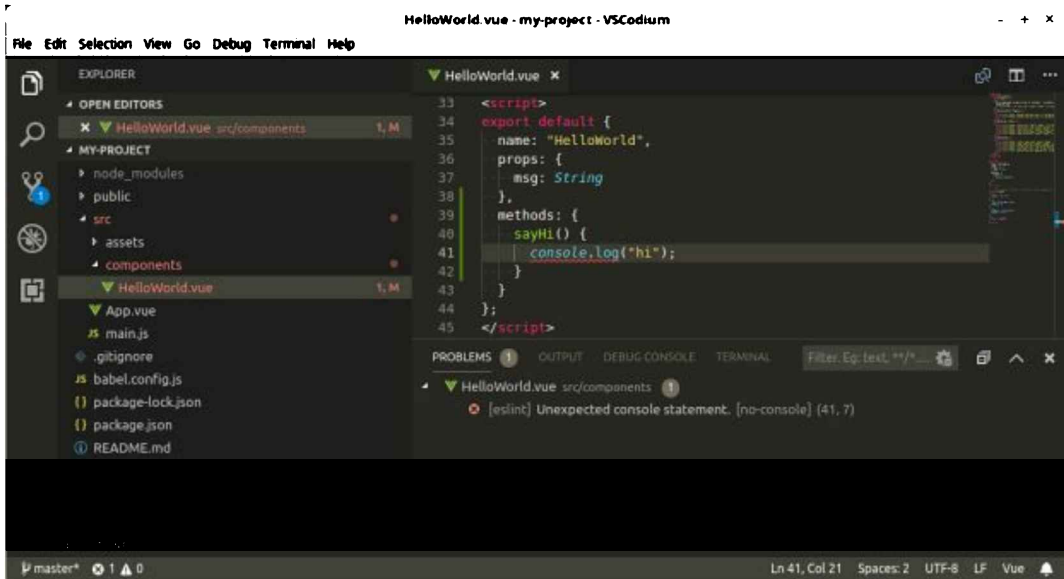
This is because, under the hood, Vue CLI has configured ESLint to use the `eslint:recommended` ruleset. This enables any rules marked with a check mark on the ESLint rules page, of which `no-console` is one.

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 ongoing 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/components/HelloWorld.vue:41:24:
 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.

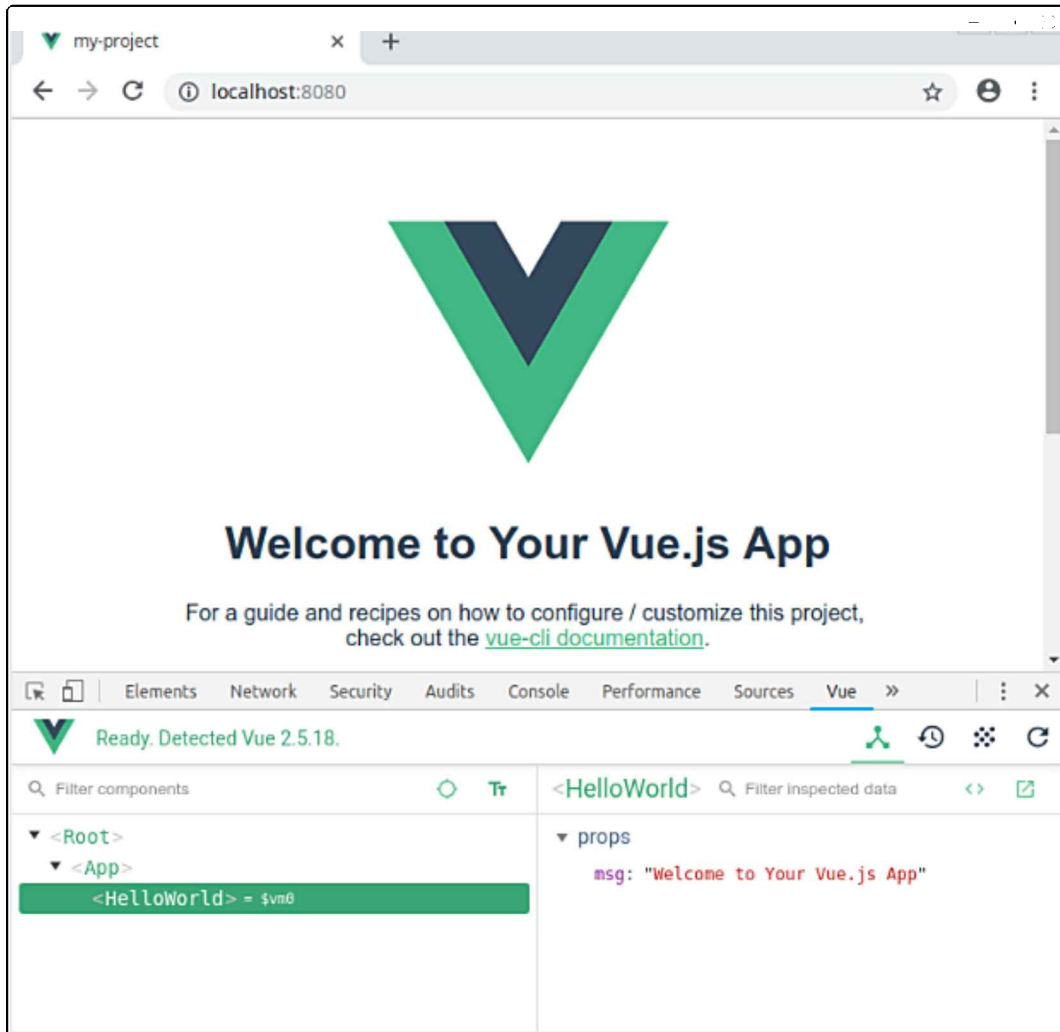
You can read more about configuration options [here](#).

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 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 `` tags around the list's `` 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-animations-js/2.2.2/web-animations.min.js"></script>

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

<!-- Set up a target to animate -->
<h1 id="hello">Hello world!</h1>

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

```

Here are a couple of things to notice:

- Since Animate.js 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.animate.animation(selector, options)`.

The options Animate.js 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.animate.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 seconds
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

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:

```
.
├── css
│   ├── lists.md
│   └── README.md
├── html
│   ├── lists.md
│   └── README.md
└── http.md
```

```

├── javascript
│   ├── functions.md
│   ├── objects.md
│   ├── README.md
│   ├── strings.md
│   └── variables.md
└── README.md

```

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:

```

.
├── components
│   ├── Footer.vue
│   ├── Hero.vue
│   ├── Message.vue
│   └── Navbar.vue
├── config.js
├── theme
│   ├── layouts
│   │   ├── Blog.vue
│   │   ├── Home.vue
│   │   └── Post.vue
│   ├── Layout.vue
│   └── styles
│       ├── code.styl
│       └── custom-blocks.styl

```

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,strings.md,variables.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,Message.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-blocks.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/0.7.1/css/bulma.min.css"
      }
    ],
    [
      "link",
      {
        rel: "stylesheet",
        href: "https://use.fontawesome.com/releases/v5.6.1/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.

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="main 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, and JavaScript</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: dismiss}">
    <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...</div>
  </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:

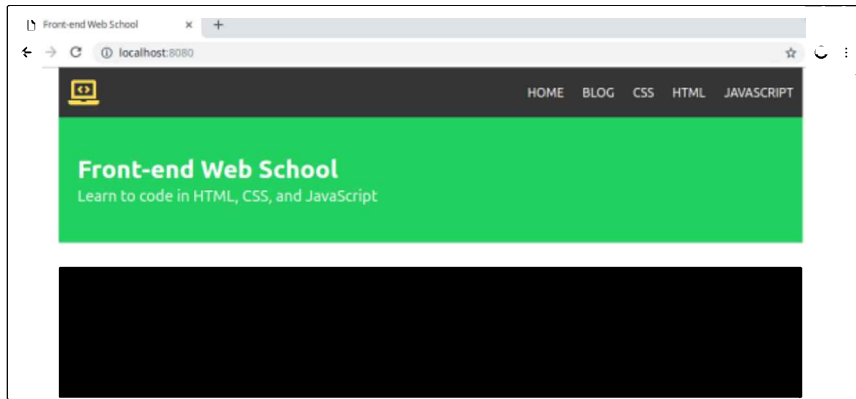
```

<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 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); currentCat = i; currentTag = null"
```

```
>
```

```
<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); currentTag = i; currentCat = null"
```

```
>
```

```
<span class="tag" :class="{ 'has-text-link': 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.path">
```

```
<router-link :to="post.path">
```

```
<div class>
```

```

        
      </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" class="tag">{{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 go java c sh yaml py;

@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
```

```
# git push -f git@github.com:<USERNAME>/<USERNAME>.github.io.git master

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

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 Mulders

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



See the Pen Ant Design 1 Timeline.

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.

Live Code

Upload ↕

See the Pen Ant Design 2.

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.

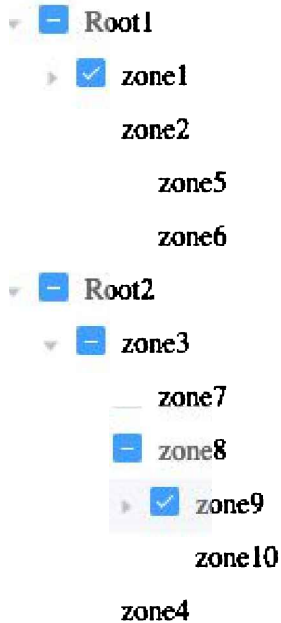
Code Snippets

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



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

Source

0/5

Target

0/1

California

Illinois

Maryland

Texas

Florida

Colorado

See the Pen Element - Transfer.

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



--



-

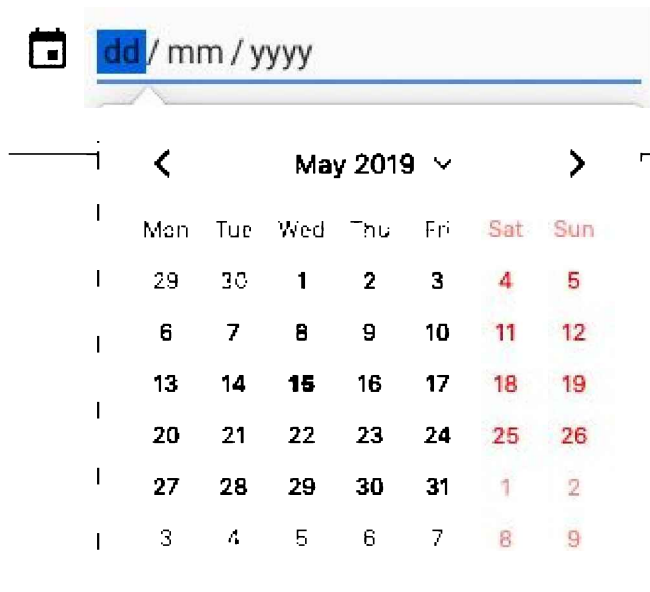
See the Pen [Vue Material - speed dial](#).

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



See the Pen [Vue Material - date picker disabled days](#).

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

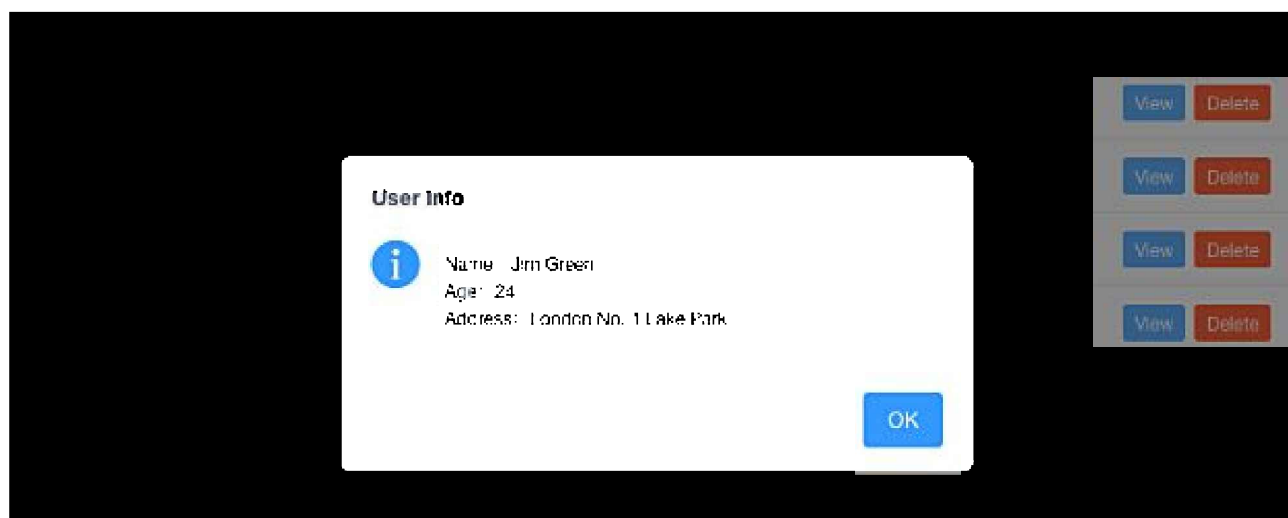
library to stand out.

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



See the Pen iView 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

| Name | Show   | The next day left  |
|-------|--|---|
| Name3 | 7181 | 377 |
| Name2 | 4720 | 1684 |
| Name1 | 7302 | 1727 |

 Export source data

 Export sorting and filtered data

 Export custom data

See the Pen [Vue.js iView Table Sort Filter CSV](#).

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

Z

Zack

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.

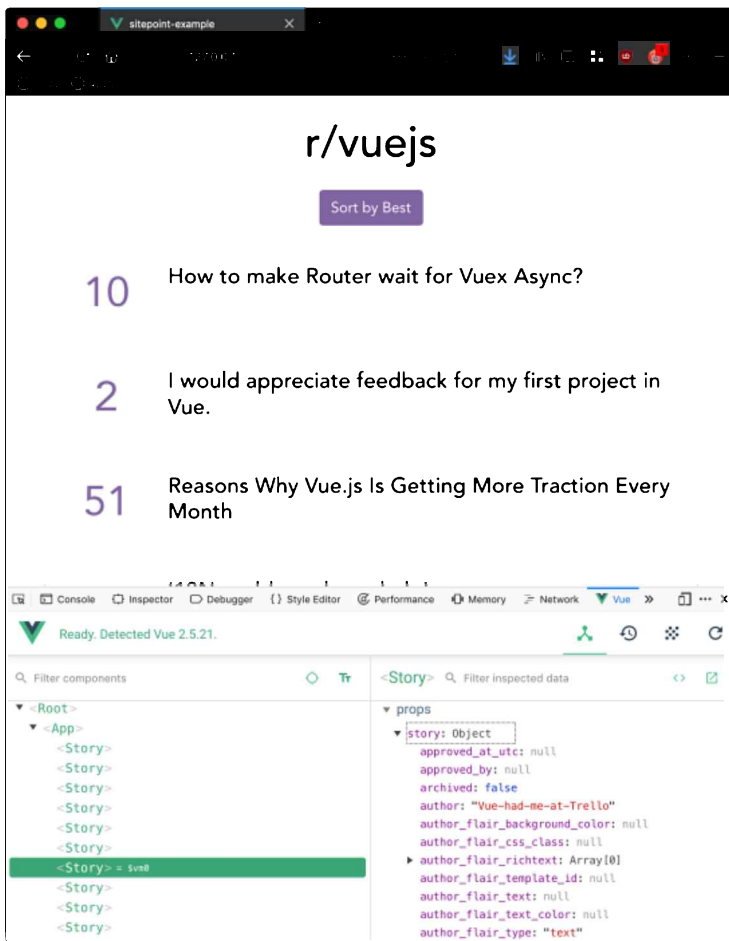
These are the things I wish I'd known when I was starting out.

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:



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.

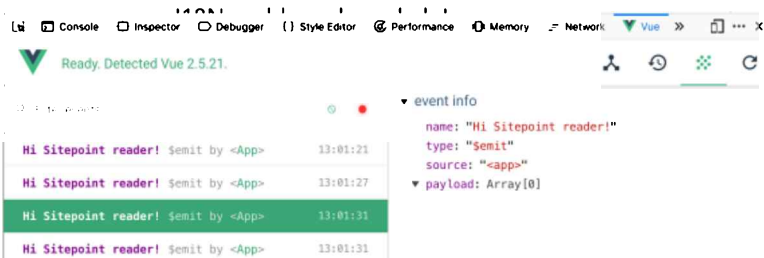
Additionally, component navigation is outstanding. Not only is there an *Inspect Vue component* option in the context menu, there's also a fuzzy search to filter your components by name, which is a very nice touch. You can also see the history of your Vuex state and "time travel" back to any given point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



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



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 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 Cypress or Nightwatch

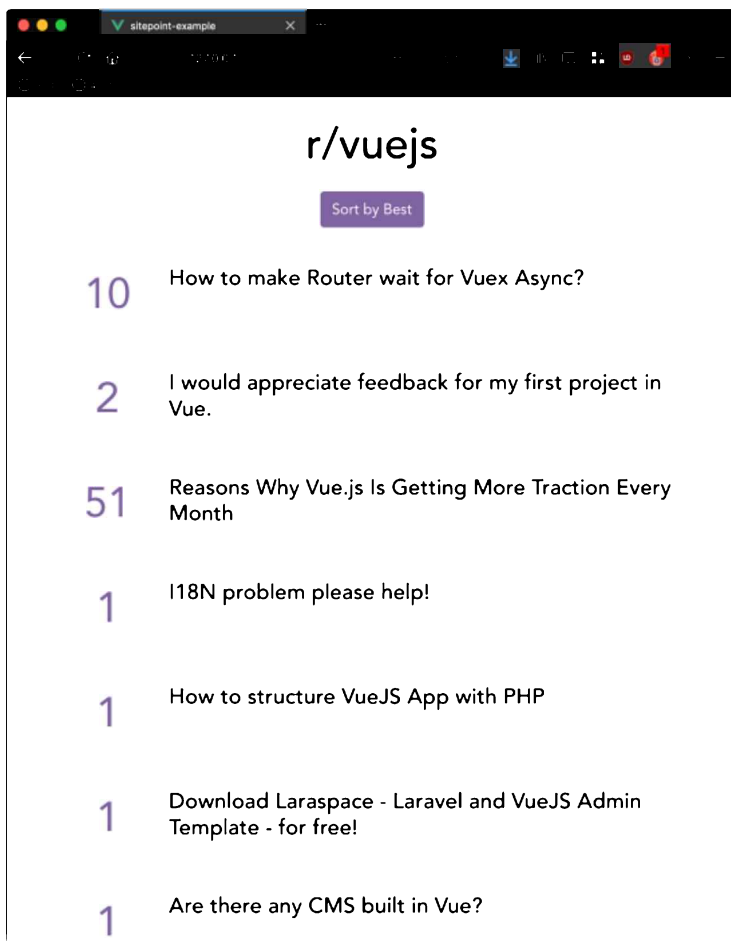
This setup even gives you the option to “Save this as a preset for future projects”, which is useful once you’ve developed a preferred stack. Check out “Getting Started with Vuex: a Beginner’s Guide” in this Vue series for more Vue CLI goodness.

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 btn-primary btn-sort">
      {{ sortHottest ? "Sort by Best" : "Sort by Hottest" }}</button>
    <div v-for="story in sortHottest ? sortByHottest : sortByBest" v-bind:key="story.data.id" >
      <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 it for an API call if you set up an API key):
      stories: stories.data.children,
      // Initial sort order of stories:
      sortHottest: true
    }
  },
  components: {
    Story
  },
  computed: {
    sortByBest: function () {
      // Sorts by score, descending. We use slice() to make a copy of the list to avoid mutating state.
      return this.stories.slice().sort((a, b) => b.data.score - a.data.score)
    },
    sortByHottest: function () {
      // The real Reddit 'Hottest' algorithm is a function of votes and submission-time over a delta.
      // We'll skip all that for this example and just return the list as we got it (which was already
      // sorted 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!