**Module Bundling with webpack**

**Lesson Duration: 75 minutes**

**Learning Objectives**

* Be able to write a simple webpack config
* Be able to use webpack to generate a bundled script of multiple modules

**Intro**

We’ve seen that we can send JavaScript files, or a module of code, to the browser using <script> tag with a src attribute set in our HTML file. This is great, we can keep our HTML, CSS, and JS in separate files. Then the <script>s and <link>s bring them together in the browser to make our site work.

As our JavaScript web applications get larger, however, we’ll want to organise our JS code a bit better. We want to put different pieces of code in different files. We want to make re-usable modules to help us avoid repeating ourselves in our code. We might have data models, maybe some helper functions, or views for our UI logic components.

This can get a bit messy and hard to follow. Lots of different files are all depending on one another, all required in our index.html. Module bundlers like webpack allow us to write our code in a more structured way while avoiding the problems that arise when writing larger applications.

**Project Without webpack**

Instructor note: Hand out the start code

This project uses a similar structure to what we’ve been using so far. It has more JavaScript files, but we don’t need to worry too much about exactly what they do. We’re just using it to look at some of the issues that module bundlers like webpack help us avoid.

**Multiple Script Tags**

Load up the project in the browser and fill in the form. When we submit the form, we should see an error in our devtools console.

Uncaught ReferenceError: TextFormat is not defined

When we look at where the error is coming from, we can see that our app.js uses the Person class from models/person.js, and in the Person class, we’re expecting an object called TextFormat to be defined. We have a dependency on some other code which our app can’t seem to find. Why might this be?

Our site only knows about the files we tell it about. In our index.html file, we have some <script> tags to tell the browser to make additional requests to load our JavaScript files.

*<!-- indexl.html -->*

<script src="js/models/person.js"></script>

<script src="js/app.js"></script>

If we look at the code in these files, TextFormat isn’t defined anywhere. TextFormat is defined in helpers/text\_format.js, so we need to add another <script> tag to load this file as well.

*<!-- indexl.html -->*

<script src="js/helpers/text\_format.js"></script> *<!-- NEW -->*

<script src="js/models/person.js"></script>

<script src="js/app.js"></script>

Let’s reload the page and try the form again.

Uncaught ReferenceError: RandomAdjective is not defined

Not again! We’ve made it further through our code, but now we can’t find the RandomAdjective class. We forgot another file, models/random\_adjective.js. So we have to add that to our index.html as well.

*<!-- index.html -->*

<script src="js/models/random\_adjective.js"></script> *<!-- NEW -->*

<script src="js/helpers/text\_format.js"></script>

<script src="js/models/person.js"></script>

<script src="js/app.js"></script>

Now we have all the code we need in our HTML file to make our app work.

This is a real pain. As our application grows and we add more files, we have to keep coming back to include new <script>s in our HTML. If we forget one, our app breaks, and not for a particularly good reason, our JavaScript code could be fine, and it still breaks because a dependency is missing.

This is also problematic because the specific order of our <script> tags can matter. They run in the order that they are listed in. It doesn’t affect this app, as we don’t do anything before the DOMContentLoaded event, by which time, all of the scripts have been run. In a different situation this could cause problems. It’s something to be aware of, just in case.

**Paired discussion: (5 minutes)**

When a class or file uses another class, file, module, function, or library, it is dependent on it. The first module relies on the dependency module. It cannot function without it.

Look through the JavaScript files in our project and draw out the hierarchy of dependencies.

Instructor note: as the students give these answers, write the dependencies of each file up on the board for use in a later task.

**Answers**

**Implicit Dependencies**

Figuring out the dependencies between files (or modules) in this project is a bit harder than it needs to be. The dependencies between different files are not explicit.

If we see TextFormat.capitalise being called in person.js, there is no information in person.js about where that object is supposed to come from. Is it an external library loaded into the page over the internet by a <script> tag? Is it another file in our js directory within our project? Is it from an npm package?

This problem only gets worse as we add more and more files to our project.

**Introducing webpack**

Using webpack is going to address both of these issues:

* The issue of maintaining a huge list of <script> tags to import our code
* The issue of dependencies between modules being implicit, rather than explicit.

Webpack is a module bundler. It takes our source code, in the form of multiple files all nicely split up in a modular, reusable way, and wraps them up into one big “bundle” file. In order for webpack to follow our dependencies between files, we’re going to need to make these dependencies explicit. Webpack supports the same syntax that we used in node, module.exports = myClass to export from a file, and require('./path/to/my\_file.js') to import one file into another.

We’ll send this bundle of JavaScript code to the user’s browser along with our HTML and CSS, instead of sending our JavaScript source code directly.

**Directory Structure**

Currently we’re sending our whole project directory with all our code to the user’s browser. With webpack, we’re going to separate our JavaScript source code from the public-facing code that the browser will actually run.

We’re going to move our current code into a new directory named “public”. This will contain the files that will actually be sent to the user’s browser:

* index.html
* main.css
* bundle.js - the bundled up .js file that webpack will generate from our JS source code.

Note: Close any files that are open in your text editor before moving them, otherwise they may be saved in their original locations again.

mkdir public

mv index.html public

mv css public

mkdir public/js

Now let’s rename our JavaScript source code directory src. This is the conventional name for a directory containing source code.

mv js src

Our new directory structure should look like this:

.

├── public

│   ├── css

│   │   └── main.css

│   ├── index.html

├── src

│   ├── app.js

│   ├── helpers

│   │   └── text\_format.js

│   └── models

│   ├── person.js

│   └── random\_adjective.js

**Installing webpack**

Webpack is a package on npm, which we’ll run in node, preparing our files before we send them to the browser environment. We’re back in the land of npm and all the amazing libraries that it gives us access to. Before we can install webpack, we need to make our project an npm project.

npm init -y

Now let’s install webpack, we also need to install the webpack-cli package.

npm i -D webpack webpack-cli

webpack is a dev-dependency as it’s not actually used while our app is running in the browser. As the developers, we use it at build-time, the user doesn’t have any webpack code running in their browser. They only need our code in the form of a bundle file created earlier, by webpack.

Let’s try it out, by writing and then running an npm script in our package.json.

// package.json

**"scripts"**: {

**"build"**: "webpack",

**"test"**: "echo \"Error: no test specified\" && exit 1"

},

We can call our npm scripts whatever we like. We’re calling this one "build". Because "build" is not a built-in npm script name (like "test" is), to get npm to find our script we have to tell npm to run a script, and then pass the name of the script we want to run.

npm run build

Ah, we’ve got a couple of warnings and errors here. We need to configure webpack to let it know where our source code is, and where to put our bundled up code.

**Configuring webpack**

Webpack looks for a config file called webpack.config.js in the root of our project, along with our package.json.

touch webpack.config.js

In this file, we tell webpack a bit about our project’s structure, and how we want webpack to work.

Paste this code into the file, and then we’ll talk about what each part is doing.

*// webpack.config.js*

**const** config **=** {

entry: `${\_\_dirname}/src/app.js`,

output: {

path: `${\_\_dirname}/public/js`,

filename: 'bundle.js'

},

mode: 'development'

};

module.exports **=** config;

First we create a configuration object and export it from the file. This is so that webpack, running in node, can import our file.

**const** config **=** {

*//...*

};

module.exports **=** config;

We tell webpack where to start looking at our code. What is the “entry point” into our application?

entry: `${\_\_dirname}/src/app.js`,

\_\_dirname is a special node variable that tells us the directory of the current file. So in webpack.config.js, it will be something like /Users/your\_username/some/more/directories/module\_bundling\_with\_webpack.

Also, we need to tell it where to put our code once it’s bundled it all up. Where do we want the bundle.js file to go?

output: {

**path**: `${\_\_dirname}/public/js`,

filename: 'bundle.js'

},

};

For the last option we need to tell webpack whether we’re in 'development' mode or 'production' mode. Are we working on developing our app, or are we ready to deploy it to our production web server and send it out into the world?

mode: 'development'

};

Webpack has thousands of configuration options and plugins, but this one option will set it up with some sensible default settings for webpack as a development tool. If we set it to 'production' it would behave differently, optimising the file size for us etc.

Every project that uses webpack will need this config file.

**Running webpack**

Now when we run webpack again, we get what we want: A bundle.js file in our public/js directory.

npm run build

**Updating our Project to use the Bundle**

This bundle file is automatically generated by webpack, based on our source code in the src directory. This means that the bundle file itself is not important, it can be re-generated at any time by running webpack. We don’t want to upload our bundle.js file to git, as it’s not necessary and will only cause problems. Every commit we would have to re-commit the updated bundle, we’ll end up getting conflicts in the bundle file when working with others on a project over github, it’ll just be messy. So let’s tell git to ignore the bundle.js file.

touch .gitignore

**node\_modules/**

**bundle.js**

**<script> Tags**

When we run public/index.html in the browser, we should see errors in the devtools console.

GET file:///Users/username/path/to/file.js net::ERR\_FILE\_NOT\_FOUND

The index.html is still looking for our individual files, and we’ve moved them. We have to update our <script> tags. We can delete all four of our old script tags, and replace them with just one, for bundle.js.

*<!-- public/index.html -->*

<script src="js/bundle.js"></script> *<!-- MODIFIED -->*

*<!-- DELETED -->*

Now we won’t need to touch our <script> tags again.

Let’s reload the project in the browser.

**Explicit Dependencies with require**

We’ve found our bundle.js file, but when we submit the form again, we see a familiar error.

Uncaught ReferenceError: Person is not defined

Our old friend “not defined” again. Why was this happening at the start of the lesson? We hadn’t included some of our code in the project. Before the reason was missing <script> tags.

Now that we’re using webpack, the solution is much nicer. Because webpack runs in node, before the code even gets to the browser, we have access to require and module.exports again 🎉

**Task: 10 minutes**

We mapped out the dependencies between the files earlier.

Instructor note: Recap the dependencies on the board

Using require and module.exports, you need to make these dependencies explicit so that webpack can bundle all of the necessary code, and our app works again.

**You should not need to modify the code at all, only adding requires and module.exports**

**To run the updated code in the browser, you’ll need to run npm run build again every time. Changing the source code won’t change the bundle file until we re-run webpack.**

Steps:

1. Add a require and/or a module.exports
2. run npm run build in the terminal to create a new bundle.js which reflects your change to the source code
3. Refresh the browser and check the console for errors
4. GOTO 1.

**Example solution**

Our dependencies are now explicit. We can look at a file, for example person.js, and right at the top we can immediately see all of its dependencies.

**Webpack’s Watch Mode**

Re-running webpack every time we make changes gets pretty annoying pretty quickly. Thankfully, there’s an option to tell webpack to watch our files for changes, and generate a new bundle whenever we save a relevant file.

Let’s update our "build" script in our package.json to use the -w flag (short for “watch”)

// package.json

**"scripts"**: {

**"build"**: "webpack -w", // MODIFIED

**"test"**: "echo \"Error: no test specified\" && exit 1"

},

If we re-run webpack one last time, it will stay running, watching our files for changes. Every time we make a change to our source code, and save our files, webpack builds a new bundle.js.

Try changing the adjectives in src/models/random\_adjective.js to test this out. Maybe we want only positive adjectives to describe the colours. When you save the file, check the terminal tab running webpack, and you’ll see it’s built the bundle again. When we refresh the app in the browser, our changes are reflected in the new bundle file without us having to touch webpack.

**Recap**

What problems did webpack solve in this project?

**Answer**

How do we run webpack?

**Answer**

What file to we need at the root of our project to configure webpack?

**Answer**

What does webpack’s -w option do?

**Answer**

**Conclusion**

Webpack allows us to structure our front-end code in multiple files and folders more easily. Allowing us to keep our front-end code organised and following sensible patterns and best practices, even as our web-apps grow larger.

Webpack also allows us to use some Node.js specific features, which usually can’t be used in the browser, such as require. This means that we can use packages from NPM in our front-end code, such as Mocha for unit testing anything that doesn’t interact with the DOM.

**Intro to Vue**

**Duration: 45 minutes**

**Learning Objectives**

* Understand why the need for frameworks has arisen
* Be able to configure and install Vue, with Webpack
* Be able to set up a single page Vue app
* Be able to use events and methods in our apps

**Introduction**

We’ve seen that manipulating the DOM with JavaScript has a few issues.

* It is repetitive
* It is mistake-prone
* It is verbose and difficult to manage

In addition, it is very challenging to architect larger JavaScript applications, while making sure they are structured properly.

In the last couple of years, a number of frameworks have arisen to solve this problem. These frameworks help us to minimise the type of DOM manipulation calls (createElement, querySelector and appendChild, to name a few) we’ve been making so far, and in some cases can increase the performance of our applications. Out of the frameworks that have arisen in recent history, React (maintained by Facebook) and Angular (largely maintained by Google) are two of the key players, and the third is [VueJS](https://vuejs.org/" \t "_blank).

Vue is a popular, community-driven framework whose job it is to give structure to front-end JavaScript applications. It is an unopinionated library that provides a number of different ways of doing things. This week, we’ll look at two approaches; one where our app lives in a single file, and one where we split our app into multiple components.

**Why Are We Learning These Two Approaches?**

Vue’s versatility means that it can be implemented in many ways. By initially creating a Vue instance in a single JavaScript file, we’re building something that could, when used correctly, be integrated into other existing apps - if you wanted Vue to simply handle how a form works in your pre-existing app, for instance, you could plug in a small Vue instance and have it do just that.

The second approach - using multiple components, means we can also build an entire Single Page Application from the ground up, in conjunction with Vue’s CLI tooling.

One of the best things about Vue is that in terms of the JavaScript frameworks out there, it’s relatively easy to learn, and fun! To use it, we’ll start by walking through an initial setup.

**Installing and Configuring Vue**

First, let’s install and configure Vue. We can do this with NPM, as normal.

npm install vue

Secondly, in order to bring in Vue, we’re going to have to add to our Webpack configuration.

*//webpack.config.js*

**const** config **=** {

entry: `${\_\_dirname}/src/app.js`,

output: {

path: `${\_\_dirname}/public/js/`,

filename: 'bundle.js'

},

resolve: { *// NEW*

alias: {

'vue$': 'vue/dist/vue.esm.js'

},

extensions: ['\*', '.js', '.vue', '.json']

},

mode: 'development'

};

module.exports **=** config;

This section tells Webpack to look in the vue folder of our node\_modules and use a version of Vue which is an EcmaScript module. (Which means we can import it!)

We should be able to run npm run build now, and open the index.html file in our browsers. We’re ready to begin!

**Setting up Vue**

Let’s open up our index.html file and take a look. What can we see?

*<!-- public/index.html -->*

<body>

<div id="app">

</div>

</body>

You should be pretty comfortable with this file; there’s really nothing unusual about it, except that we’re starting with a <div id="app"> in place. This is the div that Vue will use to hook into our HTML, to display what the user should see. We can leave this alone for the time being.

Let’s look at our app.js file now. In order to set up a Vue application, we need to set up an event listener first, so that we can be sure the DOM has loaded. (Even although we don’t have to worry about DOM manipulation directly, this is what Vue is doing behind the scenes.)

*//app.js*

document.addEventListener('DOMContentLoaded', () **=>** {

});

Next up, we need to create a new instance of a Vue application. And to do that, we have to import Vue.

The import statement is an ES6 feature and works in essentially the same way as require, however it gives us more flexibility. To import features into a file they have to have been exported elsewhere. export allows us to export various functions and objects from the same module, unlike module.exports in which you can only export the one object.

**import** Vue **from** 'vue'; *// NEW*

document.addEventListener('DOMContentLoaded', () **=>** {

**new** Vue(); *// NEW*

});

This Vue function can take in an object with several keys. Let’s add the first key we will need, el. The el key tells Vue where the root element exists; the element where our Vue app will live. Since we want it to live in the #app DOM element, that’s going to be the value of this key.

**new** Vue({

el: "#app" *// NEW*

});

el takes a CSS selector, so we’re looking for the element with ID app.

Next, let’s set up a really simple “Hello World” example. To do this, we need to understand some of the basics of Vue.

**Two Way Binding**

One of the key features of Vue is its use of “Two Way Binding”. Let’s consider a software application that has two parts:

1. Our view; the part the user sees.
2. Our model, where any logic is carried out.

Two way binding sets up a relationship between the two, so that if the model changes, the view automatically updates! (And vice-versa, changes to the view will be updated to the model. For example, within a form.)

This is *really* helpful for us - it means that we can work in JavaScript to update our model, and the result will be displayed to the user! To begin with, we’ll look at binding our Vue instance to our HTML in one direction - specifically that when our JavaScript changes, our HTML updates. We’ll look at how we can change things in the other direction in the next lesson.

Let’s see how to set this up, going back to our app.

**Binding Our Vue Instance To Our HTML**

It’s really easy to set up binding in Vue. Firstly, we’re going to declare a dynamically bound variable in our index.html file.

*<!-- index.html -->*

<div id="app">

{{ greeting }} *<!--NEW-->*

</div>

Next, we’re going to set up the variable on our Vue instance. We need to declare a key called data, and set it to be an object. You will also see this data object referred to as state.

*//app.js*

**new** Vue({

el: "#app",

data: { *// NEW*

}

});

Within this data object, we can set any variables we want to use in the DOM:

*//app.js*

**new** Vue({

el: "#app",

data: {

greeting: "Hello World!" *// NEW*

}

});

Now, if we refresh the page, we should see our greeting. That’s a lot of work to set up a “Hello World”. What have we gained here?

Any changes we make to the greeting variable will immediately be reflected in our app! To see the power of this, let’s look at events.

**Capturing your first event**

So far, our index.html file is pretty straightforward, just consisting of a single dynamically bound variable. Our Vue templates can also consist of HTML, as normal. Let’s adjust our HTML to demonstrate.

*<!-- index.html -->*

<div id="app">

<h1>{{ greeting }}</h1>

<button>Change Greeting</button> *<!--NEW-->*

</div>

So we’ve added a few HTML elements; a heading, and a button. Our next task is to make something happen when we click on the button. We know how to do this with plain JavaScript - we can use addEventListener to hook into certain events. In Vue, it’s arguably easier. Let’s add an event listener to the button directly.

*<!-- index.html -->*

<div id="app">

<h1>{{ greeting }}</h1>

<button v-on:click="updateGreeting">Change Greeting</button> *<!-- MODIFIED -->*

</div>

v-on: is a Vue *directive*. There are lots of directives, but v-on: lets us listen for events. The second part tells view that we’re listening for a click event, and this can be any of the events we’ve seen so far - click, submit, change etc.

You should note that as well as using v-on:click, you might see the following too:

*<!-- index.html -->*

<div id="app">

<h1>{{ greeting }}</h1>

<button @click="updateGreeting">Change Greeting</button> *<!-- MODIFIED -->*

</div>

In this form of the syntax, we’re simply replacing the v-on:eventName with @eventName. We’ll be using v-on:click in our course material, but you can use whatever syntax you’re more comfortable with.

There are lots of other directives, but for now, let’s focus on events.

We’ve said that when the button is clicked, we should execute a function called updateGreeting. Where should this be written?

We need to declare any methods we want to use on our Vue object, as follows:

* Add an object to your Vue instance, with the key of methods
* write any methods you need here as keys / values

*//app.js*

**new** Vue({

el: "#app",

data: {

greeting: "Hello World!"

},

methods: { *// NEW*

updateGreeting: **function**() {

**this**.greeting **=** "Changed!";

}

}

});

Notice that within these methods, we need to refer to this.variableName to get access to the greeting variable. So this.greeting, for example.

We can try this out - refresh your browser window and click the button!

**Arguments**

We can also pass arguments to the methods we call.

*<!-- index.html -->*

<button v-on:click="updateGreeting('John')">Change Greeting</button> *<!-- MODIFIED -->*

We have to watch out for single / double quotes here! Finally, let’s update our method.

*//app.js*

methods: {

**updateGreeting**: **function**(name) {

**this**.greeting **=** "Hello, " **+** name; *// MODIFIED*

}

}

So we’ve fulfilled the promise of two-way binding: *when we change the model variable, our view automatically updates*. Whatever the greeting key contains, that’s what will be displayed in the template.

**Recap**

What key does our Vue instance require to hook into our HTML?

Answer

What is the benefit of the two-way binding our Vue instance affords us?

Answer

**Conclusion**

Using Vue allows us to focus on the *logic* of our application, rather than on boilerplate DOM manipulation. Through the power of two-way binding, we can think in terms of our JavaScript, allowing our views to take care of themselves.

Next, we’ll take a look at a few more directives, and see how can work with arrays, loops, and conditionals.

# Vue: Loops and Conditionals

**Duration: 90 minutes**

### Learning Objectives

* Be able to work with Vue’s devtools
* Be able to work with Loops and Arrays in Vue
* Be able to work with Conditionals in Vue
* Be able to work with forms and Vue’s v-model directive
* Be able to implement Vue’s v-bind directive
* Understand when to implement Vue modifiers

## Introduction

We’re going to spend the next little while making something a bit more complex in Vue. But before we do, we need to look at our tooling.

One of the drawbacks of two-way binding is that it can be a little difficult to visualise the state of our program; the all-important data object.

To help with this, we’re going to install [Vue’s devtools for Chrome](https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd).

Once these are installed, we need to change one small setting.

* Right click on the Vue icon in the toolbar, click “Manage Extensions”
* Scroll down and make sure that “Allow access to file URLs” is selected.

We’ll see the benefits that this will bring shortly.

## Shopping List

We’re going to work through a slightly more complex app in order to see some key Vue concepts. In this lesson, we’ll make a shopping list that should:

* Display a list of items
* Allow the user to add new items
* Allow the user to mark items as “purchased”

Let’s start by displaying a simple list of items.

## The v-for directive

When we’re planning our app in Vue, we can either start by thinking about the model (what data is stored) - or the view (what is displayed to the user.) Different people will approach problems in different ways, so experiment and find out what works best for you.

For now, we’re going to start with the most basic approach of simply displaying a list of items, without worrying about marking items as complete, or adding items.

We know that we’re going to need to track a list of items - an array - so let’s start by declaring that on our model. Let’s put a couple of items in there to start off.

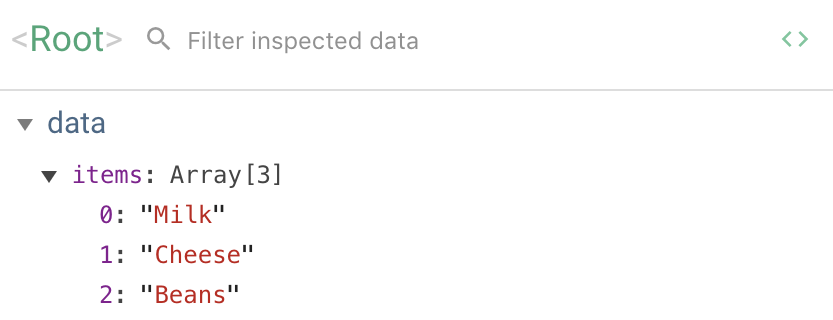
*//app.js*

data: {

**items**: ["Milk", "Cheese", "Beans"] *//NEW*

},

To see this reflected in our app, we can use our newly-installed Vue Devtools and look at the data that’s present!



Note: You might need to refresh your page or even restart your browser to see the relevant devtools pane!

To display this list, we need to use the v-for directive. If we want one <ul> element, and multiple <li> elements, one for each item - we can do something like this:

*<!-- index.html -->*

<ul> *<!-- NEW -->*

<li v-for="item in items">{{ item }}</li>

</ul>

If we run npm run build, then we should see the list! Let’s talk through v-for.

v-for sets up a loop. Of course, we need to loop over something, and in this case, we’re looping over the items array in the data object. Each individual item can be referred to inside the moustaches as {{ item }}. (Because we said "item in items").

We can also access the index of the loop - the position of the item of the array - and this will come in useful later.

Let’s go ahead and add a form, to allow us to add new items to our list.

### Adding a form

Once again, we can think about our problem from the model side, or the view side. Let’s think about what we want the user to see.

In this case, we probably want to create a form to display a text input field, and a button for the user to click. Let’s start there.

*<!-- index.html -->*

<ul>

<li v-for="item in items">{{ item }}</li>

</ul>

<form> *<!-- NEW -->*

<label for="new-item">Add a new item:</label>

<input id="new-item" type="text" />

<input type="submit" value ="Save New Item" />

</form>

Refreshing the page will show these elements, but they’re not doing anything yet. Let’s fix that.

## Introducing v-model

In order to capture input from the user, ideally, we want to know the value of the text input at all times. We want to be able to create a two-way binding between the input element and a key / value pair on our data object.

We can achieve this by using the v-model directive. Let’s change our code.

*<!-- index.html -->*

<input id="new-item" type="text" v-model="newItem" /> *<!-- MODIFIED -->*

If we refresh our browser page, we’ll see an error here.

Property or method “newItem” is not defined on the instance but referenced during render.

To set up our binding, we need to declare the newItem key on our data object too. Since it should start off being empty, we’ll give it the value of an empty string.

*//app.js*

data: {

**items**: ["Milk", "Cheese", "Beans"],

newItem: "" *// NEW*

},

That’s all we need to do! Now, whenever the value of the text input changes, the newItem variable will be automatically updated. And vice-versa, if we change the variable in our model, our view will change automatically.

If we refresh our browser, we can use devtools to watch our model dynamically update!

Now that we know what the user is typing, we can simply finish the job - we need to ensure that when the form is submitted, we take the value of newItem and add it to our items array. Before we can do that though, we probably need to remind ourselves about the default behaviour of an html form element - let’s look what happens if we enter an item into the text input and then click on our form’s submit input just now. The item disappears! Why is this?

## Modifiers

Remember, this is because by default a html form, when submitted, sends a post request to the current URL. In vanilla JavaScript, we might handle this problem using preventDefault(), like so:

**function** handleSubmit(event){

event.preventDefault()

}

But to implement this in Vue would be a bit of an anti-pattern - Vue is meant to handle our DOM manipulation for us, so surely there must be something it can provide to mitigate this problem? Thankfully, there is - and it’s called a Vue modifier. Modifiers can be added to our Vue directives to add more functionality to them. We’ll see a couple of these as we go, but our first one looks something like this:

*<!-- index.html -->*

<form v-on:submit.prevent="saveNewItem"> *<!-- MODIFIED -->*

<label for="new-item">Add a new item:</label>

<input id="new-item" type="text" v-model="newItem" />

<input type="submit" value ="Save New Item" />

</form>

Now when we click our button, whatever we’ve typed in the input is still there. The form is no longer undertaking its default behaviour, so we’re not making a post request to the current URL. We will, however, get an error to tell us that our saveNewItem method doesn’t exist, and that’s fair enough - because that’s your next task.

### Task: 10 minutes

* create a method - saveNewItem
* Inside this method, take the text entered by the user in this.newItem, and push it on to the array this.items
* After that, set this.newItem to be an empty string inside the method. What does this do? Why?

#### Solution:

*//app.js*

methods: {

**saveNewItem**: **function**(){ *//NEW*

**this**.items.push(**this**.newItem);

**this**.newItem **=** "";

}

}

Answer: Setting this.newItem = ""; clears the text box for the user, ready to enter the next item. This is because two-way binding works… both ways!

## Marking items as complete

In order to mark an item as “purchased”, we need to think about the data we store for each individual item. We want to store:

* Its name (as a string, as before)
* Its purchased status (a boolean, true or false)

How can we achieve this? Instead of storing a string, we’ll store an object instead. It might look like this:

{ **name**: "Milk", isPurchased: **true** }

Let’s amend our data object to reflect our new data structure.

*//app.js*

data: {

**items**: [ *//MODIFIED*

{name: "Milk", isPurchased: **false**},

{name: "Cheese", isPurchased: **true**},

{name: "Beans", isPurchased: **false**},

],

newItem: ""

},

And now that each item is an object, we need to update our template to use the object’s name property.

*<!-- index.html -->*

<li v-for="item in items">{{ item.name }}</li> *<!-- MODIFIED -->*

If we refresh our page, we should at least have a list of items again.

Next, we need to display something to show the user that the item has been purchased. For now, let’s just add a span element if the object’s isPurchased field is true.

To do this, we need to use Vue’s v-if directive.

*<!-- index.html -->*

<li v-for="item in items">

<span>{{ item.name }}</span>

<span v-if="item.isPurchased">Purchased!</span> *<!-- NEW -->*

</li>

v-if will output the span element **only** if item.isPurchased evaluates to true. Otherwise, it won’t.

We have two tasks remaining.

* Allow the user to create new objects again, and
* Allow the user to mark items as “purchased”

Let’s deal with the first one first, as it’s broken at the moment.

### Creating objects

Let’s look at the method we created earlier to add an item to the this.items.

*//app.js*

saveNewItem: **function**(){

**this**.items.push(**this**.newItem);

**this**.newItem **=** "";

}

At the moment, it’s simply pushing a string onto the array. We need to change this, so that it pushes an object with a name and purchased status instead.

*//app.js*

saveNewItem: **function**(){

**this**.items.push({ *//MODIFIED*

name: **this**.newItem,

isPurchased: **false**

});

**this**.newItem **=** "";

}

With this small change, we should be able to add items again.

### Allowing the user to mark items as purchased

Our last job is to let the user mark items as purchased.

Let’s add a button to each element that isn’t currently marked as purchased. To do this, we can use a negated v-if. We only want to display the button if isPurchased is false.

*<!-- index.html -->*

<li v-for="item in items">

<span>{{ item.name }}</span>

<span v-if="item.isPurchased">Purchased!</span>>

<button v-if="!item.isPurchased">Purchase</button> *<!-- NEW -->*

</li>

So if the item is purchased, we display a span element saying “Purchased!”. Otherwise, we display a button allowing the user to purchase the item.

To finish off, we need to add a click handler to the button. To figure out which item should be marked as “purchased”, v-for allows us to grab the index of the loop, and pass it to our method!

*<!-- index.html -->*

<li v-for="(item, index) in items"> *<!-- MODIFIED -->*

<span>{{ item.name }}</span>

<span v-if="item.isPurchased"> - Purchased!</span>

<button v-if="!item.isPurchased" v-on:click="buyItem(index)">Purchase</button> *<!-- MODIFIED -->*

</li>

### Task - 5 minutes

* Write the method buyItem(index)

#### Solution:

*//app.js*

buyItem: **function**(index){

**this**.items[index].isPurchased **=** **true**;

}

## Introducing v-bind

Let’s say that we wanted to add a class to our li - purchased if the item has been purchased, not-purchased otherwise.

We can use a ternary to achieve this. Let’s try it out:

*<!-- index.html -->*

<li v-for="(item, index) in items" class="item.isPurchased ? 'purchased':'not-purchased'"> *<!-- MODIFIED -->*

<span>{{ item.name }}</span>

<span v-if="item.isPurchased"> - Purchased!</span>

<button v-if="!item.isPurchased" v-on:click="buyItem(index)">Purchase</button>

</li>

If we look in our devtools, this hasn’t quite had the desired effect. Vue has done exactly as we’ve asked, and set the class as follows: class="item.isPurchased ? 'purchased':'not-purchased'". How can we get around this?

We want to make class dynamic; to evaluate a JavaScript expression to decide how to set our HTML attribute.

When we want to use a variable or expression in an HTML attribute, we have to use the v-bind directive, as follows:

*<!-- index.html -->*

<li v-for="(item, index) in items" v-bind:class="item.isPurchased ? 'purchased':'not-purchased'"> *<!-- MODIFIED -->*

This is such a common pattern that you will often see a shorthand for v-bind - simply putting a : character before the attribute in question. We can simplify what we wrote before:

*<!-- index.html -->*

<li v-for="(item, index) in items" :class="item.isPurchased ? 'purchased':'not-purchased'"> *<!-- MODIFIED -->*

We will be using v-bind extensively throughout the course in various scenarios.

## Recap

What directive should we use if we want to conditionally output a piece of HTML?

Answer

What directive should we use if we want to loop through an array and output an HTML element for each item?

Answer

What is the purpose of v-model?

Answer

What is the purpose of v-bind?

Answer

What are Vue modifiers, and what situation might be they be useful in?

Answer

## Conclusion

We’ve seen how to use v-if, v-for, v-model, v-bind and some Vue modifiers to construct a simple, dynamic JavaScript application using Vue.

Other form elements work in much the same way - for more information, see Vue’s [form guide](https://vuejs.org/v2/guide/forms.html).

# Homework: ToDo List

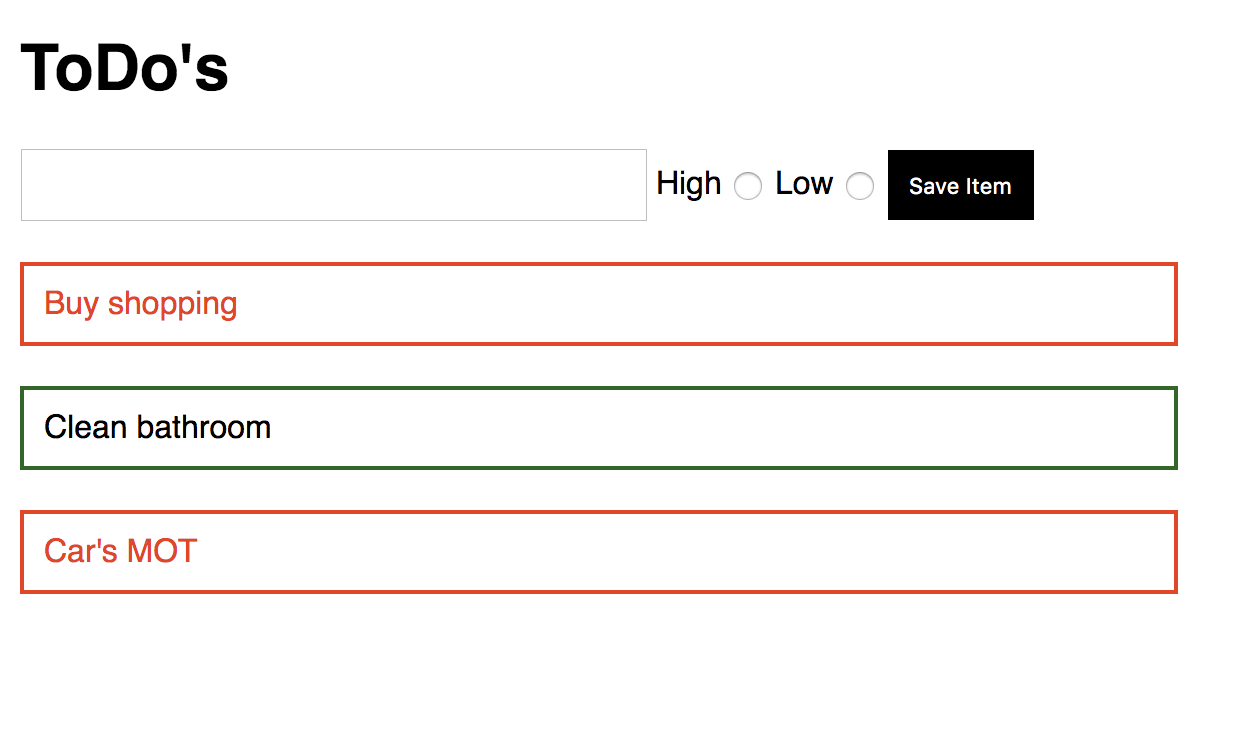
## Learning Objectives

* Be able to set up a basic Vue application
* Be able to use forms with v-model
* Be able to use v-on to manage user events

## Brief

Your task is to create a todo list application that allows a user to save an item to a list which then renders on the page.

Use Vue and Webpack to build a solution that looks like this:



## MVP

* Render a list of todos from an array of strings.
* Add the capability to add a new todo to the list.

Your todos array would look something like:

todos: [

"Buy shopping", "Clean bathroom", "Car's MOT"

]

## Extensions

* Update the todos array to take in objects with keys of name and priority.
* Add radio buttons to track the level of priority
* Give the user visual feedback on what todos are high priority

## Setup

1. npm init and install vue, webpack and webpack-cli.
2. Add build script to package.json.
3. Create folder and file structure.

Your folder structure might look like this:

* public
  + css
  + js
  + index.html
* src
  + app.js
* .gitignore
* package.json
* webpack.config.js

1. Create webpack.config.js file and add configuration.

For your convenience, here is a sample webpack configuration:

**const** config **=** {

entry: `${\_\_dirname}/src/app.js`,

output: {

path: `${\_\_dirname}/public/js/`,

filename: 'bundle.js'

},

resolve: {

alias: {

'vue$': 'vue/dist/vue.esm.js'

},

extensions: ['\*', '.js', '.vue', '.json']

},

mode: 'development'

};

module.exports **=** config;

1. Create simple HTML file with a div with id of app
2. In app.js import vue and create a new instance passing in config object with el: '#app' as starting point.
3. npm run build.
4. open index.html and make sure you can see vue dev tools.
5. Develop!!

# Fetch, Promises and Lifecycle Hooks

**Lesson Duration**: 90 minutes

### Learning Objectives

* Understand what a JSON API is and why we use them
* Be able to make a request using Fetch
* Understand the relationship between JSON and JavaScript objects
* Be able to traverse a JSON object
* Understand Vue’s lifecycle hooks and when to implement them

## Intro

In this lesson we are going to learn how to request data from an API, so that we can use it in our applications. This can save us a great deal of time, preventing us from having to create the data ourselves.

### What is an API

An API, or application programming interface, is a tool that allows one piece of software to interact with another. A lot of software provides an API, allowing developers to write software that interacts with it. The DOM, for example, is an API. The DOM allows us to write software that interacts with content of a the web page.

As well as providing us with functionality, some APIs simply provide us with data. We can use this data to enhance the functionality of our applications. Modern APIs tend to provide us with data in JSON format. Some older APIs may still use XML, but that’s becoming less popular as JSON continues to gain traction. We’ll come back to JSON shortly.

### Why Would We Use an API?

APIs often provide us with functionality or data that would be incredibly difficult and time consuming to create on our own.

Let’s imagine that you were building a travel app and needed data about all of the countries in the world. You could spend time collecting and organising the data yourself, but it would be very time consuming. It would be much more convenient to use an API like [RESTCountries](http://restcountries.eu/" \t "_blank) that can provide us with data that someone else has already prepared.

RESTCountries provides us with data about all of the countries in the world in JSON format.

We can see a list of endpoints on their website. An endpoint is essentially a URL that we can make a request to, allowing us to access the API. Let’s take a look at the [all](https://restcountries.eu/rest/v2/all) endpoint. You might install the [JSON Formatter](https://chrome.google.com/webstore/detail/json-formatter/bcjindcccaagfpapjjmafapmmgkkhgoa) Chrome extension to allow the browser to format the data in a more human readable way.

### What is JSON?

JSON, JavaScript Object Notation, is a language agnostic data interchange format derived from JavaScript. It allows us to store data in a human readable format that appears familiar to programmers of C family languages.

When we say that JSON is language agnostic, we mean that it doesn’t know anything about any other programming languages. We will be converting JSON to JavaScript today, but we could just as easily make a request from a .NET application, written in C#, and convert the resulting JSON data to C#. This means that we can use JSON as a common language allowing software that was written in different languages to speak to each other.

While JSON may look very familiar, there are a couple of key differences that we must bear in mind. All JSON is valid JavaScript, but not all JavaScript is valid JSON.

1. The keys in a JSON object must be quoted
2. Strings must be declared with double quotes ""
3. A JSON object cannot contain functions

## The Dog API

Before we dive straight in to wrangling massive data objects, let’s work with a smaller response so we can practice making our requests without worrying too much about what we’re getting back. [The Dog API](https://dog.ceo/dog-api/) bills itself as the internet’s biggest collection of open source dog pictures, and has a random endpoint that will give us a JSON object containing a URL to an image of a dog. Here’s an example response:

**{**

**"status": "success",**

**"message": "https://images.dog.ceo/breeds/malinois/n02105162\_5370.jpg"**

**}**

Before we write any code let’s think about what we’re about to do.

Hand out start point

npm run build

open public/index.html

### Task - 5 minutes

* Read over the start code to understand what our app is currently doing and come up with suggestions as to how we can approach finishing it.

At the moment, our Vue instance is set up with a button that will execute fetchDog on click, and an img in our HTML that will render, dependent on there being a dogImgURL in our instance’s data. Just now fetchDog is just logging out a message when we click it, so all we really need to do here is consider how we’re going to get that dog image from our API, retrieve an URL for an image from it and set it to be dogImgURL.

So our plan will be something along the lines of:

* We’ll make a request to the Dog API
* The Dog API will respond to our request with JSON
* We’ll then use the data in our application

And how are we going to our dog data? Appropriately enough, we’ll be doing this with another API called fetch. But to appreciate what fetch does and how it does it, first we’ll need to talk a bit about Promises.

## Promises

Network requests are asynchronous. This has two major implications.

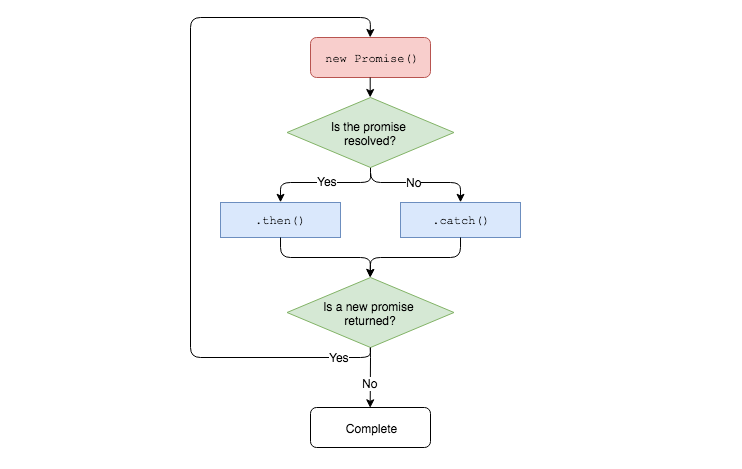
* They take an unspecified amount of time to complete
* Execution of the rest of our code is not paused in the mean time

We typically create a Promise when we want to retrieve data via some asynchronous operation. This could be requesting data from an API or a database, for example.

Just like a real life promise that represents something that will happen in the future, a JavaScript Promise object represents the result of an asynchronous operation - this can be the completion or failure of the operation. We can write a Promise and then decide what to do once the asynchronous operation has completed. This allows us to pass fewer callbacks around, meaning that our code reads a bit more as if it were synchronous, which can be easier to follow.

### How Does a Promise Work?

When we create a Promise object it will be pending until the asynchronous operation that we are wrapping has completed. Then, just like a real life promise that can be kept or broken, the Promise object will either resolve, if successful, or reject if something goes wrong, allowing us to decide what to do next. We may want to render the data that was received, or perform some kind of error handling if the Promise rejected. We can also chain multiple asynchronous operations by chaining functions which return a Promise.



## Fetch

The [fetch method](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) we’re going to implement wraps the asynchronous functionality of a request inside a Promise. That means we can write code to handle the result of our request, without having to give too much mind as to when our request is going to finish executing. There are other ways of making network requests but fetch is now widely supported across the majority of internet browsers, arguably easier to implement, and in terms of code simpler to read.

So let’s try and implement it now! Initially, we’ll assign our fetch implementation to a variable, so that we can log it out and examine it.

*///APP.JS*

*//IN METHODS*

fetchDog: **function**(){

**const** request **=** fetch("https://dog.ceo/api/breeds/image/random")*//NEW*

console.log(request);

}

In our browser’s console, we should now see a pending Promise object, which if examine a bit closer we can see has a promiseStatus of resolved. Which is good for us, because now we can try and interact with the response the API gave us.

The primary way of interaction with a Promise is through its then method, which registers callbacks to receive either a promise’s eventual value or the reason why the promise cannot be fulfilled. This allows us to treat a promise like a returned object that we can attach callbacks to, instead of passing callbacks into a function, which was previously more commonplace. Let’s see then in action.

*///APP.JS*

fetchDog: **function**(){

fetch("https://dog.ceo/api/breeds/image/random")*//MODIFIED*

.then(response **=>** console.log(response)) *//NEW*

}

We can now see the response object that’s being logged out - specifically, it’s a Response object. This contains the JSON body that we’re looking for, but to access it we need to call .json() on our response. This method also involves some asynchronous operations and therefore returns… another Promise! But that means we can chain another .then on to it and continue to work with the response till we have what we want.

*///APP.JS*

fetchDog: **function**(){

fetch("https://dog.ceo/api/breeds/image/random")

.then(response **=>** response.json()) *//MODIFIED*

.then(data **=>** console.log(data)) *//NEW*

}

And now if we look in our browser’s console we should see the response from the API we initially set out to find! Great. So how can we use this data to get the image URL to where we need it to be, in our Vue data’s dogImgURL?

### Task - 5 minutes

Take the image URL from our response and set it to be the value of dogImgURL.

*///APP.JS*

fetchDog: **function**(){

fetch("https://dog.ceo/api/breeds/image/random")

.then(response **=>** response.json())

.then(data **=>** **this**.dogImgURL **=** data.message) *//MODIFIED*

}

And there we go! Now whenever the user clicks the button, we fetch request the Dog API, handle the response and then put the URL into our Vue instance’s data, where our data-binding means that it renders automatically. Clicking a button to begin this whole process seems burdensome though - what if we could find a way so that our fetchDog function was called as soon as the app loaded, rather than waiting for user input? Well, there is and it utilises Vue’s lifecycle hooks.

At this point it’s important to make a distinction - very few APIs behave exactly the same, so it would be wrong to assume that every fetch will return an object that we can call .message on and expect the data we’re looking for. Make sure going forward that you interrogate the data you are retrieving, through an APIs documentation, or even via console.log before attempting anything with it.

## Lifecycle Hooks

From [the docs](https://vuejs.org/v2/guide/instance.html#Instance-Lifecycle-Hooks):

Each Vue instance goes through a series of initialization steps when it’s created - for example, it needs to set up data observation, compile the template, mount the instance to the DOM, and update the DOM when data changes. Along the way, it also runs functions called lifecycle hooks, giving users the opportunity to add their own code at specific stages.

In a nutshell, this is saying that our Vue instance runs a handful of specifically named methods throughout its existence, and affords us the opportunity to add our code to them. Lifecycle hooks you may see documented include created, updated, and destroyed, but the one we’re going to look at for the time being is mounted. mounted is a method that runs once our DOM is ready and available to be manipulated, so we’re using it here much in the same way that we’ve been using DOMContentLoaded. Let’s see if we can hook into it just now:

*///APP.JS*

**new** Vue({

el: "#app",

data: {

dogImgURL: **null**

},

mounted(){ *//NEW*

console.log("Hello from the mounted lifecycle hook!");

},

methods: {

fetchDog: **function**(){

fetch("https://dog.ceo/api/breeds/image/random")

.then(response **=>** response.json())

.then(data **=>** **this**.dogImgURL **=** data.message);

}

}

})

And if we look in the console now, we can see that this method is executing every time our script reloads. That means we could move our code for fetching the dog data into mounted() and not have to wait for the user to click.

*///APP.JS*

mounted(){

**this**.fetchDog() *//MODIFIED*

}

And now whenever we load our app, we have a dog image before the user does anything! Success!

## Recap

What are some of the advantages of using a promises when writing asynchronous JavaScript?

Answer

Which method do we call after receiving a Promise to determine which behaviour we want to execute it resolves successfully?

Answer

When is the mounted lifecycle method called?

Answer

## Conclusion

Now that we’ve learned how to use Promise objects and the fetch web API, we can write asynchronous JavaScript that almost reads as if it were synchronous. Our code appears more like a structured series of events, even if we aren’t sure exactly when things will happen. In short, promises allow us to write neater code. fetch also abstracts away some of the lower level implementation details of HTTP requests. We don’t really have to care exactly how fetch is implemented, we just care that we get our data back.

This isn’t all that there is to learn about promises, however. Promises really come into their own when it comes to carrying out multiple asynchronous operations either sequentially, using a then chain, or at the same time, using Promise.all.

We’ve also looked at how we can lean on Vue’s lifecycle hooks to add functionality to our apps with little effort.

## Further Resources

MDN - [Promises](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)  
MDN - [Promise.all](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all" \t "_blank)

**Computed Properties**

**Duration: 60 minutes**

**Learning Objectives**

* Understand when to use Vue’s computed properties
* Be able to implement computed properties in our apps

**Introduction**

In our Vue apps it might be necessary for us to display a variable that’s built or calculated dependent on some other data in our app, be it in our state or otherwise. While there are many ways that do this - remember, Vue is an unopinionated framework - in this lesson we’re going to look at a way of handling complex logic to dynamically update our html with the minimal fuss we’re coming to expect from Vue.

For this lesson, we’ll be working with an app that manages bank accounts.

npm run build

open public/index.html

There’s something in our HTML that you might not have seen before - v-model.number. This is another of Vue’s modifiers, which can be added to our Vue directives for extra functionality.

In this case, the modifier is ensuring that our value here comes into our Vue instance as a number, to mitigate HTML element values being strings by default - this saves us having to write any additional parseInt logic in our Vue instance.

You’ll also notice that the header of the app is intended to display the total of all the balances of the accounts held in the app’s state. At the moment however, there’s no code to provide this functionality. Let’s think about how we might do this.

**Displaying the accounts total**

Previously, we might have approached displaying the total of all the bank accounts with the tools already at our disposal - we could have a totalAccounts value in our state that could be worked out when our app fires mounted, but then that would only run once and never again. We could create a method that is called from saveAccount, whenever a new bank account is added, but then that would mean we would have to add an account before we even see the result of that calculation. We could combine both of these approaches, but to be frank we’re making a lot of work for ourselves when there’s a far more elegant solution - Vue’s computed property.

**Making a computed property**

A computed property is pretty much what it says on the tin - a property that is derived from the result of some form of computation. The rule of thumb is to use them whenever you want a property that would require some complex logic to display. To get the total of all these bank accounts, we’re going to have to call reduce on this.accounts, and while there’s nothing stopping us putting this function call straight into our template, it would make our template harder to read, and harder to maintain. The good news is that implementing a computed property isn’t too different from any other thing we’ve added to our Vue instance’s this week. Firstly, let’s add a property of computed to our Vue instance which will, to begin with, point to an empty object.

*//app.js*

data: {

...

},

computed: { *//NEW*

}

We can then add methods to this object that will return us the computed property we’re after. In this case, we’re looking for the total of all our account balances, so totalBalances seems like a pretty sensible name for the computed property.

*//app.js*

computed: {

**totalBalances**: **function**(){ *// NEW*

}

}

Now if we look in our dev tools, we should see that as well as tracking our app’s data, we now have a computed section, tracking all our computed properties. At the moment this will be showing as undefined, because that’s what a function with no return value gets us. We’ll fix that in a moment, but let’s finish wiring our computed property up first. Referring to it in our html template is as easy as referring to any other variable in our app’s state. Let’s bring the result of this function in like so:

*<!-- index.html -->*

<header>

<h1>Bank of CodeClan</h1>

<h2>Total deposits: £{{ totalBalances }} </h2> *<!-- MODIFIED -->*

<p>Total Deposits ☝️ should update dynamically when we add a new account.</p>

</header>

Great! So all we need do now is finish writing the function that will return the sum total of all of this.accounts’s account balances!

**Task (5 mins)**

Finish writing the function that will return the sum total of all of this.accounts’s account balances.

**Solution**

computed: {

**totalBalances**: **function**(){

**return** **this**.accounts.reduce((runningTotal, account) **=>** runningTotal **+** account.balance, 0); *// MODIFIED*

}

}

Wonderful! Now whenever we add a bank account to our app, the computed property totalBalances is automatically recalculating its value. Although this computed property is the result of a method, it’s important to bear in mind that it should be treated as a *property* - a value that we can call upon, and not a method that needs to be invoked, or rerun at any point. Let’s practice with one more example to finish up - let’s say that the user of this app requires the ability to filter this.accounts and only display accounts that have a minimum balance, without so much as clicking a button. How could we implement a computed property to do this?

**Task (15 minutes)**

* Add an input that allows the user to dictate the minimum balance of accounts on display
* Use a computed property to display the result of this.accounts being filtered with the inputted filter amount.

Let’s think about the steps to complete in order to achieve this task:

* We should add filterAmount - to our app’s data, and initially set that to be 0.
* We can then create an number input in our html and v-model it to the filterAmount, so when the input changes, the value in our data changes.
* We should then create a computed property - filteredAccounts - that returns this.accounts filtered so we only have accounts which have *at least* the filterAmount we’ve taken from the user - what enumeration method can we apply here?
* Once we’ve seen filteredAccounts working in our dev tools, we’ll use it in our html - instead of our accounts section looping over the accounts data in our state, we’ll now loop over our computed filteredAccounts instead.

**Solution**

*<!-- index.html -->*

<div id="filterInput"> *<!-- NEW-->*

<h3>Filter Accounts By Minimum Value</h3>

<input type="number" v-model.number="filterAmount"/>

</div>

*//app.js*

data: {

*//AS BEFORE*

**filterAmount**: 0 *// NEW*

},

computed: {

**totalBalances**: **function**(){

**return** **this**.accounts.reduce((runningTotal, account) **=>** {

**return** runningTotal **+** account.balance;

}, 0);

},

filteredAccounts: **function**(){ *//NEW*

**return** **this**.accounts.filter((account) **=>** {

**return** account.balance **>=** **this**.filterAmount;

});

}

}

*<!-- index.html -->*

<section>

<div class="account" v-for="account in filteredAccounts"> *<!-- MODIFIED -->*

<h2>{{ account.name }}</h2>

<p>Balance: £{{ account.balance }}</p>

</div>

</section>

And now whenever the user inputs a number, the accounts displayed should have at least that amount as their balance.

**Recap**

When should you implement a Computed Property?

**Answer**

**Conclusion**

We’ve seen how computed properties are an elegant way to keep complex logic out of our html, while also shouldering the responsibility of updating their own value whenever our app’s state changes; helping us manage our app’s a bit more easily.

Published with [GitHub Pages](https://pages.github.com/)

# Computed Properties

**Duration: 60 minutes**

### Learning Objectives

* Understand when to use Vue’s computed properties
* Be able to implement computed properties in our apps

## Introduction

In our Vue apps it might be necessary for us to display a variable that’s built or calculated dependent on some other data in our app, be it in our state or otherwise. While there are many ways that do this - remember, Vue is an unopinionated framework - in this lesson we’re going to look at a way of handling complex logic to dynamically update our html with the minimal fuss we’re coming to expect from Vue.

For this lesson, we’ll be working with an app that manages bank accounts.

npm run build

open public/index.html

There’s something in our HTML that you might not have seen before - v-model.number. This is another of Vue’s modifiers, which can be added to our Vue directives for extra functionality.

In this case, the modifier is ensuring that our value here comes into our Vue instance as a number, to mitigate HTML element values being strings by default - this saves us having to write any additional parseInt logic in our Vue instance.

You’ll also notice that the header of the app is intended to display the total of all the balances of the accounts held in the app’s state. At the moment however, there’s no code to provide this functionality. Let’s think about how we might do this.

### Displaying the accounts total

Previously, we might have approached displaying the total of all the bank accounts with the tools already at our disposal - we could have a totalAccounts value in our state that could be worked out when our app fires mounted, but then that would only run once and never again. We could create a method that is called from saveAccount, whenever a new bank account is added, but then that would mean we would have to add an account before we even see the result of that calculation. We could combine both of these approaches, but to be frank we’re making a lot of work for ourselves when there’s a far more elegant solution - Vue’s computed property.

## Making a computed property

A computed property is pretty much what it says on the tin - a property that is derived from the result of some form of computation. The rule of thumb is to use them whenever you want a property that would require some complex logic to display. To get the total of all these bank accounts, we’re going to have to call reduce on this.accounts, and while there’s nothing stopping us putting this function call straight into our template, it would make our template harder to read, and harder to maintain. The good news is that implementing a computed property isn’t too different from any other thing we’ve added to our Vue instance’s this week. Firstly, let’s add a property of computed to our Vue instance which will, to begin with, point to an empty object.

*//app.js*

data: {

...

},

computed: { *//NEW*

}

We can then add methods to this object that will return us the computed property we’re after. In this case, we’re looking for the total of all our account balances, so totalBalances seems like a pretty sensible name for the computed property.

*//app.js*

computed: {

**totalBalances**: **function**(){ *// NEW*

}

}

Now if we look in our dev tools, we should see that as well as tracking our app’s data, we now have a computed section, tracking all our computed properties. At the moment this will be showing as undefined, because that’s what a function with no return value gets us. We’ll fix that in a moment, but let’s finish wiring our computed property up first. Referring to it in our html template is as easy as referring to any other variable in our app’s state. Let’s bring the result of this function in like so:

*<!-- index.html -->*

<header>

<h1>Bank of CodeClan</h1>

<h2>Total deposits: £{{ totalBalances }} </h2> *<!-- MODIFIED -->*

<p>Total Deposits ☝️ should update dynamically when we add a new account.</p>

</header>

Great! So all we need do now is finish writing the function that will return the sum total of all of this.accounts’s account balances!

### Task (5 mins)

Finish writing the function that will return the sum total of all of this.accounts’s account balances.

### Solution

computed: {

**totalBalances**: **function**(){

**return** **this**.accounts.reduce((runningTotal, account) **=>** runningTotal **+** account.balance, 0); *// MODIFIED*

}

}

Wonderful! Now whenever we add a bank account to our app, the computed property totalBalances is automatically recalculating its value. Although this computed property is the result of a method, it’s important to bear in mind that it should be treated as a property - a value that we can call upon, and not a method that needs to be invoked, or rerun at any point. Let’s practice with one more example to finish up - let’s say that the user of this app requires the ability to filter this.accounts and only display accounts that have a minimum balance, without so much as clicking a button. How could we implement a computed property to do this?

### Task (15 minutes)

* Add an input that allows the user to dictate the minimum balance of accounts on display
* Use a computed property to display the result of this.accounts being filtered with the inputted filter amount.

Let’s think about the steps to complete in order to achieve this task:

* We should add filterAmount - to our app’s data, and initially set that to be 0.
* We can then create an number input in our html and v-model it to the filterAmount, so when the input changes, the value in our data changes.
* We should then create a computed property - filteredAccounts - that returns this.accounts filtered so we only have accounts which have at least the filterAmount we’ve taken from the user - what enumeration method can we apply here?
* Once we’ve seen filteredAccounts working in our dev tools, we’ll use it in our html - instead of our accounts section looping over the accounts data in our state, we’ll now loop over our computed filteredAccounts instead.

### Solution

*<!-- index.html -->*

<div id="filterInput"> *<!-- NEW-->*

<h3>Filter Accounts By Minimum Value</h3>

<input type="number" v-model.number="filterAmount"/>

</div>

*//app.js*

data: {

*//AS BEFORE*

**filterAmount**: 0 *// NEW*

},

computed: {

**totalBalances**: **function**(){

**return** **this**.accounts.reduce((runningTotal, account) **=>** {

**return** runningTotal **+** account.balance;

}, 0);

},

filteredAccounts: **function**(){ *//NEW*

**return** **this**.accounts.filter((account) **=>** {

**return** account.balance **>=** **this**.filterAmount;

});

}

}

*<!-- index.html -->*

<section>

<div class="account" v-for="account in filteredAccounts"> *<!-- MODIFIED -->*

<h2>{{ account.name }}</h2>

<p>Balance: £{{ account.balance }}</p>

</div>

</section>

And now whenever the user inputs a number, the accounts displayed should have at least that amount as their balance.

## Recap

When should you implement a Computed Property?

Answer

## Conclusion

We’ve seen how computed properties are an elegant way to keep complex logic out of our html, while also shouldering the responsibility of updating their own value whenever our app’s state changes; helping us manage our app’s a bit more easily.

# Lab: Make an app with the Countries API

**Duration: 120 minutes**

## Learning Objectives

* Be able to make requests and displaying the data in the page
* Understand how to traverse the data structure received from an API
* Be able to implement Vue’s lifecycle hooks and computed properties

Your task is to build an app that uses [this API](https://restcountries.eu/rest/v2/all) to display information on the countries of the world :earth\_africa:

## MVP

Your app should be able to:

* Display the total population of all the countries the API provides.
* Allow the user to select a single country and display information about it (at least the country’s name and flag).
* Allow users to add their favourite countries to a list that is also displayed in your app.

## Extensions

* Show a list of the selected country’s neighbouring countries.
* Display the total population of these neighbouring countries.
* Prevent the same country being added to the user’s favourite countries list twice.

# Homework - Currency Converter

## Learning Objectives

* Be able to set up a Vue application
* Be able to read data from an API
* Be able to handle user interaction and practice business logic
* Be able to implement computed properties

### Brief

You have been asked to build a currency exchange calculator.

Use the data from [this url](https://api.exchangeratesapi.io/latest) to allow the user to convert from Euro to any of the listed currencies.

### MVP

The app should load the exchange rates when the page loads. The data being retrieved is an object, so it will be beneficial to [read the docs](https://vuejs.org/v2/guide/list.html#v-for-with-an-Object) on how a v-for directive handles objects.

mounted(){

// ...

}

The user should be able to enter an amount which they can then:

* convert from Euros into a selected currency
* convert from a selected currency into Euros

Make sure you appropriately format any currencies displayed to the user (i.e. to two decimal places).

### Extensions

* Allow cross-currency conversions - from non-base to non-base currencies. (e.g. from GBP to USD)
* Research how Vue’s [filters](https://vuejs.org/v2/guide/filters.html) allow for formatting of output and implement them where appropriate.

## Preparation For Tomorrow

Please run the following command in your terminal to install Vue’s CLI - we’ll talk about this in more depth tomorrow.

npm install -g @vue/cli

To check it’s working once the install has finished, run the following command to see the help that’s available for the Vue CLI.

vue -h

**Vue CLI and Templates Introduction**

**Lesson Duration: 30mins**

**Learning Objectives**

* Understand how to use the Vue CLI to create a boilerplate app template
* Understand how to create a Vue Component

**Intro**

Up until now we have been building our Vue apps with a relatively simple folder structure. What if we wanted to make more complex apps, however? As we know, Vue is pretty unopinionated and we can build our Vue apps in various ways.

What if we wanted to integrate a testing framework or use a pre-processor like SASS or even write in TypeScript? We’d really have to think about our folder structure and all of the dependencies that we require.

Luckily for us, Vue comes with a Command Line Interface which is the standard build tool for Vue applications which aims to reduce the amount of configuration the developer has to go through.

You can read a lot more in depth about this [here](https://cli.vuejs.org/), but essentially the CLI allows us to set up our projects with a number of configuration options. We can save our chosen configuration as a preset, so that we can quickly roll a new app whenever we need to.

**Hello World**

Let’s build a simple Hello World app to see exactly what this build gives us.

By now we should have globally installed the Vue CLI so that we can create our boilerplate builds from anywhere on our computers. If you haven’t yet, you can do so by doing the following:

npm install -g @vue/cli

This gives us the vue terminal command. Let’s check it’s working, by seeing the help that’s available for the Vue CLI.

vue -h

Let’s create our app.

vue create hello\_world

At this point, depending on circumstances, the following message might display:

? Your connection the the default npm registry seems to be slow.

Use https://registry.npm.taobao.org **for** faster installation? y/N

Ensure that ‘no’ is selected here, otherwise the option to create a default configuration for our Vue projects will be bypassed.

Now, we should be asked to choose what features we want to use, in the form of a preset. We’re going to create our own.

**Use the arrow keys to select “Manually select features”, then press return.**

We can now use the arrow keys and the space bar to choose the features we want to use.

**Make sure that Babel is the only option selected, then press return.**

We’re then asked to choose where we want to store various pieces of configuration.

**Choose “In dedicated config files”, then press return.**

We’re then asked whether to save our settings as a preset.

**Type “y”, then return.**

Next, we’re asked what the name of the preset will be.

**Type “CodeClan” without the quotes, then press return.**

At this point, NPM will install our chosen dependencies. Once it’s finished, do as the CLI suggests:

cd hello\_world

npm run serve

You can then open a browser window at http://localhost:8080 to see your app. Next time, we won’t have to choose a preset, just use the CodeClan one that we’ve created.

In future you will likely wish to set your own manual configuration, however this is the most basic build we require for the time being.

Open a new tab in your terminal, and open up the project in Atom. You’ll see a few new files.

**How is this working?**

Navigate into the src folder and open main.js. This is where the app is being kicked off from. The only thing that’s new here is the render function:

*//main.js*

**new** Vue({

render: h **=>** h(App),

}).$mount('#app')

A new Vue instance has been created and is being placed in the root element of #app. The render function is rendering our app within the #app div.

Now the confusing bit. render: h => h(App). This is just shorthand for:

*//main.js*

**new** Vue({

render: **function**(createElement) { *//MODIFIED*

**return** createElement(App);

}

}).$mount('#app')

Which can be shortened to

*//main.js*

**new** Vue({

render (createElement) { *//MODIFIED*

**return** createElement(App);

}

}).$mount('#app')

Which can also be shortened to

*//main.js*

**new** Vue({

render (h) { *//MODIFIED*

**return** h(App);

}

}).$mount('#app')

Why h? h is the common convention alias for createElement in the Vue ecosystem. Why? It comes from the term “hyperscript”, which is commonly used in many virtual-dom implementations. “Hyperscript” itself stands for “script that generates HTML structures” because HTML is the acronym for “hyper-text markup language”.

Which is then shortened further using es6 fat arrow syntax to

*//main.js*

**new** Vue({

render: h **=>** h(App) *//MODIFIED*

}).$mount('#app')

You’ll notice we are also missing our DOMContentLoaded function. This is a little bit of framework magic where the $mount function is attaching our Vue instance to the DOM in the #app element.

**Component Structure**

Next, let’s look at the App component which is being rendered. You’ll notice that the file type is a .vue file rather than a .js file. It is the vue-template-compiler dependency which is converting this back into a plain JavaScript module.

There are three main parts of each .vue file.

We have the template which is where we’ll manage our component’s HTML.

Within the script tags we have all of our JavaScript functionality.

Between the style tag is where we’ll write any css relevant to this component.

**What is a component?**

We can think of components as building block of our application. Components are single, independent units of an interface. They can have their own state, layout and style. Building our apps with components allows us to ensure that we have individual modules of code that have sole responsibility for their aspects of the apps functionality.

**Writing our own components**

Now we know how the component structure works, let’s write our own.

We’ll start off by deleting everything from App.vue. Before we write our own template/script/style structure lets install a very helpful atom package called language-vue. This package gives us a shortcut to creating this structure. We’ll also install language-babel which gives us nice syntax highlighting.

apm install language-vue

apm install language-babel

For this to take effect atom will have to be restarted.

Now when we are in a .vue file, when we type template you should see the Vue Component suggestion. Selecting this will give us the boilerplate setup of a Vue component.

One thing to point out here other than the language attributes in template and style is that the style is scoped. This means that for that component you can target any DOM element and it will only style that element within that component.

For example, if we write a piece of CSS like this:

h3 {

**color**: red

}

This will only affect any h3 elements within this component - not the rest of the app! Be careful about this - ensure you organise your CSS properly.

Now, we can see that each component contains its own template (HTML), behaviour (JavaScript), and styling (CSS).

**Rendering Data**

Let’s give this component some state via the data object, within the object that we’re exporting from our script tags. We should also give it the name of the component.

*// App.vue*

**<**script**>**

**export** **default** {

name: 'app',

data(){ *//NEW*

**return** {

message: 'Hello World'

}

}

}

**<**/script>

Notice that the structure of our data variable has changed slightly - we now need to make data a function, which returns an object {}. By returning an object from our data function it ensures that any change made to this data remains local to that instance of the component. Note that this has changed slightly from the way we wrote our apps when we were working within a single component, but inside the object we can still have any keys and values we like here, as before.

And now we can render our data within the template.

*<!-- App.vue -->*

<template lang="html">

<p>{{ message }}</p>

</template>

The browser will refresh on save, so save your file and navigate back to the browser. You should now see hello world displayed.

**Recap**

How do we start a new Vue app using the CLI?

**Answer:**

 What is a “preset”?

**Answer:**

**Conclusion**

We saw that we can use the Vue CLI to quickly create applications with various settings.

# Multiple Components and Props

**Lesson Duration: 60mins**

### Learning Objectives

* Understand how to register components
* Understand how to pass data down as props

## Intro

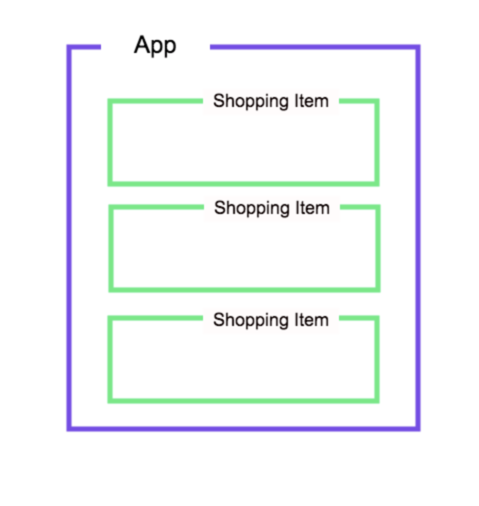
We’ve seen how to make Vue components. Next, we’re going to see how to make components that we can re-use, for example, in a list.

Doing so means that we can add individual functionality and styling, and it makes our code more maintainable.

## Shopping List

We’re going to go back to the shopping list app that we made earlier in the week, but this time, we’ll create a reusable component for each item.

Lets have a look at what the component structure is going to look like.



App is our entry component, the component that is rendered into the #app element on the DOM. App is where all of our shopping list items will live in state. For each element within the shopping list we’ll render a ShoppingListItem component.

vue create shopping-list

We’ll start off by stripping out the boilerplate code of App.vue and then creating our own component scaffolding with Atom’s language-vue package.

Let’s also give our page a header.

*<!-- App.vue -->*

<template lang="html">

<h1>My Shopping List</h1>

</template>

<script>

**export** **default** {

}

</script>

<style lang="css" scoped>

</style>

And then let’s give the component a name, and some data.

*<!-- AS BEFORE -->*

<script>

**export** **default** {

name: 'app',

data() { *//NEW*

**return** {

items: [

{name: 'Milk', isPurchased: **false**},

{name: 'Cheese', isPurchased: **true**},

{name: 'Beans', isPurchased: **false**},

]

};

}

}

</script>

### Registering Components

We’re going to render a list of shopping list items. But instead of rendering them as <li> elements, we’re going to render a shopping list item component.

First off, we’ll make that file. Open a new terminal tab so that our app keeps running.

touch src/components/ShoppingListItem.vue

In here, for the time being, let’s just generate our scaffolding and within the template render a <p> tag to indicate this is a shopping list item component.

*<!-- ShoppingListItem.vue -->*

<template lang="html">

<p>Shopping list item component</p>

</template>

<script>

**export** **default** {

name: 'shopping-list-item'

}

</script>

<style lang="css" scoped>

</style>

Next, we’re going to import this component where we need to use it.

### Using our custom component

Next, we’re going to import our ShoppingListItem within the App component. We also need to register it under the components key. Then we can render it within the template. As we’re now rendering more than one element within the template we have to wrap these elements in a root element - in this instance a div. Templates can only render one root element, without the div we would have had two root elements.

*<!-- App.vue -->*

<template lang="html">

<div class=""> *<!-- NEW -->*

<h1>My Shopping List</h1>

<shopping-list-item></shopping-list-item> *<!-- NEW -->*

</div>

</template>

<script>

**import** ShoppingListItem **from** './components/ShoppingListItem.vue'; *//NEW*

**export** **default** {

name: 'app',

data() {

**return** {

items: [

{name: 'Milk', isPurchased: **false**},

{name: 'Cheese', isPurchased: **true**},

{name: 'Beans', isPurchased: **false**},

]

};

},

components: { *//NEW*

'shopping-list-item': ShoppingListItem

}

}

</script>

You should see a shopping-list-item component being shown to the user in your browser.

### Component Naming Convention

[Vue’s Style Guide](https://vuejs.org/v2/style-guide/) recommendeds that component names should be multi-word, except for root App components. This prevents conflicts with existing and future HTML elements, since all HTML elements are a single word.

The component name can be either be kebab-case, camelCase or PascalCase. So list-item, listItem or ListItem. We’ll stick to kebab-case as it looks more html tag like and we’ll be rendering these components within our html templates.

## Passing Props

As it stands, we’re just rendering one list-item component that has no relation to the data. Let’s fix that.

Let’s use a v-for directive to render a list-item component for each item in our data object.

*<!-- App.vue -->*

<template lang="html">

<div>

<h1>My Shopping List</h1>

<ul>

<shopping-list-item v-for="(item, index) in items" :key="index"></shopping-list-item> *<!-- MODIFIED -->*

</ul>

</div>

</template>

When we render list items in Vue we have to give each element a key which we can bind to the item. This is to ensure that whenever an element of the list is changed the document will only re-render this item and not the whole list. Vue requires this attribute, and will throw a warning when it is not found.

Navigate to the browser and you should see three shopping-list-item components.

Now we want each of these shopping-list-item components to correlate with each item. How do we do this? By passing properties or props to the component.

Props are custom attributes you can register on a component. When a value is passed to a prop attribute, it becomes a property on that component instance.

Let’s give each shopping-list-item a prop of item and bind that prop to the iteration of the item in the loop.

*<!-- App.vue -->*

<template lang="html">

<div>

<h1>My Shopping List</h1>

<ul>

<shopping-list-item v-for="item in items" :key="index" :item="item"></shopping-list-item> *<!-- MODIFIED -->*

</ul>

</div>

</template>

We now have access to that item in our shopping-list-item component. Let’s update this component so that we now render item.name instead of our placeholder component message.

*<!-- ShoppingListItem.vue -->*

<template lang="html">

<li>{{ item.name }}</li> *<!-- MODIFIED -->*

</template>

If we refresh the browser we see nothing. And we now have an error message in the console saying cannot read property 'name' of undefined.

We’ve passed the prop down, but we still need to give the component a list of props it accepts. We can do this using the props key. Since we’re binding with :item in the component above, we need to use item here.

*<!-- ShoppingListItem.vue -->*

*<!-- AS BEFORE -->*

<script>

**export** **default** {

name: 'shopping-list-item',

props: ['item'] *//NEW*

}

</script>

Great, in the browser we can now see our shopping list items rendering.

## Styling Components

As it stands our shopping list doesn’t do too much. Let’s change that, by displaying items that have been purchased in a different manner than those that haven’t. One of the benefits of using .Vue files is that we can confine styling to our individual components, and see how a component is structured, how it behaves, and how it looks - all in the same file!

Let’s start by creating some styling for our purchased and unpurchased items:

*<!-- ShoppingListItem.vue -->*

<style lang="css" scoped>

**.not-purchased** {

**background-color**: lightgoldenrodyellow;

}

**.purchased** {

**text-decoration**: line-through;

**background-color**: red;

}

</style>

So if an item is unpurchased, we set the background colour of the <li> tag to lightgoldenrodyellow, and if an item has been purchased we want the background colour to be red with the text scored out. Previously when we made a shopping list we bound the class to a ternary operator that gave us a class name - but now we know a better way of abstracting our logic out of our views to help make our html more readable: computed properties!

### Task: 5 minutes

Use a computed property to provide the correct class for a purchased or unpurchased item.

### Solution

Let’s set up the computed property first. Since we’re only expecting two outcomes here - wether our item isPurchased or not, it makes sense to use a ternary operator.

*<!-- ShoppingListItem.vue -->*

<script>

**export** **default** {

name: 'shopping-list-item',

props: ['item'],

computed: { *//NEW*

purchasedClass: **function**(){

**return** **this**.item.isPurchased ? "purchased" : "not-purchased";

}

}

}

</script>

If we wanted to check that this was working at this point, we could use our devTools to inspect each individual shopping-list-item and look and see what the value of purchasedClass is under the computed section.

All that remains then is to use the return value of this computed property as the class of our li element - remember, as this is a dynamic Vue variable, we need to bind it, to avoid the value we use being parsed as a string by Vue:

*<!-- ShoppingListItem.vue -->*

<template lang="html">

<li :class="purchasedClass">{{ item.name }}</li>

</template>

This syntax means that the presence of the .purchased or .not-purchased class depends on the truthiness of item.isPurchased.

If you navigate back to the browser you should now see this style change.

## Recap

How do we make a component accessible to a parent component?

Answer:

How do we pass data down from a parent component to a child component?

Answer:

## Conclusion

We’ve seen how to build our own Vue components, which will allow us to separate the concerns of our app’s functionality and make our code more maintainable. .vue files also allow us to have sight of a component’s HTML, CSS and JavaScript all in the same file, allowing for simpler organisation of our code. Best of all, the concept of components means that we have code that can be reusable.

# Multiple Components and Event Bus

**Lesson Duration: 90mins**

### Learning Objectives

* Understand how to pass data between components

## Intro

We’ve seen how to pass data down to child components. But how do we pass data back up, or to another component? One solution is an EventBus.

An EventBus allows us to emit an event in one component and listen for that event in another. In computer science terminology a **bus** is the shared medium by which data is transfer from one place to another. This can be physical - actual circuits on a computer motherboard. For the EventBus it is purely software.

One way to think about this is like radio. Someone broadcasts on a specific frequency and anyone listening on that frequency will pick up the broadcast.

In the context of EventBus one component will broadcast data using an event name and any component listening using the same event name can pick up the data. Much the same as a button broadcasts a click event that we can add an event listener to.

### Task: 10 minutes

Read though the code and identify how the data is flowing throughout the application - for instance, where is the Munro data coming from? Where does that data then go, in terms of components and how does it get where it’s going?

Answer

At present our application could be diagrammed like this:

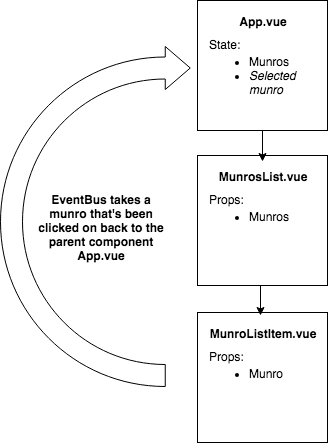


**What are we building?**

All we have is a list of names of Munros (a Munro is a Scottish mountain that is over 3000ft high). But if we look at the [API](https://munroapi.herokuapp.com/munros) you can see that it gives us a lot more information.

We want to be able to click on a list item and have details of that Munro render on screen. To do this we’ll use the eventBus to transmit the data from one component to another.

This plan could be diagrammed like this:



## Creating the EventBus

Our eventBus is going to be a new Vue instance. We’ll register it in main.js as a local variable and export it as a named [*export*](https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/export). Exporting the variable in this way allows it to be imported into other modules (files). It also avoids other undesirable ways of sharing state such as global variables.

*// main.js*

**import** Vue **from** 'vue';

**import** App **from** './App.vue';

**export** **const** eventBus **=** **new** Vue(); *//NEW*

**new** Vue({

el: "#app",

render: h **=>** h(App)

});

### Why are we making a new Vue instance?

A Vue instance comes with some out the box methods and properties. We’ve already seen some of these:

* data and props and instance properties
* lifecycle methods are instance methods

We also use the instance methods on and emit, and we’ll look at these shortly.

The Vue instance gives us a Vue object which is entirely decoupled from the DOM. All that exists on it are the instance methods.

## Using the EventBus in Components

To use the eventBus throughout the app we’ll have to import it into each component that requires it.

The first place that we’ll import it into is the ListItem component. When we click on an item, we want to emit to the rest of the app the item which has been clicked on.

*// ListItem.vue*

**<**script**>**

**import** { eventBus } **from** '../main.js' *//NEW*

**export** **default** {

name: 'list-item',

props: ['munro']

}

**<**/script>

The import {thing} syntax indicates that we’re importing an object which isn’t the default export from that file.

Next, let’s set up the click event on our list element. When we click an item we want to call a handleClick function. Within this function, let’s log out the munro that this element has access to via props.

*//ListItem.vue*

**<**template lang**=**"html"**>**

**<**li v**-**on:click**=**"handleClick"**>**{{munro.name}}**<**/li> /**/**MODIFIED

**<**/template>

**<**script**>**

**import** { eventBus } **from** '../main.js'

**export** **default** {

name: 'list-item',

props: ['munro'],

methods: { *//NEW*

handleClick(){

console.log('munro', **this**.munro);

}

}

}

**<**/script>

Navigate to the browser and click on a list item, and you should see an object being logged out that has all the details of that Munro.

Now we know we have access to the Munro when we click on the list item. Next, we want to emit that Munro to the rest of the app. We want to board this Munro data onto the eventBus.

*// ListItem.vue*

**<**script**>**

**import** { eventBus } **from** '../main.js'

**export** **default** {

name: 'list-item',

props: ['munro'],

methods: {

handleClick(){

eventBus.$emit('munro-selected', **this**.munro) *//MODIFIED*

}

}

}

**<**/script>

Here we’re calling the $emit instance method on our eventBus. The $ indicates that it is a Vue instance method.

The first argument is the name of the event that we’re emitting, as a string. You can call this event anything you like. We’ll need to re-use it when we want to listen out for this event.

The second argument is the object that we’re emitting (in this case, the details of the Munro that we’re interested in).

Navigate back to your browser and click on an item again. In your Vue dev tools you should see an events icon.



Click on this to switch to events view.

In here you can see the name of our event - munro-selected and if we click on that, within that event’s payload you should see the munro object.

We now have access to this data wherever in our app we decide to listen out for it.

## Listening for the Event

Within App.vue we can see that we have a data value of selectedMunro which is initially set as null. We want to set that value to be the data of the list item we’ve clicked on.

Let’s start off by importing the eventBus into this component.

*//App.vue*

**<**script**>**

**import** { eventBus } **from** './main.js'; *//NEW*

**import** MunrosList **from** './components/MunrosList.vue';

*// AS BEFORE*

**<**/script>

We’re now going to listen out for the data that has been emitted from ListItem. We’ll do this within the mounted function.

*//App.vue*

**<**script**>**

**import** { eventBus } **from** './main.js';

**import** MunrosList **from** './components/MunrosList.vue';

**export** **default** {

name: 'app',

data(){

**return** {

munros: [],

selectedMunro: **null**

};

},

mounted(){

fetch('https://munroapi.herokuapp.com/munros')

.then(res **=>** res.json())

.then(munros **=>** **this**.munros **=** munros)

eventBus.$on('munro-selected', (munro) **=>** { *//NEW*

console.log('within $on', munro);

})

},

components: {

"munros-list": MunrosList

}

}

**<**/script>

Here we’re calling $on on the eventBus instance.

This method takes two arguments:

* The name of the event we’re listening out for (which we designated 'munro-selected' earlier)
* A callback, which takes the payload that was emitted as an argument

Let’s log out this munro to check it is the same payload that is being emitted. This is good practice to ensure you know what data you are working with.

Now we’re happy that we’re receiving the correct information, let’s assign it to our selectedMunro data value.

*//App.vue*

*//AS BEFORE*

mounted(){

fetch('https://munroapi.herokuapp.com/munros')

.then(res **=>** res.json())

.then(munros **=>** **this**.munros **=** munros)

eventBus.$on('munro-selected', (munro) **=>** {

**this**.selectedMunro **=** munro *//MODIFIED*

})

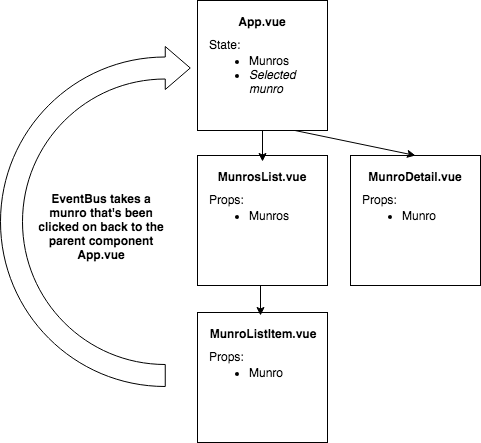
}

Navigate to the browser and click on a munro in the list. Then within your dev tools change back to component view and select the App component. You should in here that selectedMunro has changed from null to our munro object.

We’ve now successfully managed to pass data from one component to another. From here we can use this data to render details about the selected munro.

## Rendering Munro Detail

Let’s make a new component which is responsible for rendering the Munro detail. Our final product will look something like this:



touch src/components/MunroDetail.vue

Let’s start off with our component scaffolding.

*//MunroDetail.vue*

**<**template lang**=**"html"**>**

**<**p**>**I am the munro detail component**<**/p>

**<**/template>

**<**script**>**

**export** **default** {

name: 'munro-detail'

}

**<**/script>

**<**style lang**=**"css" scoped**>**

**<**/style>

Now we need to think about where we want to register this component. Where do we want it to be rendered? We’ll do it within the app component as it wouldn’t make any sense to render it anywhere within the list.

Lets import the component and then register it.

// App.vue

<script>

**import** { eventBus } **from** './main.js';

**import** MunrosList **from** './components/MunrosList.vue';

**import** MunroDetail **from** './components/MunroDetail.vue'; *//NEW*

**export** **default** {

name: 'app',

data(){

**return** {

munros: [],

selectedMunro: **null**

};

},

mounted(){

fetch('https://munroapi.herokuapp.com/munros')

.then(res **=>** res.json())

.then(munros **=>** **this**.munros **=** munros)

eventBus.$on('munro-selected', (munro) **=>** {

**this**.selectedMunro **=** munro

})

},

components: {

"munros-list": MunrosList,

"munro-detail": MunroDetail *//NEW*

}

}

</script>

And then render it next to the munros-list.

*//App.vue*

**<**template**>**

**<**div **class=**""**>**

**<**h1**>**Munros**<**/h1>

**<**div **class=**"main-container"**>**

**<**munros**-**list :munros**=**'munros'**><**/munros-list>

**<**munro**-**detail**><**/munro-detail> /**/**NEW

**<**/div>

**<**/div>

**<**/template>

Great, now in the browser you should see I am the munro detail component rendering next to the list. But we don’t want this, we want the details about the munro we’ve selected.

We want to pass details of the selectedMunro down to the munro-detail component. How do we pass data from a parent component to a child? props!

//App.vue

<template>

<div class="">

<h1>Munros</h1>

<div class="main-container">

<munros-list :munros='munros'></munros-list>

<munro-detail :munro='selectedMunro'></munro-detail> //MODIFIED

</div>

</div>

</template>

Here we’re creating a prop called munro and binding it to the selectedMunro data value of App.vue.

We now need to register this prop in munro-detail.

*//MunroDetail.vue*

**<**script**>**

**export** **default** {

name: 'munro-detail',

props: ['munro'] *//NEW*

}

**<**/script>

If we look in the dev tools, when a munro has been selected you should see the MunroDetail component’s prop of munro is set as the selectedMunro.

The last thing we have to do is use this data to render details about the Munro.

//MunroDetail.vue

<template>

<div> //MODIFIED

<h3>{{munro.name}}</h3>

<p>Height: {{munro.height}}</p>

<p>Region: {{munro.region}}</p>

<p>Meaning: {{munro.meaning}}</p>

</div>

</template>

Navigate back to the browser and refresh the page. The list no longer renders and there’s an error in the browser.

cannot read property name of null

We have an issue. The selectedMunro starts off as null and isn’t populated until we click on a list item. However, this error is preventing the app from rendering so we can’t click on an item to set the selectedMunro.

To fix this we need to only render the div containing the Munro details if there is a Munro to render. i.e selectedMunro is not null.

We can do this with an if statement on the div.

//MunroDetail.vue

<div v-if='munro'> //MODIFIED

<h3>{{munro.name}}</h3>

<p>Height: {{munro.height}}</p>

<p>Region: {{munro.region}}</p>

<p>Meaning: {{munro.meaning}}</p>

</div>

Sorted! Now our page renders and when we click on a munro its details render.

## Recap

What does an EventBus allow us to do?

Answer:

What kind of object is an eventBus?

Answer:

What methods do we use on our eventBus?

Answer:

## Conclusion

With eventBus, we now know how to pass data from child to parent component, allowing us to continue separating the concerns of our components and working towards maintainable, reusable code.

# ountries - Multi Components Lab

**Lab Duration: 120 mins**

### Learning Objectives

* Be able to create a Vue app with multiple components and templates

## Brief

Your task is to create an app that shows info for all the countries using multiple components. Use the [Countries API](https://restcountries.eu/rest/v2/all) to make a request to get the data.

### MVP

* Display a list of country names.
* Add a click event to the list item which should then render more detail about that country (name, capital, population).
* Use reusable components.

### Extensions

* Instead of rendering a list, populate a dropdown with all of the countries names.
* Add a change event to the select that renders information about the selected country.

### Advanced Extensions

* Add the countries flag and languages spoken in the country to the country detail component.
* Add a search bar to the page so that when a user enters the countries name the country detail component renders. Try to achieve this without the user having to enter the whole name.

## Planning

Draw a diagram of your files, detailing:

* the data, props, components and methods for each component.
* the flow of data throughout the application.

**Expected Components for MVP**

* CountriesList
* ListComponent
* CountryDetail

**Expected Components for Extensions**

* CountrySelect
* CountryDetail

# Energy Mix Lab

**Lab Duration: 180 mins**

## Learning Objectives

* Be able to read data from an API, and manipulate it for use
* Be able to handle user interaction and practice business logic
* Be able to use third party libraries

### Brief / MVP

The “energy mix” is the mixture of energy sources that the UK derives its energy from at any given time. For example:

Hydro: 25%

Nuclear: 18%

Wind: 40%

And so on.

You have been asked to display information relating to the UK’s energy mix, using the National Grid’s API:

https://carbon-intensity.github.io/api-definitions/#get-generation

Set up a Vue application that will display this information, and the relevant times in a suitable format - specifically, a chart. The [Vue Google Charts](https://www.npmjs.com/package/vue-google-charts) library is something you can use to do this.

The issue you will face is that your data should look like this:

generationMix: [

["Fuel", "Percentage"],

["Nuclear", 20.4],

["Hydro", 18.9],

...

]

…But the API provides it in a very different format. You will have to re-shape the data to meet your needs. You will also need to read the documentation on Vue Google Charts to fully understand how to implement it in your app.

### Extensions

Choose some or all of these:

* Allow the user to select the start and finish time and dates for the API data using [this endpoint](https://carbon-intensity.github.io/api-definitions/?shell#get-generation-from-to) - this will return a different response which will require some more in-depth manipulation to provide the chart with the data it requires.
* Investigate other types of chart - perhaps a pie chart?
* Graph out any other data you are interested in from the API

# Lab: Make an app with the Brewdog API

**Lab Duration: 180 mins**

## Learning Objectives

* Be able to make API requests in a Vue app
* Be able to design a Vue component hierarchy from scratch
* Be able to implement Vue components

Your task is to build an app that uses [this API](https://punkapi.com/documentation/v2) to display information on a variety of beers.

[This endpoint](https://api.punkapi.com/v2/beers) will provide you with some data detailing multiple beers.

**It is important for this task you consider what your views will be, but also how you can reuse components to construct these different views.**

## MVP

Your app should be able to:

* Allow the user to view all the beers
* Allow the user to view more detailed information on a selected beer
* Allow the user to mark beers as ‘favourites’
* Display the user’s favourite beers

## Extensions

* Prevent the user from marking the same beer as a favourite more than once
* Allow the user to deselect favourite beers
* Display the selected beer’s ingredients (without duplicates)
* The endpoint provided will only return 20 beers at a time. Modify your initial request to fetch all 300+ beers the API provides.

# Weekend Homework: Vue Application with Requests

### Learning Objectives

* Be able to create a multi-component web application, with Vue
* Be able to make API requests to load JSON data into your application

## Brief

Your task is to create an application that makes a request to an API and displays the data.

Suggested APIs:

1. Studio Ghibli: https://ghibliapi.herokuapp.com/
2. Reddit: (Append .json to any Reddit URL - for example https://www.reddit.com/r/javascript.json)
3. Guardian search: https://content.guardianapis.com/search?q=brexit&format=json&api-key=test

These APIs all allow browser requests without authentication or keys. (In the case of the Guardian, api-key=test should be sufficient.)

There is a more extensive list of public APIs [here](https://github.com/public-apis/public-apis) that have varying degrees of accessibility. If you choose to use a different API, make sure you are able to load the data into your application without issue, so that you can spend the time focussing on building your application. We suggest you do not use an API that requires authentication (OAuth), though using an API which requires a key is fine, as long as you are able to get a key quickly and easily.

### MVP

* The application should display data from an API request.
* The application should have a clear separation of concerns (multiple components)
* Take input from the user to update the page. You could update the page by filtering or manipulating the data on user interaction, or you might make further API requests to load more data that is then displayed.

### Extensions

Looking into a library to visual the data.

* [Leaflet](https://leafletjs.com/) is an open-source library for rendering maps
* [Google Charts](https://developers.google.com/chart/) is a library for rendering charts and graphs
* [Canvas](https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial) is a drawing surface for JavaScript.

You will need to use the library’s documentation to integrate it into your application.

## Planning

* Look at the JSON you are going to be loading into your app and based on the data, draw a wireframe
* Start by loading the data from the API into your application and checking that it has loaded, before doing any work on the views
* Plan your components and draw a diagram of the data flow through the application

### PDA Reminder:

As part of this homework you are required to take screenshots of the following:

Show an API being used within your program. Take a screenshot of:

- The code that uses or implements the API

- The API being used by the program whilst running

* Go to your [PDA Checklist](https://github.com/codeclan/pda/tree/master/Evidence%20Gathering%20Portfolio)
* Submit your PDA evidence (screenshots, etc.) to your own PDA repo

PDA Reference: P 16