

The Delightful World of Vue.js



With <3 from SymfonyCasts

Chapter 1: Encore, Symfony & API Platform

Well hey friends! Welcome to the Delightful world of Vue.js. I know I say that *every* topic we cover at SymfonyCasts is fun - and I *totally* mean that - but this tutorial is going to be a *blast* as we build a *rich* and realistic JavaScript frontend for a store.

React vs Vue

These days, the two leaders in the frontend-framework world seem to be React and Vue.js. And just like with PHP frameworks, they're *fundamentally* the same: if you learn Vue.js, it'll be much easier to learn React or the other way around. If you're not sure which one to use, just pick one and run! React tends to feel a bit more like pure JavaScript while Vue has a bit more magic, which, honestly, can make it easier to learn if you're not a full-time JavaScript developer.

Vue 2 vs Vue 3

In this tutorial, we'll be using Vue version 2. But even as I'm saying this, Vue version 3 is nearing release and might already be out by the time you're watching this. But don't worry: there are actually very few differences between Vue 2 and 3 and whenever something *is* different, we'll highlight it in the video. So feel free to code along using Vue 2 or 3.

Project Setup

Oh, and speaking of that, to get best view of the Vue goodness - I had to get *one* Vue pun in - you *should* totally code along with me: you can download the course code from this page. After unzipping it, you'll find a `start/` directory with the same code that you see here. Follow the `README.md` file for all the fascinating details on how to get your project set up. The code *does* contain a Symfony app, but we'll spend most of our time in Vue.

One of the last steps in the README will be to find a terminal, move into the project and use Symfony's binary to start a local web server. You can download this at <https://symfony.com/download>. I'll say: `symfony serve -d` - the `-d` tells it to start in the background as a daemon - and then also `--allow-http` :



```
$ symfony serve -d --allow-http
```

This starts a new web server at localhost:8000 and we can go to it using `https` or `http` . We'll talk about why I used the `--allow-http` flag later.

Copy the URL, find your browser, paste it in the address bar and... say hello to a giant error!

Yarn & Webpack Encore Setup

Let's... back up. There are two things you need to know about our project. First, to help process JavaScript and CSS, we're using Webpack Encore: a simple tool to help configure Webpack. We have an entire [free tutorial](#) about it and you'll probably want to at *least* know the basics of Webpack or Encore before you keep going.


Our Encore config is pretty basic, with a single entry called `app` . It lives in the `assets/` directory - that's where all of our frontend files will live. The `app.js` file doesn't actually *have* any JavaScript, but it *does* load an `app.scss` file that holds some basic CSS for our site, including Bootstrap. Our base layout already has a `link` tag to the built `app.css` file... which exploded because we haven't executed Encore and *built* those assets yet.

Back at your terminal, start by installing Encore and our other Node dependencies by running:



```
$ yarn install
```

If you don't have `node` or `yarn` installed, head to <https://nodejs.org> and <https://yarnpkg.io> to get them. You can also use `npm` if you want. Once this is done populating our `node_modules/` directory, we can run Encore with:



```
$ yarn watch
```

This builds the assets into the `public/build` directory and then waits and watches for more changes: any time we modify a CSS or JS file, it will automatically re-build things. The `watch` command works thanks to a section in my `package.json` file: `watch` is a shortcut for `encore dev --watch`.

Ok! Let's try the site again - refresh! Welcome to MVP Office Supplies! Our newest lean startup idea here at SymfonyCasts. Ya see, most startups take a lot of shortcuts to create their first minimum viable product. We thought: why not take that *same* approach to office furniture and supplies? Yep, MVP Office Supplies is all about delivering low-quality - "kind of" functional - products to startups that want to *embody* the minimum-viable approach in all parts of their business.

Traditional Symfony App Mixed with Vue

Everything you see here is a traditional Symfony app: there is *no* JavaScript running on this page at all. The controller lives at `src/Controller/ProductController.php`: `index()` is the homepage and it renders a Twig template: `templates/product/index.html.twig`. Here's the text we're seeing.

The point is: right now, this is a good, traditional, boring server-side-generated page.

API Platform API

The second important thing about our app is that it already has a really nice API. You can see its docs if you go to <https://localhost:8000/api>. We built this with my *favorite* API tool: API Platform. We have several tutorials on SymfonyCasts about it.

Inside our app - let me close a few files - we have 6 entities, or database tables: `Product`, `Category` and a few others we won't worry about in this tutorial. Each of these has a series of API endpoints that we will call from Vue.

For example, back on the browser, scroll down to the `Product` section: we can use these interactive docs to *try* an endpoint: let's test that if you make a request to `/api/products`, that will return a JSON collection of products. Hit Execute and... there it is! This funny-looking JSON format is called JSON-LD, it's not important for Vue - it's basically JSON with extra metadata. Under the `hydra:member` property, we see the products: a useful Floppy disk and some blank CD's - all kinds of great things for a startup in the 21st century.

We'll be using this API throughout the tutorial.


Ok, click back to the homepage. Next, let's get Vue installed, bootstrap our first Vue instance and see what this puppy can do!

Chapter 2: Installing Vue, Webpack & Eslint

To use Vue, we, of course, need to install it in our app. And, because we're using modern JavaScript practices, we're not going to include a `script` tag to a CDN or manually download Vue. We're going to install it with yarn.

But first, in addition to downloading Vue into our project, we *also* need to *teach* Webpack how to parse `.vue` files. Like React, Vue uses some special - not-actually-JavaScript - syntaxes, which Webpack needs to transform into *real* JavaScript.

To tell Webpack to parse vue files, open `webpack.config.js`. Near the bottom, though it doesn't matter where, add `.enableVueLoader()`.

A screenshot of a code editor with a dark theme. The file is named 'webpack.config.js' and has 85 lines. The editor shows line numbers 9, 64, and 82. Line 9 contains 'Encore'. Line 64 contains '.enableVueLoader()'. Line 82 contains a semicolon ';'. Navigation arrows and line ranges are visible on the left side of the editor.

```
85 lines | webpack.config.js
9  Encore
64  .enableVueLoader()
82  ;
```

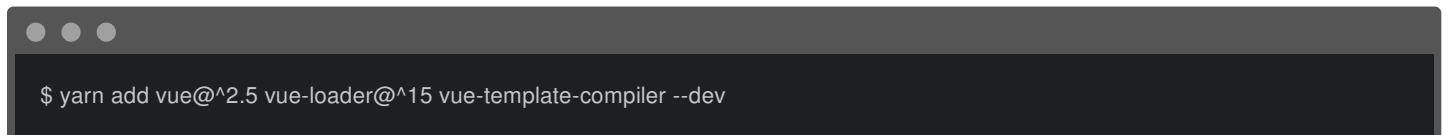
Yep! That's all you need. If you want to use Vue 3, you can pass an extra argument with a `version` key set to 3. Eventually, 3 will be the *default* version that Encore uses.

And, even though Encore is watching for changes, whenever you update `webpack.config.js`, you need to stop and restart Encore. I'll hit Control+C and then re-run:

A screenshot of a terminal window with a dark background. It shows the command '\$ yarn watch' being entered.

```
$ yarn watch
```

When we do this... *awesome!* Encore is screaming at us! To use Vue, we need to install a few packages. Copy the `yarn add` line, paste and run it:

A screenshot of a terminal window with a dark background. It shows the command '\$ yarn add vue@^2.5 vue-loader@^15 vue-template-compiler --dev' being entered.

```
$ yarn add vue@^2.5 vue-loader@^15 vue-template-compiler --dev
```

Once these are done downloading, restart Encore again with:

A screenshot of a terminal window with a dark background. It shows the command '\$ yarn watch' being entered.

```
$ yarn watch
```

It works! Nothing has really *changed* yet, but Encore is ready for Vue.

What is Vue?

In the simplest sense, Vue is a templating engine written in JavaScript. That *over-simplifies* it... but it's more or less true. In Symfony, if you go back to `ProductController`, we're accustomed to using Twig. It's easy: we tell it what template to render and we can pass variables *into* that template.

The Twig file itself is just HTML where we have access to a few Twig syntaxes, like `{{ variableName }}` to print something.

Vue works in much the same way: instead of Twig rendering a template with some variables, *Vue* will render a template with some variables. And the end-result will be the same: HTML. Of course, the one extra super power of Vue is that you can *change* the variables in JavaScript, and the template will automatically re-render.

Creating a Target Element for Vue

So instead of rendering this markup in Twig, delete all of it and just add `<div id="app">`. That `id` could be anything: we're creating an empty element that Vue will render *into*.

```
6 lines | templates/product/index.html.twig
... lines 1 - 2
3  {% block body %}
4      <div id="app"></div>
5  {% endblock %}
```

Our Non-Single Page Application

Now, what *we're* building will *not* be a single page application, and that's on purpose. Using Vue or React inside of a *traditional* web app is actually *trickier* than building a single page application. On our site, the homepage will soon contain a Vue app... but the layout - as you can see - is still rendered in Twig. We also have a login page which is rendered completely with Twig and a registration page that's the same. We'll purposely use Vue for part of our site, but not for everything... at least not in this tutorial.

Creating a Second Webpack Entry

Go back and open `webpack.config.js` again. This has one entry called `app`. The purpose of `app` is to hold any JavaScript or CSS that's used across our entire site, like to power the layout. We actually don't have any JavaScript, but the `app.scss` contains the CSS for the body, header and other things. The `app` script and link tags are included on *every* page.

But, our Vue app isn't going to be used on every page. So instead of adding our code to `app.js`, let's create a *second* entry and include it *only* on the pages that need our Vue app.

Copy the first `addEntry()` line, paste, and rename it to `products` - because the Vue app will eventually render an entire product section: listing products, viewing one product and even a cart and checkout in the next tutorial.

```
86 lines | webpack.config.js
... lines 1 - 8
9  Encore
... lines 10 - 26
27  .addEntry('products', './assets/js/products.js')
... lines 28 - 82
83  ;
... lines 84 - 86
```

Now, in `assets/js`, create that file: `products.js`. Let's start with something *exciting*: a `console.log()`:

```
Boring JavaScript file: make me cooler!
```

```
2 lines | assets/js/products.js
1  console.log('Boring JavaScript file: make me cooler!');
```

eslint

Oh, we will. But before we do, I'm going to open my PhpStorm settings and search for `ESLint`. Make sure "Automatic ESLint configuration" is selected. Because... I've already added a `.eslintrc` config file to the app. ESLint enforces JavaScript coding standards and PhpStorm can automatically read this and highlight our code when we mess something up. I *love* it! We're using a few basic rule sets including one specifically for Vue. You definitely don't need to use this exact setup, but I *do* recommend having this file.

Back in `products.js`, ha! Now PhpStorm is highlighting `console`:

```
Unexpected console statement (no-console)
```

One of our rules says that we should *not* use `console` because that's debugging code. Of course, we *are* debugging right now, so it's safe to ignore.

Ok: we added a new entry and created the new file. The last step is to include the `script` tag on our page. Open up `templates/product/index.html.twig`. Here, override a block called `javascripts`, call `parent()` and then I'll use an Encore function - `encore_entry_script_tags()` - to render all the script tags needed for the `products` entry.

```
18 lines | templates/product/index.html.twig
... lines 1 - 12
13 {% block javascripts %}
14     {{ parent() }}
15
16     {{ encore_entry_script_tags('products') }}
17 {% endblock %}
```

If you look in the base template - `base.html.twig` - it's quite traditional: we have a `block stylesheets` on top and a block `javascripts` at the bottom.

Back in our template, also override the `stylesheets` block and call `encore_entry_link_tags`. Eventually, we'll start using CSS in our `products` entry. When we do, this will render the link tags to the CSS files that Encore outputs.

```
18 lines | templates/product/index.html.twig
... lines 1 - 6
7 {% block stylesheets %}
8     {{ parent() }}
9
10     {{ encore_entry_link_tags('products') }}
11 {% endblock %}
... lines 12 - 18
```

Before we try this - because we just updated the `webpack.config.js` file - we need to restart Encore *one* more time:

```
$ yarn watch
```

When that finishes, move back over, refresh... then open your browser's debug tools. Got it! Our boring JavaScript file is alive!

Our First Vue Instance

Let's... make it cooler with Vue! Back in `products.js`, start by importing Vue: `import vue from 'vue'`. This is one of the *few* parts that will look different in Vue 3 - but the ideas are the same.

```
7 lines | assets/js/products.js
1 import Vue from 'vue';
... lines 2 - 7
```

If you imagine that Vue is a templating engine - like Twig - then all we should need to do is pass Vue some template code to render. And... that's *exactly* what we're going to do. Add `const app = new Vue()` and pass this some options. The first is `el` set to `#app`. That tells Vue to render inside of the `id="app"` element. Then, pass one more option: `template`. This is the HTML template - just like a Twig template - except that, for now, we're going to literally add the HTML right here, instead of in a separate file:

```
<h1>Hello Vue! Is this cooler?</h1>
```

```
7 lines | assets/js/products.js
... lines 1 - 2
3 const app = new Vue({
4   el: '#app',
5   template: '<h1>Hello Vue! Is this cooler?</h1>',
6 });
```

That's... all we need! Moment of truth: find your browser and refresh. There it is! We just built our first Vue app in about 5 lines of

code.

Next, let's make it more interesting by passing *variables* to the template and witnessing Vue's awesomeness first hand.

Chapter 3: Vue Instance & Dynamic Data

We've just seen the most basic thing you can do with Vue. And if you think of Vue as a templating engine like Twig, it makes a lot of sense: we instantiated a new Vue instance, told it *where* on the page to render and passed it a template. And that *totally* worked. Booya!

The data Option

When you instantiate Vue, you control it by passing a number of different *options*... and a lot of this tutorial will be about learning what options are possible. One of the *most* important ones is `data`. Unlike `el` and `template`, `data` is a *function*. It returns an array - or, really, this is an "object" in JavaScript - of variables that you want to pass into the template.

Notice that ESLint is *angry* - it's because this line is empty. Sometimes you need to ignore it until you finish: it's a bit overeager. Let's create one new "data" - one new "variable" - to pass into the template: `firstName` set to `Ryan`. That's me!

```
14 lines | assets/js/products.js
... lines 1 - 2
3  const app = new Vue({
... line 4
5    data: function() {
6      return {
7        firstName: 'Ryan',
8      };
9    },
... line 10
11  });
... lines 12 - 14
```

And now that we're passing a `firstName` variable into the template, we can say "Hello" and `{{ firstName }}`. Yes, by *complete* coincidence, Vue uses the same syntax as Twig to render things.

```
14 lines | assets/js/products.js
... lines 1 - 2
3  const app = new Vue({
... lines 4 - 9
10  template: '<h1>Hello {{ firstName }}! Is this cooler?</h1>',
11  });
... lines 12 - 14
```

Before we try this, ESLint is *still* mad at me. Sheesh! It says:

Expected method shorthand

As I mentioned, some of the options you pass to Vue are set to values - like `el` and `template`, while others are *functions*. When you have a method in an object like this, you can use a shorthand: `data() { ... }` which is just a lot more attractive.

```
14 lines | assets/js/products.js
... lines 1 - 2
3  const app = new Vue({
... line 4
5    data() {
... lines 6 - 8
9    },
... line 10
11  });
... lines 12 - 14
```


Oh, and at the bottom, temporarily add `window.app = app`. That will set our Vue `app` as a global variable, which will let us play with it in our console. Ready?

```
14 lines | assets/js/products.js
... lines 1 - 12
13 window.app = app;
```

Refresh! It rendered! But it gets better! In your browser's console, type `app.firstName`. You can *already* see that this equals Ryan! Any `data` key becomes accessible as a property on our instance. Set this to `Beckett` - my son's name... who is hopefully napping right now. Boom! The template *immediately* updates for the new data. And we can change this over and over again.

So Vue is a lot like Twig - it renders templates and can pass variables into the templates - but with this crazy-cool extra power that when we *change* a piece of data, it automatically re-renders... which, of course, is exactly what we want.

How the Vue Instance Rendering Really Works

And... at a high level... that's Vue! Yes, we're going to talk about *so* much more, but you already understand its *main* purpose. Remove the global variable and the `app =` code - we don't need that. ESLint will temporarily get mad because it thinks it's weird that we're instantiating an object and not setting to a variable, but that's fine... and it'll go away in a little while.

```
12 lines | assets/js/products.js
... lines 1 - 2
3 new Vue({
... lines 4 - 10
11 });
```

Behind the scenes, when Vue renders, it actually calls a `render()` method on the object, which you don't normally need to worry or care about. But to help this all make more sense, I want you to *see* what this method *looks* like one time. Stick with me, we're going to do some temporary experimentation.

I'm going to add a new `render()` function. As soon as I add this, when Vue renders it will call *our* function instead of rendering it internally for us. But inside, I'm going to put the *exact* code that Vue normally runs: `return Vue.compile(this.$options.template)` - that's a special way to reference the `template` option here - `.render.call(this, h)`.

```
15 lines | assets/js/products.js
... lines 1 - 2
3 new Vue({
... lines 4 - 10
11 render(h) {
12   return Vue.compile(this.$options.template).render.call(this, h);
13 },
14 });
```

I know, that's *totally* crazy, and you will *never* need to type this in a real project. What this shows us is that Vue has a `compile()` function where you can "compile" a template string and then call `render()` on it. The `.render.call` thing is a fancy way of *basically* calling `.render()` and passing it the `h` variable, which is another object that's good at dealing with DOM elements... and that you don't need to worry about.

If we refresh now, it works *exactly* like before because our `render()` method does exactly what Vue normally does. If you look at this, you might start to wonder: why do we need a `template` option at all? We're just *reading* it in `render()` ... so it could live anywhere. Let's try that! Remove the option, and, at the top, add `const template =` the template string. In `render`, reference that local variable.

```
16 lines | assets/js/products.js
... lines 1 - 2
3  const template = '<h1>Hello {{ firstName }}! Is this cooler?</h1>';
... line 4
5  new Vue({
... lines 6 - 11
12  render(h) {
13    return Vue.compile(template).render.call(this, h);
14  },
15  });
```

That should work, right? Let's find out. It totally does!

Ok, so let me tell you the big important point I'm trying to make. Vue... is simple: it's a system where you can take a template string, render it, and pass this `data` into the template... just like Twig.

Of course, as we start adding more to our app, this `template` variable is going to get *huge* and ugly. To help with that, Vue has a very special organizational concept called single file components. Let's create one next.

Chapter 4: Single File Component

As we've seen, it's *totally* possible to configure the Vue instance and put the template in the same file. But... this is going to get *crazy* as our app grows: can you imagine writing 100 lines of HTML inside this string... or more? Yikes! Fortunately, Vue solves in a unique, and pretty cool way: with single file components.

Inside the `js/` directory create a new folder called `pages/`, and then a file called `products.vue`. We'll talk more about the directory structure we're creating along the way.

Notice that `.vue` extension: these files aren't really JavaScript, they're a custom format invented by Vue.

Creating the Single File Component

On top, add a `<template>` tag. Then, copy the `h1` HTML from the original file, delete the `template` variable, and paste here.

```
14 lines | assets/js/pages/products.vue
1  <template>
2    <h1>Hello {{ firstName }}! Is this cooler?</h1>
3  </template>
... lines 4 - 14
```

Next, add a `<script>` tag. Anything in here *is* JavaScript and we'll `export default` an object that will hold our Vue options. Copy the `data()` function, delete it, and move it here.

```
14 lines | assets/js/pages/products.vue
... lines 1 - 4
5  <script>
6    export default {
7      data() {
8        return {
9          firstName: 'Ryan',
10        };
11      },
12    };
13  </script>
```

That's it! I know, the format is a bit strange, but it's *super* nice to work with. On top, the `<template>` section allows us to write HTML just like if we were in a Twig template. And below, the `<script>` tag allows us to set up our data, as well as any of the other options that we'll learn about. This is a fully-functional Vue component.

Using the Single File Component

Back in `products.js`, to use this, first, import it: `import App` - we could call that variable anything - `from './pages/products'`. Thanks to Encore, we don't need to include the `.vue` extension.

```
10 lines | assets/js/products.js
... line 1
2  import App from './pages/products';
... lines 3 - 10
```

Now, inside of `render`, instead of worrying about compiling the template and all this boring, crazy-looking code, the `App` variable already has everything we need. Render it with `return h(App)`.

10 lines | assets/js/products.js

```
... lines 1 - 3
4  new Vue({
5    el: '#app',
6    render(h) {
7      return h(App);
8    },
9  });
```

That feels good! Let's try it: move over, refresh and... it still works!

Adding a Component Name

From here on out, we're going to do pretty much *all* our work inside of these `.vue` files - called single file components. One option that we're going to add to *every* component is `name`: set it to `Products`. We could use *any* name here: the purpose of this option is to help debugging: if we have an error, Vue will tell us that it came from the `Products` component. So, always include it, but it doesn't change how our app works.

15 lines | assets/js/pages/products.vue

```
... lines 1 - 5
6  export default {
7    name: 'Products',
8  };
13 };
... lines 14 - 15
```

`$mount()` the Component

Before we keep working, there are two small changes I want to make to `products.js`. First, the `el` option: it tells Vue that it should render into the `id="app"` element on the page. This works, but you *usually* see this done in a different way. Remove `el` and, after the `Vue` object is created, call `$.mount()` and pass it `#app`.

9 lines | assets/js/products.js

```
... lines 1 - 3
4  new Vue({
5    render(h) {
6      return h(App);
7    },
8  }).$mount('#app');
```

I also like this better: we first create this `Vue` object - which is a template and set of data that's ready to go - and *then* choose where to mount it on the page.

Shorthand `render()` Method

Second, because the `render()` method only contains a `return` line, we can shorten it: `render` set to `h => h(App)`.

7 lines | assets/js/products.js

```
... lines 1 - 3
4  new Vue({
5    render: (h) => h(App),
6  }).$mount('#app');
```

That's effectively the same: it uses the arrow function to say that `render` is a function that accepts an `h` argument and will return `h(App)`. I'm *mostly* making this change because this is how you'll see Vue apps instantiated on the web.

Next, let's get to work inside our single file component: we'll add the HTML markup needed for our product list page and then learn how we can add styles.

Chapter 5: CSS: Styling a Component

Our main focus in this tutorial will be to build a rich product listing page inside of `products.vue`. To get that started, I'm going to replace the `h1` with some new markup - you can copy this from the code block on this page.

```
68 lines | assets/js/pages/products.vue
1  <template>
2    <div class="container-fluid">
3      <div class="row">
4      ... lines 4 - 53
54    </div>
55  </div>
56 </template>
57 ... lines 57 - 68
```

Notice that there is nothing special yet. We're not rendering any variables: this is 100% static HTML. If you refresh the page, ok! We have a sidebar on the left, and an area on the right where we will *eventually* list some products. Good start!

Global CSS and Vue Components

And, though it's not pretty, it *does* already have *some* styling. If you look back at the HTML I just pasted, the basic styling is thanks to some Bootstrap classes that are on the elements. This works because, in the `assets/scss/app.scss` file, we're importing Bootstrap and I've decided to include the built `app.css` file on every page. So, naturally, if we use any Bootstrap classes in Vue, those elements get styled.

Custom Component style Section

But I also want to add some extra styling that's *specific* to the sidebar. The question is: where should that CSS live? We could, of course, add some classes to `app.scss` and use those inside our Vue component.

But Vue gives us a better option: because we want to style an element that's created in this *component*, Vue allows us to also put the *styles* in the component.

First, inside the `aside` element, give this `<div>` a new class called `sidebar`.

```
76 lines | assets/js/pages/products.vue
1  <template>
2    <div class="container-fluid">
3      <div class="row">
4        <aside class="col-xs-12 col-3">
5          <div class="sidebar p-3 mb-5">
6          ... lines 6 - 29
30        </div>
31      </aside>
32      ... lines 32 - 53
54    </div>
55  </div>
56 </template>
57 ... lines 57 - 76
```

Next, at the bottom - though it doesn't matter where - there is *third* special section that any `.vue` file can have: you can have a `<template>` tag, a `<script>` tag and also a `<style>` tag. Inside, we're writing CSS: add `.sidebar` and let's give it a border, a `box-shadow` and a `border-radius`.

76 lines | assets/js/pages/products.vue

... lines 1 - 68

```
69 <style>
70 .sidebar {
71   border: 1px solid #efefee;
72   box-shadow: 0px 0px 7px 4px #efefee;
73   border-radius: 5px;
74 }
75 </style>
```

Styling done! Remember: we're still running `yarn watch` in the background, so Webpack is constantly re-dumping the built JavaScript and CSS files as we're working. Thanks to that, without doing *anything* else, we can refresh and... the sidebar is styled!

This works thanks to some Vue and Webpack teamwork. When Webpack sees the `style` tag, it grabs that CSS and puts it into the entry file's CSS file, so `products.css`.

View the page source: here's the `link` tag to `/build/products.css`. Whenever we add any styles to any component that this entry uses, Webpack will find those and put them in here. Like a lot of things with Webpack, it's not something you really need to think about: it just works.

Using Sass Styles

So this is awesome... but I *do* like using Sass. If you look at `webpack.config.js`, down here... yep! I've already added Sass support to Encore with `.enableSassLoader()`.

Open up `assets/scss/components/light-component.scss`. This is a Sass mixin that holds the exact CSS I just added manually. If we could use Sass code inside the `style` tag, we could *import* this and save ourselves some duplication!

And that's *totally* possible: just add `lang="scss"`.

76 lines | assets/js/pages/products.vue

... lines 1 - 68

```
69 <style lang="scss">
... lines 70 - 74
75 </style>
```

Now that we're writing Sass we can `@import '../..scss/components/light-component'` and inside `.sidebar`, `@include light-component;`.

76 lines | assets/js/pages/products.vue

... lines 1 - 68

```
69 <style lang="scss">
70 @import '../..scss/components/light-component';
71
72 .sidebar {
73   @include light-component;
74 }
75 </style>
```

Let's try it! Back over on your browser, refresh and... we have Sass!

To finish off the styling, I'll use Sass's nesting syntax to add a hover state on the links. Now after we refresh... got it!

82 lines | assets/js/pages/products.vue

... lines 1 - 71

72 .sidebar {

... lines 73 - 74

75 ul {

76 li a:hover {

77 background: \$blue-component-link-hover;

78 }

79 }

80 }

... lines 81 - 82

Being able to put your styles *right* inside the component is one of my *favorite* features of Vue. And in a bit, we're going to do something even *fancier* with modular CSS.

Next, let's add some dynamic data back to our app and play with one of the *coolest* things in Vue: the developer tools.

Chapter 6: data() and Vue Dev Tools

The template I pasted in is 100% hardcoded. Boring!

See this little "shipping" message down here? Let's pretend that sometimes we need this message to change - like maybe if the user is on the page for longer than 60 seconds, we want our app to get desperate and change the shipping time to be faster.

The point is: we want this message to be dynamic, which means that it needs to be a `data`. Copy the message. Then, in `data()`, remove `firstName`: we're not using that anymore. Call the new data key `legend` and set it to the shipping message.

```
82 lines | assets/js/pages/products.vue
↑ ... lines 1 - 58
59 export default {
↑ ... line 60
61   data() {
62     return {
63       legend: 'Shipping takes 10-12 weeks, and products probably won't work',
64     };
65   },
66 };
↑ ... lines 67 - 82
```

Now that we have a `data` called `legend`, back up on the template, we're allowed to say `{{ legend }}`.

```
82 lines | assets/js/pages/products.vue
↑ ... lines 1 - 48
49     <span class="p-3">
50       {{ legend }}
51     </span>
↑ ... lines 52 - 82
```

Beautiful! And if we move over to our browser and refresh... it even works!

Vue Dev Tools

The *first* time we played with data, we set our entire Vue application onto a global variable so we could change the `firstName` data from our browser's console and watch the HTML update. Being able to see and play with the data on your components is a *pretty* handy way to debug. And fortunately, there's a much easier way to do this than manually setting a global variable.

It's called the "Vue.js Dev Tools": a browser extension for Chrome or Firefox that gives you tons of Vue information. If you don't already have it, install it - it's amazing.

Once you have the dev tools... and once you're on a page that contains a Vue app running in dev mode, you can open your browser's debugging tools - I actually need to close and re-open mine so that it sees my Vue app - and... boom! New Vue tab. Click it!

I *love* this. On the left, it shows the component "hierarchy" of my app. In a few minutes, we're going to create *more* components and start nesting them inside of each other, just like HTML. If you click on `<Products>`, ah: on the right, you can see the `data` for this component. *And* we can change its value! Add quotes and replace this with whatever message is in your heart. When you hit the save icon... the HTML updates! Vue calls this "reactivity": the idea that Vue *watches* your data for changes and re-renders a component when necessary.

Anyways, the Vue dev tools will be a *powerful* way to visualize our app, see its data and even change data to see how things update.

The `data()` Function versus `data: () => { }` Function

Before we create our *second* component, I need to point out a small detail. We configure Vue by passing it options. Some of

these options are set directly to values, while others - like `data()` - are functions.

And because the `data()` function is always just a `return` statement, you'll often see it written with the shortcut, arrow syntax: `data: () => {`, arrow, `{`, remove the `return` and fix the ending.

```
80 lines | assets/js/pages/products.vue
... lines 1 - 60
61   data: () => ({
62     legend: 'Shipping takes 10-13 weeks, and products probably won't work',
63   }),
... lines 64 - 80
```

Just like with the `render` shortcut we used in `products.js`, this is *effectively* the same: it says that the `data` property is set to a function that returns this object. The return is implied.

You can use this if you want... but I'm going to go back to the original way. It's a bit longer, but I'll use this syntax consistently for *all* Vue options that are set to functions. The shorter syntax also has a limitation where you can't use the `this` variable: something that we *will* need later.

Next: let's extract part of our product listing markup into a *second* component and include it from `Products`. This will start to show off one of Vue's key concepts: having multiple components that communicate to each other.

Chapter 7: Creating a Child Component

The `products.vue` file is known as a component. And, as we can see, a component represents a set of HTML, complete with CSS for that HTML and even *behavior* for those elements, which we'll learn about soon.

In the same way that *HTML* elements can be placed inside of each other, Vue components can *also* be nested inside of each other. For example, we could move an entire section of HTML into another `.vue` file and then *include* that component in the same spot.

And... this is often a *great* idea! Because, if we built our *entire* app in this one file, it would become *huge* and complex!

When & Why to Extract to a Component

We have this same problem in PHP: if you have a large or complex function, you might decide to extract part of it into a *different* function. You might do this because you want to re-use the extracted code *or* just because separating things keeps your code more readable and better organized.

As a general rule, if an area of your DOM has (1) special functionality, (2) needs to be reused *or* (3) would just make the original template easier to read, you should consider extracting it into its own component.

Creating the Second Component

It's not *really* very complex, but let's pretend that we want to reuse the "legend" functionality in other places and pass different text each time we use it.

Let's go! Inside the `js/` directory, create a new sub-directory called `components/`. I'm not going to put this new component in `pages/`: `pages/` is sort of meant to hold "top-level" components, while `components/` will hold everything else: Vue components that contain little bits and pieces of HTML and that, in theory, could be re-used.

Inside `components/`, create a new file called `legend.vue`. Start the exact same way as before: add `<template>` and, inside, copy the `` from `products.vue` and paste. To keep things simple, I'm going to temporarily copy the old shipping message and hardcode it: the component will *start off* completely static.

```
12 lines | assets/js/components/legend.vue
1 <template>
2   <span class="p-3">
3     Shipping takes 10-12 weeks, and products probably won't work
4   </span>
5 </template>
... lines 6 - 12
```

The other tag we need is `<script>` with `export default {}`. The *only* option that we should *always* have is `name`. Set it to `Legend`.

```
12 lines | assets/js/components/legend.vue
... lines 1 - 6
7 <script>
8   export default {
9     name: 'Legend',
10  };
11 </script>
```

Rendering a Component inside a Template

Component... done! Using this in the `Products` component is a three step process. First: inside the `<script>` tag - just like we normally do in JavaScript - import it: `import LegendComponent...` - we could call that anything - `from '../components/legend'`.

85 lines | assets/js/pages/products.vue

... lines 1 - 55

56 <script>

57 import LegendComponent from '../components/legend';

... lines 58 - 69

70 </script>

... lines 71 - 85

Second, to make this *available* inside the template, add a new option called `components`. As I've mentioned, there are a number of options that you can add here to configure Vue, like `name`, `data()` and `components`. There aren't a *tons* of them - so don't worry - and we'll learn the most important ones little-by-little.

85 lines | assets/js/pages/products.vue

... lines 1 - 58

59 export default {

... line 60

61 components: {

62 LegendComponent,

63 },

... lines 64 - 68

69 };

... lines 70 - 85

If we stopped now, *nothing* would change: this makes `LegendComponent` *available* to our template, but we're not using it. In fact, that's why ESLint is so mad at me:

The "LegendComponent" has been registered but not used.

The last step is to go to our template, remove the old code, and *use* the `LegendComponent` as if it were an HTML tag. Type `<`. You might be expecting me to say `<LegendComponent/>` ... after all, PhpStorm *is* recommending that. Instead, use the kebab-case option: `<legend-component />`.

85 lines | assets/js/pages/products.vue

1 <template>

... lines 2 - 47

48 <div class="row">

49 <legend-component />

50 </div>

... lines 51 - 53

54 </template>

... lines 55 - 85

Using `LegendComponent` *would* have worked, but when we add a key to `components`, like `LegendComponent`, Vue *also* makes it available in its kebab-case version: so `legend-component`. It does that so our template can look cool by using the HTML standard of lowercase and dashes everywhere.

Anyways, let's try it! Move over, refresh and... it works! But the *real* prize is over on the Vue dev tools. Nice! Our component hierarchy is growing! We now have a `<Legend>` component *inside* of `<Products>`.

Of course... its text is now *static*... so we *also* kinda took a step backwards. Next, let's learn how to pass info from one component into another with `props`.

Chapter 8: Props: Passing Info into a Child Component

The text inside the `Legend` component is static. That won't work! Remember: our goal is to be able to re-use the `Legend` component from other places in our app and *pass* it the text that it should render. Somehow, we need to pass a value from the `Products` component down *into* the `Legend` component.

In PHP, if we created a function and needed some info to be passed *into* it, we would add an argument to the function. Simple! In Vue, components have a *similar* concept called "props".

Here's how it works. As soon as I need something to be passed *into* a component, that component needs a `props` option. Set this to an array with one key for each prop that an outside component should be able to pass. Call the one prop we need, how about, `title`.

```
13 lines | assets/js/components/legend.vue
... lines 1 - 6
7 <script>
8 export default {
... line 9
10   props: ['title'],
11 };
12 </script>
```

Thanks to this, any component that includes this component is now *allowed* to pass a `title` prop to it. We'll see what that looks like in a minute.

To use these, Vue makes all `props` available as variables in the template. Replace the hardcoded text with `{{ title }}`.

```
13 lines | assets/js/components/legend.vue
1 <template>
2   <span class="p-3">
3     {{ title }}
4   </span>
5 </template>
... lines 6 - 13
```

This component is now perfect: it says that it accepts a prop called `title` and then we use its value in the template. Lovely!

Back in `products.vue`, how can we *pass* that prop to `legend-component`? By adding an *attribute*: `title=""` and then whatever value you want, like `TODO PUT LEGEND HERE`.

```
85 lines | assets/js/pages/products.vue
1 <template>
... lines 2 - 47
48 <div class="row">
49   <legend-component title="TODO PUT LEGEND HERE" />
50 </div>
... lines 51 - 53
54 </template>
... lines 55 - 85
```

props are Read-Only

That should do it! When we refresh the page... it works! And the Vue dev tools are even *more* interesting! If you click on `<Products>`, you can still see the `legend` data, though, we're not using this anywhere at the moment. And when you click on `<Legend>`, nice! You can see its `props`!

We've just seen two of the *most* important things in Vue: `data` and `props`. `data` are values that will *change* while your app is

running, which is why the Vue dev tools gives us the little pencil icon to modify them. But `props` are different: when a component receives a `prop`, it *can't* change it: the Vue dev tools doesn't give us a cute pencil icon for a prop. Props are just a way for a component to *receive* data.

I like to think of "props to a component" like "arguments to a function" in PHP. Think about it: in PHP, if we add an argument to a function, its main purpose is to allow whoever calls us to *pass* us data. Once we receive an argument, we don't usually *change* the argument's value. I mean, we *could*, but the main purpose of an argument is to *read* data, not modify it. Props are *just* like that.

Items in `data` are almost more like variables that you create *inside* the function to hold some new dynamic data: you create them for your own purposes and can set and change them as much as you want. Of course, if you were to call *another* function that needed one of these variables, you would pass it *to* that function as an argument. The same is true in Vue: in a few minutes, we're going to pass the `legend` data in the `Products` component into the `Legend` component as a prop.

Props and Data are Available in Templates

Oh, and one other thing about `props` and `data`. In a template, all props are available as variables, which means we can say `{{ title }}`.

But you might remember that earlier, when we used the `legend` data in a template, we did the same thing: we said `{{ legend }}`. It turns out that Vue makes *both* `props` and `data` available as variables in your template. Which, again, is a lot like a function in PHP: you have access to any arguments that were passed to you *and* any local variables that you create.

Next: it's cool that we can pass a prop from `Products` into `<legend-component>`. But what we *really* want to do is pass this dynamic `legend` data as the `title` prop so that when that `legend` changes, the text updates. We'll do this with an important concept called `v-bind`?

Chapter 9: v-bind: Dynamic Attributes

We added a `title` prop to `legend` to make it dynamic: whenever we use this component, we can pass it different text. It's like an *argument* to the `legend` component.

But in this situation, I want to go one step further: I want to pretend that the legend text on this page needs to *change* while our app is running - like the shipping time magically starts to decrease if the user is on the site for awhile... and we *really* want them to buy.

Whenever we need a value to change while the app is running, that value needs to be stored as `data`. In `products.vue`, we already have a `legend` data from earlier, but we're temporarily not using it. So what we really want to do is this: pass the `legend` data as the `title` prop, instead of the hardcoded text.

Easy enough! We know that anything in `data` and `props` is available in our template. So, for the `title` attribute, or technically `prop`, say `title="{{ legend }}"`

```
85 lines | assets/js/pages/products.vue
1  <template>
  ... lines 2 - 47
48    <div class="row">
49      <legend-component title="{{ legend }}" />
50    </div>
  ... lines 51 - 53
54 </template>
  ... lines 55 - 85
```

As *soon* as we do that, Webpack is *mad*! Go check out the terminal. Yikes, it doesn't like this syntax at all - it can't even *build* our assets:

interpolation inside attributes has been removed. Use v-bind or the colon shorthand instead. For example, instead of `id="{{ val }}"`, use `:id="val"`.

That's... a pretty *awesome* error message.

Using v-bind for Dynamic Attributes

Basically, the `{{ }}` syntax that we've grown to know and love... can't be used inside an attribute. If you need a dynamic value in an attribute, you need to prefix the attribute with `v-bind:`. And then, inside the attribute, just use the variable name.

```
85 lines | assets/js/pages/products.vue
  ... lines 1 - 47
48    <div class="row">
49      <legend-component v-bind:title="legend" />
50    </div>
  ... lines 51 - 85
```

Now it builds successfully. Before we talk more about this, let's try it! Refresh and... it works! Over on the Vue dev tools, the `Products` component has a `legend` data... and it looks like the `Legend` component is *receiving* that as its `title` prop.

The *cool* thing is that if we modify the `legend` data - "Will ship slowly!" - the text updates! The modified data is passed to `Legend` as the `title` prop, and Vue re-renders it. So while *we* will *never* change a prop directly, if we change a data... and that data is passed to a prop, the prop *will* update to reflect that.

v-bind is Full JavaScript

Back at our editor, let's talk more about this `v-bind` thing. There will actually be *several* of these `v-` things in Vue: they're used whenever Vue needs to do something special, including if statements and for loops. `v-bind` is probably the most important one.

Very simply: if you want an attribute to be set to a dynamic value, you must prefix the attribute with `v-bind`.

As *soon* as you do that, the attribute is no longer just text, like `pb-2` up here. We're now writing *JavaScript* inside the attribute.

I'll prove it: add `+` open quote and say

```
this is really JavaScript.
```

```
85 lines | assets/js/pages/products.vue
... lines 1 - 47
48     <div class="row">
49       <legend-component v-bind:title="legend + ' this is really JavaScript!'" />
50     </div>
... lines 51 - 85
```

And... when we try this... yea! That text shows up! It's a mixture of our data and that string.

So... that's really it! `v-bind` is meant to "bind" an attribute to some dynamic value, often a data or prop. But honestly, I don't even think of it like that. I just think: if I want to use JavaScript inside of an attribute, I need to use `v-bind`.

The v-bind Colon Shorthand

We're going to use this *all* the time: we're *constantly* going to be setting attributes to dynamic values. Vue understand this. And so, they've provided us with a nice shortcut. Instead of `v-bind:title`, just say `:title`.

```
85 lines | assets/js/pages/products.vue
... lines 1 - 47
48     <div class="row">
49       <legend-component :title="legend" />
50     </div>
... lines 51 - 85
```

That means the *exact* same thing: it's still *really* `v-bind` behind the scenes.

I like this *way* more. In my mind, an attribute without a `colon` is literally set to that string. Prefixing the attribute with `:` transforms it into JavaScript. Simple.

Anyways, when we try it now it... of course, works brilliantly.

Specifying the type of a Prop

While we're talking about the `legend` component, I want to make one small tweak to the `props` option. Remember: to *allow* a `title` prop to be passed to us, we needed to first define the prop here.

If you hover over this, ESLint is mad:

```
prop title should define at least its type.
```

In addition to just saying:

```
please allow a prop called title to be passed to me
```

We can *also* tell Vue what *type* we expect this prop to be and whether or not it's *required*.

Change `props` to an object where `title` is a key. Set this to another object with two items: `type` set to `String` and `required` set to `true`. Valid types are things like `String`, `Number`, `Boolean`, `Array`, `Object` and a few others - all with upper case names.

18 lines | assets/js/components/legend.vue

```
↑ ... lines 1 - 6
7   <script>
8   export default {
↑ ... line 9
10    props: {
11      title: {
12        type: String,
13        required: true,
14      },
15    },
16  };
17 </script>
```

Oh, and ESLint is still mad because I messed up my indentation! Thanks!

This won't change how our app *works*: it just helps to document our code *and* Vue will give us some nice validation. Back in `products.vue`, temporarily "forget" to pass the `title` prop.

85 lines | assets/js/pages/products.vue

```
↑ ... lines 1 - 47
48     <div class="row">
49       <legend-component :title="legend" />
50     </div>
↑ ... lines 51 - 85
```

When we reload... error!

Missing required props: "title".

Remember: props are basically arguments you pass to a component. But since they're super dynamic, *this* is the way that we, sort of, type-hint each prop and mark it as required or not. There's also a `default` option you can pass for optional props.

Next: let's make our styles more hipster - *and* less likely to cause accidental side effects - by using *modular CSS*.

Chapter 10: Modular CSS

I love that I can put my styles right inside the component. In `products.vue`, we render an element with a `sidebar` class... and then immediately - without going anywhere else - we're able to add the CSS for that. We *can* still have external CSS files with *shared* CSS, but for any styling that's *specific* to a component, it can live right there.

CSS Name Conflicts

But... we need to be careful. The class `sidebar` is a pretty generic name. If we accidentally add a `sidebar` class to *any* other component - or even to an element in our Twig layout - this CSS will affect it!

Find your browser, refresh, and open the HTML source. On top, here's our stylesheet: `/build/products.css`. Open that up. Yep! That's what I expected: a `.sidebar` class with the CSS from the component. It's easy to see that these styles would affect *any* element on the page with a `sidebar` class... whether we want it to or not.

Hello Modular CSS

To solve this problem, Vue, well really, the CSS world - because this is a generic CSS problem - has created two solutions: scoped CSS and modular CSS. They're... two *slightly* different ways to solve the same problem and you can use either inside of Vue. We're going to use modular CSS.

What does that mean? It means that whenever we have a `style` tag in a Vue component, we're going to include a special attribute called `module`.

```
85 lines | assets/js/pages/products.vue
↑ ... lines 1 - 71
72 <style lang="scss" module>
↑ ... lines 73 - 83
84 </style>
```

That's it. Back at your browser, leave the CSS file open, but close the HTML source and refresh the homepage. Ah! We *lost* our sidebar styling! It's a modular CSS "feature", I promise!

To see what's going on, go back to the tab with the CSS file and refresh. Woh. The class names *changed*: they're now `.sidebar_` and then a random string. Back on the main tab, if we inspect element... the `div` *still* has the normal `sidebar` class.

Here's what's going on. When you add `module` to the `style` tag, Vue generates a random string that's specific to this *component* and adds that to the end of all class names. The *great* thing is that we can use generic class names like `sidebar` *without* worrying about affecting other parts of our page. Because... in the final CSS, the class name is *really* `sidebar_` then that random string.

Of course, now that this is happening, we can't just say `class="sidebar"` in the template anymore. We need to somehow use the *dynamic* name - the one that includes the random string.

Rendering the Dynamic CSS Class

As soon as you add `module` to a `style` tag, Vue makes a *new* variable available in your template called `$style`, which is a map from the original class name - like `sidebar` - to the new dynamic name - like `sidebar_abc123`.

I'm going to delete the `p-3 mb-5` classes temporarily... just to simplify.

Ok: we no longer want to set the `class` attribute to a simple string: it needs to be dynamic. And whenever an attribute needs to contain a dynamic value, we prefix it with `:`, which is *really* `v-bind` dressed up in its superhero costume.

Now, we're writing JavaScript. Use that new `$style` variable to say `$style.sidebar`.

85 lines | assets/js/pages/products.vue

```
1 <template>
  ... lines 2 - 4
5   <div :class="$style.sidebar">
  ... lines 6 - 53
54 </template>
  ... lines 55 - 85
```

Unfortunately, at this time, the Vue plugin in PhpStorm doesn't understand the `$style` variable, so it won't be much help here. But because we have a class called `sidebar` inside the `style` tag, we can say `$style.sidebar`.

Let's try it! Move over, refresh and... we're back! Our class renders with the dynamic name.

The Powerful class Attribute Syntaxes

Of course, it looks a *little* weird because our element is missing the two classes we removed. How can we add them back? We could do some ugly JavaScript, a plus, quote, space... but... come on! Vue almost *always* has a nicer way.

In fact, Vue has special treatment for the `class` attribute: instead of setting it to a string, you can *also* pass it an array. Now we can include `$style.sidebar` and the two static classes inside quotes: `p-3` and `mb-5`.

85 lines | assets/js/pages/products.vue

```
1 <template>
  ... lines 2 - 4
5   <div :class="[$style.sidebar, 'p-3', 'mb-5']">
  ... lines 6 - 53
54 </template>
  ... lines 55 - 85
```

That should do it! Back at the browser... much better.

So... that's modular CSS! We're going to use it throughout the project and we'll learn a *couple* more tricks along the way.

Controlling the "ident" (modular CSS Class Name)

You may have noticed that, when you look at the DOM, it's not super clear which component the `sidebar` class is coming from: that random string doesn't tell us much. That's not a *huge* deal, but with a little config, we can make this friendlier in dev mode.

Open up `webpack.config.js`. It doesn't matter where, but down here after `enableVueLoader()`, I'm going to paste in some code. You can get this from the code block on this page.

93 lines | webpack.config.js

```
  ... lines 1 - 8
9  Encore
  ... lines 10 - 66
67  // gives better module CSS naming in dev
68  .configureCssLoader((config) => {
69    if (!Encore.isProduction() && config.modules) {
70      config.modules.localIdentName = '[name]_[local]_[hash:base64:5]';
71    }
72  })
  ... lines 73 - 93
```

I admit, this *is* a bit low-level. Inside of Webpack, the `css-loader` is what's responsible for understanding and processing styles. This `localIdentName` is how the random string is generated when using modular CSS. This tells it to use the component name, then the class name - like `sidebar` - and *then* a random hash. And we're only doing this in `dev` mode because when we build for production, we don't care what our class names look like.



Tip

Actually, the `[name]` part will be the filename of the Vue component, not its name.

To make this take effect, at your terminal, hit Ctrl+C to stop Encore and then restart it:

A terminal window with a dark background and three light gray window control buttons (minimize, maximize, close) in the top-left corner. The terminal displays the command `$ yarn watch` in a light gray font.

Once that finishes, I'll move back to my browser. Refresh the CSS file first. Nice! The class name is `products_sidebar_` random string. And when we try the real page, it works too.

Next, oh, we're going to try something that I'm so excited about. It's called "hot module replacement"... which is a pretty cool-sounding name for something even cooler: the ability to make a change to our Vue code and see it in our browser without - wait for it - reloading the page. Woh.

Chapter 11: Webpack dev-server: Faster Updating

Before we keep going further with Vue, I want to show you a really fun feature of Webpack that Vue works *perfectly* with. It *does* require some setup, but I think you're going to love it!

Find your terminal. Right now, we're running `yarn watch`. Hit Control + C to stop it.

When you run `yarn watch` or `yarn dev` or `yarn build` - which builds your assets for production - Webpack reads the files in the `assets/` directory, processes them, and dumps *real* files into `public/build/`. If you look at the HTML source of the page, our `script` and `link` tags point to these physical files. So we have a build process, but it builds real files and our browser ultimately downloads static CSS and JS content.

This is great, but there's *another* way to run Webpack while developing. It's called the `dev-server`. It *should* be easy to use but, as you'll see, if you have a more complex setup or use Docker, it might require extra messing around.

Running the dev-server

Anyways, let's try this fancy `dev-server`. Instead of `yarn watch`, run, you guessed it, `yarn dino-server`:



```
$ yarn -server
```

Nope, that won't work. But dang! I wish we had added that to Encore! Missed opportunity. Run:



```
$ yarn dev-server
```

Just like with `yarn watch`, this `dev-server` command works thanks to a `scripts` section in our `package.json` file, which Encore gives you when you install it. `dev-server` is a shortcut for `encore dev-server`.

Back at the terminal, interesting. It says "Project running at <http://localhost:8080>" and "webpack output is served from <http://localhost:8080/build>".

When you run the `dev-server`, Webpack does *not* output physical files into the `public/build` directory. Well, there are two JSON files that Symfony needs, but no JavaScript, CSS or anything else. These do *not* exist.

Instead, if you want to access these, you need to go to <http://localhost:8080> - which hits a web server that the command just launched. Well, this "homepage" doesn't work - but we don't care. Try going to `/build/app.js`.

This is our built `app.js` JavaScript file! Here's the idea: instead of outputting physical files, Webpack makes everything available via this `localhost:8080` server. If we, on *our* site, can change our script and link tags to *point* to this server, it will load them.

And... I've got a surprise! When we refresh the homepage... woh! The site still works! Check the HTML source. Nice! All the `link` and `script` tags *automagically* point to `localhost:8080/`.

So, on a *high* level, this is cool, but not *that* interesting yet. Instead of Webpack creating physical files, it launches a web server that hosts them... and our Symfony app is smart enough to point all of our script and link tags at this server. But the end result is more or less the same as `yarn watch`: when we update some JavaScript or CSS code, we can immediately refresh the page to see the changes.

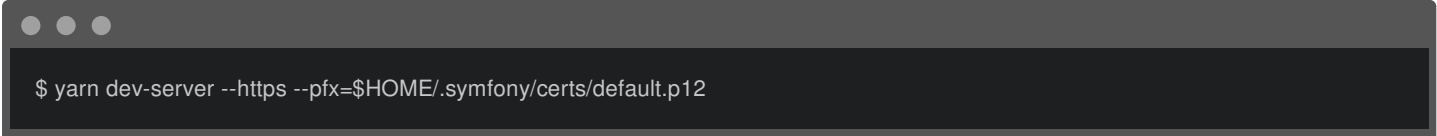
dev-server & HTTPS

But... the dev-server opens up some interesting possibilities. First, you might notice that a bunch of AJAX requests are failing on this page. It's some `sockjs-node` thing from that dev server. One of the super powers of the dev server is that it can automatically update the JavaScript and CSS in your browser *without* you needing to reload the page. To do that, it makes a

connection back to the dev-server to look for changes.

This is failing because it's trying to use https for the AJAX call and, unless you configure it, the dev-server only works for http. And it's trying to use https because *our* page is running on https.

So there are 2 ways to fix this. First, you could configure the Webpack dev server to allow https. If you're using the Symfony web server like we are, this is actually pretty easy - the `yarn dev-server` command just gets longer:



```
$ yarn dev-server --https --pfx=$HOME/.symfony/certs/default.p12
```

That tells the dev-server to allow https and to use the same SSL certificate as the Symfony dev server.

The other option, which is not quite as cool, but will *definitely* work, is to access your site with http so that this request *also* uses http.

When we originally started the Symfony web server, we started it with `symfony serve -d` and then `--allow-http`. This means that the web server supports https, but we're *allowed* to use http. Once we change to http in the URL... the AJAX call starts working!

If you have a more complex setup, like you need to change the host name or have CORS issues, check out the Encore dev-server docs or drop us a question in the comments. But ultimately, if the dev-server is giving you problems, don't use it! It's just a nice thing to have.

Automatic Reloading

Why? Head over to your terminal, open `products.vue` and... let's see... I'll make my favorite change: adding some exclamation points! I'll save then move back to my browser.

Nice! The page refreshed *for* me. If I remove those exclamation points and come back, it did it again! Ok, that's kinda cool: as soon as it detects a change, it automatically reloads.

But... we can get cooler! We can avoid the site from refreshing at *all* with one simple flag. Let's see that next.

Chapter 12: HMR: See Changes without Reloading

If you use Webpack's `dev-server`, then as soon as it detects a change, our page automatically reloads. Cool... but not cool enough! Find your terminal, hit Control + C to stop Encore, then re-run the dev server with a `--hot` option.

```
$ yarn dev-server --hot
```

That stands for hot module replacement. Once that finishes... I'll move over and reload the page one time just to be safe.

Let's do the same trick: add two exclamation points, then quick! Go back to the browser! Woh! It's there... but I didn't even see it reload! And no matter *how* fast you are, you'll see the update but you'll *never* see a page refresh: it's just not happening anymore! How cool is that? Webpack is able to *dynamically* update the JavaScript without reloading. It even keeps any Vue *data* the same.

This hot module replacement thing doesn't work with *all* JavaScript, but it *does* work well with Vue and React.

Disabling CSS Extraction

But... there's one *tiny* problem. Don't worry it's not a big deal. Back in our editor, at the bottom, let's make a CSS change: I'm pretty sure a designer just told me that the hover background should be pink.

This time, back on the browser, hmm: the style did *not* update. But if we refresh, it *is* there. HMR isn't working for styles.

This is easy to fix by disabling a feature in Encore. Let me show you. Open up `webpack.config.js` and go all the way to the bottom so we can use an `if` statement. Here, say if not `Encore.isProduction()` - a nice flag to see if we're building our assets for production or not - then `Encore.disableCssExtraction()`.

```
97 lines | webpack.config.js
... lines 1 - 91
92 if (!Encore.isProduction()) {
93   Encore.disableCssExtraction();
94 }
... lines 95 - 97
```

CSS extraction means that any CSS that Webpack finds should be *extracted* into an external CSS file. That's why a `products.css` file is being output. When you *disable* CSS extraction, it tells Webpack *not* to do this anymore. Instead, it embeds the CSS *into* the JavaScript. Then, when the JavaScript loads, it *adds* the CSS as `style` tags on the page.

I'll explain that a bit more in a minute... but let's see how our page changes first. Because we updated our Webpack config, at your terminal, stop Encore and restart it:

```
$ yarn dev-server --hot
```

Now, refresh the page. Did you see that? It looks the same, but it was unstyled for *just* a second. If you view the HTML source, there are *no* CSS link tags anymore. The CSS is being added by our JavaScript files. And because it takes a moment for those to load, our page looks ugly for an instant.

Two important things about this. First, disabling CSS extraction should *only* ever be done in dev mode: you *always* want *real* CSS files on production. And second, the *only* reason we're going to all the trouble of disabling CSS extraction at *all*, is because hot module replacement *only* works when it's disabled. Hopefully, someday, that won't be the case... and we won't need to do this.

The end result is pretty sweet though. Our hover links are still pink but now let's change them to green. And... yes! You could

see it change from pink to green *almost* instantly. If we remove the extra background entirely... that time it was faster than me!

I'm going to do the rest of the tutorial using the dev-server with hot module replacement because I love it! But it *did* require some work - like using `http://` or getting the ssl certificate setup and disabling CSS extraction.

Your situation may require even *more* work. If it does, consider using `yarn watch` instead. The `dev-server` is supposed to make your life easier. If it doesn't leave it behind.

Next, let's *really* start organizing our app - and making it more realistic - by splitting our code into several new components.

Chapter 13: Organizing into more Components

One of the huge benefits of Vue is being able to organize your DOM elements and their behavior into smaller pieces. Our app is still pretty simple but you can *already* see that our template is getting a bit crowded. Could we break this into sub-components?

Let's see: this really looks like two parts: a sidebar and a main area that will eventually list products - I'm going to call that the "catalog". Ok! Let's create two new components to keep things organized.

Creating the Catalog Component

Start with the catalog. Inside the `components/` directory - because this will technically be a component that we could re-use, create a new file called `catalog.vue`.

Then we *always* start the same - I *love* when things are boring! Add a `<template>` tag - with an empty `<div></div>` to start - and a `<script>` tag. The *minimum* we need here is `export default` an object. To help debugging, we always include at least a `name` key that identifies this component.

```
36 lines | assets/js/components/catalog.vue
1  <template>
  ... lines 2 - 20
21 </template>
  ... lines 22 - 25
26 export default {
27   name: 'Catalog',
  ... lines 28 - 33
34 };
35 </script>
```

Nice start team! Let's go grab the parts that we want to move here. Let's see: I want the Products title, the area where that will list products and the legend. Basically, I want to move this entire `col-xs-12` div. But... I won't *exactly* do that. It's up to you, but I think the `col-xs-12` belongs in *this* template because it defines the "layout" for `products` page. In theory, the catalog component could be embedded into *any* area on the site, so I won't put this element in there.

Instead, copy the 3 divs *inside* of this element. Back in `catalog.vue`, I'm going to keep the empty div. Why? One of the rules of Vue is that you *must* have a *single* outer element in each component. If we deleted the div, we would have *three* outer elements and... you won't be friends with Vue anymore! There *is* a way to fix this in Vue 3 - called Fragments - and you can even use a fragments plugin for Vue 2 if you really want to avoid this. But we'll keep the extra div.

36 lines | assets/js/components/catalog.vue

```
1 <template>
2   <div>
3     <div class="row">
4       <div class="col-12">
5         <h1>
6           Products
7         </h1>
8       </div>
9     </div>
10
11    <div class="row">
12      <div class="col-xs-12 col-6 mb-2 pb-2">
13        TODO - load some products!
14      </div>
15    </div>
16
17    <div class="row">
18      <legend-component :title="legend" />
19    </div>
20  </div>
21 </template>
22 ... lines 22 - 36
```

In the template, we're using `<legend-component />` so we need to import that just like we did before - `import LegendComponent from './legend'` - and add a `components` option with that inside.

36 lines | assets/js/components/catalog.vue

```
23 <script>
24   import LegendComponent from './legend';
25
26   export default {
27
28     components: {
29       LegendComponent,
30     },
31
32   };
33
34 </script>
```

Perfect: `LegendComponent` down here, allows us to use `legend-component` in the template.

The last thing we need is our data: this is the text that we pass to `LegendComponent`. Copy the `data()` function from `products.vue` and paste it here.

36 lines | assets/js/components/catalog.vue

```
26 export default {
27
31   data: () => ({
32     legend: 'Shipping takes 10-12 weeks, and products probably won't work',
33   }),
34 };
35 ... lines 35 - 36
```

Now we *could* have kept this `data` key in `products` and passed it into `catalog` as a prop. We're going to talk more later about *where* a piece of `data` should live and why. But don't worry about that yet.

Creating the Sidebar Component

Ok! I think this component is ready! Let's do the sidebar. Create a new file called `sidebar.vue` and start the same way: with a `<template>` tag. This time, let's immediately grab the elements we need. Like last time, I'm going to leave the `col-xs-12` here so that the parent component can determine the layout. Copy the `<div>` inside and paste it into `sidebar.vue`.

```
49 lines | assets/js/components/sidebar.vue
1  <template>
2    <div :class="[$style.sidebar, 'p-3', 'mb-5']">
3      <h5 class="text-center">
4        Categories
5      </h5>
6
7      <ul class="nav flex-column mb4">
8        <li class="nav-item">
9          <a
10            class="nav-link"
11            href="/"
12          >All Products</a>
13        </li>
14        <li class="nav-item">
15          <a
16            class="nav-link"
17            href="#"
18          >Category A</a>
19        </li>
20        <li class="nav-item">
21          <a
22            class="nav-link"
23            href="#"
24          >Category B</a>
25        </li>
26      </ul>
27    </div>
28  </template>
↑ ... lines 29 - 49
```

Perfect! Next, the `<script>` tag with `export default`, `name: 'Sidebar'` and... that's all the config this component needs!

```
49 lines | assets/js/components/sidebar.vue
↑ ... lines 1 - 29
30 <script>
31 export default {
32   name: 'Sidebar',
33 };
34 </script>
↑ ... lines 35 - 49
```

But the sidebar *does* need one more thing: some styles. In `products.vue`, all the way at the bottom, copy the *entire* `style` tag and put it into `sidebar.vue`.

```

49 lines | assets/js/components/sidebar.vue
... lines 1 - 35
36 <style lang="scss" module>
37 @import '../scss/components/light-component';
38
39 .sidebar {
40   @include light-component;
41
42   ul {
43     li a:hover {
44       background: $blue-component-link-hover;
45     }
46   }
47 }
48 </style>

```

Cleaning up the Parent Component

I think we're ready! In `products.vue`, this is going to be so satisfying. Remove the old import and replace it with `import Catalog from` and go up one level `../components/catalog`. Copy that and also import `Sidebar` from `sidebar`. Add both of these to the `components` option to make them available in the template.

```

27 lines | assets/js/pages/products.vue
... lines 1 - 14
15 <script>
16 import Catalog from '../components/catalog';
17 import Sidebar from '../components/sidebar';
18
19 export default {
20
21   components: {
22     Catalog,
23     Sidebar,
24   },
25 };
26 </script>

```

Ready to delete some code? Remove *all* the sidebar markup and instead say: `<sidebar />`. Do the same for the 3 catalog divs: just `<catalog />`.

```

27 lines | assets/js/pages/products.vue
1 <template>
2   <div class="container-fluid">
3     <div class="row">
4       <aside class="col-xs-12 col-3">
5         <sidebar />
6       </aside>
7
8       <div class="col-xs-12 col-9">
9         <catalog />
10      </div>
11    </div>
12  </div>
13 </template>
... lines 14 - 27

```

In the component itself, if you look at `data()`, the `legend` key is no longer used directly in this component... so we can delete the whole function. Also remove the `style` tag.

Wow. Look at that! Down to about 25 lines of code! Deciding *when* a component should be split into smaller components isn't a

science. In this case, none of the code we removed was *complex*, but splitting it allowed us to organize a lot of HTML and styles. I can *think* more clearly now.

Let's see if it works! Thanks to the `dev-server` that's running in hot mode, we don't even need to refresh: it *does* work. But... I'll refresh just to prove it.

Next: as our app grows, there will be more and more directories and paths to keep track of, like going to `../components` or `../scss`. That's not a huge deal, but we can add a shortcut to make life easier.

Chapter 14: Aliases

As our app grows, there's going to be more and more directories and paths to think about. In `products.vue`, we go *up* one directory to get to `components`. And in `sidebar`, we need to go up two directories to get to our CSS files. This isn't a *huge* deal, but it's only going to get worse as we add even more directories and sub-directories.

To help with this, the Vue world commonly uses a feature called Webpack aliases. Open up your `webpack.config.js` file. It doesn't matter where... but I'll go after `.enableSingleRuntimeChunk()`, add `.addAliases()` and pass an object. Add a key called `@` set to `path.resolve()`.

Oh, but stop right there: PhpStorm is mad! This `path` thing is a core Node module and we need to require it first. At the top: `var path = require('path');`.

The `path.resolve()` function is the *least* important part of this whole process: it's a fancy way in Node to create a path. Pass it `__dirname` - that's a Node variable that means "the directory of this file" - then `assets` and finally `js`.

```
104 lines | webpack.config.js
↑ ... line 1
2  var path = require('path');
↑ ... lines 3 - 9
10  Encore
↑ ... lines 11 - 38
39  // This is our alias to the root vue components dir
40  .addAliases({
41    '@': path.resolve(__dirname, 'assets', 'js'),
42    styles: path.resolve(__dirname, 'assets', 'scss'),
43  })
↑ ... lines 44 - 104
```

Before I explain this, duplicate the line and create one more alias called `styles` that points at the `scss` directory. And... I don't need those quotes around `styles`.

So... what the heck does this do? An alias is kind of like a fake directory. Thanks to this, when we import files in our code, we can prefix the path with `@` and Webpack will know that we're referring to the `assets/js` directory. We can do the same with `styles`: that's a shortcut to the `assets/scss` directory.

Let's see this in action. First, because we just made a change to our Webpack config, at your terminal, hit Control + C to stop Encore and then restart it:

```
$ yarn dev-server --hot
```

Once that finishes.. just to be safe, let's refresh. Everything still works. *Now* let's use our shiny new, optional alias shortcut!

Using the Alias

Start in `products.vue`. Instead of `../`, which gets us up to the `js/` directory, we can say `@/components/catalog` ... because `@` is an *alias* to the same directory.

```
27 lines | assets/js/pages/products.vue
↑ ... lines 1 - 14
15  <script>
16  import Catalog from '@/components/catalog';
17  import Sidebar from '@/components/sidebar';
↑ ... lines 18 - 25
26  </script>
```

The nice thing is that if we move our code to a different directory, this path will keep working: `@` always points to the `js/` folder.

We don't have to use this *everywhere*, but let's update a few other spots, like `catalog.vue`. Same thing: `@/components/legend`.

```
38 lines | assets/js/components/catalog.vue
... lines 1 - 22
23 <script>
24 import LegendComponent from '@/components/legend';
... lines 25 - 36
37 </script>
```

And then in `sidebar.vue`, it's a bit different. Down in the `style` tag, we can use the `styles` alias. But when you're inside CSS code and want to use an alias, you need *one* extra thing: a `~` prefix. So in this case, `~styles/components`.

```
49 lines | assets/js/components/sidebar.vue
... lines 1 - 35
36 <style lang="scss" module>
37 @import '~styles/components/light-component';
... lines 38 - 47
48 </style>
```

Oh, and I totally messed up! You can see a build error from Webpack. When I set up the `styles` alias, the path *should* be `scss`, not `css`.

```
104 lines | webpack.config.js
... lines 1 - 9
10 Encore
... lines 11 - 39
40 .addAliases({
... line 41
42   styles: path.resolve(__dirname, 'assets', 'scss'),
43 })
... lines 44 - 104
```

Over at the terminal, here's the fully angry error: file to import not found or unreadable... because we gave it a bad path. I'll stop and restart Encore one more time:

```
$ yarn dev-server --hot
```

Now... it's happy! Let's update two more files to get a feel for this. Open `products.js`. Instead of `./pages`, we can say `@/pages`.

```
7 lines | assets/js/products.js
... line 1
2 import App from '@/pages/products';
... lines 3 - 7
```

And one more in `app.js`. To load the CSS file, we can say `styles/` and then we don't the `scss` directory.

```
10 lines | assets/js/app.js
... lines 1 - 8
9 import 'styles/app.scss';
```

But... maybe you were expecting me to say `~styles` like we did earlier? Here's the deal: when you're inside of a JavaScript file, you can just use the word `styles` even if you *referring* to a CSS file. The `~` thing is only needed when you're doing the import from *inside* of CSS itself, like in the `style` tag.

So... those are Webpack aliases. If you love them, great! If you think they're some sort of strange sorcery, don't use them. The `@` alias *is* common in the Vue world. So at the *very* least, if you see Vue code importing `@/something`, now you'll understand the dark magic that makes this work.

Next: let's see our *second* custom Vue syntax: the `v-for` directive for looping.

Chapter 15: Looping with v-for

Let's start to make our app a bit more dynamic. See these categories on the sidebar? They are 100% hardcoded in the template. Boring!

Eventually, when the page loads, we'll make an AJAX request to dynamically load the *real* categories from our API. This means that the categories will be empty at first and then will *change* to be the *real* categories as soon as that AJAX call finishes. And so technically, the categories are something that will *change* during the lifetime of our Vue app. And anything that changes must live as a key on `data`.

Creating categories Data

Cool! Let's add our first piece of `data` to sidebar: `data()`, then `return` an object with `categories` set to an array. We're going to worry about the AJAX call in a few minutes. For now, let's set the initial data to some hardcoded categories. Each category can look however we want - how about an object with a `name` property set to our top-selling category - "Dot matrix printers" and a `link` property set to `#` for now.

Copy this and create a second category. Oh! But I can't misspell the cool "Dot Matrix" category! Shame on me! Set the second category to something just as modern: "Iomega Zip Drives". If you don't know what that is... you're definitely younger than I am.

```
63 lines | assets/js/components/sidebar.vue
... lines 1 - 30
31 export default {
... line 32
33   data() {
34     return {
35       categories: [
36         {
37           name: 'Dot Matrix Printers',
38           link: '#',
39         },
40         {
41           name: 'Iomega Zip Drives',
42           link: '#',
43         },
44       ],
45     };
46   },
47 };
48 </script>
... lines 49 - 63
```

Using v-for

Now that our component has a data called `categories`, we have a variable called `categories` in the template! But... we can't just render it with `{{ categories }}`. Nope, we need to loop *over* the categories.

Like Twig and its `{% for` or `{% if` tags, Vue has a number of custom syntaxes, which are called *directives*. We'll see the most important ones in this tutorial.

And while some of the Vue directives *work* a lot like Twig tags, they *look* a bit different. The directive for looping is called `v-for`, but instead of being a standalone tag, we put it right *on* the element that we want to loop.

Check it out: I'll split the `` onto multiple lines for readability, and then say `v-for="category in categories"`.


```

63 lines | assets/js/components/sidebar.vue
1  <template>
  ... lines 2 - 14
15  <li
16    v-for="category in categories"
17    class="nav-item"
18  >
  ... lines 19 - 24
25  </li>
  ... lines 26 - 27
28 </template>
  ... lines 29 - 63

```

That *totally* custom syntax will loop over the `categories` data and make a new variable called `category` available inside the `li`.

Now... life is easy! For the `href=""`, remember: each `category` is an object with `name` and `link` properties. So we basically want to say `category.href`. I mean, `category.link`! I'll catch that mistake in a minute!

But... this won't work yet because it would *literally* print that string. To make it *dynamic*, use `:href`.

```

63 lines | assets/js/components/sidebar.vue
  ... lines 1 - 14
15  <li
16    v-for="category in categories"
17    class="nav-item"
18  >
19    <a
20      :href="category.link"
21      class="nav-link"
22    >
  ... line 23
24    </a>
25  </li>
  ... lines 26 - 63

```

Now the contents of the attribute are JavaScript, and `category.link` is perfectly *valid* JavaScript.

Below, we can print the name with `{{ category.name }}`.

```

63 lines | assets/js/components/sidebar.vue
  ... lines 1 - 18
19  <a
  ... lines 20 - 21
22  >
23    {{ category.name }}
24  </a>
  ... lines 25 - 63

```

That's it! I'll remove the other hardcoded `li`.

Let's go check it out! When we move over, thanks to our dev-server in hot mode, we don't even need to refresh! The "All Products" is hardcoded, but the two categories below this are dynamic! Oh, but let me fix my mistake from earlier: use `category.link` because the data has a `link` property. That looks better.

The Purpose of the key Attribute

Back in my editor, ESLint is, once again, mad at me! And that's *usually* a good hint that I'm doing something silly! It says:

Elements in iteration expected to have `v-bind:key`.

That's a fancy way of saying that whenever you use `v-for` you're *supposed* to give that element a `key` attribute, which is any

unique identifier for each item in the loop.

Obviously... our code *works* without it. The *problem* comes if we *updated* the `categories` data. Without a `key` attribute, Vue has a hard time figuring out *which* category updated... and which element to re-render.

To help Vue out, after `v-for`, always add a `key=""`. Normally you'll set this to something like `category.id` - some unique key for each item. In this case, because we're looping over an array we invented, we can use the Array *index*.

To get access to the array index, change the `v-for` to `category, index`. Now say `key="index"`. But... *once* again... this isn't *quite* what we want: this would set the `key` attribute to the *string* `index`. To make it dynamic, change it to `:key="index"`.

```
64 lines | assets/js/components/sidebar.vue
... lines 1 - 14
15     <li
16       v-for="(category, index) in categories"
17       :key="index"
... line 18
19     >
... lines 20 - 25
26   </li>
... lines 27 - 64
```

Hey! Now we even get autocomplete!

Back on the browser, we won't see any difference, but if we updated the categories, Vue would re-render perfectly.

The Directives API Docs

We've now seen *two* Vue directives. Google for "Vue API" to find a page called [Vue.component - API](#). I *love* this page! It... well... talks about pretty much *everything* that exists in Vue.

On the left, I'm going to scroll past *tons* of things to a section called "Directives". Notice that there aren't *that* many directives in Vue - about 15 or so. And we've already seen both `v-bind` and `v-for`. We'll talk about the other important directives later. For each one, you can get info about how it works.

The custom key Attribute

Oh, and there's one *tiny* detail I want to mention before we keep going. Back on the site, inspect element on the `li`. It has `class="nav-item"`, but it does *not* have a `key` attribute?

There's some magic going on. Normally, if you add `foo="bar"` to an element, that element - of course - *will* have a `foo` attribute! We know that the *true* purpose of the `key` attribute was to help Vue internally... but shouldn't that also cause our element to have a `key` attribute?

Back on the docs, right below the Directives section is another one called "Special Attributes" and you see that `key` is one of them. Basically, if you add an attribute in Vue, it *will* render as an attribute... unless it's one of these *special* attributes. It's not really an important detail, but if you were wondering *why* everything *other* than `key` is rendered as an attribute, this is your answer. Vue is hiding it, because it knows it's internal.

Next: Vue is more than data and re-rendering: it's also about responding to user interaction. Let's add our first *behavior*: a link that will collapse and expand the sidebar.

Chapter 16: v-on & methods: User Interaction

We've talked a lot about data, props and using those in a template. But what we *haven't* done yet is add any *behavior*. So here's our next big goal: add a link to the bottom of the sidebar that, when the user clicks it, will collapse or expand the sidebar.

Adding the collapsed Data

Cool! But... where should we start? Let's think about it: in the sidebar template, we're going to need to know whether or not we are currently collapsed or expanded... because we will probably need to render different classes or styles to make that happen. And since this "is collapsed" information is something that will *change* while our Vue app is running, it needs to live in `data`.

Ok! Inside `data()`, add a new property called, how about, `collapsed`. Set it `false` so that the component starts *not* collapsed.

Back on the browser, it doesn't do anything yet, but we *can* see the new data.

Let's update the template to use this. Add a `style` attribute. Basically, if `collapsed` is true, we'll set a really small width. Of course we can't just start referencing the `collapsed` variable right now because we need to add the `:` in front of `style` to make it dynamic.

To set the width, we *could* put `width:` in quotes - I'm missing the quotes actually, then end quote, plus, and some dynamic logic that uses the `collapsed` variable. But... yuck! Because the `style` attribute can get complex, instead of creating a long string of styles, Vue allows us to set this to an object with a key for each style, like `width: '500px'` and `margin: '10px'`. In our case, I'll use the ternary syntax: if `collapsed` is true, set the width to `70px`, else use `auto`.

Ok! We have a `collapsed` data and we're using it in the template. Testing time! In the Vue dev tools, click on Sidebar, change `collapsed` to `true` and... oh that looks awful! We'll clean that up soon. But it *is* working: the width dynamically changes.

Adding the Button

Of course, we're not going to expect users to change the `collapsed` data via their Vue dev tools. Nope: they'll need a *button* that will change this state.

Back on the template, let's see... after the ``, I'll add an `<hr>`, a div with some positioning classes and a button with `class="btn btn-secondary"`.

Nothing interesting yet! For the button text, if we're currently collapsed, use `>>`, else, use `<< Collapse`.

Oh ESLint is mad! Hmm, it thinks that the `<` sign is me trying to open an HTML tag! I could escape this and use `<` - that's probably a great solution! But as you'll see, Vue will escape this *for* us.

Back on the browser, I'll inspect the new button, and right click to "Edit as HTML". Yep! Vue automatically escapes any text that you render. That `<` *becomes* `<` automatically.

Hello v-text

Oh, and, by the way, when you have an element and the *only* thing inside of it is some dynamic text - like our button - there is one other way to render that text. Copy our dynamic code, add a new attribute called `v-text`, set it to our dynamic code... and delete the curly braces.

Now Vue is happy. `v-text` is the *third* Vue directive that we've seen, after `v-bind` and `v-for`. It's not a particularly important one... it's just an alternative way to set the "text" of an element and... it's a *tiny* bit faster. I mostly just wanted you to see it.

Adding a new Method

Let's get back to the *real* goal: when the user clicks this button, we need to do something! Specifically, we need to change the `collapsed` data.

We know that the variables that we have access to in the template are the keys from `data` and `props`... though we don't have any props in this component. It turns out that there are a couple *other* ways to add stuff to your template. One allows you to add *functions*.

Check it out: it doesn't matter where... but I'll do it after `data()`, add a `methods:` option set to an object. Create a function called `toggleCollapsed()` and, to start, just say `console.log('CLICKED!')`.

The idea is that, "on click" of this button, we will tell Vue to call this method.

OnClick with v-on

How... do we do that? By using our *fourth* directive - and a very, very important one. It's called `v-on`. Inside the `button` element, add `v-on:` and then the name of the normal DOM event that you want to listen to, so: `click`. Set this to the name of the *method* that should be called on click: `toggleCollapsed`.

Yep, the `toggleCollapsed` function is *now* available in the template because we added it to the `methods` options. PhpStorm even lets you Ctrl or Cmd click to jump to it!

So... let's try it! Back on your browser, click and... then scroll down. These errors up here were from when we were in the middle of working. And... yes! There's our log!

The v-on Shortcut

`v-on` is one of the *most* important directives and we're going to use it *all* the time. Scroll up a little to the `:href` attribute. We know that this is *really* `v-bind` in disguise: `v-bind` is *such* a common directive, that Vue gives us this special shortcut.

Vue *also* has a shortcut for `v-on`: the `@` symbol. Want to run some code "on click", use `@click` or `@mouseover` or `@ whatever` event you want.

Updating the State

Ok: we have a button and when we click it, it calls the `toggleCollapsed` method. The *final* step is to *change* the `collapsed` state so that the template updates. How? `this.collapsed = !this.collapsed`.

If you were eagerly waiting for some complex & impressive code to change the data, I'm sorry to disappoint you.

The secret to this working is that any `data` keys are accessible as a *property* on `this`. In the next chapter, we'll look deeper into how that works. But for now, it feels good: we change the value of `this.collapsed` and... apparently, Vue will re-render.

Sound too good to be true? Let's try it! Click the button. It works! Over on the Vue dev tools, on the `Sidebar` component, we can watch the `collapsed` state change.

And... congratulations! We've now talked about the *core* parts of Vue. Seriously! To get this working, we added a `collapsed` data, used that in the template, and, on click, called a `toggleCollapsed` method that we created... which changed the `collapsed` data. Vue was then smart enough to re-render the template.

We'll see this basic flow over and over again.

But before we do, I want to talk more about how the `this` variable works in Vue. It's at the *heart* of Vue's magic. And if we understand it, we'll be just a *little* bit closer to being unstoppable. That's next.

Chapter 17: Magic "this" & its Properties

The best and worst part of Vue is its magic. We know that if you update a data key, Vue *somehow* knows that it needs to re-render. That's why we can play with data in the Vue dev tools and the HTML automatically updates.

We even know that if we pass that data as a prop to *another* component, Vue knows to re-render that one too! We can see this on the `Catalog` component: it has a `legend` data... there it is... which we pass to the `legend-component` as the `title` prop. When we update the `legend` data inside `Catalog`, it updates the `title` prop *and* re-renders that component.

console.log(this)

But... the magic I want to talk about is not this re-rendering stuff. I *love* that part! The *real* magic of Vue is the Vue instance, the `this` variable. Let's take a few minutes to dive into the `this` variable. I promise, it will go a *long* way to helping us *truly* master Vue.

Let's play around in the `Sidebar` component. Add a new option called `created`, which is a function. We're going to talk more about this later, but basically, if you add an option called `created` to any component, Vue will automatically call that function when your instance is being created. We're going to use this as an easy way to dig into the `this` variable: `console.log(this)`.

Let's go check it out! On my console... there it is! But I'll refresh anyways just to clear things out. Cool! It's a `VueComponent`: an object with... a *ton* of properties. If you're using Vue 3, instead of `VueComponent`, you'll see something called a Proxy. I'll talk about what a "Proxy" is in a few minutes. For now, if you're using Vue 3, click the "Target" property to see the real object.

The Properties of this

Ok, so `this` is an object with a bunch of properties... and the *vast* majority of these are things that you will never need to worry about. If a property starts with a `$`, you're *technically* allowed to use it, but you probably won't need to except in advanced situations. If a property starts with `_`, then it's meant to be internal and should *not* be used. In Vue 3, all of the internal keys live below a property called `_`.

Oh! But check this out: the object *also* has a property called `categories` and another called `collapsed`! Woh! Those are the keys we have in `data`! That explains why we can reference `this.collapsed`: the `this` variable really *does* have a `collapsed` property! And if you look back at the log, the instance *also* has a method called `toggleCollapsed`. This is here because, under the `methods` option, we added one with that name.

Before we talk more about this, let's try one more thing. Right now, sidebar doesn't have any `props`. Let's temporarily add one. Say `props:` and create one called `testProp` with `type: String`. One of the *other* things you can do here is give a prop a *default* value in case it's not passed to the component. Set this to a misspelled version of "I am the default value".

Perfecto-ish. Back on the browser, our code was already updated and... if we scroll down on the console... here's the latest log. Inside... yes! It now has a property called `testProp`!

How the "this" Object is Created

Here's the big picture. We configure Vue by passing it a set of options, like `name`, `data`, `created`, `props` and `methods`. Behind the scenes, Vue takes these options and creates a Vue object - the object that's in the console. When it does that, it takes all of the keys from `data`, all of the keys from `props` and all of our `methods` and *adds* those to the instance! This is why we can say `this.collapsed` to reference the `collapsed` data or `this.testProp` to reference that prop. Heck, we can even say `this.toggleCollapsed()`! Our `methods` become *real* methods on the object.

So the first thing I want you to understand is exactly this: Vue reads our options and uses them to create a Vue object where `data`, `props` and `methods` are *real* keys on that object. Later, we'll see one *more* option - called `computed` properties - that are added to the object in the same way.

Variables in the Template are called on this

Now that we understand this, we can demystify how the template works. In a template, we know that we can magically reference `categories` because `categories` is in data. Or we can magically reference `toggleCollapsed` because that's a key under `methods`.

But in reality, whenever you reference a variable or call a method in a template, Vue, sort of, prefixes it with `this.`. So the `@click` is really `this.toggleCollapsed` and when we're referencing `collapsed`, it's really `this.collapsed`.

Now you can't *actually* use the, `this` keyword in a template, but it *is* what's happening behind the scenes. And I can prove it!

Back on our browser... if you scroll down on the log, the object has a property called `_uid`. On Vue 3, it's `__uid`. This is an internal, unique identifier for the component that we normally don't care about. But *technically*, because the instance has a property called `_uid`, we should, in theory, be able to say `{{ _uid }}` to print it. If I'm telling the truth about Vue, this should call `this._uid` on the object.

And... it does! It prints "7"! Well, the value was 6 before, but when it re-rendered, `_uid` was 7.

Let's try something else: print `definitelyNotARealProperty`. Back on the browser... the error is *wonderful*:

```
Property or method definitelyNotARealProperty is not defined on the instance but was referenced during render.
```

Vue is literally saying: this is not a property on the instance! And the way you *add* something to the instance is by defining it under `data`, `props`, `methods` or the `computed` option that we'll talk about later. I love it!

Next, Vue has *one* other piece of magic I want to explore called reactivity. It's all about how Vue is smart enough to re-render a template at the *instant* that we change a simple piece of data.

Chapter 18: Reactivity

We now understand that we pass Vue a set of options and it takes all the keys in `data`, `props` and `methods` and adds those onto the Vue instance. We can see this in our console thanks to the `console.log(this)` that we added: this has `categories`, `collapsed` and `testProp` properties plus a `toggleCollapsed` method. Vue effectively *copies* these onto the instance so that we can say things like `this.collapsed`.

But there's still a mystery: when we *change* a data key - like `this.collapsed = !this.collapsed` - Vue instantly re-renders the template. How does that happen? Think about it: if `collapsed` is just a simple property, then how is Vue aware that we changed it?

Data & Props are Getter Properties

The answer is... magic! Sorcery! Potions! Ok... really it's just clever JavaScript. Look back at the `categories` and `collapsed` properties on the console. Notice that, instead of showing the value, it shows `(...)` and says "invoke property getter" when I hover. If you click that, *then* it shows the value.

It turns out that `categories`, `collapsed` and also `testProp` are *not* real properties on the object! Scandal! They're "getter" properties. If you scroll near the bottom of the log, you see `get categories` and `get collapsed`. What you're seeing is a special feature of JavaScript where you can make an object *look* like it has a property, even if it really doesn't. There is *no* property called `categories`, but we *are* allowed to reference `this.categories` thanks to this `get categories` function. When we say `this.categories` or `this.collapsed`, it calls this `proxyGetter`, which is some low-level Vue function. And when we *set* that property - like `this.collapsed = something`, it will call this `proxySetter` function.

The point is: Vue makes our data accessible, but *indirectly* via these getter and setter functions. It does that so that it can hook into our calls and re-render. When we say `this.collapsed =`, that calls `proxySetter`, which updates the data *and* tell Vue that it needs to re-render any affected components.

And... this is great! We get to run around just saying `this.collapsed =` not even realizing that Vue is intercepting that call and intelligently re-rendering.

Proxy Objects

By the way, in Vue 3, this magic is done by something called a Proxy. A Proxy is a native JavaScript object - it's a feature built into the language itself, not invented by Vue. With a Proxy, you can *wrap* another object and intercept all method calls, property gets and property sets. The result is the same... it's just a bit of a cleaner way to get the job done. By the way, this *whole* idea of hooking into us changing data and then automatically re-rendering any affected templates has a cool name: reactivity. Oooo.

Reactivity in Arrays

Reactivity gets even *more* amazing when you look at the `categories` data. Back in `created`, let's also log `this.categories` for simplicity. Back on the console... find that log - it's on the right - and expand it.

Check it out: it's a normal Array with 0 and 1 keys. So... if we dynamically added a *third* item to this array, would Vue know to re-render? Yep!

In Vue 3, instead of an array, this would be a Proxy *around* an array, which would let Vue hook into the new item being added. How is it done in Vue 2? Check out this `__proto__` key. This is a fancy place where we can see all the methods that exist on this `Array`, which is an object in JavaScript. So you can call `.pop`, `.push()` or any of these. But check it out: each is set to a function called `mutator()`. That is *not* a native JavaScript function: it comes from Vue!

Yea, Vue has *replaced* all of the `Array` functions with their *own* functions! If we `push()` a new item onto the array, it will call this `mutator()` function, which will push the item on the array *and* trigger any re-rendering. It's... pretty crazy. And actually, because of how this is done in Vue 2, there *are* a *few* edge case ways that you update an Array in a way that Vue *cannot* detect. The most common by far is if you set an item directly to a specific index. That's not a problem in Vue 3 thanks to the Proxy object.

Deep Reactivity

But wait, there's more! Check out one of the individual category objects. Remember, in `data`, each category has `name` and

`link` properties. Guess what? Once Vue loads the data, these are *not* real properties anymore: it says "invoke getter". Vue creates an object that *looks* like the data we created, but instead of having *real* `name` and `link` properties, it adds getters and setters for them - the same `reactiveGetter` and `reactiveSetter` functions we saw earlier. So if we changed the `name` property of the `0` index category, that will update that property *and* trigger any re-rendering needed.

Phew! So this is the *real* magic of Vue, and one of the things that sets it apart from React. In React, you handle data - or state - a bit more manually, but with less magic. It's always a tradeoff. But if you can understand how Vue's magic works on a high level, it will go a *long* way to helping you do great stuff.

Before we keep going - remove the `created` and `props` options. Next, we know that `props`, `data` and `methods` are copied onto the Vue instance and so, made available in the template. Let's talk about the fourth and last thing that is added to the instance: computed properties.

