

The Delightful World of Vue



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 window titled 'webpack.config.js' showing 85 lines. The code is in a dark theme. Line 9 has 'Encore'. Line 64 has '.enableVueLoader()'. Line 82 has a semicolon ';'. The editor shows line ranges like '... lines 1 - 8', '... lines 10 - 63', '... lines 65 - 81', and '... lines 83 - 85'.

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. The prompt is '\$ 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. The prompt is '\$ 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. The prompt is '\$ 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  });

```

For Vue 3, the code should look like this:

```

import { createApp } from 'vue';
import App from './pages/products';

createApp(App).mount('#app');

```

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  }
... lines 8 - 12
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.

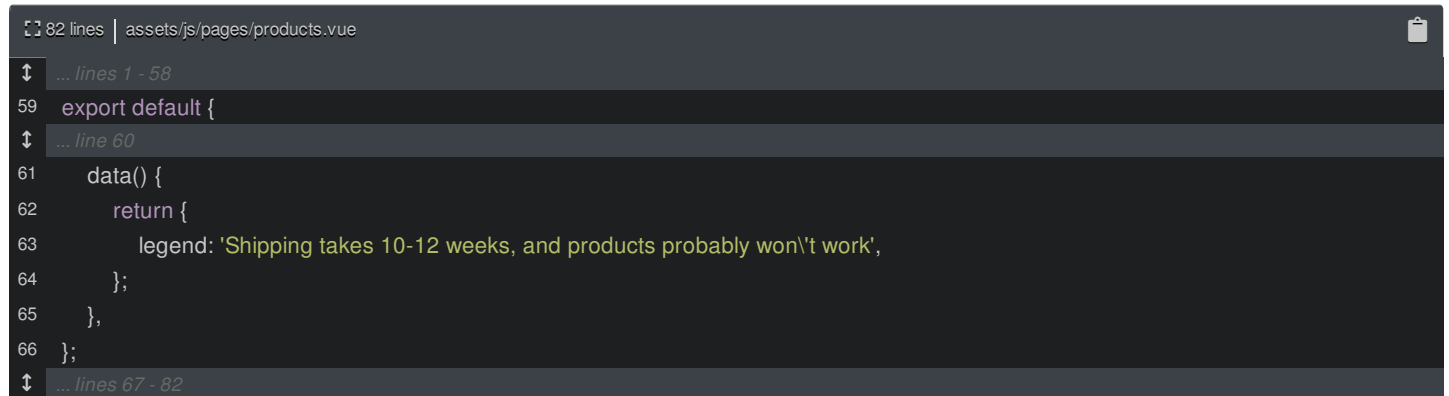
Next, let's add some dynamic data back to our app and play with one of the *coolest* things in Vue: the developer tools.

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 {
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.

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:



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`:



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



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.

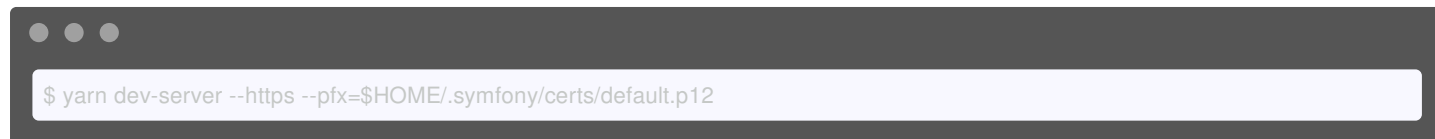
[dev-server & HTTPS](#)

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:

A terminal window with a dark background and three window control buttons (red, yellow, green) in the top-left corner. The command prompt shows the command: `$ 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.

If you're using Webpack Encore 1.0 or higher, the page will not automatically reload anymore. But that's ok: that's been disabled to show something even cooler, which we'll talk about in the next video.

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.

If you're using Webpack Encore 1.0 or higher, you do not need to pass the `--hot` flag: it's already, automatically, enabled.



```
$ 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.

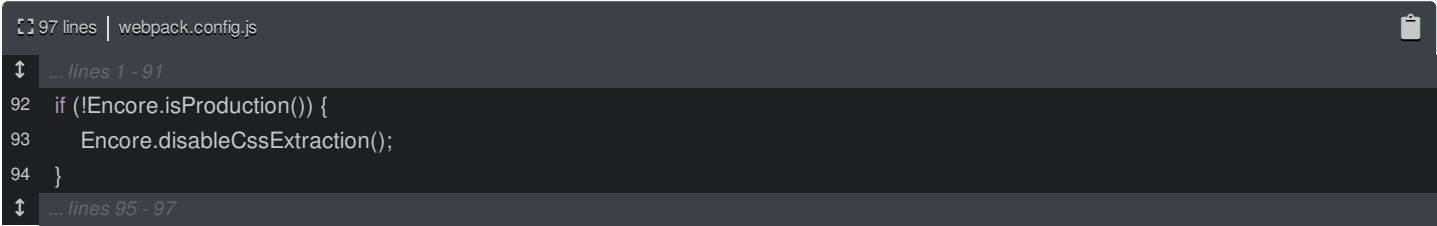
This hot module replacement thing doesn't work with *all* JavaScript, but it *does* work well with Vue and React.

[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()`.



```
92 if (!Encore.isProduction()) {  
93   Encore.disableCssExtraction();  
94 }
```

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>
... 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
... lines 1 - 22
23 <script>
24 import LegendComponent from './legend';
... line 25
26 export default {
... line 27
28   components: {
29     LegendComponent,
30   },
... lines 31 - 33
34 };
35 </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
... lines 1 - 25
26 export default {
... lines 27 - 30
31   data: () => ({
32     legend: 'Shipping takes 10-12 weeks, and products probably won't work',
33   }),
34 };
... 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>
14
```

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 need 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're 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"`.

'. Line 19 is '>'. Line 20 is ''. Line 21 is ''. Line 22 is ''. Line 23 is ''. Line 24 is ''. Line 25 is ''. Line 26 is ''." data-bbox="42 186 945 346"/>

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

```
68 lines | assets/js/components/sidebar.vue
34 <script>
35 export default {
36
37   data() {
38     return {
39       collapsed: false,
40
41     };
42   },
43 };
44
```

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 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`.

```
68 lines | assets/js/components/sidebar.vue
1  <template>
2    <div
3
4      :style="{ width: collapsed ? '70px' : 'auto' }"
5    >
6
31 </div>
32 </template>
33
```

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"`.

```
78 lines | assets/js/components/sidebar.vue
... lines 1 - 33
34     <div class="d-flex justify-content-end">
35       <button
36         class="btn btn-secondary btn-sm"
37       >
38
39     </button>
40   </div>
... lines 41 - 78
```

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

```
78 lines | assets/js/components/sidebar.vue
... lines 1 - 34
35     <button
36       class="btn btn-secondary btn-sm"
37     >
38       {{ collapsed ? '>>' : '<< Collapse' }}
39     </button>
... lines 40 - 78
```

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.

```
77 lines | assets/js/components/sidebar.vue
... lines 1 - 34
35     <button
36
37       v-text="collapsed ? '>>' : '<< Collapse'"
38     />
... lines 39 - 77
```

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!')` .

```
83 lines | assets/js/components/sidebar.vue
... lines 1 - 43
44 <script>
45 export default {
... lines 46 - 61
62   methods: {
63     toggleCollapsed() {
64       console.log('CLICKED!');
65     },
66   },
67 };
68 </script>
... lines 69 - 83
```

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**.

```
83 lines | assets/js/components/sidebar.vue
1 <template>
... lines 2 - 33
34 <div class="d-flex justify-content-end">
35   <button
... line 36
37     v-on:click="toggleCollapsed"
... line 38
39   />
40 </div>
... line 41
42 </template>
... lines 43 - 83
```

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.

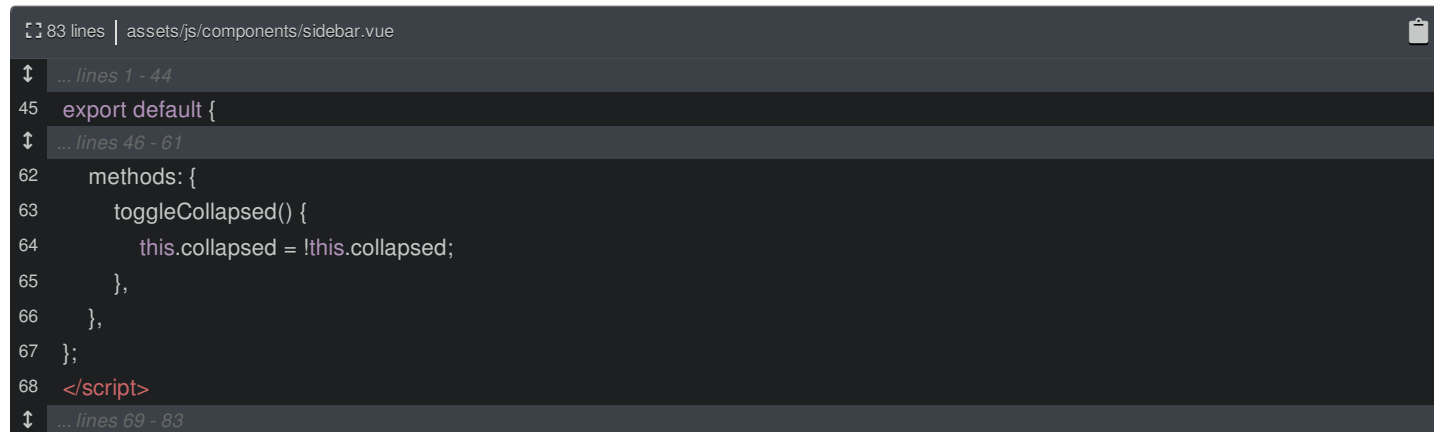
Vue *also* has a shortcut for **v-on**: the **@** symbol. Want to run some code "on click", use **@click** or **@mouseover** or **@** *whatever* event you want.

```
83 lines | assets/js/components/sidebar.vue
... lines 1 - 34
35 <button
... line 36
37   @click="toggleCollapsed"
... line 38
39 />
... lines 40 - 83
```

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` .



```
83 lines | assets/js/components/sidebar.vue
... lines 1 - 44
45 export default {
... lines 46 - 61
62   methods: {
63     toggleCollapsed() {
64       this.collapsed = !this.collapsed;
65     },
66   },
67 };
68 </script>
... lines 69 - 83
```

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)`.

```
86 lines | assets/js/components/sidebar.vue
... lines 1 - 43
44 <script>
45 export default {
... lines 46 - 61
62   created() {
63     console.log(this);
64   },
... lines 65 - 69
70 };
71 </script>
... lines 72 - 86
```

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".


```
92 lines | assets/js/components/sidebar.vue
... lines 1 - 44
45 export default {
... lines 46 - 61
62   props: {
63     testProp: {
64       type: String,
65       default: 'I am the default value',
66     },
67   },
... lines 68 - 75
76 };
... lines 77 - 92
```

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` .

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.

```
93 lines | assets/js/components/sidebar.vue
1 <template>
... lines 2 - 5
6 <h5 class="text-center">
7   Categories
8   {{ _uid }}
9 </h5>
... lines 10 - 42
43 </template>
... lines 44 - 93
```

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*:

93 lines | assets/js/components/sidebar.vue

... lines 1 - 5

```
6     <h5 class="text-center">
7       Categories
8       {{ definitelyNotARealProperty }}
9     </h5>
```

... lines 10 - 93

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.

[Tracking Changes with the Observer](#)

You see, when Vue renders a component, it keeps track of *every* data or prop that you *access* during that process, whether you access it directly in the template or reference it in a method. It's able to do that because when we access a property, it actually calls that `reactiveGetter` thing.

The point is: by the time Vue finishes rendering a component, it has a nice list of all of the data and props that this component *depends* on. Each component has an "observer" object internally that keeps track of this. Then, when any of those *change*, the observer is notified and it re-renders the component. All we need to do is focus on rendering our component. But behind-the-scenes, Vue is observing which properties we're using so that it knows *exactly* when it needs to re-render.

[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.

```
92 lines | assets/js/components/sidebar.vue
... lines 1 - 43
44 <script>
45 export default {
... lines 46 - 67
68   created() {
69     console.log(this, this.categories);
70   },
... lines 71 - 75
76 };
77 </script>
... lines 78 - 92
```

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.

Chapter 19: v-if, v-show and Conditional Classes

Our collapsing sidebar works, but *yikes*, it looks terrible! We *need* to fix this! Let's hide the text and links *entirely* when the sidebar is in its collapsed state.

v-if directive

Look at `sidebar.vue`. In the template, we want to hide the `<h5>`, the `` and the `<hr />`: basically everything except the button.

In Vue, there are two ways to do this and both of them come in the form of *directives*. To make this work, I'm going to add a new `<div>` to wrap all the elements that we need to hide.

The first directive is `v-if`. We can write `v-if=""` and, inside, use a JavaScript expression. We'll say `!collapsed`. So: show this if we are *not* collapsed. If we go over to our browser, we don't even need to refresh because we're using hot module replacement. And... it works! Much better!

```
85 lines | assets/js/components/sidebar.vue
1  <template>
2    <div
3  ... lines 3 - 4
5    >
6      <div v-if="!collapsed">
7  ... lines 7 - 33
34    </div>
35  ... lines 35 - 42
43    </div>
44  </template>
45  ... lines 45 - 85
```

inspecting v-if and the v-show directive

The key thing here, if you inspect the HTML, is that with `v-if`, the elements are removed *entirely*. So if we click the button again, the elements come back, and if we collapse, they're gone!

The other way to hide things is by using `v-show`. With `v-show` things will *look* the same. The difference is that, when we click `collapse`, the HTML is still there. `v-show` just adds `display: none` to the element when it needs to. *Sneaky*!

```
85 lines | assets/js/components/sidebar.vue
1  ... lines 1 - 5
6    <div v-show="!collapsed">
7  ... lines 7 - 33
34    </div>
35  ... lines 35 - 85
```

So `v-show` and `v-if` accomplish the same thing. Which you should use just depends on the situation. `v-show` is usually better if you're hiding and showing a lot, like a drop-down menu. That's because `v-show` is *super* fast.

But if what you're hiding isn't shown often or is complex - like a modal - `v-if` might be better because Vue won't spend any time even rendering the element until you need it.

.component style

Ok! While we're here, I also want to clean things up. Let's refactor our `width` away from `style` to a proper class. But before we do that, there's one thing I wanted to do earlier that I forgot about. Notice that down here in our styles, we have a single class called `sidebar` ... and we're applying that to the *root* element of our template. This is great! But sometimes, as a convention, when you have a class that is applied to the root element of a component, you'll use the generic name `.component`

```
85 lines | assets/js/components/sidebar.vue
... line 1
2   <div
3     :class="[$style.component, 'p-3', 'mb-5']"
... line 4
5   >
... lines 6 - 42
43 </div>
... lines 44 - 85
```

Down in `style` , update the class name to match.

```
85 lines | assets/js/components/sidebar.vue
... lines 1 - 71
72 <style lang="scss" module>
... lines 73 - 74
75 .component {
... lines 76 - 82
83 }
84 </style>
```

This won't make any difference, it's just a nice standard when you need a class on your outer element.

To remove this `width` style, go back down to the style block and, inside `.component` add a new class called `&.collapsed` with `width: 70px` .

```
93 lines | assets/js/components/sidebar.vue
... lines 1 - 78
79 .component {
... lines 80 - 81
82   &.collapsed {
83     width: 70px;
84   }
... lines 85 - 90
91 }
... lines 92 - 93
```

If you're not familiar with this syntax, it means that if an element has both the `component` and the `collapsed` classes, then it will get this width.

Back in the template, delete the `style` attribute.

The tricky part here is that we need to conditionally add the `collapsed` class to our root element. But... there's not really a good way to do that.

Conditional :class

Luckily `Vue` is here again to rescue us! It turns out, `:class` has *another* superhero syntax.

Instead of an array, pass an object where every *key* in the object will be the class name that you want to add, and the *value* will be true or false for whether or not you *want* this class. Since we always want these three classes, we'll say `$style.component: true` , `'p-3': true` and `'mb-5': true` .

And... Webpack is furious with me! The reason is that, in an object, you can't use variable names as keys. The problem is that JavaScript thinks that all keys are *strings*, even if you don't have quotes around them. If you want tell JavaScript that your key is some sort of variable, you need to put square brackets around it. That's a bit ugly, but *now* JavaScript understands it.

To add the `collapsed` class conditionally. I'll use that funny syntax again to say `$style.collapsed` , because every time we reference a class within modular CSS, we need to use `$style` . And since we want the `collapsed` class to show if the collapsed state is `true` , we'll just set it to `collapsed` .

```
93 lines | assets/js/components/sidebar.vue
1  <template>
2    <div
3      :class="{
4        [$style.component]: true,
5        [$style.collapsed]: collapsed,
6        'p-3': true,
7        'mb-5': true
8      }"
9    >
10  ... lines 10 - 46
47  </div>
48  </template>
49  ... lines 49 - 93
```

Back in the browser, click on [>> Collapse](#) and... perfect! You can see our class hiding and showing in the DOM.

But... there's a slightly *better* way to accomplish all of this... and it will use one of my *favorite* features of Vue: computed properties. Let's talk about them next.

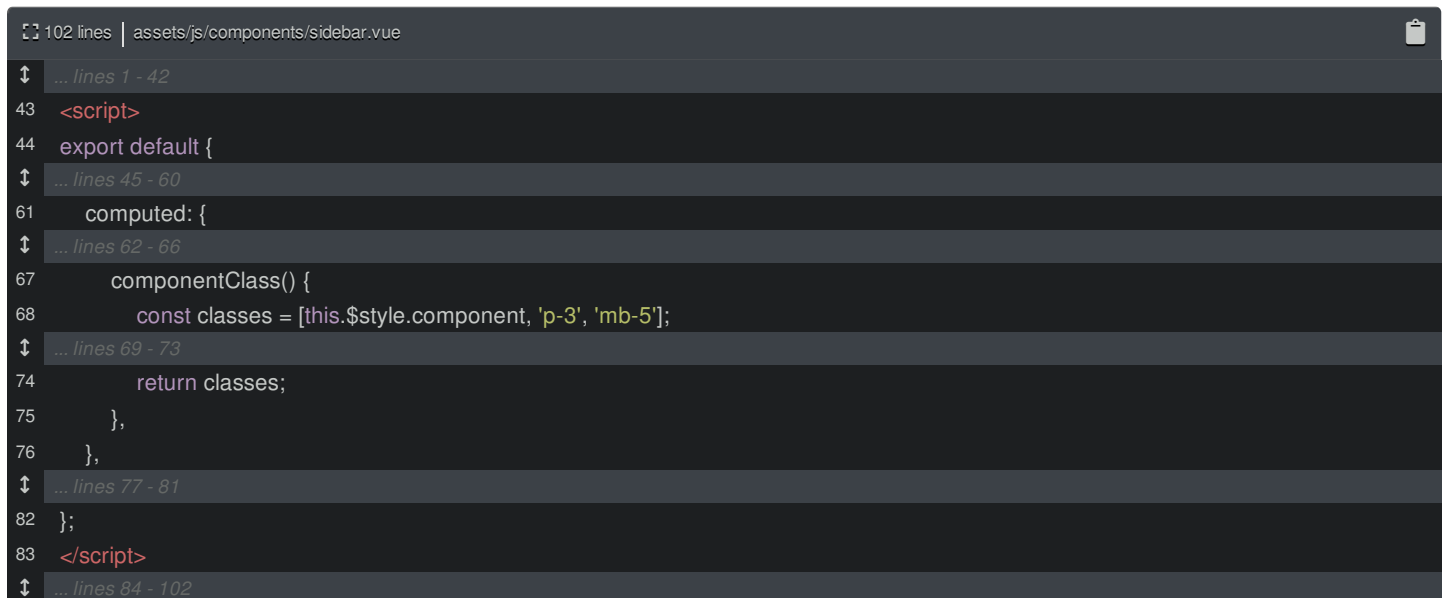
Chapter 20: Computed Properties

This special `class` syntax gave us the power to conditionally add the `collapsed` class. This works nicely, but it *is* kind of a lot of logic to have in our template. It's gettin' a bit ugly. A better solution might be to calculate which classes this component should have using *JavaScript* and then *pass* that value into our template as a variable.

Whenever you need to calculate a value, like an array of classes, based off some `props` or `data`, the way to do that is with a *computed property*. Here's how it works: Down inside our JavaScript code, right below `data`, add a new option called `computed` and set it to an object.

Just like with `methods` down here, we'll populate this with *functions*. Add our first computed property called `componentClass()` - you can name that *anything*. This will return the array of classes that our component should have. But don't worry about how we're going to *use* it yet: let's just start by filling in the logic. First we'll say `const classes =` and set that to an array with the three classes that we always need. `this.$style.component` (we'll talk about that in a second), `p-3`, and `mb-5`. Perfect!

At the bottom, return `classes`. We'll worry about the conditional class in a minute.



```
102 lines | assets/js/components/sidebar.vue
... lines 1 - 42
43 <script>
44 export default {
... lines 45 - 60
61   computed: {
... lines 62 - 66
67     componentClass() {
68       const classes = [this.$style.component, 'p-3', 'mb-5'];
... lines 69 - 73
74       return classes;
75     },
76   },
... lines 77 - 81
82 };
83 </script>
... lines 84 - 102
```

[.this magic](#)

But let's talk about `this.$style.component` real quick:

We know that as soon as we add `module` to our style tag, `Vue` makes a new `$style` variable available in our template. We use that to say things like `$style.component`.

A few minutes ago, we learned that anytime you reference a variable inside of a template, internally, what that *actually* does is call `this.$style`. We just don't *have* to say `this.` in the template because Vue adds it for us automatically.

So even if we knew *nothing* else, the very fact that we can reference the `$style` variable in a template means that the `vue` instance must have a `$style` property on it. In other words, we are allowed to say `this.$style` inside of JavaScript. That's why this works in our computed property method.

For the conditional class logic, let's say if `this.collapsed`, to reference our collapsed state, then `classes.push(this.$style.collapsed)`.


```

102 lines | assets/js/components/sidebar.vue
... lines 1 - 60
61   computed: {
... lines 62 - 66
67     componentClass() {
... lines 68 - 69
70       if (this.collapsed) {
71         classes.push(this.$style.collapsed);
72       }
... lines 73 - 74
75     },
76   },
... lines 77 - 102

```

That's it! And of course, with *any* methods, adding some documentation is always nice.

Here, PhpStorm *tries* to guess the return type... but gets a little confused since Vue is so dynamic. Let's help it: this returns an array of strings.

```

102 lines | assets/js/components/sidebar.vue
... lines 1 - 60
61   computed: {
62     /**
63      * Computes the component classes depending on collapsed state
64      *
65      * @return string[]
66      */
67     componentClass() {
... lines 68 - 74
75   },
76 },
... lines 77 - 102

```

Excellent!

[How computed works](#)

So here's the deal. As soon as you have a key under the **computed** option, it becomes available in the template as a *variable*.

Copy **componentClass** and, up in the template, very simply, we'll say **:class="componentClass"**.

```

102 lines | assets/js/components/sidebar.vue
1   <template>
2     <div :class="componentClass">
... lines 3 - 39
40   </div>
41 </template>
... lines 42 - 102

```

Up until now, we know that Vue adds all keys under **data**, **props** and **methods** to the Vue instance, which means that we can reference those inside our template. Well, **computed** is the fourth and *final* thing that gets added to the instance. **Vue** adds each key under **computed** as a *property*.

This means that, up in the template, we can just reference **componentClass**, which is really **this.componentClass**. But behind the scenes, when we access that property, Vue will *actually* call the **componentClass** function to get it. It's able to do that thanks to the fake getter property trick that we saw earlier when we console.logged the **this** variable.

And really, the only difference between **methods** and **computed** is the syntax: we use methods like methods and computed like properties. Oh, and also, Vue caches computed properties so that it only needs to call our function when something actually *changes*.

Anyways, because computed keys are added to the Vue instance, just like **methods**, **data** or **props**, it means that we can

reference them with the `this` variable. To prove it, inside of our `toggleCollapsed` method, let's say `console.log(this.componentClass)`.

```
104 lines | assets/js/components/sidebar.vue
... lines 1 - 76
77   methods: {
78     toggleCollapsed() {
... lines 79 - 80
81       console.log(this.componentClass);
82     },
83   },
... lines 84 - 104
```

Notice that PhpStorm tries to autocomplete that with parentheses, but that's not right! We need to reference it like a property, even though we know that Vue will call our method.

[Check it out!](#)

So if we go over to the browser now... check this out! You can see the log and it shows the component classes correctly every time we change it!

You can also see this over in the `Vue` dev tools! Click down on `Sidebar`. Under `data`, you now have a `computed` section with `componentClasses`. This changes when you hide and show the sidebar!

Back at our editor, remove the `console.log()`. We just mastered one of the most *powerful* tools in Vue: computed properties! Nice work!

Next, what happens when we need to access a piece of data - like `collapsed` - in a *different* component? If that component is a *child*, we can pass it down as a prop. But... what if it's not?

Chapter 21: Where should a Piece of Data Live?

This collapsing sidebar looks better... except that the layout doesn't change. Hmm, it would make more sense if the main content moved over to take up more space.

Right click and inspect element on this area. When we collapse the sidebar, what we *need* to do is change the classes on the `<aside>` to take up less space and change the classes on the main `<div>` to take up *more* space.

Pff. No problem: we just need to make a few classes dynamic!

Yes... but... it's not so simple. The `collapsed` data lives inside of `sidebar`. But the elements where we need to use that information do *not*. They live inside `products.vue`: here's the `<aside>` and this is the `<div>`. Somehow we need to access the collapsed data right here.

Never Duplicate Data

Let me *first* say one important thing: a piece of data - like `collapsed` - should *never* be duplicated. It must always live in exactly *one* spot. So what we're *not* going to do is create another `collapsed` data inside of `products` ... and then - somehow - try to keep the new `collapsed` data in sync with the `collapsed` data in sidebar. Yuck!

Nope, a piece of data should always live in exactly one spot.

Now, if you use something like Vuex - or some strategies in Vue 3 that we'll talk about in a future tutorial - then it's possible to store your data in a central location, *outside* of your components. Then multiple components can get and set that data directly. But in our case, there's only one place that data can live: inside a component.

In which Component should Data Live?

And so, each time you need to introduce a new piece of data into your app, there's a natural question: which component should this data live in? Like, why did we put `collapsed` in `sidebar` instead of, maybe inside `products.vue`?

Here's the rule: you should add data to the deepest component that needs it. By deepest, I'm referring to the component hierarchy that you can see in the Vue Dev tools: `Sidebar` is a deep component and `Products` is higher.

Now, until this moment, the *only* component that needed access to the `collapsed` data was `Sidebar`. So it made *perfect* sense to put it inside `Sidebar`. But *now* we realize that we *also* need access to the `collapsed` data inside of `Products`.

To make this work, we actually need to *move* the `collapsed` data from `sidebar` up into the `products` components. That will allow us to *pass* it down into `sidebar` as a prop. You can't pass data *up*, but you *can* pass it down.

Moving the collapsed Data

Let's get to work! Inside of `products.vue`, let's add the new data. Create the `data()` function. Call the data `sidebarCollapsed` - so we know *what* is collapsed - and initialize it to `false`.

```
32 lines | assets/js/pages/products.vue
↑ ... lines 1 - 14
15 <script>
↑ ... lines 16 - 18
19 export default {
↑ ... lines 20 - 24
25   data() {
26     return {
27       sidebarCollapsed: false,
28     };
29   },
30 };
31 </script>
```

Now that the data lives here, we need to pass it to `sidebar`. No problem! Say `:collapsed` - because we need this to be set to a dynamic value - `= "sidebarCollapsed"`.

```
32 lines | assets/js/pages/products.vue
1  <template>
  ... lines 2 - 3
4    <aside class="col-xs-12 col-3">
5      <sidebar :collapsed="sidebarCollapsed" />
6    </aside>
  ... lines 7 - 12
13 </template>
  ... lines 14 - 32
```

Perfect! To be able to *receive* this `collapsed` prop, in `sidebar`, we need to define it. Add `props`: set to an object with a `collapsed` key set to *another* object. We need `type: Boolean` and I like adding `required: true` to make sure it's passed.

```
107 lines | assets/js/components/sidebar.vue
  ... lines 1 - 42
43 <script>
44 export default {
  ... line 45
46   props: {
47     collapsed: {
48       type: Boolean,
49       required: true,
50     },
51   },
  ... lines 52 - 86
87 };
88 </script>
  ... lines 89 - 107
```

We now temporarily have a `collapsed` prop *and* a `collapsed` data and PhpStorm is *mad*! And... it's right: we can't do that and we don't want to. Delete the `collapsed` data.

Technically, we CAN have a data and a prop by the same name, but at runtime, only the prop will be present!

```
107 lines | assets/js/components/sidebar.vue
  ... lines 1 - 51
52 data() {
53   return {
54     categories: [
  ... lines 55 - 62
63   ],
64   };
65 },
  ... lines 66 - 107
```

The *cool* thing is that, because we named the `collapsed` prop the same as the data we had before, most of our code is just going to work! Vue doesn't care if the `collapsed` variable is a data or prop.

Let's try it! Find your browser and I'll refresh just to be safe. Click on the `Products` component in the dev tools and change the new `sidebarCollapsed` data to true.

Yes! The sidebar collapsed! If you click on `Sidebar`, you can see that the prop is true. Each time we change the `sidebarCollapsed` data, the prop in `Sidebar` *also* changes.

[What Happens when you Modify a Prop](#)

Even clicking the collapse button works! Well... *sort of*. Click on the console to find... ah! A horrible error!

Avoid mutating a prop directly since the value will be overwritten whenever the parent component re-renders.

So... here's what's happening. We *are* able to reference the `collapsed` prop inside `sidebar`. But down on the button, when we click it, this calls `toggleCollapsed()`. Jump down to that method. Ah... then this method *changes* the `collapsed` *prop*.

Earlier, I said that you should *never, ever* change the value of a prop. Props are meant to be *read* but not *modified*. Of course, you might be thinking:

Yeah... but it worked! When we changed the prop, the sidebar *totally* re-rendered correctly!

And that's true! But not everything is right with the world. Look back at the Vue Dev tools and the `Sidebar` component. The `collapsed` prop is false. When we click the button, that *correctly* changes to `true`.

But now look at the `Products` component. Under data, `sidebarCollapsed` is false! When we click the button it does *not* change! By modifying the prop, we've cause our `sidebarCollapsed` data and `collapsed` prop to get out of sync. If we were using the `sidebarCollapsed` data in the `Products` component, the sidebar would probably look half collapsed and half *not* collapsed.

The point is, each piece of data must live in exactly *one* location and *that* location is the "source of truth". *That* is where the value needs to change.

What we *really* need to do is somehow have the `Sidebar` *tell* the `Products` component:

Hey! The button was clicked so... the collapsed data should change!

And *then* the `Products` component could change its *own* data. The *way* to do this is with `$emit()`. Let's talk about it next.

Chapter 22: Communication UP with \$emit

The `sidebarCollapsed` data now lives inside of `Products` because, in a minute, we're going to use it to dynamically change the classes on the sidebar and content elements. We're also already passing this data as a `collapsed` prop into `Sidebar` so we can happily reference it inside the template.

The problem is that, when we click the Collapse button, what we *really* want to do is change the data on the *parent* component: we want to change `sidebarCollapsed` on `products.vue`. But... you *can't* do that. A component can *only* change data on itself.

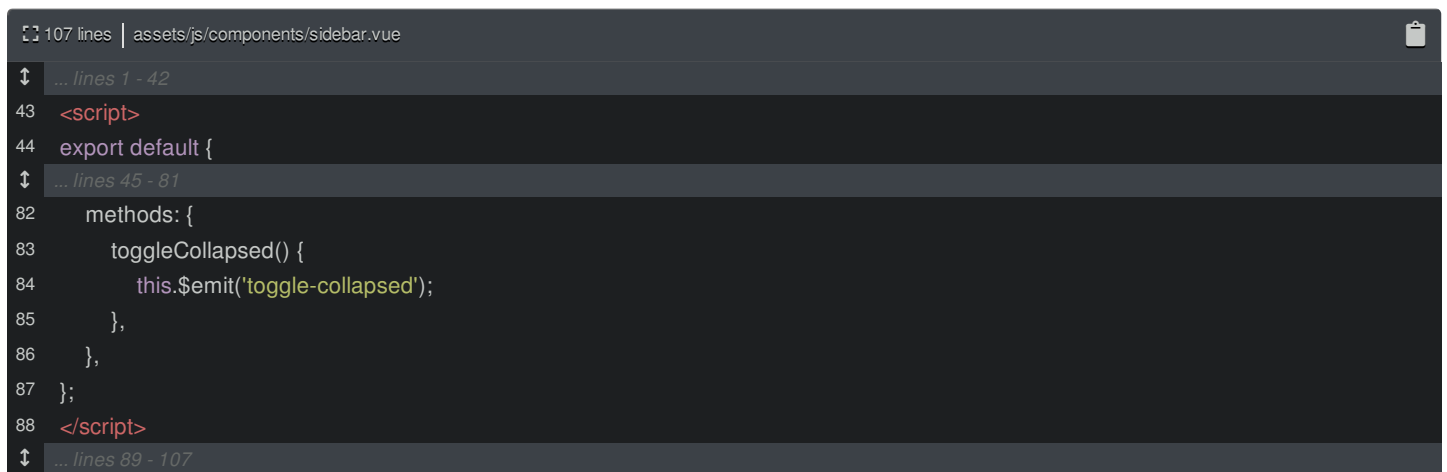
Here's the deal: we already know how to communicate information *down* the component tree. We do that via props: we communicate the `sidebarCollapsed` state down into `sidebar` with a prop. Cool! But how can we communicate *up* the tree?

[Communicating to a Parent Component with \\$emit\(\)](#)

Whenever a child component needs to change data on a parent component... or, more abstractly: whenever a child component needs to *communicate* that something happened to a parent component - like "the collapsed button was clicked" - it does that by emitting an event.

Check this out: inside of `sidebar`, go down to the `toggleCollapsed()` method that's being called on click. Instead of modifying the prop, say `this.$emit()`. Yes, in addition to props, data, methods and computed props, the `Vue` instance - the `this` variable - *also* has a few built-in methods and properties. There aren't a *ton* of methods, but there *is* one called `$emit` and it's one of the *most* important.

Inside of `$emit()`, let's emit a custom event called `toggle-collapsed`, which I totally just made up.



```
107 lines | assets/js/components/sidebar.vue
... lines 1 - 42
43 <script>
44 export default {
... lines 45 - 81
82   methods: {
83     toggleCollapsed() {
84       this.$emit('toggle-collapsed');
85     },
86   },
87 };
88 </script>
... lines 89 - 107
```

This won't work yet, but we can *already* see it in action. Back on your browser, over in the Vue dev tools, select the `Sidebar` component. So far, we've been paying a lot of attention to the `props`, `data` and `computed` info. But there is *also* a section called `events`. This will show any *events* that our Vue components are emitting. So now, when we hit the Collapse button, nothing *happens* yet, but we can see the `toggle-collapsed` event being emitted! That's pretty cool. The next step will be to *listen* to this event from the parent component.

[Emitting Directly in v-on](#)

Before we do that, calling `this.$emit()` is *totally* fine from inside a method. But we can simplify. Copy the `$emit()` code then delete the method entirely. Up in the template, find the `@click`, which we know is really `v-on:click`. Set it to `$emit('toggle-collapsed')`.

```

102 lines | assets/js/components/sidebar.vue
1  <template>
2  ... lines 2 - 32
33  <div class="d-flex justify-content-end">
34    <button
35  ... line 35
36    @click="$emit('toggle-collapsed')"
37  ... line 37
38  />
39  </div>
40  ... line 40
41  </template>
42  ... lines 42 - 102

```

Because... remember! Whenever you reference a variable or function inside a template, Vue will really call `this.$emit()` behind the scenes... which is *exactly* what we were doing a moment ago. This is just shorter... and it emits the event just like before.

[Listening to the Event on the Parent Component](#)

The second step to this process is to go into the `products` component and *listen* to that event. I'll move the `<sidebar>` onto multiple lines.

To listen to the `toggle-collapsed` event, we're going to use `v-on`. Because, really, listening to a custom event is *no* different than what we're doing in `sidebar`. To listen to the `click` event of a button, we use `v-on:click`, or `@click` for short. Then, on click, we run some code!

`click` is a native DOM event, but things work *exactly* the same for an event that we're emitting manually.

Let's do this the long way first: say `v-on:toggle-collapsed=` and set this to call a new `toggleSidebarCollapsed` method when that event happens.

```

40 lines | assets/js/pages/products.vue
1  <template>
2  ... lines 2 - 3
4    <aside class="col-xs-12 col-3">
5      <sidebar
6  ... line 6
7      v-on:toggle-collapsed="toggleSidebarCollapsed"
8    />
9  </aside>
10 ... lines 10 - 15
16 </template>
17 ... lines 17 - 40

```

Copy that name and go down to the `script` section. We don't have any methods yet so first add `methods`, then create the `toggleSidebarCollapsed()` function inside. Very simply - just like we did before in `sidebar` - set `this.sidebarCollapsed` to `!this.sidebarCollapsed`.

```

40 lines | assets/js/pages/products.vue
1  ... lines 1 - 21
22 export default {
23  ... lines 23 - 32
33  methods: {
34    toggleSidebarCollapsed() {
35      this.sidebarCollapsed = !this.sidebarCollapsed;
36    },
37  },
38 };
39 </script>

```

I love that. Back on the browser, I'll refresh to be safe... and then click Collapse. It works! You can see the events and, back on

the `Products` component, we can watch the `sidebarCollapsed` data change. *That* is a nice setup.

Using Shortcut v-on Everywhere

Now that we have this working, go back and find the `v-on:` attribute. What I *love* about this directive is how clear it is:

on toggle-collapsed, run this code

But in practice, we're going to use the shortcut syntax everywhere. So: `@toggle-collapsed`.

```
40 lines | assets/js/pages/products.vue
1  <template>
  ... lines 2 - 3
4    <aside class="col-xs-12 col-3">
5      <sidebar
  ... line 6
7        @toggle-collapsed="toggleSidebarCollapsed"
8      />
9    </aside>
  ... lines 10 - 15
16 </template>
  ... lines 17 - 40
```

Just remember that the `@` symbol means "on" - so "on toggle collapsed".

Our communication between the `Products` component and `Sidebar` - using props to communicate down and events to communicate up - is a pattern that you'll see *many* times in Vue. It keeps a single source of data, but *effectively* allows anyone to read *or* change it.

Next, let's *use* the new `sidebarCollapsed` data in `products.vue`. When we do that, we're going to add some custom styles but decide to *not* make them modular.

Chapter 23: Using Shared CSS

So let's *finally* use the new `sidebarCollapsed` data inside `products`. Right now, when we collapse, we add a `collapsed` class... which makes the element really small - just 70px.

Remove the `.collapsed` class entirely: we're going to simplify.

```
82 lines | assets/js/components/sidebar.vue
... lines 1 - 68
69 <style lang="scss" module>
70 @import '~styles/components/light-component';
71
72 .component {
... lines 73 - 79
80 }
81 </style>
```

Inside the `componentClass` computed property, copy the three classes that the sidebar *always* has. Then delete that completely. We're going to move *all* of the size logic to the parent component. Paste the three classes onto the outer div.

```
82 lines | assets/js/components/sidebar.vue
1 <template>
2 <div :class="[$style.component, 'p-3', 'mb-5']">
... lines 3 - 39
40 </div>
41 </template>
... lines 42 - 82
```

Over in `products.vue`, the classes on the `<aside>` and this `<div>` now need to be dynamic based on the `sidebarCollapsed` data. Hey! That's a *perfect* use-case for computed properties!

Down in our code, add a `computed` option with *two* computed properties. Call the first `asideClass`: it will determine the classes for the `<aside>` element. Inside, return `this.sidebarCollapsed`. If we *are* collapsed, use a class called `aside-collapsed` - that doesn't exist yet, we'll create it in a minute. If we are *not* collapsed, use the normal `col-xs-12` and `col-3`.

```
48 lines | assets/js/pages/products.vue
... lines 1 - 17
18 <script>
... lines 19 - 21
22 export default {
... lines 23 - 32
33   computed: {
34     asideClass() {
35       return this.sidebarCollapsed ? 'aside-collapsed' : 'col-xs-12 col-3';
36     },
... lines 37 - 39
40   },
... lines 41 - 45
46 };
47 </script>
```

Oh, and Webpack is mad because, of course, this needs to be a *function*. That's better.

Copy `asideClass()`, paste and call it `contentClass()`. For this one, when we're collapsed, use `col-xs-12 col-11` so that it takes up *almost* all of the space. And then when it's not collapsed, use the normal `col-xs-12 col-9` so that it shares the space.

```
48 lines | assets/js/pages/products.vue
... lines 1 - 32
33   computed: {
... lines 34 - 36
37     contentClass() {
38       return this.sidebarCollapsed ? 'col-xs-12 col-11' : 'col-xs-12 col-9';
39     },
40   },
... lines 41 - 48
```

Perfect! Well... no, not perfect: ESLint isn't happy about this `computed` option:

The `computed` property should be above the `methods` property.

This makes no difference *technically*, but there are some general best-practices about the *order* of your options. Let's move this above `methods` to follow those.

```
48 lines | assets/js/pages/products.vue
... lines 1 - 21
22 export default {
... lines 23 - 32
33   computed: {
... lines 34 - 39
40   },
41   methods: {
... lines 42 - 44
45   },
46 };
... lines 47 - 48
```

Ok: now that we have 2 keys under `computed`, we have two new variables inside of our template. Scroll up to the top. For the `<aside>` element, we can say `class="asideClass"`. Ah, but I'm sure you're starting to spot my mistake - we need `:class` to make that dynamic.

Do the same for the div below: `class="contentClass"` and then make it `:class`.

```
48 lines | assets/js/pages/products.vue
1  <template>
... lines 2 - 3
4    <aside :class="asideClass">
... lines 5 - 8
9    </aside>
... line 10
11   <div :class="contentClass">
... line 12
13   </div>
... lines 14 - 15
16 </template>
... lines 17 - 48
```

Sweet! Let's give it a try! I'll refresh just to be safe. And... ah! It works! It might feel smoother with some CSS transitions, but on a Vue level, this is working brilliantly!

Adding a Global Class

Though, I *could* use a little more padding on the sidebar when it's collapsed... I don't want it *all* the way against the edge.

Go back to the `computed` property. When the sidebar is collapsed, it has an `aside-collapsed` class... which I *totally* made up: that does *not* exist yet. To fix our padding issue, we *could* have said `this.$style['aside-collapsed']` and then added a new `.aside-collapsed` class to the `style` tag of this component.

But... to make this more interesting, let's pretend that we're going to have multiple components across our app that will need to use this class. And so, I don't want to add it as a modular style to this component: I'd rather put this CSS in a *central* spot and *share* it.

That is why I used `.aside-collapsed` instead of using the `$style` variable. Open up `scss/app.scss` and, at the bottom, add the style: `aside-collapsed` with `padding: 0 15px`.

```
60 lines | assets/scss/app.scss
... lines 1 - 56
57 .aside-collapsed {
58   padding: 0 15px;
59 }
```

Cool! When we move over to our browser... yes! It looks better already. So this is just a reminder that while modular styles are *cool*, if you want to re-use something, you *can* continue to use normal CSS files.

[Importing Shared, Non-Modular CSS Files](#)

By the way, you *could* also use `@import` to import CSS files from inside the `style` tag of your component... you can even do it in a way that *prevents* the styles from getting the module prefix. To do that, add a second `style` tag, leave off the `module` and make the language `css`. You can still *import* SASS files, but if you make the language `scss`, for some reason, your CSS rules will get duplicated inside Webpack.

Next, it's finally time to make some Ajax calls and bring in some *truly* dynamic data!

Chapter 24: Ajax with Axios

Our app is *really* starting to come together. I think it's time to make our data *dynamic*: the categories on the sidebar are hardcoded and the products aren't even loading yet. Let's make some Ajax calls!

Installing Axios

To make those, we're going to use a library called Axios. To install it, open a new terminal tab and run:

A terminal window with a dark background and three window control buttons (red, yellow, green) in the top-left corner. The command prompt shows a dollar sign followed by the text 'yarn add axios --dev'.

The `--dev` part isn't very important.

The other popular option for AJAX calls is to use `fetch()` instead of Axios. `fetch()` is actually a built-in JavaScript function, which means you don't need any outside library. However, if you need to support IE 11, then you *will* need a polyfill to use it. Both Axios and fetch are great options.

Investigating our API

For our first trick, let's load products onto the page. Our app already has an API - powered by API Platform - and you can see its docs by going to `/api`. Scroll down to the section about products and expand the endpoint for `GET /api/products`.

The best way to see how this works is... to try it! Let's see... hit Execute and... here's the response body. We already have a set of products in the database.

If you haven't used API Platform before - it's *no* problem. But, the structure with `@id`, `@context`, `hydra:member` and other keys might look odd. This *is* JSON, but it's using a format called JSON-LD Hydra, which is basically JSON with extra metadata: each response will have the same structure with extra fields to give you more info. It's super handy.

Now, notice that the URL to the endpoint is `/api/products`. But if we put `/api/products` in our browser... we don't see JSON! It's the same documentation page! That's because API Platform *realizes* - by reading the `Accept` header - that we're requesting this from a browser and so it returns HTML. When we request this from Axios with no `Accept` header, we'll get back JSON.

But if you ever want to see the JSON in a browser to see how it looks, there's a hack: add `.jsonld` to the end of the URL. *This* is our endpoint.

Let's go *all* the way back to our homepage and... I'll re-open the browser dev tools.

Making the AJAX Call from mounted()

Ok, when we load products, which component is going to *need* that data? The top-level `products.vue` component renders the sidebar and catalog. We *could* load the products here... but we won't *actually* need them. Hold Command or Ctrl and click `<catalog` to jump to that component.

Ah, *this* is the component that needs the products data.

Here's the goal: as *soon* as Vue loads this component, we'll start the AJAX call so that we can load the products as quickly as possible. Fortunately, Vue allows us to run code during its startup process, and there are two main "hook" points: `mounted` and `created`. We'll talk more about these later but Vue considers your component `mounted` when it's actually added to the page - like, in `products.js` when we call `.$mount()`.

To run code right *after* our component is mounted, all we need to do is create a function called `mounted()`. Inside, we'll make the AJAX call.

How? First, at the top of the `script` section `import axios from 'axios'`.

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

... lines 1 - 22

23 <script>

24 import axios from 'axios';

... lines 25 - 42

43 </script>

Then, *using* Axios is beautifully simple: `axios.get('/api/products')` . And like every AJAX library, this will return a *Promise*, which you can learn *all* about in a [JavaScript Tutorial](#) here on SymfonyCasts.

To use the Promise, add `.then()` , and pass an arrow function with `response` as the argument. Let's `console.log(response)` to see what it looks like.

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

... lines 1 - 36

37 mounted() {

38 axios.get('/api/products').then((response) => {

39 console.log(response);

40 });

41 },

... lines 42 - 44

Testing time! Back over on the browser, click to view the console. Thanks to hot module replacement... that already ran! But to make the flow more realistic, let's refresh the page.

Now... boom! The log shows up almost instantly. The `response` is an object with `headers` , `status` and other things. What we want is `data` . One of the nice features of Axios is that it decodes the JSON automatically.

When you're working with JSON-LD Hydra like this, the collection of items is stored on a `hydra:member` property. Yep, it's an array with 12 products. We have product data!

Next, this is working *great*, but I'm going to choose a slightly *different* syntax for handling Promises: `async` and `await`. Then, we'll use our brand new data to render those products onto the page.

Chapter 25: The await Keyword

Before we start using the products data from the AJAX call, there's *one* other way to work with promises... and I really like it! It's the `await` syntax. We know that Axios returns a Promise... and that we *normally* run code *after* a promise has finished - or "resolved" - by calling `.then()` on it.

This works great. But instead add `const response =` before the axios call and then remove the callback.

If we stopped right now, response would actually be a *Promise* - not a response. But if we put `await` in front of it, it *will* be a response! The `await` keyword causes your code to *wait* for that Promise to resolve. And whatever data is *normally* passed to your callback as an argument is instead *returned*. There is *still* an asynchronous AJAX call happening, but our code *reads* a bit more like synchronous code. The `await` keyword is syntactic sugar.

```
44 lines | assets/js/components/catalog.vue
... lines 1 - 22
23 <script>
... lines 24 - 26
27 export default {
... lines 28 - 36
37   mounted() {
38     const response = await axios.get('/api/products');
... lines 39 - 40
41   },
42 };
43 </script>
```

[Why do we need async?](#)

Hmm... PhpStorm looks mad... but... let's ignore it and try this anyways! Move back over to the browser and scroll to the bottom of the console. Ah!

Cannot use keyword await outside an async function.

So... first: I hope that the general idea of `await` makes sense to you. If you have something that is asynchronous, you put `await` in front of it and that says:

Please wait for this to finish, get the return value and *then* keep executing my code.

That's great. But this comes with one rule: whenever you use the `await` keyword, whatever function you're *inside* of needs to have an `async` keyword in front of it. Let me put back the `console.log(response)`.

```
44 lines | assets/js/components/catalog.vue
... lines 1 - 36
37   async mounted() {
... lines 38 - 39
40     console.log(response);
41   },
... lines 42 - 44
```

When you make a function `async`, it means that your function will *now* automatically and *always* return a *Promise*. If your function has a `return` value, that will be the *data* of the Promise.

This... can be confusing at first: when `mounted()` is called, our code *will* freeze on the `async` line and wait for the AJAX call to finish. But this doesn't freeze our entire JavaScript app. In reality, the `mounted()` function will almost *immediately* finish and will return a Promise. That Promise will *resolve* once all of our code executes.

To say this a different way: if we called `mounted()` directly from our code - we won't do that, but just pretend - then `mounted()` would finish before the AJAX call and it would *now* return a Promise. If we wanted to do something *after* the AJAX call and the rest of the code in `mounted()` finished, we could chain a `.then()` from that Promise.

But in reality, `Vue` is responsible for calling `mounted()` and `Vue` doesn't care about or use any value that we might return from `mounted()`. So basically, `Vue` couldn't care less that we just changed this to `async` and so, caused our method to return a Promise.

The key thing to know about `async` and `await` is that even though our code will wait on this line, *really* the `mounted()` function will finish nearly instantly. `Vue` isn't going to call `mounted()` then freeze our *entire* `Vue` app waiting for it to finish. It starts our AJAX call then keeps going. That's *perfect*.

Anyways, now when we refresh... yes! The AJAX call finishes and logs the response. Feel free to use Promises directly or with `await`: we'll use `await` in this tutorial.

[Adding the products Data & hydra:member](#)

Ok, let's use our *real* products data! Inside `catalog`, if you think about it, the products are something that will *change* during the lifecycle of our component. At the *very* least, when the `catalog` component first loads, the products will be empty. And then, as soon as the Ajax call finishes, they will *change* to be the real products.

That's a *long* way of saying that products need live in `data`. Head up to to the `data` option and add a new `products` key set to an empty array as its default data.

In `mounted()`, instead of logging the response, now we can say `this.products = response.data` and... let's see. Inside `response.data`, we want the `hydra:member` property. Cool! Add `.hydra:member`. Oh but that `:` mucks things up. We need to use square brackets here with that in quotes.

```
45 lines | assets/js/components/catalog.vue
↑ ... lines 1 - 37
38   async mounted() {
↑ ... lines 39 - 40
41     this.products = response.data['hydra:member'];
42   },
↑ ... lines 43 - 45
```

Ok, let's check out the `products` data. Move over... I'll refresh just to be safe... then on the `Vue` dev tools, find the `Catalog` component. Yes! Our `products` data has 12 items in it!

So... let's celebrate and *use* this data! Up in the template, remove that pesky `TODO`. In our design, we need to have one of these divs for each product. Break it into multiple lines and then loop with `v-for="product in products"`. And every time we use `v-for`, we need to add a `key` attribute set to some unique, non-changing key for each item.

```
49 lines | assets/js/components/catalog.vue
1  <template>
↑ ... lines 2 - 11
12    <div
13      v-for="product in products"
14      :key="product['@id']"
15      class="col-xs-12 col-6 mb-2 pb-2"
16    >
↑ ... line 17
18    </div>
↑ ... lines 19 - 24
25  </template>
↑ ... lines 26 - 49
```

[The Useful @id IRI](#)

If you look at the products data on the dev tools, you'll notice that we *do* have an `id` property. It's not very important in this case, but I'm actually going to use this `@id` instead. This is called an "IRI" - it's a unique key that API Platform adds to *every* resource. It's... just more *useful* than a database ID or UUID because it's a *real* URL: we could make a request to this address to

fetch that specific product.

That "usefulness" won't be... well... *useful* in this situation. But because of this, in general, I'm going to use `@id` everywhere as my unique identifier. The only problem is that when you say `product.@id`, JavaScript gets mad because you can't use an `@` sign in this syntax. Once again, use square brackets and wrap this in quotes.

Inside the div, start by printing the name: `{{ product.name }}` ... because name is one of the keys in the data. And... yea! You can already see it being used! There is our list of *real*, high-quality, products.

```
49 lines | assets/js/components/catalog.vue
1  <template>
  ... lines 2 - 11
12  <div>
  ... lines 13 - 15
16  >
17    {{ product.name }}
18  </div>
  ... lines 19 - 24
25 </template>
  ... lines 26 - 49
```

Next, if we added all of the markup and data we need for each product directly into the `catalog` component... things could get big and ugly fast. Let's split the product listing into two, smaller, sleeker components.

Chapter 26: Product Listing Components

We're only printing the *name* of each product, but we could *easily* start adding more content right here: the product image, price and a button to view the product. But if we did that, this template we start to get pretty big. And later, we're going to add a search bar... which will make this component even busier!

Exactly *when* you should split a component into *smaller* components is not a science. It's a subjective decision and there's no wrong answer. But I *am* going to refactor this `<div class="row">` into a `product-list` component: a component whose *entire* job will be to just... list products! At the end of the next chapter, I'll give you *another* reason why I did this.

Creating a Component Sub-Directory

You might expect me to go into the `components/` directory and create a new file called `product-list.vue`. But... I won't. That's not wrong, but instead, I'm going to create a new *directory* called `product-list/`. Because, in a few minutes, we're going to have *two* components that help us build this product-list area.

Now create a new file called `index.vue`. I'll talk about that name soon.

Start the same way as always: add a `<template>` tag and, inside, since we are going to move this entire area here, use the `<div class="row">` as the outer element.

```
18 lines | assets/js/components/product-list/index.vue
1  <template>
2    <div class="row">
3
4    </div>
5  </template>
... lines 6 - 18
```

Next, add the `<script>` tag with `export default` the options object. Give this a `name` set to, how about, `ProductList`.

```
18 lines | assets/js/components/product-list/index.vue
... lines 1 - 6
7  <script>
8    export default {
9      name: 'ProductList',
... lines 10 - 15
16 };
17 </script>
```

Move the products Data?

Poetry! I already know that we're *definitely* going to need access to the array of `products` so that we can loop over them. To get this, we have two options. First, we could *move* the `products` data from `catalog` into here. After all, I said that a piece of data should live in the *deepest* component that needs it. And when we're done, that would be *this* component!

But... I *won't* do that. I'll talk about why in the next chapter. Moving the `products` data into here wouldn't be *wrong*, but *leaving* it in `catalog` will help us follow a "smart component, dumb component" design pattern. More on that soon.

The Imprecise products Prop

The second option - and the one we'll choose - is to have the `products` passed to us as props. Add a `props` option with a `products` key. The type will be `Array` - because this will be an array of product objects - and I'll also add `required: true`.

```

18 lines | assets/js/components/product-list/index.vue
... lines 1 - 7
8   export default {
... line 9
10    props: {
11      products: {
12        type: Array,
13        required: true,
14      },
15    },
16  };
... lines 17 - 18

```

This is one of the downsides of `props` ... or kind of JavaScript in general: I can say that this should be an `Array`, but I can't really enforce that it's an array of product objects: an array of objects that all have certain keys. Well, you *can* do this with custom prop validator logic, but I think it's more work than it's worth. So, I'll just use `Array` and say "Good enough for JavaScript!"

In the template, since we already have the outer div, copy the `div` with the `v-for` and paste it here. In theory, we shouldn't need to change *anything*: we're going to pass that *same* array of product objects into this component.

```

24 lines | assets/js/components/product-list/index.vue
1   <template>
2     <div class="row">
3       <div
4         v-for="product in products"
5         :key="product['@id']"
6         class="col-xs-12 col-6 mb-2 pb-2"
7       >
8         {{ product.name }}
9       </div>
10    </div>
11  </template>
... lines 12 - 24

```

Importing and index Component

So... let's use this! Back in `catalog`, start by importing the component: `import ProductList` - that seems like a pretty good name - from `@/components/product-list`.

```

43 lines | assets/js/components/catalog.vue
... lines 1 - 18
19  <script>
... lines 20 - 21
22  import ProductList from '@components/product-list';
... lines 23 - 41
42  </script>

```

And I can stop right there. Because we have an `index.vue` inside of `product-list/`, we can import the *directory* and Webpack will know to use that `index.vue` file. That's a nice trick for organizing bigger components: create a sub-directory for the component with an `index.vue` file and add other sub-components inside that directory. We'll do that in a minute.

Anyways, add `ProductList` to components so that we can reference it in the template. Here, our job is pretty simple: delete the entire `<div>` and say `<product-list />`. PhpStorm is already suggesting the one `prop` that we need to pass: `products="products"`. But we know that's wrong: we don't want to pass the *string* products, we want to pass the variable.

It's kind of fun to see what it looks like if you forget the colon. In the console of our browser, if we refresh, yikes!

```
Invalid prop: type check failed for prop "products": expected Array got String with value "products".
```

Add the `:` before `products` to make that attribute dynamic. And back over on the browser... yea! All the products are printing just

like before.

```
43 lines | assets/js/components/catalog.vue
1  <template>
2    <div>
3    ... lines 3 - 10
11    <product-list :products="products" />
12    ... lines 12 - 15
16  </div>
17 </template>
18 ... lines 18 - 43
```

Creating a product-card Component

And once again, we could stop now and start adding more product info right here. But I'm going to go one step further and refactor each *individual* product into its own component.

Inside the `product-list/` directory, create a second component called, how about, `product-card.vue`. Start like we always do: with the `<template>`. Each product will use this div with the `col` classes on it. Copy the classes but *keep* the `v-for` here. In `product-card`, add a `div` and paste those classes.

Let's also go steal the `{{ product.name }}` and put that here.

```
18 lines | assets/js/components/product-list/product-card.vue
1  <template>
2    <div class="col-xs-12 col-6 mb-2 pb-2">
3      {{ product.name }}
4    </div>
5  </template>
6  ... lines 6 - 18
```

That's a good enough start. Next add the `<script>` tag with `export default` and `name` set to `ProductCard`. And once again, we know that we're going to need a product passed to us. So I'll jump straight to saying `props` with `product` inside. The product will be an *object*... and like with the Array, there's not an easy way to enforce *exactly* what that object looks like. But we can at least say `type: Object` and also `required: true` in case I do something silly and forget to pass the prop entirely.

```
18 lines | assets/js/components/product-list/product-card.vue
... lines 1 - 7
8  export default {
9    name: 'ProductCard',
10   props: {
11     product: {
12       type: Object,
13       required: true,
14     },
15   },
16 };
17 </script>
```

Ok! This is ready! Back in `index.vue`, we'll follow the *same* process to use the new component. This starts to feel a bit repetitive.. and I *love* that! It means we're getting comfortable.

Start with `import ProductCard from` and you can either say `./product-card`, which is shorter, or `@/components/product-list/product-card`, which is more portable.

```
27 lines | assets/js/components/product-list/index.vue
... lines 1 - 10
11 <script>
12 import ProductCard from '@/components/product-list/product-card';
13 ... lines 13 - 25
26 </script>
```

To make this component available in the template, add the `components` key with `ProductCard` inside.

```
27 lines | assets/js/components/product-list/index.vue
... lines 1 - 13
14 export default {
... line 15
16   components: {
17     ProductCard,
18   },
... lines 19 - 24
25 };
... lines 26 - 27
```

Finally, in the template, this is pretty cool: we now want to loop over the products and render the `ProductCard` *itself* each time. And putting a `v-for` on a custom component is *totally* legal. Replace the `<div>` with `<product-card>`, remove the `class` attribute and replace it with `:product="product"`. And since this element no longer has any content, it can be self-closing.

```
27 lines | assets/js/components/product-list/index.vue
1  <template>
2    <div class="row">
3      <product-card
4        v-for="product in products"
5        :key="product['@id']"
6        :product="product"
7      />
8    </div>
9  </template>
... lines 10 - 27
```

That's *really* nice: we loop over `products` and render an individual `ProductCard` for each one. When we check the browser, it looks like it's working! I'll refresh just to be sure... I don't *always* trust hot module replacement. And... yep! It *does* work.

What I *really* like about the `product-card` component is that it's focused on rendering just *one* thing: a single product. Next, let's *really* bring the product to life by adding more data, styles and a `computed` property.

Chapter 27: Product Details & Smart vs Dumb Components

Let's *really* make these products come to life! Now that we have a component whose *only* job is to render a *single* product, this is going to be fun & clean.

The Product Card Template

I'll start by pasting some HTML into the template: you can copy this from the code block on this page. But there's nothing too interesting yet. We *are* referencing a few styles - `$style['product-box']` and `$style.image` and we'll add a `style` tag soon for those. If you're wondering why I'm using the square bracket syntax, that's because JavaScript doesn't like dashes with the object property syntax: you can't say `$style.product-box` ... which, yes, is annoying.

```
46 lines | assets/js/components/product-list/product-card.vue

1  <template>
2    <div class="col-xs-12 col-6 mb-2 pb-2">
3      <div :class="$style['product-box']">
4        <div :class="$style.image">
5          
10
11      <h3 class="font-weight-bold mb-2 px-2">
12        {{ product.name }}
13      </h3>
14    </div>
15
16    <div class="p-2 my-3 d-md-flex justify-content-between">
17      <p class="p-0 d-inline">
18        <strong>${{ product.price }}</strong>
19      </p>
20
21      <button
22        class="btn btn-info btn-sm"
23      >
24        View Product
25      </button>
26    </div>
27  </div>
28  <hr>
29  <div class="px-2 pb-2">
30    <small>brought to you by {{ product.brand }}</small>
31  </div>
32 </div>
33 </template>
```

A little below this, we *are* using some product data. If you go back to the Vue dev tools and click on Catalog, each product has several fields on it, like `brand` , `image` - which is the URL to an image - `name` , `price` , and even `stockQuantity` .

So, for the image, we're using `src="product.image"` - with the `:` that makes this attribute dynamic - and we're rendering more data for the `alt` attribute, the product name and the `product.price` . We also have a button to *view* the product page... which isn't doing anything yet... then we print `product.brand` .

Hopefully this all feels pretty simple.

When you Forget the style Tag

If we move over now and check the console, oooooo:

```
Cannot read property product-box of undefined
```

Coming from **product-card**. Vue is telling us that the **\$style** variable is undefined... which makes sense: we don't have a **style** tag yet! No problem: add the **<style>** with **lang="scss"**. In fact, **\$style** will be undefined until you have a **style** tag *and* that tag has the **module** attribute.

```
67 lines | assets/js/components/product-list/product-card.vue
... lines 1 - 46
47 <style lang="scss" module>
... lines 48 - 65
66 </style>
```

For the styles itself, I'm going to import this **scss/components/light-component.scss** file, which is a Sass mixin. Add **@import**, then, to use the Webpack alias we created earlier, say **~** and the name of the alias. So **~styles/** to point to the **scss** directory - then **components/light-component**.

```
67 lines | assets/js/components/product-list/product-card.vue
... lines 1 - 46
47 <style lang="scss" module>
48 @import '~styles/components/light-component';
... lines 49 - 65
66 </style>
```

Excellent! Now that we've done the hard work, I'll paste in a few more styles. This adds the **.product-box** and **.image** classes that correspond with the **\$style** code in the template.

```
67 lines | assets/js/components/product-list/product-card.vue
... lines 1 - 46
47 <style lang="scss" module>
... line 48
49 .product-box {
50   border: 1px solid $light-component-border;
51   box-shadow: 0 0 7px 4px #efefee;
52   border-radius: 5px;
53 }
54 .image {
55   img {
56     width: 100%;
57     height: auto;
58     border-top-left-radius: 5px;
59     border-top-right-radius: 5px;
60   }
61
62   h3 {
63     font-size: 1.2rem;
64   }
65 }
66 </style>
```

Ok, I think we're ready! When we move over... hmm... I don't see any products. Let's refresh to be sure. And... yes! There they are! Each has an image, title, price and button.

Computed price Property

But, hmm: these prices aren't right. I would *love* to be able to sell blank CDs for \$1,300... but that's not the *real* price. When you deal with prices, it's pretty common to store the prices in "cents", or whatever the lowest denomination of your currency is. The

point is, this is 2,300 cents, so \$23.

Yep, we have a formatting problem: we need to take this number, divide it by 100 and put the decimal place in the right spot. Yes, we have a situation where we need to render something that's *based* on a prop, but needs some *processing* first. Does that ring a bell? That's the *perfect* use-case for one of my absolute *favorite* features of Vue: computed properties!

Let's do this! Add a **computed** option with one computed property called **price()**. Inside, return **this.product.price** - to reference the price of the **product** **prop** and divide this by 100. Good start! To convert this into a string that always has two decimal points, we can use a fun JavaScript function that exists on any Number: **.toLocaleString()**. Pass this the locale - **en-US** or anything else - and then an options array with **minimumFractionDigits: 2**.

```
77 lines | assets/js/components/product-list/product-card.vue
... lines 1 - 34
35 <script>
36 export default {
... lines 37 - 43
44   computed: {
... lines 45 - 48
49     price() {
50       return (this.product.price / 100)
51         .toLocaleString('en-US', { minimumFractionDigits: 2 });
52     },
53   },
54 };
55 </script>
... lines 56 - 77
```

Pretty cool, right? I'll even add some docs to our function. I'm over-achieving!

```
77 lines | assets/js/components/product-list/product-card.vue
... lines 1 - 43
44   computed: {
45     /**
46      * Returns a formatted price for the product
47      * @returns {string}
48      */
49     price() {
... lines 50 - 51
52   },
53 },
... lines 54 - 77
```

Now that we have a computed property called **price**, we can use it with **{{ price }}**, as *if* **price** were a prop or data.

```
77 lines | assets/js/components/product-list/product-card.vue
1 <template>
... lines 2 - 15
16 <div class="p-2 my-3 d-md-flex justify-content-between">
17   <p class="p-0 d-inline">
18     <strong>{{ price }}</strong>
19   </p>
... lines 20 - 25
26 </div>
... lines 27 - 32
33 </template>
... lines 34 - 77
```

We know that computed properties - similar to **props** and **data** - are added as *properties* to the Vue instance, which is why we can say things like **this.price**. But behind the scenes, when we access this property, it will call our method. As a bonus, it even *caches* that property in case we refer to it multiple times.

Computed Properties with Arguments?

Oh, and by the way: this is *one* of the reasons why I created a specific component for rendering each product. If we did *not* have this component... and we were rendering this data inside the `ProductList` component, we wouldn't be able to use a computed property... because we would need to pass an *argument*: the product whose price we need to calculate. Instead, we would have needed to create a *method*... which isn't the end of the world, but is less efficient. Any time that you're creating a method to *return* data, it's a signal that you should considering refactoring into a smaller component that could use a computed property.

Anyways, now when we move over... we don't even need to refresh: there is our *beautiful* 30.00 price. What a bargain!

Smart and Dumb Components

Before we keep going, I want to circle back on a controversial decision I made earlier: the fact that we kept the `products` data inside `catalog.vue` even though the `product-list` component is *technically* the deepest component that needs it.

If you look at `catalog.vue`, it holds the AJAX call and pretty soon it will hold logic for a search bar. But... it doesn't render a lot of markup. I mean, yeah, it has an `<h1>` up here and a `<div>` down here, but its *main* job is to contain data and logic.

Compare this to the product-list component: `index.vue`. This doesn't have *any* logic! It receives props and renders.

Well... surprise! This separation was *not* an accident: it's a design pattern that's often followed in Vue and React. It's called smart versus dumb components, or container versus presentational components.

This pattern says that you should try to organize some components to be smart - components that make AJAX calls and change state - and other components to be dumb - that receive props, render HTML and maybe emit an event when the user does something.

`product-card` is another example of a dumb, or "presentational" component. Sure, it has a computed property to do some basic data manipulation, but this is just a component that receives a prop and renders, maybe with some minor data formatting.

To compare this to the Symfony world, one way to think about this is that a smart component is like a controller: it does *all* the work of getting the data ready. That might involve calling other services, but that's not important. Once it has all the data, it passes it into a template, which is like a dumb component. The template simply receives the data and renders it.

Like all design patterns, keep this in the back of your mind as a guide, but don't obsess over it. We're doing a good job of making this separation in some places, but we're not perfect either, and I think that's great. However, if you can *generally* follow this, you'll be happier with your components.

Next, now that we're loading data via AJAX, we need a way to tell the *user* that things are loading... not that our server is on fire and they're waiting for nothing. Let's create a Loading component that we can re-use anywhere.

Chapter 28: Loading Component

Google for "vue lifecycle hooks" and click into [The Vue Instance](#) page. About half way down, you'll find a spot that talks about "Instance Lifecycle Hooks".

If you look back at `catalog.vue`, "lifecycle hooks" is referring to things like the `mounted()` function, which is called when the component is added to the page.

[The Important Lifecycle Hooks](#)

If you scroll down a bit on the docs, they have a *huge* diagram that you can *really* nerd out on: this shows all the different things that happen between a Vue instance being instantiated, mounted onto the page, rendered, updated and removed from the page if you programmatically remove a component.

Feel free to study this - it's fun stuff. But I'll highlight the three most important hooks: `created`, `mounted` and `destroyed`, which is called after your component is completely removed.

[mounted vs created](#)

We used `mounted` earlier to start our AJAX call. That means that our Vue instance was created, mounted into the DOM, and *then* the function was called. It turns out that a better place to load data is actually `created`.

Let's try this: change `mounted` to `created` and then I'll refresh to be safe. That works *just* fine.

```
43 lines | assets/js/components/catalog.vue
... lines 1 - 23
24 export default {
... lines 25 - 35
36   async created() {
... lines 37 - 39
40   },
41 };
42 </script>
```

The `created()` function is called as *soon* as the Vue instance for our component is instantiated. That lets us start our AJAX call as *early* as possible. By the time it's mounted onto the page, the `products` data may or may *not* yet be available, probably they aren't. But it doesn't really matter. And we can see this when we refresh: the products are *missing* for a moment.

The point is: `created` is the best place to do data setup like this. And `mounted` is the correct hook if you need to do something that will manipulate the DOM.

[Creating the Loading Component](#)

Now even though this is loading a *tiny* bit faster, it's still not instant. And so, to give our users confidence that our server isn't on vacation, we need a loading message.

To help us have a *consistent* loading message whenever we need one, let's create a shiny new `Loading` component. Inside `components/`, add a new `loading.vue` file. Give this a `<template>` with an `<h4>` that says: - wait for it - `Loading...`. That's good writing.

Let's also give this a class: `:class="$style.component"`.

```
19 lines | assets/js/components/loading.vue
1  <template>
2    <h4 :class="$style.component">
3      Loading...
4    </h4>
5  </template>
... lines 6 - 19
```

Before we *add* that style, create the `<script>` tag with `export default` and the options object. This only needs a `name` key set to `Loading`.

```
19 lines | assets/js/components/loading.vue
... lines 1 - 6
7  <script>
8  export default {
9    name: 'Loading',
10 };
11 </script>
... lines 12 - 19
```

Now let's add the `<style>` tag with `lang="scss"` and `module`. Add just one class: `.component`.

```
19 lines | assets/js/components/loading.vue
... lines 1 - 12
13 <style lang="scss" module>
14 .component {
... lines 15 - 16
17 }
18 </style>
```

Referencing an Image in CSS

If you look at the `assets/` directory, it has an `images/` directory with a `loading.gif`. In the `.component` class, we're going to set this as the background image. We can do that with `background: url()` and then the relative path to the file from here: `../../images/loading.gif`. We could also add a Webpack alias for `images/` if we want. Finish this with `no-repeat left center` and add a little padding to get the positioning just right.

```
19 lines | assets/js/components/loading.vue
... lines 1 - 13
14 .component {
15   background: url('../../images/loading.gif') no-repeat left center;
16   padding: 0 0 4px 50px;
17 }
... lines 18 - 19
```

Say hello to our *super* fancy loading component!

Using the Component

Over in `index.vue`, time to put it to work! Start by adding some markup to hold the loading message.

```
36 lines | assets/js/components/product-list/index.vue
1  <template>
2    <div class="row">
3      <div class="col-12">
4        <div class="mt-4">
5          ... line 5
6        </div>
7      </div>
8    </div>
9  </template>
10 ... lines 17 - 36
```

Next, import it with `import Loading from '@components/loading'` and add `Loading` to `components` : the order doesn't matter.

```
36 lines | assets/js/components/product-list/index.vue
18 <script>
19 import Loading from '@components/loading';
20 ... lines 20 - 34
35 </script>
```

Finally, celebrate in the template with: `<Loading />` .

```
36 lines | assets/js/components/product-list/index.vue
1  <template>
2    ... lines 2 - 3
3
4    <div class="mt-4">
5      <loading v-show="products.length === 0" />
6    </div>
7  ... lines 7 - 15
16 </template>
17 ... lines 17 - 36
```

We're not *conditionally* hiding and showing that yet but... there it is! Not bad!

[Hiding / Showing the Loading Animation](#)

Ok: we only want to show the `Loading` component when the products AJAX call hasn't finished. The two different ways to conditionally hide or show something are `v-show` and `v-if` . In this case, especially because we're *eventually* going to be loading the product list multiple times when we have a search bar, let's use `v-show` so we can hide & show it quickly. Add `v-show=""` . And, let's see: the easiest way to know if the products are still loading is to check if `products.length === 0` .

```
36 lines | assets/js/components/product-list/index.vue
1  <template>
2    ... lines 2 - 4
3
4    <loading v-show="products.length === 0" />
5  ... lines 6 - 15
16 </template>
17 ... lines 17 - 36
```

That's not a *perfect* solution - we'll see why later - but it's good enough for now. And when we reload... that's nice!

We can also add a `v-show=""` on the `product-card` element with `products.length > 0` . It's not *really* needed since this won't even loop if there are no products, but it balances things.

We now have dynamic products *and* a loading animation while the AJAX call is finishing. I'm super happy about that! But our categories are *not* dynamic yet. Wah, wah. Let's fix that next. But after we do, we'll explore a *faster* way to load them.

Chapter 29: Dynamic Categories via AJAX

I've gotta say, now that we have dynamic products, these hardcoded categories are starting to *stress me out!* So let's make those dynamic too!

If you look at `/api`, just like how we have a `/api/products` endpoint for the collection of products, we also have one for `/api/categories`. We can use that! And now that we know the correct way to make Ajax calls to load data, this "should" be simple! Famous last words!

Add an Ajax Call to Sidebar

Open `sidebar.vue`. We created a `data` property earlier called `categories`, but... it's just hardcoded. Set this to an empty array to start.

```
73 lines | assets/js/components/sidebar.vue
... lines 1 - 42
43 <script>
44 export default {
... lines 45 - 51
52 data() {
53   return {
54     categories: [],
55   };
56 };
57 };
58 </script>
... lines 59 - 73
```

To make the AJAX call, head over to `catalog.vue` to celebrate one of programming's oldest arts: stealing code. Copy the entire `created()` function and paste it in `sidebar.vue` under `data`. We just need to change the URL to `/api/categories` and the data from `this.products` to `this.categories`.

```
80 lines | assets/js/components/sidebar.vue
... lines 1 - 42
43 <script>
44 import axios from 'axios';
... line 45
46 export default {
... lines 47 - 58
59 async created() {
60   const response = await axios.get('/api/categories');
61
62   this.categories = response.data['hydra:member'];
63 },
64 };
65 </script>
... lines 66 - 80
```

Awesome! Let's... just see what happens! Refresh... whoa! I think it worked! No errors in the console... and in the dev tools, if you click on `Sidebar`, we have real `categories` data!

The data for each category is pretty simple: it has the normal `@id` and `@type` that comes from JSON-LD and it also has `id` and `name`.

Oh, and one thing I forgot to mention: when we pasted the `created()` function, PhpStorm *automatically* added the `axios` import for us. Sweet! Oh, but it used double quotes... which ESLint does *not* like. Let's fix this. Much better! If you want, you can tweak your PhpStorm settings to use single quotes automatically.

[v-for index and :key](#)

Anyways, head up to the `v-for` directive. We learned earlier that every `v-for` element must have a `key` attribute. Until now, since our categories were hardcoded, we used the array `index` for the `:key`. But now we can be *smarter* because we know that each category has a unique `@id` property.

Let's simplify the `v-for` - we don't need `index` anymore - and then say `:key="category['@id']"`.

```
80 lines | assets/js/components/sidebar.vue
1  <template>
  ... lines 2 - 7
8  <ul class="nav flex-column mb4">
  ... lines 9 - 15
16  <li
17    v-for="category in categories"
18    :key="category['@id']"
  ... line 19
20  >
  ... lines 21 - 27
28  </ul>
  ... lines 29 - 40
41 </template>
  ... lines 42 - 80
```

[Linking Properly](#)

The last thing that doesn't work is... our links! We originally set the `href` to `category.link` ... but there is *no* `link` property on the real category data.

Here's the plan: we will eventually create a separate page that will display all of the products for a specific category - the URL will be `/category/` and then the `id` of the category. We'll worry about making sure that page exists later, but let's get the links working now.

If you're a Symfony user, then you're used to *generating* a URL to a route. But from JavaScript, I'm going to keep it simple and hardcode the URL to this new page. I think it's *totally* ok to do this: it keeps your JavaScript simpler. The tradeoff, of course, is that if you ever changed a URL, you would need to update your JavaScript.

Ok: what we want to do here is say `/category/` and then print `category.id`. But since we have the colon in front of `href` to make this dynamic, we would need to have single quotes around the string, a plus sign and then `category.id`.

That *would* work: in the browser, when I hover over categories, the right URL *is* display at the bottom of my browser.

[JavaScript Dynamic Strings](#)

But yikes! This code *hurts* to look at! Can we make it nicer? Of course! Replace the quotes with `ticks`. Now, if we need dynamic code, write `${}` - so `${category.id}`.

```
80 lines | assets/js/components/sidebar.vue
... lines 1 - 7
8      <ul class="nav flex-column mb4">
... lines 9 - 15
16      <li>
... lines 17 - 19
20      >
21      <a
22        :href="`/category/${category.id}`"
23        class="nav-link"
24      >
... line 25
26      </a>
27    </li>
28  </ul>
... lines 29 - 80
```

This is a superpower of modern JavaScript, not Vue. Look over now and... yes! All of the links look *perfect*!

[Too much Dynamic Data!](#)

This is working nicely! We have dynamic products and dynamic categories. The *only* thing that bothers me is all the loading! Notice that each time we refresh, I see the products *and* categories loading!

When we only loaded the products, that was probably okay: it was just one spot that loaded pretty fast. But *also* having the categories waiting to load is starting to look a bit jarring. Plus, we're eventually going to have multiple pages that will use the same categories sidebar. This means that on *every* page, the user will wait for the *same* list of categories to be fetched via AJAX. We can do better!

So next, let's investigate how we can get data from the server in a way that *avoids* an AJAX call. We'll use this at *first* to highlight the current category on the sidebar. Then later, we'll replace the AJAX call completely.

Chapter 30: Passing data From the Server to Vue

In our sidebar, we're looping over the categories and creating a link for each one to `/category/` plus the category `id`. When we did this, I said that this would link to a *future* page that we would create. Well... guess what! If you click on `office supplies`, this... actually *works*! Well, sort of: the URL changed but... it looks like it just loaded the same page.

Let me show you what's going on: I've done a little bit of work behind the scenes. In `src/Controller/ProductController.php`, we have an `app_homepage` route that renders a `product/index.html.twig` template. This is the page that we've been using so far. Open `templates/product/index.html.twig` so we can see it. Nothing special here: we have our target `div` for the Vue app, and the `script` and `link` tags for our `products` Webpack entry.

Head back to the controller. Below this, we have *another* route called `app_category`. That's why this page works! I've already created a route and controller for `/category/{id}` that loads the *same* Twig template as our other page!

[About Multiple Page Applications](#)

So here's the idea: we're purposely *not* building a single page application. Part of the reason is that multi page applications are, in a lot of ways, *trickier* to work with in Vue than single page apps. And also, it's *totally* legal to have a traditional multi page application with Vue mixed in only where you need it.

In this app, we're going to have a homepage - which is basically the `All Products` category - where you can click on any category to go to a *totally* different page. That page will render the *same* Vue app, but only show products for *that* category. And in a future tutorial, we're also going to create a "product page" that will do the same thing: be a separate URL that's handled by the same Vue app.

So yes: our *one* Vue app will behave *differently* based on which page we're on.

To achieve that, when we're on a specific category page, our Vue code needs to do two special things.

One: we're probably going to want to highlight *which* category we're on in the sidebar so that the user knows that we're on the `office supplies` category and not in `All Products`, for example.

And two: we're going to need to filter the product list because, right now, no matter what category I click, I am *always* getting the same list of products. We need to somehow *realize* that we are on a *category* page and then use that information to make an API request for *only* products from that category.

[Server Variables](#)

How can we do that? I have no idea! I'm kidding! In some ways, it's a simple problem: the one piece of information that we need to know in our Vue code is what category ID we are currently on. Are we on *no* category ID? Meaning: show all products? Or are we on category ID 23?

To do that, we need to communicate this information from the server *to* Vue. There are multiple ways to do this, but my favorite approach is to set a global JavaScript variable in the template and then *read* that from inside of our Vue app.

[Adding the Category @id to our Controller](#)

Start by adding a second argument to the `showCategory()` action. Don't worry, we're not going to go *too much* into Symfony and I'll explain what I'm doing along the way. Add `IriConverterInterface $iriConverter` to get a service from API Platform.

41 lines | src/Controller/ProductController.php

```
... lines 1 - 5
6 use ApiPlatform\Core\Api\IriConverterInterface;
... lines 7 - 11
12
13 class ProductController extends AbstractController
14 {
... lines 15 - 25
26 public function showCategory(Category $category, IriConverterInterface $iriConverter): Response
... lines 27 - 39
40 }
```

Remember: when you click on the **Catalog** component in the Vue Dev tools and look at the **products** data, each item that we get back from our API - whether it's a product, category or something else - has an **@id** property. That is known as the IRI. I like to use this IRI string instead of the integer ID from the database because it's more useful: it's a real, functional URL!

So on the category page, instead of setting a JavaScript variable that says we're on category **23**, I'm going to use the IRI of that category. This **\$iriConverter** will help me get that.

Add a second argument to the template called **currentCategoryId** - in reality, this is the "current category IRI" - set to **\$iriConverter->getIriFromItem()** and then pass the **\$category** object.

41 lines | src/Controller/ProductController.php

```
... lines 1 - 25
26 public function showCategory(Category $category, IriConverterInterface $iriConverter): Response
27 {
28     return $this->render('product/index.html.twig', [
29         'currentCategoryId' => $iriConverter->getIriFromItem($category),
30     ]);
31 }
```

Printing JavaScript Values Directly in our Template

Obviously, we're going to use **currentCategoryId** in the template. But when we do that, we need to be careful: there are *two* pages that render the *same* template... and the **currentCategoryId** variable will only be available on *one* of those pages, not both.

In **index.html.twig**, above where I'm rendering my **products** JavaScript, add a **script** tag and say **{% if currentCategoryId is defined %}** with **{% else %}** and **{% endif %}**. When the variable *is* defined, use it to set a global JavaScript variable: **window.currentCategoryId =**, a set of quotes, and then **{{ currentCategoryId }}**.

That's it! Oh, but in theory, if **currentCategoryId** contained a single quote, this would break. To be *extra* safe, pipe this to **e('js')**.

26 lines | templates/product/index.html.twig

```
... lines 1 - 12
13 {% block javascripts %}
... lines 14 - 15
16 <script>
17     {% if currentCategoryId is defined %}
18         window.currentCategoryId = '{{ currentCategoryId|e('js') }}';
19     {% else %}
... line 20
21     {% endif %}
22 </script>
... lines 23 - 24
25 {% endblock %}
```

That will escape the string so that it's always safe for JavaScript. In the **else**, if no current category is set, that means we're on the homepage. Let's say **window.currentCategoryId = null;**


```
26 lines | templates/product/index.html.twig
... lines 1 - 16
17     {% if currentCategoryId is defined %}
... line 18
19     {% else %}
20         window.currentCategoryId = null;
21     {% endif %}
... lines 22 - 26
```

The end result of this *long* journey is that when we refresh the page, view the source and scroll down to where the JavaScript is... yes! We have `window.currentCategoryId = '/api/categories/23'`. The back slashes are just escaping the forward slashes.

Next, let's use this global variable in our Vue code to highlight which category we're on in the sidebar.

Chapter 31: Reading Server Data & Global Classes

We just set a global `currentCategoryId` variable in JavaScript. Let's use this in our Vue app to highlight which category we're currently viewing.

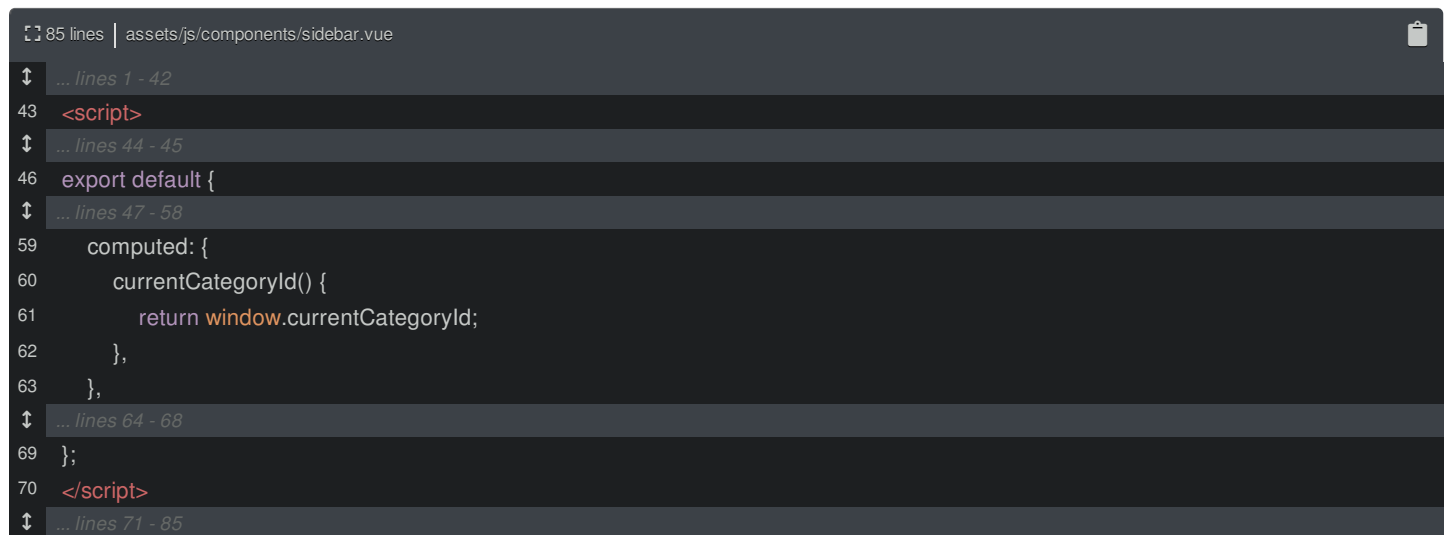
Open up `sidebar.vue`. How can we get the current category info here? The simplest way is... to reference the global variable! We *will* talk soon about ways to organize this, but since we created a global variable in Twig, we can absolutely use that here!

Well... we can't use it *immediately* in our Vue template. If we tried to use `currentCategoryId` to add a dynamic class, Vue would think that we were trying to reference `this.currentCategoryId`.

We have two options. First, we could add a `currentCategoryId` *data* and initialize it to `window.currentCategoryId`. That would be *fine*. But because I don't intend for this value to *change* while my Vue app is running - at least not yet - it doesn't *really* need to live as data, though it wouldn't hurt anything,

Instead, let's leverage a computed property... which is a *great* way to expose any extra variables you need in a template.

Below `data`, add `computed` and create a new computed property called `currentCategoryId`. Inside, `return window.currentCategoryId`.



```
85 lines | assets/js/components/sidebar.vue
... lines 1 - 42
43 <script>
... lines 44 - 45
46 export default {
... lines 47 - 58
59   computed: {
60     currentCategoryId() {
61       return window.currentCategoryId;
62     },
63   },
... lines 64 - 68
69 };
70 </script>
... lines 71 - 85
```

It's just that simple!

[Adding the .selected Style](#)

Before we use this in the template, let's add a new class that we can use for the "selected" category. Inside the `ul`, create a new style called `li a.selected`. This is the class that we will need to dynamically add in the template. Inside, say `background: $light-dash-component-border;`.

```

95 lines | assets/js/components/sidebar.vue
... lines 1 - 77
78 <style lang="scss" module>
... lines 79 - 80
81 .component {
... lines 82 - 83
84   ul {
... lines 85 - 88
89     li a.selected {
90       background: $light-component-border;
91     }
92   }
93 }
94 </style>

```

This comes indirectly from the `light-component` mixin we're using - it's actually a variable set in `colors.scss` .

Ok! Let's use this `selected` class up in the template. Start with the "all products" item: this should have the `selected` class *if* the `currentCategoryId` is `null` . And it should *always* have the `nav-link` class.

To do that, change this to `:class` and set it to an *object*. Inside, add a key called `nav-link` set true so that it *always* shows up. For the dynamic class, add `[$style.selected]` - to reference our new `selected` class - and make that render if `currentCategoryId === null` .

```

95 lines | assets/js/components/sidebar.vue
1 <template>
... lines 2 - 8
9   <li class="nav-item">
10     <a
11       :class="{
12         'nav-link': true,
13         [$style.selected]: currentCategoryId === null,
14       }"
... line 15
16     >All Products</a>
17   </li>
... lines 18 - 46
47 </template>
... lines 48 - 95

```

Remember: the ugly square bracket syntax is needed so that JavaScript knows that our key is a dynamic expression. That's... unfortunate, but we're going to fix that in a minute anyways!

Now copy the `:class` attribute and, down inside the loop, paste over the existing `class` . In this case, we want to show the class if `category['@id'] === currentCategoryId` .

```
95 lines | assets/js/components/sidebar.vue
1 <template>
  ... lines 2 - 18
19 <li
20   v-for="category in categories"
  ... lines 21 - 22
23 >
24 <a
  ... line 25
26   :class="{
27     'nav-link': true,
28     [$style.selected]: category['@id'] === currentCategoryId,
29   }"
30 >
31   {{ category.name }}
32 </a>
33 </li>
  ... lines 34 - 46
47 </template>
  ... lines 48 - 95
```

Testing time! Back on the browser... yes! It already works! I'm on the "Office Supplies" page and that category *is* highlighted! Let's click on "All Products" and... it works *beautifully*!

So even though we don't have access to the global variable directly in our template, it's very simple to create a computed property that grabs that *for* us and makes it available!

SASS Globals

There's *one* thing I want to improve before we talk about a *better* way to manage global variables.

It bothers me a *little bit* that I have to use `$style.selected` ... especially because I'm forced to use the ugly `[]` syntax! If you look down at our CSS, we're already inside of a modular `.component` class. On the top of our template: yep! We're using `$style.component` on the root element.

In SASS, because I put the `.selected` class *inside* of `.component`, the style will only apply to elements that are *inside* the root element.

Move over to your browser and "inspect element" on the selected link. Not surprisingly, this has a modular `selected` class: `sidebar_selected_` and then a dynamic hash. But check out how the CSS is generated for this: it's `.sidebar_component_` a hash and then `ul li.sidebar_selected_` and *another* hash! We don't actually need that second hash! The first `.sidebar_component_` hash selector is *already* enough to make sure that our `selected` class doesn't affect anything else on the page.

So here's what I would *love* to be able to do: I want to be able to write CSS like we're doing now - but *not* have any classes inside `.component` render in a modular way. Up in the template, I want to be able to just type `selected` and have it work.

```
95 lines | assets/js/components/sidebar.vue
1 <template>
  ... lines 2 - 9
10 <a
11   :class="{
  ... line 12
13     'selected': currentCategoryId === null,
14   }"
  ... line 15
16 >All Products</a>
  ... lines 17 - 46
47 </template>
  ... lines 48 - 95
```

Make our child style classes into globals

And here's how: down in the style tag, you can tell SCSS:

Hey! I'm using a `selected` class, but I *don't* want you to treat this like a modular class: I *don't* want you to add the prefix and hash.

The way you do that is by adding `:global` in front of it.

```
95 lines | assets/js/components/sidebar.vue
... lines 1 - 77
78 <style lang="scss" module>
... lines 79 - 80
81 .component {
... lines 82 - 83
84   ul {
... lines 85 - 88
89     :global li a.selected {
... line 90
91   }
92 }
93 }
94 </style>
```

As *soon* as we do that, if I inspect element on the selected link and look at the generated CSS selector on the right... yes! It has the modular `sidebar` class but *then* it only says `.selected`. We're not worried about that affecting any *other* parts of the page because it's *still* inside the modular sidebar class.

And... we can go one step further! If you think about it, because `.component` will be converted into a modular class, we don't need *any* classes inside of it to be modular. We can move the `:global` up and after the `.component` class.

```
95 lines | assets/js/components/sidebar.vue
... lines 1 - 80
81 .component :global {
... lines 82 - 83
84   ul {
... lines 85 - 88
89     li a.selected {
... line 90
91   }
92 }
93 }
... lines 94 - 95
```

Now *everything* inside will *not* be modular. This means that the *only* time I should have to use the `$style` variable is on the root element! The rest can be normal classes. This already looks much better!

Next, we have successfully referenced a global variable via a computed property. But I don't *love* having global variables hidden inside my code. We can do this in a better, more organized way, by refactoring it into a service!

Chapter 32: Page Context "Service"

Setting a global variable in JavaScript and then reading it from inside our Vue app is, really, a nice way to communicate information from our server to our front-end app. But global variables are still global variables and we should *try to at least* isolate and organize them as much as possible. Because, for example, what if we changed our app to use the Vue router? Instead of full page refreshes where we set the `currentCategoryId` as a global variable in Twig, *now* that data would be returned in a *different* way: via an AJAX call.

The point is: the way we get `currentCategoryId` could change. And if we have `window.currentCategoryId` sprinkled around our code everywhere, it's... not ideal. No problem! Let's isolate our global variable into a central spot. Enter JavaScript services!

JavaScript Services

I think you're *really* going to like this. Inside my `js/` directory, create a new folder called `services/`. So far, everything we've worked on has been Vue components... but there's a lot more to our app. We have code for making AJAX calls - which we will eventually centralize - and we're also going to have generic logic that we want to reuse from multiple places.

In our app, the `services/` directory is going to hold files that help us fetch data. So, it's a bit different than services in Symfony, which are any classes that do work. By services here, I mean API services... though you could also fetch data from local storage... or even by reading a global variable that we set in the template! Those are *all* sources of data.

Creating the page-context service

So inside `services/`, create a new file called `page-context.js`. I *totally* invented that name: the purpose of this file will be to help read data related to what "page" we're on - like the current category id.

Inside, instead of exporting a default, like we've been doing so far with Vue, we're going to export *named* functions. As we start adding more functions to this file, you'll see how we can use whichever one or two functions we need. Say `export function`, call it `getCurrentCategoryId` and, inside, *very* simply, `return window.currentCategoryId` !

```
9 lines | assets/js/services/page-context.js
1  /**
2   * Returns the current category id that's set by the server.
3   *
4   * @returns {string|null}
5   */
6  export function getCurrentCategoryId() {
7    return window.currentCategoryId;
8  }
```

Just like that! We have a central place to read our global variable! I'll celebrate by adding some JSDoc above this.

Thanks to this, if we get this information some *other* way in the future, we will *only* need to update code in this one spot. I love that!

Now, in `sidebar.vue`, we can use this like any normal JavaScript module. But I'm going to type this import a little backwards: `import from '@services/page-context'`. I left the part after `import` blank, which is totally *not* valid JavaScript. But now, I can add `}` and autocomplete the `getCurrentCategoryId` function.

```
96 lines | assets/js/components/sidebar.vue
↑ ... lines 1 - 48
49 <script>
↑ ... line 50
51 import { getCurrentCategoryId } from '@services/page-context';
↑ ... lines 52 - 76
77 </script>
↑ ... lines 78 - 96
```

Nice! Down in the computed property, use this: `return getCurrentCategoryId()` .

```
96 lines | assets/js/components/sidebar.vue
... lines 1 - 52
53 export default {
... lines 54 - 65
66   computed: {
67     categoryId() {
68       return getCurrentCategoryId();
69     },
70   },
... lines 71 - 75
76 };
... lines 77 - 96
```

That is lovely! When we move over and refresh... it works!

[Loading categoryId in the Correct Component](#)

I'm really happy that we've centralized this global variable into a shiny new module. But I want to do just a *little* bit of future proofing. In a real app, you may or may not choose to do this - but it'll be a good exercise and it will help us later.

Here we go: `currentCategoryId` is *not* something that will change while our app is running. Because, when we click on a different category, the page refreshes and the entire Vue app restarts. For the duration of our page view, `currentCategoryId` *never* changes!

This means that it *isn't* something that *needs* to live in `data` : we don't need anything to re-render when it changes. That's why it's *totally* legal to grab this value in `sidebar.vue` or anywhere else that needs it.

But I want to *kind of* future proof our app... and plan ahead for a future where `currentCategoryId` *will* change while my Vue app is running. If you pretend that `currentCategoryId` *did* need to be in `data` , what component would that `data` live in?

Remember: the answer to this question is always find the *deepest* component that needs the data. If I look in `products.vue` , we know that `sidebar` needs to know the `currentCategoryId` so that it can highlight that category. And `catalog` is *also* going to need to know the current category soon so that we can print the category title *and* filter the product list to show only those in that category.

This means that if `currentCategoryId` were data, it would need to live on the `products` components so that we could pass it down into `sidebar` and `catalog` as a prop.

[Replace categoryId Computed with a Prop](#)

Now, I don't *actually* want to turn the `currentCategoryId` into `data` right now because... I don't *need* to. But I *do* want to structure my app with this in mind. To start, copy the computed property from `sidebar.vue` , and, in `products.vue` add it there.

```
52 lines | assets/js/pages/products.vue
... lines 1 - 17
18 <script>
... lines 19 - 20
21 import { getCurrentCategoryId } from '@services/page-context';
... line 22
23 export default {
... lines 24 - 33
34   computed: {
... lines 35 - 40
41     categoryId() {
42       return getCurrentCategoryId();
43     },
44   },
... lines 45 - 49
50 };
51 </script>
```

Oh, and this is cool! When I pasted, check it out! It added the import for me automatically! It *did* mess up the code style, but you can fix that in PhpStorm if you want. That's better.

So, instead of having `currentCategoryId` as data, we will have it as a computed property... but *inside* the component where it *would*, in theory, need to live as data. That will make it *super* easy to *change* to data later if we need to.

Now, pass this to sidebar with `:current-category-id="currentCategoryId"`.

```
53 lines | assets/js/pages/products.vue
1  <template>
  ... lines 2 - 4
5  <sidebar
  ... line 6
7    :current-category-id="currentCategoryId"
  ... line 8
9  />
  ... lines 10 - 16
17 </template>
  ... lines 18 - 53
```

And in `sidebar.vue`, instead of a computed property, we'll set this as a prop. Add `currentCategoryId` with type `String` - this is the IRI string - and also `default: null`.

```
94 lines | assets/js/components/sidebar.vue
  ... lines 1 - 48
49 <script>
  ... lines 50 - 51
52 export default {
  ... line 53
54   props: {
  ... lines 55 - 58
59     currentCategoryId: {
60       type: String,
61       default: null,
62     },
63   },
  ... lines 64 - 73
74 };
75 </script>
  ... lines 76 - 94
```

The reason I'm using default `null` is that this will allow the prop to be a `String` *or* `null`, which is what it will be on the homepage. You can add more customized prop validation if you want... but this is good enough for me!

If you scroll down, our `currentCategoryId` computed property is angry! It says duplicate key `currentCategoryId` because we don't want to have this as a prop and *also* as a computed prop. Delete the computed property... and we can also delete the import to celebrate. Our code is happy!

[Check it in the browser!](#)

Moment of truth! When we move over... yes! It's *still* working. Yay for not breaking things!

If you're not sure *why* we did this, here's what's going on. By moving `currentCategoryId` up to `products.vue` and passing it as a prop to `sidebar`, it would now be *very* easy for us to change the `currentCategoryId` computed prop into `data`. In fact, if we did that, everything else would... well... magically not break!

Next, let's get to work on the `catalog` component. Let's pass `currentCategoryId` as a prop so we can filter the list of products to *only* those for that category.

Chapter 33: Filtering the Products

Ok team! We have the `currentCategoryId` and we're using it on the sidebar to highlight which page we're on. That is *awesome!* But our products on the right aren't being *filtered* at all yet! Yikes!

Go to `/api` to check out our API docs and scroll down to the `GET /api/products` endpoint. I've already done a bit of work behind the scenes in API platform to allow us to fetch all the products for a specific category.

Here's how it works: we can enter a category's IRI into this box: so, `/api/categories/` - and then 24 to get the Furniture category. When we hit Execute... yes! It only returned 5 items! And if I scroll up, you can see the URL that gave us this result: if you ignore the url-encoded parts, this is `/api/products?category=/api/categories/24` .

All we need to do is make that same request from inside of Vue!

Passing `currentCategoryId` as a Prop

Head back to our app. From the top level `products.vue` , hold Command or Ctrl and click `<catalog />` to jump into that component. Catalog is responsible for loading our products on `created` by making a request to `/api/products` . Hmm... I *wish* we had access to the `currentCategoryId` here... because we could use that to change the URL in the AJAX call! Well then... let's go get it!

Back in `products.vue` , add `:current-category-id="currentCategoryId"` to pass it as a prop... *just* like we did with the sidebar.

```
53 lines | assets/js/pages/products.vue
1  <template>
  ... lines 2 - 11
12    <div :class="contentClass">
13      <catalog :current-category-id="currentCategoryId" />
14    </div>
  ... lines 15 - 16
17 </template>
  ... lines 18 - 53
```

Now, in `catalog.vue` , *add* that as a prop. Actually, let's go steal the prop definition from sidebar - it's perfect there - and paste it here.

```
56 lines | assets/js/components/catalog.vue
  ... lines 1 - 18
19 <script>
  ... lines 20 - 23
24 export default {
  ... lines 25 - 29
30   props: {
31     currentCategoryId: {
32       type: String,
33       default: null,
34     },
35   },
  ... lines 36 - 53
54 };
55 </script>
```

Using `currentCategoryId` to Filter on the AJAX call

Wonderful! We are *now* receiving `currentCategoryId` . To use this, down in `created` we *could* add `?category=` and then the `currentCategoryId` . But with Axios, there's a *better* way. Create a new `params` variable set to an object: this will hold all the query parameters that we want to send. Now, `if (this.currentCategoryId) {` , then `params.category = this.currentCategoryId` .

```
56 lines | assets/js/components/catalog.vue
... lines 1 - 23
24 export default {
... lines 25 - 41
42   async created() {
43     const params = {};
44     if (this.currentCategoryId) {
45       params.category = this.currentCategoryId;
46     }
... lines 47 - 52
53   },
54 };
... lines 55 - 56
```

To pass that to axios, add a second parameter, which is an options object. One of the options you can pass is called `params`. So: `params: params, .`

```
56 lines | assets/js/components/catalog.vue
... lines 1 - 41
42   async created() {
... lines 43 - 47
48     const response = await axios.get('/api/products', {
49       params: params,
50     });
... lines 51 - 52
53   },
... lines 54 - 56
```

Object Shorthand: Keys without the Key

That *should* work... but yikes! ESLint is *mad*! It says:

```
expected property shorthand.
```

This is referring to something that we've actually already done many times, but I want to highlight it in case you haven't noticed. In JavaScript, if you are trying to add a property to an object... and that property's name is the same as the *variable* being used for its value, you can use a shorthand syntax: just `params`. This sets a property *named* `params` to the `params` variable.

Check it out!

If we go over now, you can already see it! It reloaded! Yes! Furniture shows furniture! Breakroom shows breakroom products! Office supplies shows office supplies! And Snacks shows... uh... did this just break? Where are my snacks?

It turns out that our snacks category is currently empty! Gasp! If that's not bad enough, instead of saying - "No snacks! You're on your own!" - we see the loading screen forever.

That's... my fault. When `catalog` renders `product-list` - hold Command or Ctrl and click to jump to that - the loading is showing based on whether the `products` length is zero or not. An empty category *looks* like it's still loading!

We *need* to improve this: we need to *truly* know whether or not the AJAX call has finished! Let's make a *smarter* loading mechanism next!

Chapter 34: Smarter Loading: AJAX status as State

Oh no! The snacks category is *empty*! That's a *huge* problem on its own! To make things worse, you can't even tell! It looks like it's loading *forever*... while I'm sitting here getting hungrier and hungrier.

The reason is that, in the `product-list` component, we're showing the loading animation by checking if the products length is zero.

Using the length of an array to figure out if something is done loading doesn't really work if it's *possible* that the thing really *can* be empty in some cases! And that's *totally* the situation we have: sometimes a category *has* no products. And later when we add a search, sometimes no products will match.

So... the easy solution didn't work. What we need *instead* is a flag that *specifically* tracks whether or not the products AJAX call is finished.

[Add loading to Catalog](#)

We know Catalog is the smart component that takes care of making the AJAX request. This means that it is *also* aware of whether or not we are *currently* making an AJAX call for the products. To track this, let's add a new data: call it `loading` and set it to `false` by default.

```
57 lines | assets/js/components/catalog.vue
... lines 1 - 18
19 <script>
... lines 20 - 23
24 export default {
... lines 25 - 35
36   data() {
37     return {
... line 38
39       loading: false,
... line 40
41     };
42   },
... lines 43 - 54
55 };
56 </script>
```

Now, very simply, in `created()`, say `this.loading = true` right before the AJAX call and, right after, `this.loading = false`.

And *just* like that, we have a flag that we can use to render things based on the *true* loading status!

```
60 lines | assets/js/components/catalog.vue
... lines 1 - 42
43 async created() {
... lines 44 - 48
49   this.loading = true;
50
51   const response = await axios.get('/api/products', {
52     params,
53   });
54
55   this.loading = false;
... line 56
57 },
... lines 58 - 60
```

[Try...catch](#)

While we're here, we can *also* add some simple error handling *just* in case the AJAX call fails. To do that, wrap all of this in a `try...catch` block. Then, inside `catch`, set `this.loading = false`.

```
64 lines | assets/js/components/catalog.vue
... lines 1 - 42
43   async created() {
... lines 44 - 50
51     try {
... lines 52 - 57
58   } catch (e) {
59     this.loading = false;
60   }
61 },
... lines 62 - 64
```

If we *really* didn't trust our API... we could add a data called `error`, change that in `catch` to a message and render it. But with this, we will *at least* fail *somewhat* gracefully, and avoid the loader from spinning forever.

As easy as that was, this could be a bit dangerous! The problem right now is that if *any* of these lines have an error - like if our response doesn't have a `data` key on it - then the `catch` will be called and we will *not* show it. We could be hiding a bug in our code. I *hate* bugs! I think pizza is much better...

So instead, above the `try`, add `let response`. This simply *declares* the variable outside of the `try...catch` scope so that it's available in the entire `created` function. Now, remove the `const` from `response` and then I'll `return` from the `catch`. So if we hit the `catch`, just exit. Finally, move the `this.products = response.data` code outside of the `catch`. Now if *that* line has a problem, it won't be silenced: we'll have to deal with it!

```
68 lines | assets/js/components/catalog.vue
... lines 1 - 42
43   async created() {
... lines 44 - 48
49     this.loading = true;
50
51     let response;
52     try {
53       response = await axios.get('/api/products', {
54         params,
55       });
56
57     } catch (e) {
58       this.loading = false;
59     }
60
61     return;
62
63     this.products = response.data['hydra:member'];
64   },
... lines 66 - 68
```

Whether or not you should use the `try...catch` just depends on your situation. I probably *wouldn't* do this because, if my API endpoint is failing, I have bigger problems: my site is broken! Giving the user a graceful error is nice, but maybe I'll save that for V2.

However, if you *do* have a valid situation where an AJAX request might fail - like if you're *sending* data to the server that might fail validation - then *this* is how you can catch that error and deal with it. We'll talk about sending data in the next tutorial.

[Pass loading down to product-list](#)

Okay, we now have the `loading` data on our smart catalog component. Let's pass that into the `product-list` component so that we

can use it to hide or show the loading spinner. Split the `product-list` onto multiple lines and then add `:loading="loading"` .

```
71 lines | assets/js/components/catalog.vue
1  <template>
2    <div>
3  ... lines 3 - 10
11    <product-list
12      :products="products"
13      :loading="loading"
14    />
15  ... lines 15 - 18
19    </div>
20  </template>
21  ... lines 21 - 71
```

And now that we're passing the `loading` prop, in `index.vue` , update the `props` so we can receive it: add a new `loading` prop with `type: Boolean` and `required: true` .

```
40 lines | assets/js/components/product-list/index.vue
18 <script>
19 ... lines 19 - 21
22 export default {
23 ... lines 23 - 27
28   props: {
29     loading: {
30       type: Boolean,
31       required: true,
32     },
33 ... lines 33 - 36
37   },
38 };
39 </script>
```

We can *now* simplify the template: we want to show the loading animation *if* `loading` is true. And we also want to show these product cards down here, if we are `!loading` . This second spot isn't *super* important, but it doesn't hurt to have it!

```
40 lines | assets/js/components/product-list/index.vue
1  <template>
2    <div class="row">
3      <div class="col-12">
4        <div class="mt-4">
5          <loading v-show="loading" />
6        </div>
7      </div>
8
9      <product-card
10 ... line 10
11        v-show="!loading"
12 ... lines 12 - 13
14      />
15    </div>
16  </template>
17  ... lines 17 - 40
```

[Adding a "No Products" Message after Loading](#)

Time to check things out! Yep! You can *already* see that the snacks page no longer has the loading spinner. And my other pages work *just* fine.

Well... except it would be even *better* with a "no products found" message! And now, we can easily add that.

After the `<loading />` component, add an `h5` with a `v-show` directive. This will hold that "no products found" message... which means that we want it to show if we are *not* loading but `products.length === 0`.

```
47 lines | assets/js/components/product-list/index.vue
1  <template>
2  ... lines 2 - 3
4      <div class="mt-4">
5          <loading v-show="loading" />
6
7          <h5
8              v-show="!loading && products.length === 0"
9              class="ml-4"
10         >
11             Whoopsie Daisy, no products found!
12         </h5>
13     </div>
14 ... lines 14 - 22
23 </template>
24 ... lines 24 - 47
```

If that's our situation, print a helpful message. And... there it is! Our snacks page - except for the fact that there are *no* snacks - works great.

[Adding Loading to the Sidebar](#)

The products loading part is now works flawlessly. But there is one other spot that we're loading with AJAX that does *not* have any loading info: the categories sidebar!

We're actually going to fix this soon by making the categories load instantly. But since they *are* still loading via AJAX, let's add the loading component there as well. Open up `sidebar.vue` : this is the component that makes the AJAX request for the categories and renders them in its template.

To do this right, should we add another `loading` data like we just did in catalog? We *totally* could! And that's probably a great option. But... I'm going to cheat because I know that my app will *never* have zero categories. If that ever happened, it would probably mean I accidentally emptied my database. Yikes!

Instead, I *am* going to use the `categories.length` to figure out if we're loading. But to be extra organized, let's do this via a computed property called `loading`. Inside `return this.categories.length === 0`.

```
99 lines | assets/js/components/sidebar.vue
49 <script>
50 ... lines 50 - 51
52 export default {
53 ... lines 53 - 68
69   computed: {
70     loading() {
71       return this.categories.length === 0;
72     },
73   },
74 ... lines 74 - 78
79 };
80 </script>
81 ... lines 81 - 99
```

If there are no categories, then we are loading! The nice thing about using a computed property is that it will let us use a simple `loading` variable in the template. And later, if we *did* want to change this to `data`, that would be *super* easy.

Ok: to use this in the template, first import the loading component: `import Loading from '@components/loading'`. Then add the `components` key with `Loading` inside.

103 lines | assets/js/components/sidebar.vue

```
... lines 1 - 48
49 <script>
... line 50
51 import Loading from '@components/loading';
... line 52
53 export default {
... line 54
55   components: {
56     Loading,
57   },
... lines 58 - 82
83 };
84 </script>
... lines 85 - 103
```

Finally, up in the template, right after the `h5`, we'll say `<loading v-show="loading">`.

105 lines | assets/js/components/sidebar.vue

```
1 <template>
2   <div :class="[$style.component, 'p-3', 'mb-5']">
3     <div v-show="!collapsed">
... lines 4 - 7
8     <loading v-show="loading" />
... lines 9 - 38
39   </div>
... lines 40 - 47
48 </div>
49 </template>
... lines 50 - 105
```

I love it!

And when we move over to the browser... I'm *hoping* to see the loading animation right before the categories load. That was super quick! But it *was* there. We have proper loading on both sides!

Next, I want to start organizing our AJAX calls: we currently make them from inside of `sidebar.vue` and `catalog.vue`. That's maybe ok, but I'd like to explore a *better* way to organize these.

Chapter 35: AJAX Services

Head over to `sidebar.vue` where we make the categories AJAX call. In Symfony, we often isolate complex logic - or logic that we need to reuse - into services. One of the *most* common places that we do that is for database queries: we almost *always* have a *repository* class that holds all the database queries for a specific table.

It's optional, but I'd like to do the same with my frontend code! I'd like to isolate all of my AJAX requests for a specific *resource* - like categories or products - into its own, reusable JavaScript module. Then, instead of having AJAX calls inside my components, all that logic will be centralized.

[Adding a Service!](#)

In Symfony, "Services" is *kind of* a generic word for any class that does work. But in this context, I'm using "services" to mean something slightly different. These "services" are *API* services... or, really, any code that loads data - whether that's via an AJAX call, local storage or reading global variables that we set in Twig.

Inside of `js/services/`, create a new file called `categories-service.js`. The `-service` on the end is *totally* redundant since we're in a `services/` directory, but I like to have descriptive filenames.

The `services/` directory already holds one other file called `page-context`. This has *nothing* to do with AJAX calls or APIs but it *is* something that returns data, which is why I put it here. Right now it reads a global variable, but if we decided later to load this via AJAX, it would *still* be a service.

In `categories-service.js` export a function called, how about, `fetchCategories()`. For the logic, copy the `axios` line from `sidebar` ... and paste it here. PhpStorm helpfully imported `axios` for me... but I'll tweak the quotes. Back in the function, return `axios.get()`.

```
9 lines | assets/js/services/categories-service.js
1  import axios from 'axios';
2  ... lines 2 - 5
6  export function fetchCategories() {
7      return axios.get('/api/categories');
8  }
```

This is a really, *really* simple AJAX call, but at *least* we're centralizing the URL so that we don't have it all over the place. Let's also be good programmers and add some documentation above this: it returns a `Promise` ... which, actually, PhpStorm already knew without us saying anything.

But I won't add a description because the function name already describes this pretty well.

```
9 lines | assets/js/services/categories-service.js
1  ... lines 1 - 2
3  /**
4   * @returns {Promise}
5   */
6  export function fetchCategories() {
7  ... lines 7 - 9
```

[Use it from sidebar.vue](#)

Ok: in `sidebar.vue`, let's use this! First, import the service: `import {} from '@services/categories-service'`. Inside the curly braces, grab `fetchCategories`.

105 lines | assets/js/components/sidebar.vue

```
... lines 1 - 50
51 <script>
... line 52
53 import { fetchCategories } from '@services/categories-service';
... lines 54 - 85
86 </script>
... lines 87 - 105
```

Now down in `created`, life gets *much* simpler: `const response =` - keep the `await` because we *still* want to wait for the function to finish - then `fetchCategories()`.

105 lines | assets/js/components/sidebar.vue

```
... lines 1 - 54
55 export default {
... lines 56 - 79
80   async created() {
81     const response = await fetchCategories();
... lines 82 - 83
84   },
85 };
... lines 86 - 105
```

I love this! And to clean up, since we're not using `axios` directly in this component, we can remove the import.

Create the Products Service

The *other* place where we're making an AJAX call is in `catalog.vue` to fetch the products. This one is a *bit* more complex because *if* we have a category, we need to pass a `category` query parameter.

Since this AJAX call is for a different API resource, inside the `services/` directory, create a third file called `products-service.js`.

Start the same way: `export function fetchProducts()` with a `categoryIri` argument. I've been calling this `categoryId` so far, but in reality, this *is* the IRI, so I'll give it the proper name here.

17 lines | assets/js/services/products-service.js

```
1 import axios from 'axios';
... lines 2 - 6
7 export function fetchProducts(categoryIri) {
... lines 8 - 15
16 }
```

For the logic, go back to `catalog.vue`, copy the `params` code... and paste it here. Let's also copy the response line, paste that here too and return `axios.get()`.

Finally, for the `params`, it's not `this.currentCategoryId` but `categoryIri`. So `if (categoryIri)` then `params.category = categoryIri`.

17 lines | assets/js/services/products-service.js

```
... lines 1 - 6
7 export function fetchProducts(categoryIri) {
8   const params = {};
9   if (categoryIri) {
10     params.category = categoryIri;
11   }
12
13   return axios.get('/api/products', {
14     params,
15   });
16 }
```

And... I need to fix my import code to use single quotes on `axios`.

Before we use this, let's add some docs: the `categoryId` will be a `string` or `null` and this will return a Promise.

```
17 lines | assets/js/services/products-service.js
... lines 1 - 2
3 /**
4  * @param {string|null} categoryId
5  * @returns {Promise}
6  */
7 export function fetchProducts(categoryId) {
... lines 8 - 17
```

That's looking great!

[Use the Service in `catalog.vue`](#)

Let's put it to use in `catalog.vue`. Like before, start by importing it: `import { } from '@services/products-service'` and then bring in the `fetchProducts` function.

```
64 lines | assets/js/components/catalog.vue
... lines 1 - 21
22 <script>
23 import { fetchProducts } from '@services/products-service';
... lines 24 - 62
63 </script>
```

Now, down in `created()`, we don't need *any* of the `params` stuff anymore. And the `response` line can now just be `response = await fetchProducts(this.currentCategoryId)`.

```
64 lines | assets/js/components/catalog.vue
... lines 1 - 26
27 export default {
... lines 28 - 45
46   async created() {
47     this.loading = true;
48
49     let response;
50     try {
51       response = await fetchProducts(this.currentCategoryId);
52
53       this.loading = false;
54     } catch (e) {
... lines 55 - 57
58   }
... lines 59 - 60
61 },
... lines 62 - 64
```

This is *lovely*: the `created()` function reads like a story: set the `loading` to true, call `fetchProducts()`, then set `loading` to false.

To finish the cleanup, remove the unused `axios` import.

Phew! We just made a lot of changes so... let's make sure we didn't break anything. Do a full page refresh to be sure and... yea! Everything loads. The products and sidebar work on every page and our code is better organized.

[Service: Return the Full Response? Data?](#)

Before we keep going, I want to mention a design decision that I made. In a service like `products-service`, we are returning the `Promise` from Axios. That means that when we use `await` or chain a normal `.then()`, what we will ultimately receive is the response... which we can then use to say things like `response.data`.

But if you want, you *could* go a bit further in the service and add `.then((response) => response.data)`.

By doing this, our function *still* returns a Promise. But instead of the payload of the promise being the full AJAX response, it will be the JSON data. To make the code work in `catalog`, we would set the function directly to the `data` variable.

That makes the function a *bit* nicer to use. But... I'm going to completely undo all of that. Why? Changing the function to directly give you the JSON data is nice. The *problem* is if you ever needed to read something from the response, like a header. If we only returned the data, reading a header wouldn't be possible.

That's why I typically keep my functions simple and return the Promise that resolves to the entire Response. I need to do more work in my component, but I also have more power.

Refresh one more time to make sure nothing broke! All good.

Next, hmm... I don't mind *some* of the loading on this page, but the categories in the sidebar are starting to get on my nerves! It makes the page look incomplete while it loads because the categories *feel* like part of the *initial* page structure. We can fix that by passing the categories from the server *directly* into Vue!

Chapter 36: Skipping AJAX: Sending JSON Straight to Vue

With *both* the categories and products loading dynamically, our app is starting to get really exciting! But there's a part of the user experience that I'm *not* happy about: there are a lot of things loading!

[The "too much loading" Problem](#)

When we get to a page, it's probably okay for some things to load. But right now, the page basically looks empty at first. The categories form part of the page layout... and it's a bit jarring when the sidebar is empty.

And... it could get worse! What if we wanted to include the current category name in the page title... or as the h1 on the page! In that case, *both* of those would be missing on load! And if we started to render info about the authenticated user in Vue - like a user menu, if we loaded that data via AJAX, then we would need to hide that menu at first and *then* show it.

The point is: too much loading can be a big problem.

What's the solution? Well, we're already making a request to the server each time we visit a category. When we do that, our server is *already* primed to make fast database queries. So, in theory, we should be able to fetch data - like for the categories or user information - *during* that page load and avoid the slow AJAX request.

In general, there are two solutions to this problem of "too much loading". The first is called server-side rendering where you render the Vue app on your *server*, get the HTML and deliver that on the initial page load.

That's a great solution. But it's also a bit complex because you need to install and execute Node on your server.

[Passing the Categories from the Server to Vue](#)

The second option, which is a lot simpler and almost as fast, is to pass the *data* from our server into Vue. Literally, in the controller, we're going to load all the categories, pass them into Twig and set them on a variable that we can read in JavaScript. That will make the data *instantly* available: no AJAX call needed!

Ok, let's do this! Remember: the controller for this page is `src/Controller/ProductController.php`. And actually, there are *two* controllers: `index()` - which is the homepage - and `showCategory()` for an individual category.

So if we're going to pass the categories to Vue, we'll need to pass it into the template for *both* pages.

Start in `index()`: autowire a service called `CategoryRepository $categoryRepository`. Now, add a second argument to Twig so that we can pass in a new variable called `categories` set to `$categoryRepository->findAll()`.

```
44 lines | src/Controller/ProductController.php
... lines 1 - 12
13 class ProductController extends AbstractController
14 {
... lines 15 - 17
18 public function index(CategoryRepository $categoryRepository): Response
19 {
20     return $this->render('product/index.html.twig', [
21         'categories' => $categoryRepository->findAll(),
22     ]);
23 }
... lines 24 - 42
43 }
```

That will query for *all* the categories.

Do the same thing down in `showCategory()`: add the `CategoryRepository $categoryRepository` argument, go steal the `categories` variable... and paste it here.

```

44 lines | src/Controller/ProductController.php
... lines 1 - 27
28 public function showCategory(Category $category, IriConverterInterface $iriConverter, CategoryRepository $categoryRepository): Res
29 {
30     return $this->render('product/index.html.twig', [
... line 31
32         'categories' => $categoryRepository->findAll(),
33     ]);
34 }
... lines 35 - 44

```

Woo! We now have a `categories` variable available in the Twig template.

Serializing to JSON in the Template

Open it up: `templates/product/index.html.twig`. We're already setting a `window.currentCategoryId` global variable to an IRI *string*. But this situation is more interesting: the `categories` variable is an array of `Category` *objects*. And what we *really* want to do is transform those into JSON.

Go to `/api/categories.jsonld`: that's a quick way to see what the API response for categories looks like. So if we're going to send categories data from the server instead of making an AJAX call, that data should, ideally, look *exactly* like this.

This means that, in our Symfony app, we somehow need to serialize these `Category` objects into the JSON-LD format.

Open the `src/Twig/` directory to find a shiny class called `SerializerExtension`. I created this file, which adds a filter to Twig called `jsonld`. By using it, we can serialize *anything* into that format.

```

30 lines | src/Twig/SerializerExtension.php
... lines 1 - 8
9 class SerializerExtension extends AbstractExtension
10 {
... lines 11 - 17
18 public function getFilters(): array
19 {
20     return [
21         new TwigFilter('jsonld', [$this, 'serializeToJsonLd'], ['is_safe' => ['html']]),
22     ];
23 }
24
25 public function serializeToJsonLd($data): string
26 {
27     return $this->serializer->serialize($data, 'jsonld');
28 }
29 }

```

Awesome! Back in the template, add `window.categories` set to `{{ categories|jsonld }}`.

```

27 lines | templates/product/index.html.twig
... lines 1 - 12
13 {% block javascripts %}
... lines 14 - 15
16 <script>
... lines 17 - 21
22     window.categories = {{ categories|jsonld }};
23 </script>
... lines 24 - 25
26 {% endblock %}

```

Let's go see what that look like! Find your browser, refresh and view the page source. Near the bottom... there it is! It's has the *same* JSON-LD format as the API! In the console, try to access it: `window.categories`. Yes! Here are the four categories with the

normal `@context` , `@id` and `@type` .

Well, *technically* this is a *little* bit different than what the API returns. Go back to `/api/categories.jsonld` . In the true API response, the array is *actually* under a key called `hydra:member` . And if this were a long collection with pagination, the JSON would have extra keys with information about how to get the rest of the results.

The JSON we're printing is *really* just the stuff inside `hydra:member` . But most of the time, this is all you really need.

But if you *did* need *all* of the data, you could pass a 3rd argument to `serialize()` - an array - with a `resource_class` option set to whatever class you're serializing, like `Category::class` . *That* would give you more structure. If you need pagination info, that's also possible. Let us know in the comments if you need that.

But for us this data is going to be *perfect*, because all we need are the categories. Next, let's use this data in our Vue app to avoid the AJAX call! When we do, suddenly, our AJAX service function will *change* to be synchronous. But by leveraging a Promise directly, we can *hide* that fact from the rest of our code.

Chapter 37: Faking AJAX calls: Reading Synchronously

The `categories` data is now available as a global variable: `window.categories` .

In `sidebar.vue` , we're calling `fetchCategories()` , which lives in our fancy new `categories-service` module. You can find this at `assets/js/services/categories-service.js` .

To switch this from an AJAX call to the global variable, just `return window.categories` . Celebrate by removing the `axios` import on top.

```
7 lines | assets/js/services/categories-service.js
1  /**
2   * @returns Array
3   */
4  export function fetchCategories() {
5    return window.categories;
6  }
```

This is what I like about centralizing the `fetchCategories()` method: we don't need to run around and change our code in a bunch of places: just here. Well... that's not *quite* true yet. One problem is that our function does *not* return a `Promise` anymore! It now returns an array. That could affect code that uses this.

[Whoops! We're Returning Different Data](#)

But... let's ignore that for a moment! Move over and refresh. Bah! It's broken:

```
Cannot read property hydra:member of undefined
```

We *did* change the function from returning a `Promise` to returning an array. But that's actually *not* what broke our code! The *real* problem is that, down in `created()` , `fetchCategories()` no longer gives us a `response` object with a `data` key!

Let's back up: it's actually *ok* to use `await` on a non-async function: JavaScript just grabs the return value from the function and gives it to us: no error. The *real* problem is that the value that the *previous* code gave us was a `response` object... but *now* we're returning an *array* of categories. This means that `response.data` is undefined!

The simplest way to fix this is right here: comment out the old `this.categories` line and directly say `this.categories = fetchCategories()` . We don't need the `await` anymore, but it doesn't hurt anything.

```
105 lines | assets/js/components/sidebar.vue
↑ ... lines 1 - 50
51 <script>
↑ ... lines 52 - 54
55 export default {
↑ ... lines 56 - 79
80   async created() {
81     this.categories = await fetchCategories();
82
83     //this.categories = response.data['hydra:member'];
84   },
85 };
86 </script>
↑ ... lines 87 - 105
```

Simple enough! Back on the browser, it's *already* showing the categories on the left. And if we reload... yes! The categories are *instantly* there! Woo! There *is* still a slight delay before the styles load, but that will only happen in dev mode: we did that to allow hot module replacement. So, mission accomplished!

Returning a Promise from the Sync Method

One of the cool things about isolating our AJAX calls into functions like `fetchCategories()` is that, when we call that function, we don't need to know or care if it's talking to an API, or which API, or if we're just loading the data locally from local storage or a variable.

But... that's not really true right now. Because when we changed this from using the API to the global variable, we changed what this function returned! We changed it from returning a `Promise` to an array and we *also* changed the actual *data* that's returned from a `response` object with `data` and `hydra:member` properties to an array. This meant that, once this function was synchronous, we needed to update any code that called this.

And... that's fine. If you know that you're going to load categories synchronously via `window.categories`, then you can just update your code to reflect that. It's not a huge problem.

But when we changed to the global variable, we could have *also* written our function in a way that *did* not break any code that used it: we *could* have returned a `Promise`.

Let's try that: `return new Promise()`. This needs one argument: a callback with `resolve` and `reject`. If you've never seen a `Promise` like this, check out our [ES6 tutorial](#) all about them. They're *fascinating*.

```
13 lines | assets/js/services/categories-service.js
... lines 1 - 3
4  export function fetchCategories() {
5    return new Promise((resolve, reject) => {
... lines 6 - 10
11  });
12 }
```

Anyways, once our work is done - which will be *instantly* since we don't need to make any AJAX calls, call `resolve()` and pass it the *data* that we want the promise to return. Now, we're not *really* making an AJAX request... so we can't *exactly* return the same data as before because... we don't have a response. But we know that what we *really* care about is that this `Promise` returns a object with a `data` key and a `hydra:member` key below it. Let's at least fake that here: add `data` set to an object and `hydra:member` set to `window.categories`. I'll remove the extra return at the bottom and, above the function, once again, advertise that we return a `Promise` !

```
13 lines | assets/js/services/categories-service.js
... lines 1 - 3
4  export function fetchCategories() {
5    return new Promise((resolve, reject) => {
6      resolve({
7        data: {
8          'hydra:member': window.categories,
9        },
10     });
11   });
12 }
```

Now, back in `sidebar.vue`, we can revert all of our changes and use the *exact* code we had before:
`const response = await fetchCategories()`.

And... that's it! The `Promise` is a little funny... because it will *always* resolve immediately. But that's fine! When it resolves, it will return data that's not *exactly* the same as an Axios response, but it's close.

Moment of truth! At our browser... it *looks* like it's working. Let's refresh. Yes! It *definitely* works. So this is a *great* way to get dynamic data from your server *without* needing an AJAX call when you *really* want something to be available almost instantly.

Next, let's use the categories data and `currentCategoryId` to print the *name* of the category on each page. To do that, we'll need to make sure all of that data lives in the right component.

Chapter 38: Passing Props vs Fetching Directly

At this point, we have access to which category we're currently on and *all* the category data. Let's use all that goodness to print a better title: we can now print the category name when we're on a category page.

Check out `catalog.vue` : *this* is where the `h1` lives. Ok, all we need to do is use the `currentCategoryId` to find that category in the `categories` data, and then print its name.

To do this, we *could* create a computed property right in `catalog` with all of this logic... and then print it right here. But instead, I'm going to isolate this title area into its own component. Again, *when* you should move something into its own component is subjective, but I'm planning to reuse this title area in the next tutorial.

Creating the Title Component

In the `components/` directory, create a new `title.vue` file and start the same way as always: with the template. Set this to a `<div>` and immediately add `:class="$style.component"` . Inside, put the `h1` with, for now, a hardcoded "Products".

```
22 lines | assets/js/components/title.vue
1  <template>
2    <div :class="$style.component">
3      <h1>
4        Products
5      </h1>
6    </div>
7  </template>
8  ... lines 8 - 22
```

Next, add the `<script>` tag with the minimum needed: `export default` and `name: 'Title'` .

```
22 lines | assets/js/components/title.vue
8  ... lines 1 - 8
9  <script>
10 export default {
11   name: 'Title',
12 };
13 </script>
14  ... lines 14 - 22
```

Finally, at the bottom, because we're already referencing `$style` , add `<style lang="scss" module>` . We only need one thing: `.component {}` , an `h1 {}` inside, and `font-size: 1.7rem` .

```
22 lines | assets/js/components/title.vue
8  ... lines 1 - 14
15 <style lang="scss" module>
16 .component {
17   h1 {
18     font-size: 1.7rem;
19   }
20 }
21 </style>
```

Perfecto! A nice, simple component to render the title.

Watch out for Protected Element Names

In `catalog.vue` , let's use this! But when I import it, I'm going to call it `TitleComponent` from `@/components/title` . The reason is that, when we add `TitleComponent` to `components` , it *really* means `TitleComponent: TitleComponent` .

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

```
... lines 1 - 19
20 </script>
... lines 21 - 23
24 import TitleComponent from '@/components/title';
... line 25
26 export default {
... line 27
28   components: {
... lines 29 - 30
31     TitleComponent,
32   },
... lines 33 - 61
62 };
63 </script>
```

Anyways, the key - `TitleComponent` in this case - determines the HTML element that we can use in the template.

`TitleComponent` means that we can say `<title-component />` .

That's great. But if we had called it `Title` , then, in the template, we would need to use `<title />` . Do you see the problem? `<title>` is a *real* HTML tag! And so, instead of rendering our `title` component, Vue would just... render a `title` tag.

In general, if you want to avoid collisions, a best practice - which I admit we have *not* been following - is to include a `-` every time you render a component. There's a W3C spec that recommends that custom component always have a dash.

Anyways, when we use `<title-component>` ... I think it's working! I still see "Products".

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

```
1 <template>
... line 2
3   <div class="row">
4     <div class="col-12">
5       <title-component />
6     </div>
7   </div>
... lines 8 - 17
18 </template>
... lines 19 - 64
```

To make that print the actual *category* name, *something* needs to use the `currentCategoryId` and `categories` info to find the current category's name. For now, I'm going to put that logic directly into `title` . That means that `title` will need to receive both of these pieces of data as props. Add `props:` with `currentCategoryId` , `type: String` and `default: null` so that null is allowed. Then add `categories` with `type: Array` and `required: true` .

```
32 lines | assets/js/components/title.vue
... lines 1 - 8
9 <script>
10 export default {
... line 11
12   props: {
13     categoryId: {
14       type: String,
15       default: null,
16     },
17     categories: {
18       type: Array,
19       required: true,
20     },
21   },
22 };
23 </script>
... lines 24 - 32
```

This is everything we need.

[categories Prop vs Use the Service!](#)

But I want to talk about that `categories` prop. Obviously, our `title` component will need the `categories` Array so it can do its job. By adding it as a prop, you can already see that my plan is to pass this in from the parent component.

But in our app, we know that categories are *actually* being loaded from the global `windows.categories` variable: they are *not* being loaded via Ajax. They're also not something that will ever *change*: once we get the categories, those are the categories forever.

So in reality, because the categories are *instantly* available and never change, instead of using a prop, we *could* just use our `categories-service` to get that data directly from this component!

[Props vs Grabbing a Value Directly](#)

This is a *key* thing that I want you to understand: we use props for two reasons.

The first reason is to directly pass configuration to a component. For example, our `Legend` component has a `title` prop. If you want to use this component, then you need to pass that configuration, which could be a piece of data, a hardcoded string or anything. It's like an argument to a method.

The second reason we use props is related, but more subtle. When something will *change* during the lifecycle of our Vue app, then that "thing" *must* live as data on a component. And once something lives as data on a component, the only way to make it accessible to child components is by passing the value as a prop. Props are the *only* mechanism to pass down data.

Now, side note - with something like Vuex or some new features in Vue 3, it's possible to store data *outside* of a component in a central location. That helps remove this second reason for using props. But we'll talk about that in our Vue 3 tutorial.

Anyways, in this situation, because `categories` is boring and static, it doesn't really need to live as data anywhere. And that means, we do *not* need to pass it down as props. If we want to, it would be *totally* ok fetch the categories directly from `categories-service` wherever we need it.

Now that you know the *easy* solution, let's take the *harder* path. Let's pretend that our categories *are* still loading via AJAX... which means that *technically* they *do* change during our app's lifecycle: they're empty for a moment, and *then* they populate. Let's tackle this next.

Chapter 39: Hoisting Data Up

We just discussed that, because the `categories` in our app are static - we don't load them with AJAX and they never change - they don't *really* need to live as data on a component and it would be *totally* ok to fetch them directly - wherever we need them - from `categories-service` .

But... we're going to take the more complex path by pretending that our categories *are* still loading via AJAX... which means they *do* change during the lifecycle of our app... which means that they *do* need to live as data on a component.

Computed property for Category Name

But... let's ignore that for a moment and finish *this* component. We know that the `title` component will receive `currentCategoryId` and `categories` props. To find the *current* category name, we'll need to write some logic. And... hey! That's a *perfect* case for a computed property.

Add `computed` with one key inside, how about, `categoryName()` . For the logic, if `this.currentCategoryId` - to reference that prop - `=== null` , then return "All Products".

```
43 lines | assets/js/components/title.vue
... lines 1 - 8
9 <script>
10 export default {
... lines 11 - 21
22   computed: {
23     categoryName() {
24       if (this.currentCategoryId === null) {
25         return 'All Products';
26       }
... lines 27 - 30
31   },
32 },
33 };
34 </script>
... lines 35 - 43
```

If we *are* on a category page, find the correct one with `const category = this.categories.find()` . Pass this an arrow function with a `cat` argument. We want to find the category whose `@id` property - which is the IRI string - matches `this.currentCategoryId` .

```
43 lines | assets/js/components/title.vue
... lines 1 - 21
22   computed: {
23     categoryName() {
... lines 24 - 27
28       const category = this.categories.find((cat) => (cat['@id'] === this.currentCategoryId));
... lines 29 - 30
31   },
32 },
... lines 33 - 43
```

The `find()` function *effectively* loops over all the categories, calls this function for each one, and returns the first that makes this expression true.

At the bottom, add `return` and use the ternary syntax: if a category was found, which... it should be unless the `categories` data is loading via AJAX and is empty at first - then return `category.name` . Else, use an empty string... or you could say "Loading..." .

```
43 lines | assets/js/components/title.vue
... lines 1 - 21
22   computed: {
23     categoryName() {
... lines 24 - 29
30     return category ? category.name : "";
31   },
32 },
... lines 33 - 43
```

Perfect! Up in the template, use this: `{{ categoryName }}` .

```
43 lines | assets/js/components/title.vue
1  <template>
2    <div :class="$style.component">
3      <h1>
4        {{ categoryName }}
5      </h1>
6    </div>
7  </template>
... lines 8 - 43
```

Hoisting the categories Data

And... this component is done! Let's get to the interesting part. We need to pass `currentCategoryId` and `categories` into this component. Open the parent component - `catalog.vue` - and let's scroll down a little. It already has access to `currentCategoryId` - yay! - but it does *not* have access to `categories` .

Where *does* the `categories` data live? Head over to the Vue dev tools. The `Catalog` component is rendered by `Products` ... but it doesn't have access to `categories` either. Ah yes, that's because `categories` *currently* live as data in `Sidebar` .

And that makes sense! Until this moment, the `Sidebar` component was the *only* one that needed the categories. But now that `catalog` *also* needs that info - so it can pass it to the `Title` component - we need to *hoist* - or "pull up" - the `categories` data to a higher component: we need to move it into the `Products` component so that we can pass it to both `Sidebar` and `Catalog` as `props` .

This is a fairly common situation: you start by putting your data in one component, then later you need to move it *higher* so that more parts of your app can use it. We did this once earlier: we moved the `collapsed` boolean data from `Sidebar` *up* to `Products` so that we could use it in more places.

So let's get to work! Open up `products.vue` and then `sidebar.vue` . Copy the `categories` data and then remove the `data` option entirely. In `products` , find `data` and paste `categories` .

```
64 lines | assets/js/pages/products.vue
... lines 1 - 22
23 <script>
... lines 24 - 28
29 export default {
... lines 30 - 34
35   data() {
36     return {
... line 37
38     categories: [],
39   };
40 },
... lines 41 - 61
62 };
63 </script>
```

The other thing we need move is the `created()` function. Copy that full function and delete it. In `products` it looks like we don't have a `created()` function yet, so we can paste this one.

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

```
... lines 1 - 28
29 export default {
... lines 30 - 51
52   async created() {
53     const response = await fetchCategories();
54
55     this.categories = response.data['hydra:member'];
56   },
... lines 57 - 61
62 };
... lines 63 - 64
```

When we did that, PhpStorm automatically added the import... but I *do* need to fix how that code looks.

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

```
... lines 1 - 22
23 <script>
... lines 24 - 26
27 import { fetchCategories } from '@services/categories-service';
... lines 28 - 62
63 </script>
```

So far so good. Now in **sidebar**, because we don't have **categories** as data anymore, we will need it as a prop. Inside **props**, add **categories**, **type: Array** and **required: true**.

99 lines | assets/js/components/sidebar.vue

```
... lines 1 - 54
55 export default {
... lines 56 - 59
60   props: {
... lines 61 - 68
69     categories: {
70       type: Array,
71       required: true,
72     },
73   },
... lines 74 - 78
79 };
... lines 80 - 99
```

Finally, back over in **products**, pass this to sidebar: **:categories="categories"**.

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

```
1 <template>
... lines 2 - 3
4   <aside :class="asideClass">
5     <sidebar
... lines 6 - 7
8       :categories="categories"
... line 9
10     />
11   </aside>
... lines 12 - 20
21 </template>
... lines 22 - 64
```

Unless I tripped over my keyboard somewhere, that *should* make the sidebar happy again! Back at the browser... I'll refresh just to be sure and... it works! We have some errors due to a missing prop in **TitleComponent** - but that's ok: we're working on passing that!

Passing categories down to Title

Now that we have access to `categories` inside `products`, we can *also* pass that to `catalog` : `:categories="categories"` .

```
64 lines | assets/js/pages/products.vue
1  <template>
  ... lines 2 - 12
13    <div :class="contentClass">
14      <catalog
  ... line 15
16        :categories="categories"
17      />
18    </div>
  ... lines 19 - 20
21  </template>
  ... lines 22 - 64
```

To add that prop, let's steal the prop code from `sidebar` ... and paste it into `catalog` .

```
71 lines | assets/js/components/catalog.vue
  ... lines 1 - 28
29  export default {
  ... lines 30 - 35
36    props: {
  ... lines 37 - 40
41      categories: {
42        type: Array,
43        required: true,
44      },
45    },
  ... lines 46 - 68
69  };
  ... lines 70 - 71
```

Use this shiny new prop to pass the `categories` *again* to their final location. The `TitleComponent` needs *two* things actually: `:currentCategoryId="currentCategoryId"` and `:categories="categories"` .

```
71 lines | assets/js/components/catalog.vue
1  <template>
  ... lines 2 - 3
4    <div class="col-12">
5      <title-component
6        :current-category-id="currentCategoryId"
7        :categories="categories"
8      />
9    </div>
  ... lines 10 - 20
21  </template>
  ... lines 22 - 71
```

Phew! I think we've got all the wires connected. When we move over... yes! It's working... and there are *no* errors. We can go to "all products" and that title works too.

But that *was* a *lot* of prop passing. We moved the `categories` data to `products` ... so we could pass it to `catalog` ... so we could pass it to `title` . This one of the common ugly parts of a traditional Vue or React app: a lot of prop passing. It's not the end of the world, but as I've mentioned a few times, it *is* something that can be solved by centralizing your data, which is possible in Vuex or in Vue3. I'm particularly excited about the possibilities in Vue 3.

Next: this product listing page is *really* looking good. But since we're going to have a *ton* of these... um... "useful" products in our store, let's add a search bar. This will be a *perfect* opportunity to talk about the *last*, *super* important directive: `v-model` .

Chapter 40: The Formidable v-model

Ya know what just occurred to me? We *haven't* talked *at all* about form elements yet! And what a wonderful coincidence! Because our *next* challenge is to add a search bar input to filter the product list.

The search bar will, of course, contain some HTML. It will also need to manage the value of the search bar and help us know when we should filter the product list. That's *enough* that I think we should isolate this in a new component.

Creating the search-bar Component

In `components/`, create a new file called `search-bar.vue`. Add the `<template>` with `<div>` and an `<input>` with `class="form-control"`, a `placeholder` and `type="search"`.

```
16 lines | assets/js/components/search-bar.vue
1  <template>
2    <div>
3      <input
4        class="form-control"
5        placeholder="Search products..."
6        type="search"
7      >
8    </div>
9  </template>
... lines 10 - 16
```

So nothing special. If you're wondering why I added the `div`, it's just because we're going to have *more* than just the input later.

At the bottom add the `<script>` section with the basic `export default` and `name: 'SearchBar'`.

```
16 lines | assets/js/components/search-bar.vue
... lines 1 - 10
11 <script>
12 export default {
13   name: 'SearchBar',
14 };
15 </script>
```

Love it! Over in `catalog.vue`, let's see... change the `div` around the title to `col-3`, and then, below, add a new `<div class="col-9">`. Inside, we haven't imported the `search-bar` component yet... but ah! Let's try to use it anyway! Type `<sea` and hit tab to auto-complete that.

```
76 lines | assets/js/components/catalog.vue
1  <template>
... line 2
3    <div class="row">
... lines 4 - 9
10      <div class="col-9">
11        <search-bar />
12      </div>
13    </div>
... lines 14 - 23
24  </template>
... lines 25 - 76
```

When we did that, because PhpStorm is *awesome*, it added the import *and* put this down in the `components` section. PhpStorm, did we just become best friends?

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

... lines 1 - 25

26 <script>

... lines 27 - 29

30 import SearchBar from '@components/search-bar';

... lines 31 - 74

75 </script>

Let's check the browser. Boom! That's a *sweet* search bar. Now let's bring it to life

Binding a data to the Input

To filter the product list, we need to know the *value* of the input. There are a few ways to get this, but the simplest is to add a piece of **data** to the component that we keep "in sync" with the text of the input. Add **data** and return an object with one item: **searchTerm** set to an empty string to start.

22 lines | assets/js/components/search-bar.vue

... lines 1 - 12

13 export default {

... line 14

15 data() {

16 return {

17 searchTerm: "",

18 };

19 },

20 };

... lines 21 - 22

To "bind" this data to the input, we need to do two things. First, set the input value to the data: **:value="searchTerm"** .

22 lines | assets/js/components/search-bar.vue

1 <template>

2 <div>

3 <input

4 :value="searchTerm"

... lines 5 - 7

8 >

9 </div>

10 </template>

... lines 11 - 22

Now with *just* that, if move over to the browser and look at the Vue dev tools, we can click on **SearchBar** , change the **searchTerm** data and... voilà! The input text updates... which shouldn't be too surprising. That's Vue goodness in action.

But what we *can't* do yet is type in the box and have it *update* that data. That's the *second* part of binding an input to data.

@input for when the Input Changes

How can we do that? It's lovely: by listening to an event on the input.

Remember: the way we listen to an event is with **v-on:** and then the name of the event, like **click** or **keydown** . Well, in practice, we use the shortcut **@** syntax, so **@click** or **@keydown** .

In this case, use **@input** . The **input** event is a native, normal JavaScript event that's *similar* to **keyup** : it will trigger any time the input's value changes.

Inside the quotes, *so far*, we've set this to something like **someMethod** . Then, we've added a **methods** option, with a **someMethod** function, and put whatever code we want to run right there.

That's *totally* valid, but if what you need to do is simple, you can *also* write an expression right inside the attribute. In this case, when the input changes, we only need to set the **searchTerm** data to the new string.

Inline v-on Expression

To do that, say `searchTerm` - which we know *really* means `this.searchTerm` - equals `$event.target.value` .

```
23 lines | assets/js/components/search-bar.vue
... lines 1 - 2
3   <input
... lines 4 - 7
8     @input="searchTerm = $event.target.value"
9   >
... lines 10 - 23
```

That... deserves an explanation. If we set this to a method name, then the method will receive an `event` argument. Then we can say `event.target.value` to get this input's value.

When you write an inline expression, Vue magically makes the `event` object available as a `$event` variable.

But... hmmm... ESLint is mad: it doesn't like how I ordered my attributes. From a technical standpoint, we can put these in *whatever* order we want. But as a standard, ESLint likes to have listeners - like `@input` - on the bottom.

Ok, let's take our input for a test drive! Back at the browser, find `SearchBar` on the dev tools. Now, as we type... yea! The data is updating!

What we *effectively* just did is took a piece of data - `searchTerm` - and *bound* it to a form input. When the data changes, the input changes. When the input changes, the data changes.

Hello v-model!

This is a *very* common thing to do in Vue. In fact, it's *so* common that Vue created a special directive *just* for it.

Check this out: we can delete the `@input` and `:value` lines and replace them with *one* line that will do the *exact* same thing. It's `v-model="searchTerm"` .

```
22 lines | assets/js/components/search-bar.vue
... lines 1 - 2
3   <input
4     v-model="searchTerm"
... lines 5 - 7
8   >
... lines 9 - 22
```

This is one of the *last* important directives that we haven't already talked about. `v-model="searchTerm"` *literally* means: set the value attribute to `searchTerm` and, on input, *update* the `searchTerm` data with the input value. It's identical to what we had before.

Now `v-model` *does* act a little bit different with things like checkboxes or select elements, but those are minor normalizations to make `v-model` work how you *want* it to work for those elements. Back at the browser, everything is hooked up just like before.

You're probably going to see `v-model` *all* the time. For me, it helps to remember what it's *really* doing behind the scenes: setting the input's `value` to the `searchTerm` data and, on change, *updating* that data for you.

Next: let's use this search term to filter our product list as the user is typing!

Chapter 41: Pass Data in a Custom Event & Internal Data

Our search bar is working really well! We can type in it and it keeps things synchronized with the `searchTerm` data. We now need to communicate when the `searchTerm` changed *up* to catalog so that it can know to filter the product list.

We might think that, since `Catalog` *needs* to know the `searchTerm` data, we should move it up to `Catalog` then pass it back into `search-bar` as a prop. We did this in a few cases already: we moved a piece of data *up* the tree to make it accessible to more components.

We *could* do that... but I won't. Trust me for now - I'll explain why soon.

But regardless of where the data lives, what we need to do in `Catalog` is update the product list *when* the search term *changes*. In other words, we need to perform an action when an *event* happens. So for our first order of business, whenever we update the search box, we need the `search-bar` component to say "Yo! I changed!" and for it to *pass* us the new search term.

[\\$emit a Search Event](#)

In `search-bar`, this *specifically* means that on the `input` event of the search box, we need to emit a custom event. To do that, add `@input=""` and this time, set it to a method name: `onInput`.

```
28 lines | assets/js/components/search-bar.vue
1  <template>
2    <div>
3      <input
4      ... lines 4 - 7
5      ...
6      ...
7      ...
8      @input="onInput"
9    >
10  </div>
11 </template>
12 ... lines 12 - 28
```

As we talked about earlier, when you have `v-model`, one of the things it does behind the scenes is add its *own* `@input` which sets the `searchTerm` data to the input's value. In this situation, because we need to do something *else* on input, we're adding a *second* `@input`. The original one that's set by `v-model` is *still* going to update the `searchTerm` data. Then, after it finishes, our custom code will run, which is pretty cool!

Below, add `methods` and `onInput()`. Inside, emit a custom event with `this.$emit()`. Let's call it `search-products`. If we intended to re-use this component as a generic search box around the site, we could call it just `search`.

And it turns out that `$emit` has an optional *second* argument, which can hold *any* data that we want to *include* on the event object. That's good news! Because, in addition to just saying that the `search-products` event has happened, we *also* need to pass what the `searchTerm` *is*. Do that with `term: this.searchTerm`.

```
28 lines | assets/js/components/search-bar.vue
... lines 1 - 12
13 <script>
14 export default {
15 ... lines 15 - 20
16 ...
17 ...
18 ...
19 ...
20 ...
21 methods: {
22   onInput() {
23     this.$emit('search-products', { term: this.searchTerm });
24   },
25 },
26 };
27 </script>
```

[Check the Event in Vue DevTools](#)

Before we touch anything else, move over to the browser. On the Vue Dev Tools, click on `SearchBar`. Actually, click on the Events tab.

Now... type! Yes! We see one event for *each change* that we made to the text! Check the last event: it has a `payload` property that, if we explore, has the *term* on it. Awesome! We'll see how to use that in a few minutes.

Products Filtering Strategies

Back to `Catalog` ! There are two ways to filter the products. The first and *easiest* is to just take the `products` array and filter it using JavaScript. The second is to do a server-side search. That's *much* more powerful and we *will* try that next. But to start, let's keep things simple.

When the `searchTerm` changes, we don't *actually* want to *change* the `products` array because we don't want to lose that information: if the user cleared out the search box, we will need to show the *original* list. What we *really* need to do is keep track of the `searchTerm` and use it to *compute* a new array of filtered products.

Duplicating the searchTerm Data???

Add a new `searchTerm` data set to empty quotes.



```
77 lines | assets/js/components/catalog.vue
... lines 1 - 32
33 export default {
... lines 34 - 50
51   data() {
52     return {
... line 53
54       searchTerm: "",
... lines 55 - 56
57     };
58   },
... lines 59 - 74
75 };
... lines 76 - 77
```

Now you might be yelling:

Ryan! You dummy! You just duplicated the `searchTerm` data! You have `searchTerm` in `search-bar` and you *also* have `searchTerm` in `Catalog` ! You said never to do that!!!

That is an *excellent* point - even the dummy part, a lot of the time. But, I'm doing this on *purpose*. The `searchTerm` inside of `search-bar` is really an *internal* piece of data. It exists *just* to help that component do its job. Anyone who *uses* this component doesn't really need to know or care that it exists. All *they* care about is that this component emits an event when the search changes and passes us the new `term`.

Then, in `Catalog`, in order to do our work here, it just so happens that we *also* need to store the search term as data.

Now, we *could just* put the `searchTerm` data in `Catalog` and pass it to `search-bar` as a prop. But there are two reasons why I *won't* do that. First, because I don't have to: `Catalog` will never need to *change* the `searchTerm`. So because the *only* place it will *ever* change is `search-bar`, the two pieces of data won't get out-of-sync. The *second* reason is that, in a few minutes, we're going to add *debouncing*, which basically means that the `searchTerm` inside `search-bar` will update *immediately*, but that component will *wait* before it emits the event. It's subtle, but in that situation, the two search terms will *not* always match each other.

If this is confusing... don't worry. There's not a wrong way to do things: if you *just* put `searchTerm` in `Catalog` and passed it as a prop to `search-bar`, that might be a bit unnecessary, but it would work great! If you added debouncing later, you might *then* realize that you need two *separate* pieces of data. I'm planning ahead, but it's *always* ok to choose a path, move forward, and let the design figure itself out naturally.

Anyways, we're sort of half way done. Next, we need to *listen* to the custom `search-products` event, *update* the `searchTerm` data in `Catalog` and use that to print a *filtered* list of products.

Chapter 42: Filtering Products

Ok: we have `products` and `searchTerm` data. Let's *update* the `searchTerm` when the `search-bar` component tells us it's changed.

[Listen to `search-products` Event in Catalog](#)

Remember, in that component, we're dispatching an event called `search-products`. In `Catalog`, up in the template, find the `<search-bar` element and add `@search-products="onSearchProducts"`, which is a method that we now need to create.

```
82 lines | assets/js/components/catalog.vue
1  <template>
  ... lines 2 - 9
10  <div class="col-9">
11    <search-bar @search-products="onSearchProducts" />
12  </div>
  ... lines 13 - 23
24 </template>
  ... lines 25 - 82
```

Down in the code, do it: add `methods: {}` and then `onSearchProducts()`. Since this method is going to be called when an event is emitted, it will receive an `event` argument. And because, when we emitted the event, we added a `term` key, we can use that here! We can say `this.searchTerm = event.term`.

```
82 lines | assets/js/components/catalog.vue
  ... lines 1 - 25
26 <script>
  ... lines 27 - 32
33 export default {
  ... lines 34 - 74
75   methods: {
76     onSearchProducts(event) {
77       this.searchTerm = event.term;
78     },
79   },
80 };
81 </script>
```

[Check it in Vue Dev Tools](#)

Let's go check it out! Back on the dev tools... if I type `disc`, the `searchTerm` in `SearchBar`, of course, updates. But if we look in `Catalog`, the search term *also* changed here! Yes!

[Filter the Products Client-Side](#)

We can *finally* filter the product list. Scroll up to the template. To render the products, we pass them into the `<product-list>` component: we're currently passing in *all* of the products. But when there is a `searchTerm` we *now* want this to be a *subset*.

This is, yet again, a situation where we need to reference a value in the template that requires some custom logic. In other words, it's computed property time! In preparation, pass `filteredProducts` to `product-list`.

```
93 lines | assets/js/components/catalog.vue
1  <template>
2    <div>
3    ... lines 3 - 14
15    <product-list
16      :products="filteredProducts"
17    />
18  />
19  ... lines 19 - 22
23  </div>
24  </template>
25  ... lines 25 - 93
```

Copy that name, go down and, above `created`, add `computed` with `filteredProducts()`. Inside the function, `if (!this.searchTerm)`, then we can just return `this.products`: the normal array of products.

```
93 lines | assets/js/components/catalog.vue
... lines 1 - 32
33 export default {
34  ... lines 34 - 58
59  computed: {
60    filteredProducts() {
61      if (!this.searchTerm) {
62        return this.products;
63      }
64  ... lines 64 - 67
68    },
69  },
70  ... lines 70 - 90
91  };
92  ... lines 92 - 93
```

But if there *is* a search term, return `this.products.filter()` and pass an arrow function with a `product` argument. I'm going to use the *super* hipster shortcut syntax: because I don't have any curly braces - just parentheses - this has an implied `return` statement. So, return `product.name.toLowerCase().includes(this.searchTerm.toLowerCase())`.

```
93 lines | assets/js/components/catalog.vue
... lines 1 - 58
59  computed: {
60  ... lines 60 - 64
65    return this.products.filter((product) => (
66      product.name.toLowerCase().includes(this.searchTerm.toLowerCase())
67    ));
68  },
69  },
70  ... lines 70 - 93
```

Basically, loop over all the products and return a new array containing *only* the products whose name includes the search term.

[Try it in the browser!](#)

Let's try it! Over in the browser, this is the *full* list of office supplies. If I type in `disk` ... yes! It shows just one! Try `dis` ... 2!

But... as cool and fast as this is... our JavaScript filtering has some serious downsides. First, if our products were paginated, this would *not* work: the user would only be searching through a single page of products! Yikes! And second, what if we wanted the search to *also* match on fields that are *not* shown in this list? Or maybe we have a super-cool Elasticsearch system that we want to use?

The point is: filtering on the client-side *might* work in some simple cases... but most of the time, you'll probably want to perform a search on the server via an AJAX call. Let's do that next!

Chapter 43: Async Computed Properties

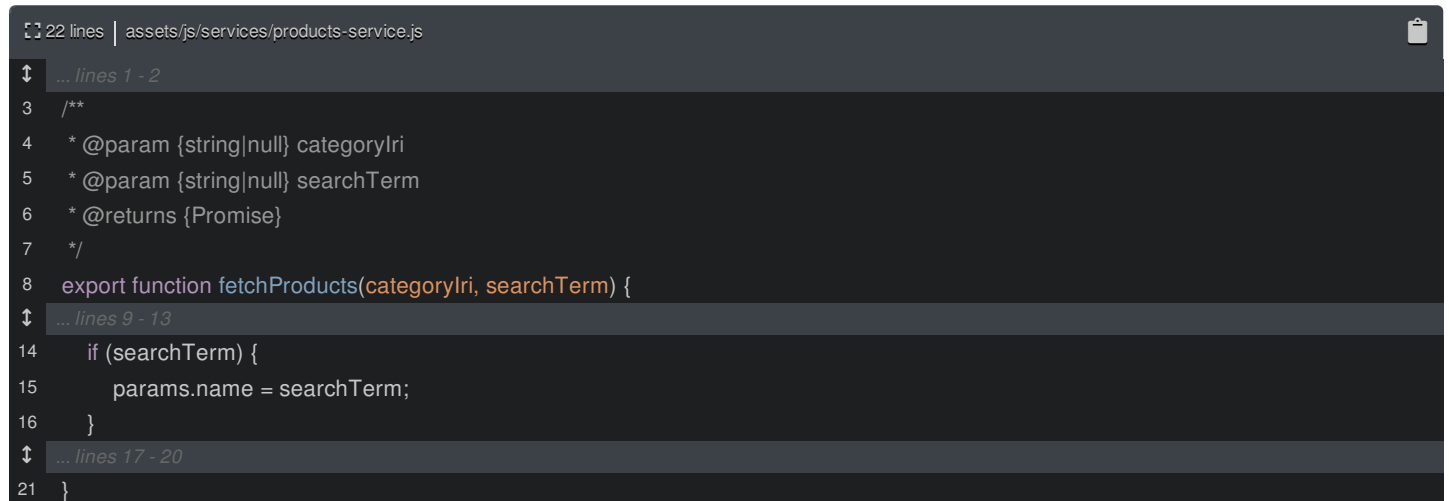
Our search is cool and fast... but it won't work if we start paginating the products... and the logic used in the search is *super* basic: we're just comparing the product name. In most cases, I'll want my search to be smarter than this: like via a more complex database query or by using something like Elasticsearch.

For both of these reasons, I want to refactor our search to fetch the data from the *server*, instead of doing it all in JavaScript.

Open a new tab and go to `/api/products.jsonld`: our shortcut to see what our API looks like. I've already added a basic filter to the API: you can add `?name=` and a term, and it will filter the results. This actually only filters on the `name` property. But you can make the search in your API as rich as you need. The important part is that our API has a way for us to send a search query and get a sub-set of results.

Adding Search to the Product Service

Back in our code, open up `services/products-service.js`. This function is responsible for making the AJAX request for products and it can already filter by category. Now, add a *second* argument called `searchTerm`. Then, very simply, if we have a `searchTerm`, say `params.name = searchTerm` to add the query parameter. I'll even document the new param: it will be a `string` or `null` and it's called `searchTerm`.



```
22 lines | assets/js/services/products-service.js
... lines 1 - 2
3  /**
4   * @param {string|null} categoryIri
5   * @param {string|null} searchTerm
6   * @returns {Promise}
7   */
8  export function fetchProducts(categoryIri, searchTerm) {
... lines 9 - 13
14    if (searchTerm) {
15      params.name = searchTerm;
16    }
... lines 17 - 20
21  }
```

Very nice!

AJAX inside a Computed Property?

Let's use this in `catalog.vue`. I'm going to do this in the *simplest* way possible first... and it's totally *not* going to work! We currently have a `filteredProducts` computed property, which *computes* the correct array of Products to use based on the `searchTerm` and `products`.

So the easiest thing to do is: if there is *no* `searchTerm`, return `this.products`. Else, if there *is* a `searchTerm`, let's make our API call! I'll copy the `fetchProducts()` line from below and say `const response = await fetchProducts()`. And, of course, Webpack is *mad* because this method now needs to be `async`.

```
93 lines | assets/js/components/catalog.vue
... lines 1 - 32
33 export default {
... lines 34 - 58
59   computed: {
60     async filteredProducts() {
... lines 61 - 64
65       const response = await fetchProducts(this.currentCategoryId, this.searchTerm)
... lines 66 - 67
68     },
69   },
... lines 70 - 90
91 };
... lines 92 - 93
```

Much better! To finish the function, I'll go steal some more code and say `return response.data['hydra:member']` .

```
93 lines | assets/js/components/catalog.vue
... lines 1 - 58
59   computed: {
60     async filteredProducts() {
... lines 61 - 66
67       return response.data['hydra:member'];
68     },
69   },
... lines 70 - 93
```

So... this makes sense, right? When we reference `filteredProducts` in the template, that will call our function, we make the AJAX call, wait for it to finish and then return the new array of products. Genius!

But... you can already see that we have an angry underline below `await` . Bah... let's ignore that and try it! I'll refresh and... oh... we *are* broken. *And* we have an error:

```
Invalid prop: type check failed for prop "products". Expected Array got Promise.
```

This is coming from `ProductList` and that prop is *passed* by `Catalog` . Yep, when we render `<product-list>` , the `filteredProducts` is no longer an Array. It's a Promise!

And... yea. That makes sense. When you make a function `async` , when someone calls that function, it finishes *immediately* and returns a `Promise` : it does *not* actually wait for your code to run. Then, later, when your function *does* finally finish its work, that promise *resolves*.

So this is a *long* way of saying that computed properties can *only* do synchronous work: they must return a value immediately. If you try to do something async, then you'll end up returning a `Promise` instead of the real value.

Nope, if you need to do something asynchronous, then you *can't* use a computed property. So, computed properties are "yay!" for calculating synchronous stuff but are "boo!" for async stuff.

Adding a new Data

The solution is to abandon your project and take up a peaceful career herding sheep. Or, you'll need to convert the computed property into a piece of data. And then, whenever that data needs to change - like whenever the `searchTerm` changes - you'll call a method that will *make* the AJAX call and *update* that data once its done.

So... let's do this! The first step is to add a piece of data that can hold the filtered products. But, actually, now that we're not trying to keep *all* the products in the `products` data so that we can filter based off of it, whenever our AJAX call finishes, it's now ok to *change* that `products` array directly. So instead of adding a *new* piece of data - we'll just change products.

Sweet! This means that, up in the template, we should change `filteredProducts` back to `products` . And back down, we can remove the computed section entirely.

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

```
1  <template>
2    <div>
3  ... lines 3 - 14
15    <product-list
16      :products="products"
17      :loading="loading"
18    />
19 ... lines 19 - 22
23  </div>
24 </template>
25 ... lines 25 - 86
```

Updating products on searchTerm Change

Here's the plan then: whenever the `searchTerm` changes, we basically want to re-run all of the code that makes the AJAX call and updates the `products` data... but with a minor addition to *also* include the search query.

To help re-use this, create a new method called `loadProducts()` with a `searchTerm` argument.

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

```
... lines 1 - 32
33 export default {
34 ... lines 34 - 61
62   methods: {
63 ... lines 63 - 66
67     async loadProducts(searchTerm) {
68 ... lines 68 - 82
83   },
84 };
85 ... lines 85 - 86
```

Now, copy the entire `created()` function... and paste. To include the `searchTerm`, pass that as a second arg to `fetchProducts()`. Oh and, of course, make this method `async`.

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

```
... lines 1 - 61
62   methods: {
63 ... lines 63 - 66
67     async loadProducts(searchTerm) {
68       this.loading = true;
69
70       let response;
71       try {
72         response = await fetchProducts(this.currentCategoryId, searchTerm);
73
74         this.loading = false;
75       } catch (e) {
76         this.loading = false;
77
78         return;
79       }
80
81       this.products = response.data['hydra:member'];
82     },
83   },
84 ... lines 84 - 86
```

Up in `created`, we only need `this.loadProducts(null)`. I'm using `null` because when we first load, there will be *no* search term. We *could* pass `this.searchTerm` ... but I'm going to delete that data in a minute.

```
86 lines | assets/js/components/catalog.vue
... lines 1 - 32
33 export default {
... lines 34 - 58
59   async created() {
60     this.loadProducts(null);
61   },
... lines 62 - 83
84 };
... lines 85 - 86
```

This was just a simple refactoring. And... if we reload the page! Yay, our refactoring did *not* make the site catch on fire. A win!

Back in the editor, the *last* step is to call `this.loadProducts()` whenever the search changes... which is *exactly* when `onSearchProducts()` is called! Add `this.loadProducts()` and pass `event.term`.

```
85 lines | assets/js/components/catalog.vue
... lines 1 - 60
61   methods: {
62     onSearchProducts(event) {
63       this.loadProducts(event.term);
64     },
... lines 65 - 81
82   },
... lines 83 - 85
```

Thanks to this, when `onSearchProducts()` is called, this will *start* the AJAX call. Later, when it finishes, the `products` data will get updated and the component will re-render.

And hmm. If you think about it: we don't even need the `searchTerm` data anymore. I'll look for it: we're setting it here... and initializing it in `data`. The `loadProducts()` method doesn't need it because we pass it as an argument.

So celebrate by removing the `searchTerm` data.

Moment of truth! Let's refresh the page to be safe and... type. Yes! This matches 2 products and `disk` matches one. In Symfony's web debug toolbar, you can see the AJAX calls. We now have the ability to make our search as *powerful* as our heart desires.

[Destructured Event Arg](#)

Oh, but there's one *tiny* other thing I want to show you. In `onSearchProducts`, we get the `term` with `event.term`. And... that's actually the *only* part of the `event` object that we're using. In the JavaScript world, you'll often see a method like this written as `{ term }`.

This is object destructuring: it grabs the `term` property from the `event` argument that's being passed and sets it as a `term` variable. It allows us to just say `term` below. We can even document this: the `term` param is a string. Extra credit if you describe the function above.

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

```
↑ ... lines 1 - 60
61   methods: {
62     /**
63      * Handles a change in the searchTerm provided by the search bar and fetches new products
64      *
65      * @param {string} term
66      */
67     onSearchProducts({ term }) {
68       this.loadProducts(term);
69     },
70   },
71   ... lines 70 - 86
87 },
88 ... lines 88 - 90
```

After this change... the search still works. But... wow! This is making a *lot* of AJAX calls! Even if we type *really* fast, it makes one AJAX call per letter! Let's fix that next by adding debouncing.

Oh, but before we do, it's not hurting anything, but since the `created()` function does *not* directly use `await` anymore, it doesn't need to be `async`.

Ok, onto the debouncing!

Chapter 44: Debouncing: Data can Hold Anything

The only problem is that we made our search *too* awesome. When I type... wow! Look at those are AJAX requests - one for *every* character I type. It's, sort of unnecessarily flooding our API.

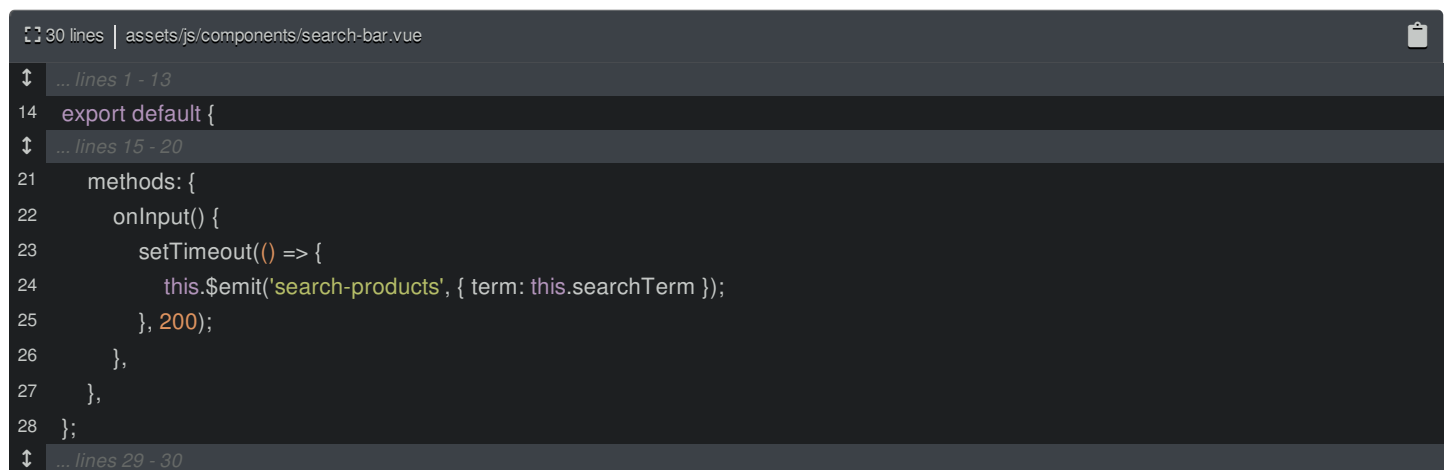
This is a common problem with a common solution: debouncing, which is *almost* as fun as it sounds. With debouncing, instead of sending an AJAX request after *every* letter, we wait until the user *stops* typing - maybe 200 milliseconds - and *then* make the request.

The *cool* thing is that we're going to be able to add debouncing *entirely* inside of the `search-bar` component. This means that our catalog code won't need to change at *all*; it will *instantly* be able to take advantage of it.

How can we do that? By delaying the custom `search-products` event until the user is done typing.

[Adding the setTimeout\(\)](#)

Start by adding `setTimeout()`, passing it an arrow function, then moving the `$emit` call inside. Set the timeout to 200 milliseconds. Now, instead of emitting the event immediately, it will be *slightly* delayed.



```
30 lines | assets/js/components/search-bar.vue
14 export default {
21   methods: {
22     onInput() {
23       setTimeout(() => {
24         this.$emit('search-products', { term: this.searchTerm });
25       }, 200);
26     },
27   },
28 };
```

Easy peasy! Oh, and the arrow function is important: if we used a traditional function, the `this` variable wouldn't be our Vue instance. Silly JavaScript!

Now, some of you probably realize that this isn't going to *quite* work yet. If we refresh... and then type really fast. Ah! It *still* sent four AJAX requests... it just waited 200 milliseconds before making each of them. Whoops!

[Storing and Clearing the Timeout](#)

To get debouncing to work, what we *really* need to do is, `onInput()`, if a timeout is currently active and *waiting* to be called, we need to *cancel* it and restart the timer.

To do that, we need to keep track of the return value of `setTimeout()`: it's a "timeout id". Then, the next time `onInput()` is executed we will call `clearTimeout()` and pass it that id.

None of this is too complex... but I *do* have one question: where should we store this "timeout id" so that we can reference it later? The easiest thing to do is to store it as data.

Add a new data called `searchTimeout` set to `null`.

36 lines | assets/js/components/search-bar.vue

```
... lines 1 - 13
14 export default {
... line 15
16   data() {
17     return {
... line 18
19       searchTimeout: null,
20     };
21   },
... lines 22 - 33
34 };
... lines 35 - 36
```

Then, in the function, say `this.searchTimeout = setTimeout()` . Now that this is stored, at the top of the function, *check* if it has a value: if `this.searchTimeout` , then `clearTimeout(this.searchTimeout)` .

36 lines | assets/js/components/search-bar.vue

```
... lines 1 - 21
22   methods: {
23     onInput() {
24       if (this.searchTimeout) {
25         clearTimeout(this.searchTimeout);
26       }
... lines 27 - 31
32   },
33 },
... lines 34 - 36
```

Oh, and to round this all out nicely, once the callback *is* finally called, we can reset the `searchTimeout` back to null.

36 lines | assets/js/components/search-bar.vue

```
... lines 1 - 22
23   onInput() {
... lines 24 - 27
28     this.searchTimeout = setTimeout(() => {
29       this.$emit('search-products', { term: this.searchTerm });
30       this.searchTimeout = null;
31     }, 200);
32   },
... lines 33 - 36
```

Now, if we type really fast, the second time `onInput()` is called, it will *clear* the timeout, and then, below, start a *new* one.

Let's try it! I'll refresh to be sure then... type super fast. Yes! Just one AJAX call down here to the categories API. That's *beautiful*!

[Non-Reactive Stuff on Data?](#)

For me, the most interesting thing about what we just did is that, in some ways, storing `searchTimeout` as *data* seems like an *abuse* of data. Normally we put something in data because we want it to be *reactive*: when that value changes, we want any component that uses it to re-render. For `searchTimeout` ... we don't really need that! We just needed a place to *stash* that value. But... this is *fine*. The main purpose of *data* is to store "reactive" data. But if you need a place to store something else, go nuts.

Next, if we have some business logic that we want to re-use between components, where should that live? Let's take our organization up to the next level!

Chapter 45: Business Logic Helpers

We've already been organizing our code in several ways. The biggest way is that we've been breaking our components down into smaller pieces, which is *awesome*! We also created services for fetching data, whether that's via AJAX calls or by grabbing a global variable.

But these aren't the only ways we can organize our code. In JavaScript, like with most languages, if you have a chunk of code that's complex or that you want to reuse, you can *totally* isolate that into its own file. This is *exactly* what we often do in Symfony with service classes.

Why Isolate Logic?

Look at `product-card.vue`. We created a computed property that takes the product's price, divides it by a hundred and then converts it into decimal digits to display in the template.

Having logic inside your components is a bit like having logic inside of controllers in Symfony. It's not the *worst* thing ever, but it makes your controllers harder to read. It also means that you can't reuse that code from *other* parts of your app *or* unit test it.

The same is true in JavaScript. By isolating logic like this into a separate file, we can keep our components readable, re-use logic and, if you want, unit test it.

But... I'm not going to put this logic into the `services/` directory because, at least in this project, I'm using `services/` to mean "things that fetch data". In functional programming, a *helper* is a term that's often used for functions that take input, process it and return something else. And *this* is *exactly* what our new function will do.

Create a Helper Function

So, inside of `js/`, create a new directory called `helpers/` and then a new file called `format-price.js`. In here, `export default` and, actually, let's use the more *hipster* arrow syntax to say that I `export default` a function.

```
5 lines | assets/js/helpers/format-price.js
1 export default (price) => {
2   ... lines 2 - 3
4 };
```

For the body of that function, go to `product-card`, copy the formatting code and... paste. But change to use the `price` argument.

```
5 lines | assets/js/helpers/format-price.js
1 export default (price) => {
2   return (price / 100)
3     .toLocaleString('en-US', { minimumFractionDigits: 2 });
4 };
```

Brilliant! Now, you might notice that ESLint is *angry*. Does it NOT like hipster code? How *dare* you, ESLint? Oh, no! Phew...! It says

Unexpected block statement surrounding arrow body. Move the return value immediately after the arrow.

My ESLint rules are set up so that if I have an arrow function that only has *one* line, and that one line a return statement, we should add parentheses around the statement then remove the `return` and the semi-colon at the end.

```
5 lines | assets/js/helpers/format-price.js
1 export default (price) => (
2   (price / 100)
3     .toLocaleString('en-US', { minimumFractionDigits: 2 })
4 );
```

That's now an *implied* return.

If you don't like that, just use the normal function syntax. And, to earn the admiration of our teammates, let's add some JSDoc: the price is a number and let's even describe what the function does.

```
11 lines | assets/js/helpers/format-price.js
1  /**
2   * Formats a price by adding a dot and normalizing decimals
3   *
4   * @param {number} price
5   * @returns {string}
6   */
7  export default (price) => (
8  ... lines 8 - 11
```

And... woo! We now have a *nice* reusable function! Oh, and there are two ways to organize your helpers... or JavaScript modules in general. First, you can have a file, like `format-price`, which exports `default` a *single* function. Or, if you have *several* different helper functions related to pricing or number manipulations, you could create a file called, maybe, `number.js` and then export *named* functions. That second idea is what we're doing inside of `services/`. It's up to you to decide which you like better.

[Using the Helper Function the Wrong Way](#)

Ok! Let's go use this inside of `product-card.vue`. The first thing we need to do is import it into the component. Do that with `import formatPrice from '@helpers/format-price'`

```
79 lines | assets/js/components/product-list/product-card.vue
... lines 1 - 34
35 <script>
36 import formatPrice from '@helpers/format-price';
... lines 37 - 56
57 </script>
... lines 58 - 79
```

Now, when I *first* started using Vue, I thought:

Hey! I now have a local variable called `formatPrice` in this file! So let's go right up to the template and use it! `formatPrice()` with `product.price` to reference the `product` object and its `price` property.

```
79 lines | assets/js/components/product-list/product-card.vue
1 <template>
... lines 2 - 16
17 <p class="p-0 d-inline">
18 <strong>${{ formatPrice(product.price) }}</strong>
19 </p>
... lines 20 - 32
33 </template>
... lines 34 - 79
```

But... it was not to be! If you move over to the browser's console... our dreams are crushed! It says:

Property or method `formatPrice` is not defined on the instance, but referenced during render.

Of course! When you reference a variable or function in a template we know that what it *really* does is call `this.formatPrice()`. It does *not* try to find some local `formatPrice` variable. On our instance, we *do* have a computed property called `price` but no methods.

[Using the Helper in our Component code](#)

So... we *can't* just import `formatPrice` and expect it to magically be available in the template. But we *can* use it in our JavaScript code, like in our computed property.

Change the template code back to `price` . Now, in the computed method, use the new helper: `return formatPrice()` , and pass the same thing we did before: `this.product.price` .

```
78 lines | assets/js/components/product-list/product-card.vue
... lines 1 - 37
38 export default {
... lines 39 - 45
46   computed: {
... lines 47 - 50
51     price() {
52       return formatPrice(this.product.price);
53     },
54   },
55 };
... lines 56 - 78
```

This time, when we move over... yes! It works perfectly! Let me get rid of the search term and... nice! No errors in the console.

Next: Let's make our search bar a *little bit* fancier by adding an X icon on the right to clear the search!

Chapter 46: Adding an [x] to our Search Bar

When we type inside the search bar, it actually already has a little X icon on the right. That's because we're using an `<input type="search"/>` for the search bar. And in *my* browser, this adds an X icon.

But... can we pretend that *doesn't* exist for a moment? Because I want to see if we can add our *own* little X icon that, on click, will clear the search term. This will be a *great* opportunity to practice our new dangerous skills!

Add the Button!

Inside of `search-bar.vue`, add a class to the outer div called `input-group`. That will let us add a new `<div>` element at the bottom, which will hold the "X" button. Give it `class="input-group-append"` and a `v-show`. Let's see: we only want to show the "X" button if the search term is *not* empty. Do that with `v-show="searchTerm" - != ""`.

```
48 lines | assets/js/components/search-bar.vue
1  <template>
2    <div class="input-group">
3    ... lines 3 - 11
12   <div
13     class="input-group-append"
14     v-show="searchTerm != ""
15   >
16   ... lines 16 - 20
21  </div>
22  </div>
23  </template>
24  ... lines 24 - 48
```

Now, inside the div, add a `<button>` - I'll talk about that ESLint error soon - with an X as the text and `class="btn btn-outline-secondary"`.

Ok ESLint: what's up? Hmm: `v-show should go before class`. Ah, we've seen this a few times before: attributes can go in *any* order, but there *are* some best-practices. Swap these two attributes and... ESLint is happy. I'm happy!

```
48 lines | assets/js/components/search-bar.vue
... lines 1 - 11
12  <div
13    class="input-group-append"
14    v-show="searchTerm != ""
15  >
16    <button
17      class="btn btn-outline-secondary"
18    >
19      X
20    </button>
21  </div>
... lines 22 - 48
```

Over in the browser... there we go! It looks a *little* silly because of the double X, but remember! We're ignoring that! I want to see if we can get *our* button working: on click, we want to clear the search bar.

Adding Behavior to the [x]

And... I know how to do this! I'll just ask Siri to do it for me! Or, we can listen to the click event! So we need `v-on`, but of course we'll use the shortcut. Say: `@click=""` set to a new method called `eraseSearchTerm`. Copy that method name and, below - we already have a `methods` section - so paste this as a second key.

```
52 lines | assets/js/components/search-bar.vue
1 <template>
... lines 2 - 15
16 <button
17   class="btn btn-outline-secondary"
18   @click="eraseSearchTerm"
19 >
... lines 20 - 23
24 </template>
... lines 25 - 52
```

Inside, we just need `this.searchTerm = ''`.

```
52 lines | assets/js/components/search-bar.vue
... lines 1 - 25
26 <script>
27 export default {
... lines 28 - 34
35   methods: {
... lines 36 - 45
46     eraseSearchTerm() {
47       this.searchTerm = '';
48     },
49   },
50 };
51 </script>
```

We're *crushing* it!

[Back to Testing in the Browser](#)

Time to test! Let's refresh... do a quick search... and click the X button. Boom! The search cleared. Oh, but the *products* did *not* update! Siri, could you.. update those for me?

[Emit on Clear Search](#)

When we think about the `search-bar` component, the `searchTerm` data is *entirely* internal. The only reason we have a `searchTerm` data at *all* is just to make our life easier... *inside* that component. For example, it helps us up here to know whether or not we should hide or show the X button.

But for people that *use* this component, all *they* know and care about is that `search-bar` *emits* a `search-products` event. And *that* is what we're missing in `eraseSearchTerm()` ! We correctly updated the internal data, but we *forgot* the most important part: emitting the event.

Copy the `$emit()` statement and use it here. Of course we know `searchTerm` will always be empty, but this will work fine.

```
53 lines | assets/js/components/search-bar.vue
... lines 1 - 34
35   methods: {
... lines 36 - 45
46     eraseSearchTerm() {
47       this.searchTerm = '';
48       this.$emit('search-products', { term: this.searchTerm });
49     },
50   },
... lines 51 - 53
```

[Check that it all Works!](#)

Now when we go over... search for something cool... and clear it... yea! We got it! Making this work was nothing new, but it was a great exercise to think about the internal and external parts of our `search-input` component.

Next let's do something I'm *really* excited about! The `currentCategoryId` is something that we set on page load and then never change. But we've organized our app in *such* a good way that with a *little* bit of logic and a new concept called a watcher, we're going to be able to dynamically change the category and have the whole page update. Woh. Let's do this!

Chapter 47: Watchers: The Good, The Bad & The Useful!

One of the things that we need to know on every page load is what the current category is. To get that, we pass it from the server to Vue by setting a `currentCategoryId` global variable. To make that a bit nicer, we even created a `page-context` service that reads the global variable for us. We use this info to highlight which category is active on the sidebar and *also* to filter the products.

One thing that we *could* have done is just call the `getCurrentCategoryId()` function in every component that needed that info. For example, in `sidebar.vue`, where we need the current category, we could have imported the `page-context` service and called `getCurrentCategoryId()`. And we could have done the same thing in `catalog`. That would be safe because the `currentCategoryId` is *not* something that changes: it's not one of those things where, when it changes, we need our component to re-render. We don't need it to be *reactive*.

But now, I *do* want to do this. Here's our new mission: make it possible to change `currentCategoryId` while our app is running and for the sidebar and products to instantly update. This *now* means that we *do* need `currentCategoryId` to be "reactive". And *that* means it needs to live as `data` in a component.

Look at the component tree in the Vue dev tools. You know the drill: if both `<Catalog>` and `<Sidebar>` need `currentCategoryId`, then it needs to live as `data` in `<Products>`. *Then*, we can pass it down via `props`.

By the way, in Vue 3, it *is* possible to have reactive objects outside of a Vue component.

The good news is: we planned ahead! Look inside of `products.vue`. Hey! This has a `currentCategoryId` computed property, which calls - I'll hold command or control and click - the `page-context` service. Then, we pass this down to the other components via a prop.

[Replace the Computed Property for Data](#)

This means that all we need to do is change `currentCategoryId` from a computed prop to `data` and... everything should just work! What could go wrong?

Remove this computed property and, up in `data`, add up a new variable called `currentCategoryId`. I'll set its initial value to `getCurrentCategoryId()`. We *can* still use the service to get the initial value.

```
62 lines | assets/js/pages/products.vue
... lines 1 - 22
23 <script>
... lines 24 - 28
29 export default {
... lines 30 - 34
35   data() {
36     return {
... lines 37 - 38
39       currentCategoryId: getCurrentCategoryId(),
40     };
41   },
... lines 42 - 59
60 };
61 </script>
```

If we go to the browser, everything still works wonderfully! It's *still* passing down `currentCategoryId` in *exactly* the same way. The cool thing now is, I can go the Vue Dev tools, click on `<Products>`, scroll down to the `currentCategoryId` and change it. Let's, say `24` and... boom! Our app updated! The new category is highlighted in the sidebar *and* we can see the correct title. Victory!

Oh... but it did *not* change the list of products! Let's try this again: I'll change to `25` - your id numbers will probably be different - and... again! It updated the `sidebar` and `title` ... but *not* the product list. What's going on?

[Products Data Depends on a Changing Prop](#)

Put on your debugging hat and dive into `catalog.vue`, the component that holds the `products` data. The `products` data depends on two things. If we go down to `loadProducts()` - the method that actually makes the AJAX call - we can see that `products` depends on the `searchTerm` and *also* on `currentCategoryId`. This means that when *either* of these change, we need to *re-call* `loadProducts()` so that it will make the new AJAX request.

Making sure that `loadProducts()` was called when the `searchTerm` changes was easy because the `search-bar` component already emits an event whenever the search changes. We then call the `onSearchProducts()` method, *that* calls `this.loadProducts()` and *that* makes the AJAX call and changes the `products` data. *That* part is wonderful.

The question *now* is: how can we run code when `currentCategoryId` changes? How can we detect when a prop changes?

Here's the answer: When there is *no* other event or hook that you can listen to and you simple *must* execute code when a prop or data changes, you can use something called a `watcher`! Very simply: a watcher is a function that's called by Vue whenever a specific prop or data changes.

[Watchers](#)

I'm not gonna lie. Watchers *kind of* get a bad reputation. But there's nothing wrong with them. The problem is that, a lot of times, people use watchers when there's actually a *better* solution available. For example, look back at `search-bar`. Remember: we're using `v-model` in the template to bind our input to the `searchTerm` data. I also added `@input` so that, down here, we could emit the custom event.

Another way that we could have done this is, instead of listening with `@input`, we could have added a watcher for `searchTerm`: a function that is called whenever that data changes. Then, whenever `searchTerm` changes and our watcher function was called, we would emit the custom event.

The reason I *didn't* do that is, as I just said, watchers are kind of your last resort. They're not as performant as other parts of your system. In this case, we *did* have another option: listening via `@input`.

But looking back in `catalog`, there's simply *no* other solution. We need to run code when the `currentCategoryId` prop changes. And we can't create a computed prop called `products` because the code we need to run is *async*. *That* is why we're going to use a watcher.

Let's do it next!

Chapter 48: Adding a Watcher

Let's add a watcher! Create a new option called `watch: {}` set to an object of property names. For instance, since we want to run code when `currentCategoryId` changes, we'll add `currentCategoryId` set to the function that will be called. When Vue executes this, it will pass us two arguments: the new value of `currentCategoryId` and the old value. For now, just `console.log(newVal, oldVal)` .

```
95 lines | assets/js/components/catalog.vue
... lines 1 - 25
26 <script>
... lines 27 - 32
33 export default {
... lines 34 - 57
58   watch: {
59     currentCategoryId(newVal, oldVal) {
60       console.log(newVal, oldVal);
61     },
62   },
... lines 63 - 92
93 };
94 </script>
```

Let's see if this works! Move over to the browser: the console looks clear. Since we haven't *actually* added a way to change the `currentCategoryId` from within our app, head over to the Vue Dev Tools and change it there. I'll set it to 24, go back to the console and... yes! You can see the new value first and then the old value.

[Load Products When currentCategoryId Changes](#)

Back to `catalog.vue` ! I'm actually going to delete these two arguments because I don't need them. Instead, our code can reference the new value directly via the `currentCategoryId` prop. Ok: when the `currentCategoryId` changes, we want to call `this.loadProducts()` . For the search term pass `null` for now.

```
95 lines | assets/js/components/catalog.vue
... lines 1 - 57
58   watch: {
59     currentCategoryId() {
60       this.loadProducts(null);
61     },
62   },
... lines 63 - 95
```

This will trigger `loadProducts()` ... which in turn will *read* the new `currentCategoryId` , get our products back from the server and update the `products` data. It's a fool-proof plan!

So let's check it out! I'll go back to the Vue Dev Tools, change the data to 23 and... ah! It works! With the loading screen and everything! I *love* that we can even change it to `null` and it goes to "All products". Awesome!

The only rough part is that if I, for example, search for "disk"... Actually let's try this under "office supplies". Search for "disk"... Then go find the `<Products>` component in the Vue Dev Tools and change the `currentCategoryId` to... let's say 24 .

Ah! It returns everything! It did *not* apply the `searchTerm` on the products. The reason, of course, is that, in our watcher, we're passing `this.loadProducts(null)` . Yep, we're saying:

```
Hey Vue! Load the products with no searchTerm .
```

And darn it, Vue is following our directions perfectly!

Suddenly we Do Want a searchTerm Data

If you look at data, we do *not* have `searchTerm` as a data key. Why? Because, until now, we didn't need it! All we needed to do - when `onSearchProducts()` was called - was use the `term` to immediately load the products. There was no need to store the search term anywhere for later. But now we *do* have a need! We *do* need to store the `searchTerm` so that when the `currentCategoryId` changes, we *know* what the `searchTerm` is.

Ok! Re-add `searchTerm` to data. And, down in `onSearchProducts`, say `this.searchTerm = term`. We can now *remove* the `term` argument from `loadProducts` and just say `this.searchTerm` instead.

```
97 lines | assets/js/components/catalog.vue
... lines 1 - 32
33 export default {
... lines 34 - 50
51 data() {
52   return {
... lines 53 - 55
56     searchTerm: null,
57   };
58 },
... lines 59 - 66
67 methods: {
... lines 68 - 72
73   onSearchProducts({ term }) {
74     this.searchTerm = term;
75     this.loadProducts();
76   },
77
78   async loadProducts() {
... lines 79 - 81
82     try {
83       response = await fetchProducts(this.currentCategoryId, this.searchTerm);
... lines 84 - 85
86     } catch (e) {
... lines 87 - 89
90     }
... lines 91 - 92
93   },
94 },
95 };
... lines 96 - 97
```

That looks good! Now... let's see: up in `created()`, we don't need any arguments... and same in the `currentCategoryId()` watcher.

```
97 lines | assets/js/components/catalog.vue
... lines 1 - 32
33 export default {
... lines 34 - 58
59 watch: {
60   currentCategoryId() {
61     this.loadProducts();
62   },
63 },
64 created() {
65   this.loadProducts();
66 },
... lines 67 - 94
95 };
... lines 96 - 97
```

Each will now automatically use the current search term. Let's try it!

[Try it one more time!](#)

I'll click on "Office Supplies" to get a full page refresh. Search for "disk"... and go over to the Vue Dev Tools one more time. Click on `<Products>` . change the `currentCategoryId` to 24 and... wow! No products found! If I change this back to `null` for "all categories"... it *does* find the product!

Of course, it's not updating the URL when we change the category... which it probably should. For that, we would need a Vue Router: a key component of single page apps - or even "mini" single-page "sections" that you might create on part of your site. With the Router, we could click on these links and have the URL change *without* a full page refresh. We'll save that topic for another time.

People! Friends! You made it through our first, *gigantic* Vue tutorial! Congrats! You deserve a snack... and probably some outside time.

There *are* more things to talk about - we'll save those for a future tutorial - but wow! You are already *incredibly* dangerous. I hope you enjoyed this process as much as I have! Vue is a *very* powerful, *very* fun tool to work with. So go build something *awesome*... then tell us about it!

And, as always, if you have any questions, comments or kitten videos, let us know down in the comments. All right, friends, seeya next time!

