Lab 7: Introduction to VueJS

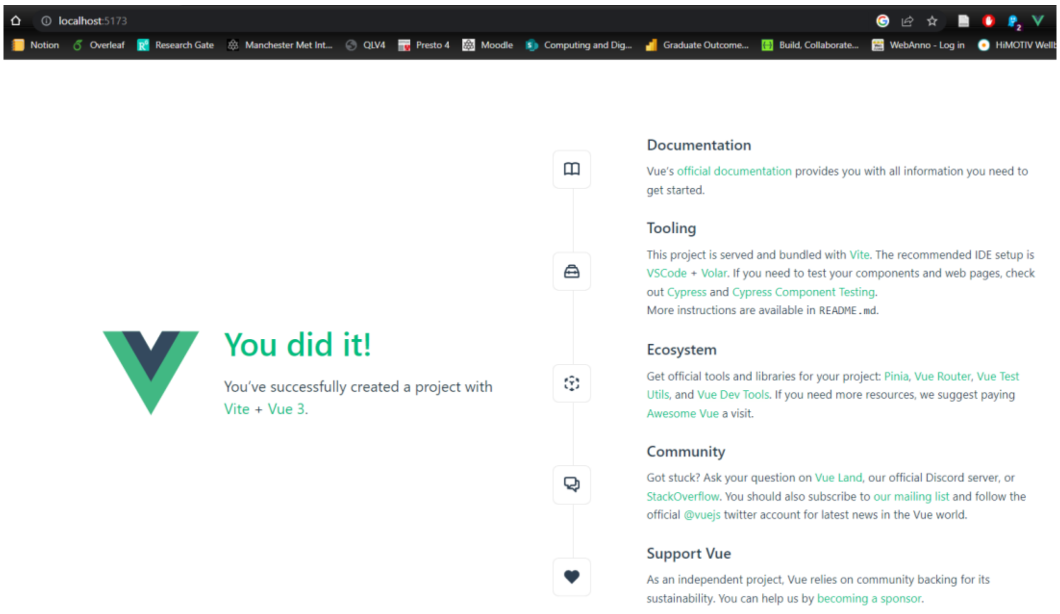
Learning Objectives

- Get started with VueJS and the Front End
- Practice creating components and understand how reusable components can be created using props
- Understand how routing works within our front end applications

Exercise 1: Create a Vue app

First things first, we need to learn how to create VueJS apps so that we can start to build our front-end applications. When we create a Vue app, the template adds in some starter code, so we also need to know how to clean the app up.

1. Create a new directory and open it in our terminal (remember the front end app is separate from the back end applications that we have been building so far, so make sure the directories are separate too)
2. In the terminal, run “npm init vue@latest” – the terminal will ask you some questions about the application and optional installs. Feel free to just skip through them for now by pressing “enter” on your keyboard.
3. Once the project has been initialised, “cd” into the project directory and run “npm install” and “npm run dev”
4. Your terminal should give you a URL for accessing your site. For me, it opens up on port 5173 (but yours might be different). Open it up in your browser to check:



You did it!

You’ve successfully created a project with Vite + Vue 3.

Documentation

Vue’s official documentation provides you with all information you need to get started.

Tooling

This project is served and bundled with Vite. The recommended IDE setup is VSCode + Volar. If you need to test your components and web pages, check out Cypress and Cypress Component Testing.
More instructions are available in README.md.

Ecosystem

Get official tools and libraries for your project: Pinia, Vue Router, Vue Test Utils, and Vue Dev Tools. If you need more resources, we suggest paying Awesome Vue a visit.

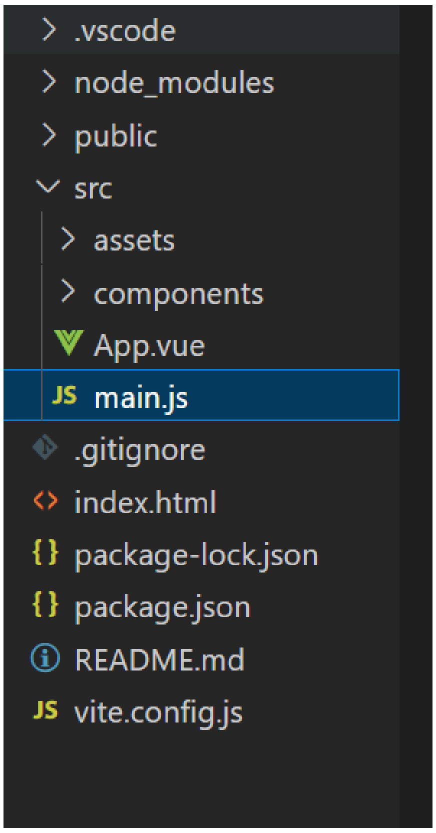
Community

Got stuck? Ask your question on Vue Land, our official Discord server, or StackOverflow. You should also subscribe to our mailing list and follow the official @vuejs twitter account for latest news in the Vue world.

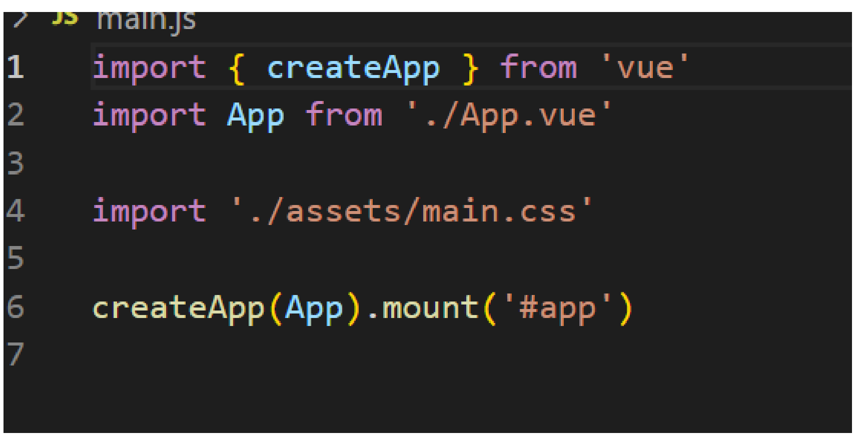
Support Vue

As an independent project, Vue relies on community backing for its sustainability. You can help us by becoming a sponsor.

5. We want to remove all of the starter templating code (the bits you can see in the browser at the moment). For this, first open the application directory in VS Code.

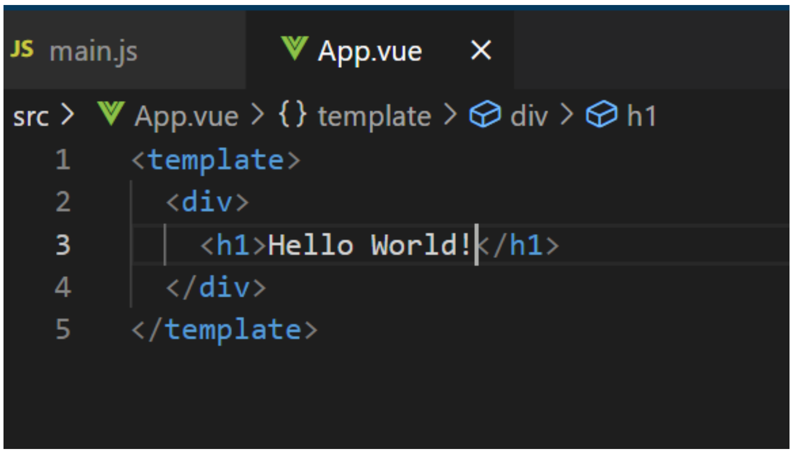


6. The “src” directory contains the template code. Delete the “assets” and “components” directories.
7. In the main.js file, remove the line that imports the CSS file – because we have just deleted it.



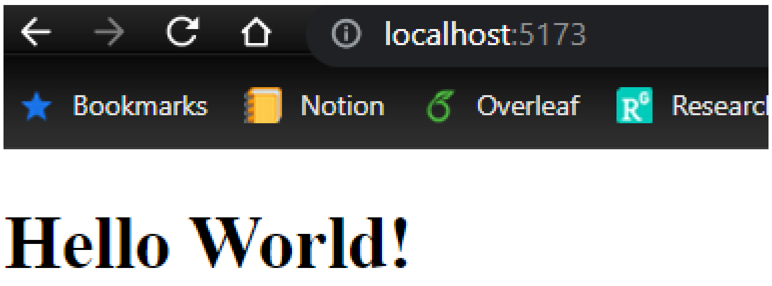
```
import { createApp } from 'vue'
import App from './App.vue'

import './assets/main.css'

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

```

8. This main.js file creates our app and runs the root component (App.vue). However, we want to clean up our App.vue component too. In the App.vue file, delete everything and replace it with the following:


main.js App.vue
src > App.vue > {} template > div > h1

```
<template>
  <div>
    <h1>Hello World!</h1>
  </div>
</template>
```

9. Test your app in the browser. Make sure that there are no errors.



Hello World!

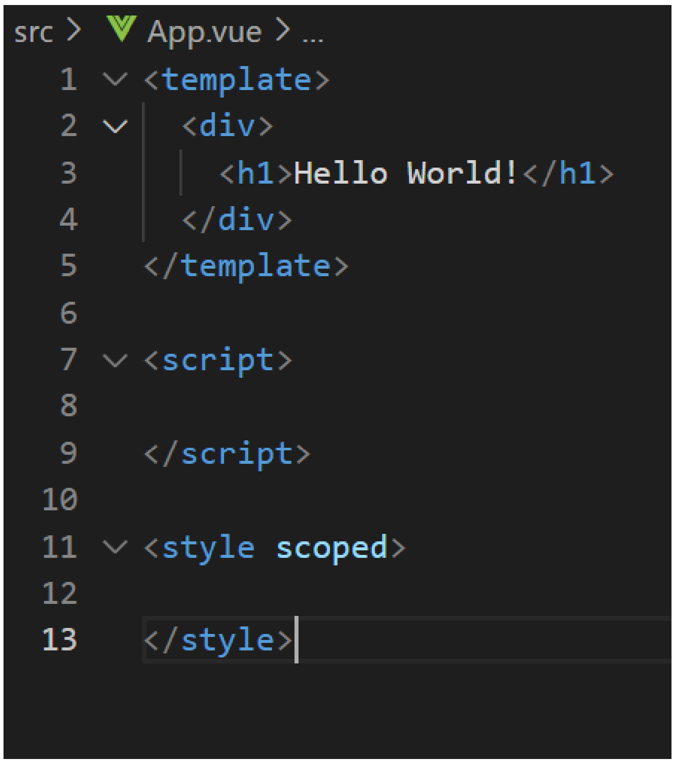
Exercise 2: Create our first component

Vue components are split into three sections: template, script and style.

Template: contains all of the HTML (i.e., the structure of our component)

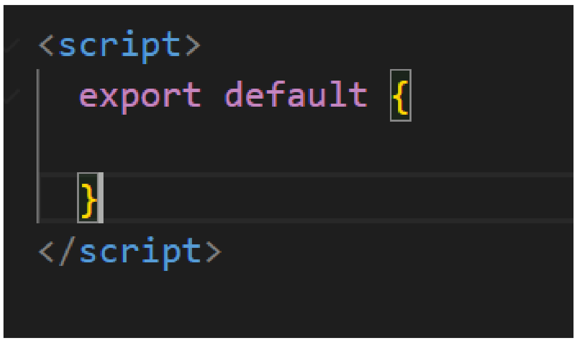
Script: contains all of the JS to give our component its functionality (e.g., API calls, form validation, data storage)

Style: contains all of the style rules for making a component pretty.



```
src > App.vue > ...
<template>
  <div>
    <h1>Hello World!</h1>
  </div>
</template>

<script>

</script>

<style scoped>

</style>
```

The script portion of our component is exported as an object:



```
<script>
  export default {

  }
</script>
```

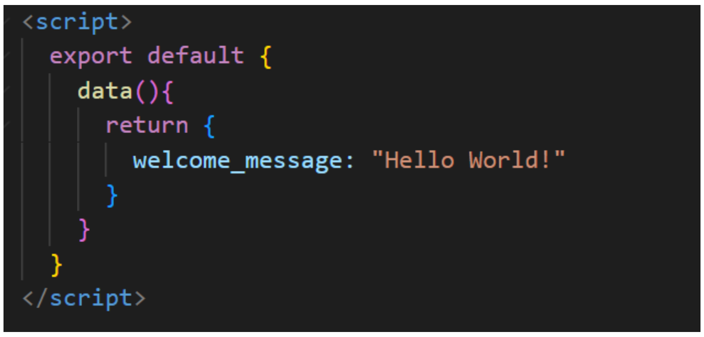
And contains three main parts (at least three main parts that are relevant to us for now):

data(): a function which returns an object. This object contains all of the data relevant to the component (i.e., our components variables)

created(): a function that is executed when the component first loads (a bit like a constructor in an object)

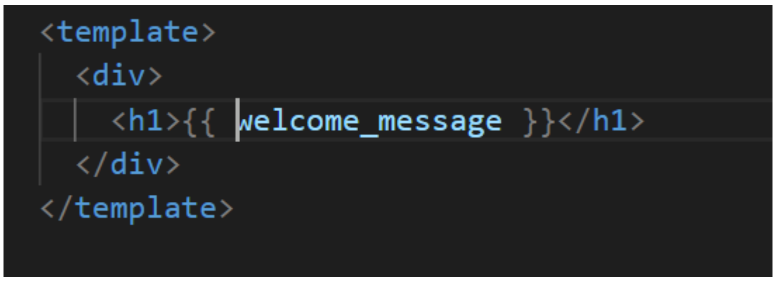
methods: an object containing all of the relevant methods for our component

1. Copy the code below. Here we are creating a variable for our component.



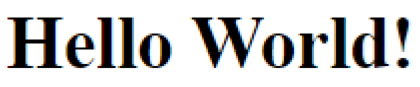
```
<script>
  export default {
    data(){
      return {
        welcome_message: "Hello World!"
      }
    }
  }
</script>
```

2. We can view this variable in our template section by using double curly-brackets:



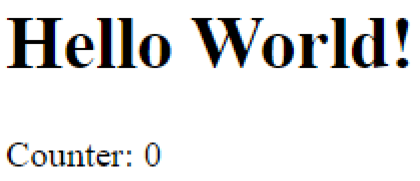
```
<template>
  <div>
    <h1>{{ welcome_message }}</h1>
  </div>
</template>
```

3. Test it in your browser



Hello World!

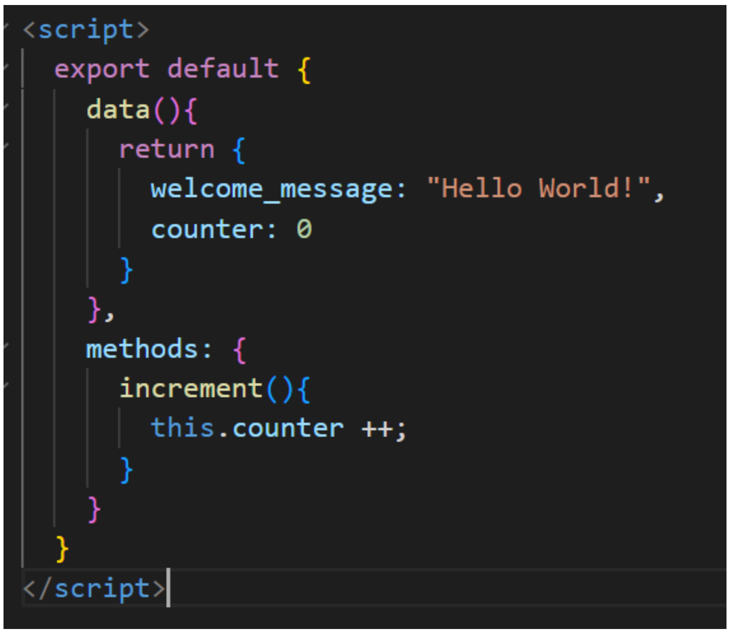
4. Create another variable called "counter", initialise it to 0
5. Add a <p> tag to your template to display the counter in your browser



Hello World!

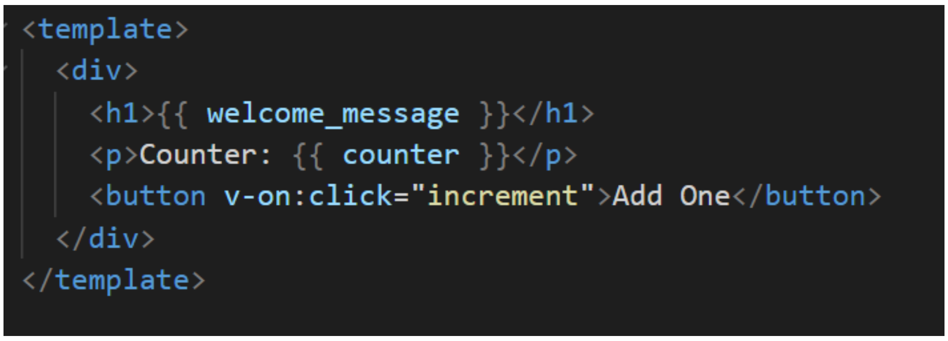
Counter: 0

6. In the methods object, we can create a function to increment the counter. Alter your script so that it looks like the below:



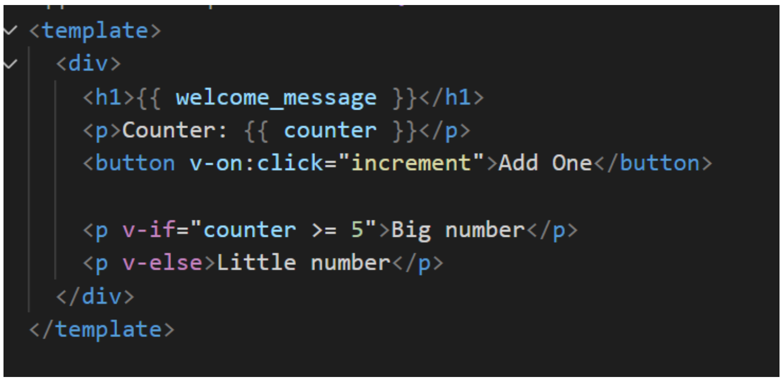
```
<script>
  export default {
    data(){
      return {
        welcome_message: "Hello World!",
        counter: 0
      }
    },
    methods: {
      increment(){
        this.counter ++;
      }
    }
  }
</script>
```

7. We have created the increment function, but it is never called. Let's add a button to our component which will call the increment function when clicked:



```
<template>
  <div>
    <h1>{{ welcome_message }}</h1>
    <p>Counter: {{ counter }}</p>
    <button v-on:click="increment">Add One</button>
  </div>
</template>
```

8. Test the button in your browser.
9. The "v-on:click" that we saw in the button is called a Vue directive. There are many different Vue directives that allow us to add interactivity to our components. For example, let's conditionally render another <p> tag depending on whether the counter has reached a certain value. For conditionally rendering content, we can use the "v-if" and "v-else" directives:



```
<template>
  <div>
    <h1>{{ welcome_message }}</h1>
    <p>Counter: {{ counter }}</p>
    <button v-on:click="increment">Add One</button>

    <p v-if="counter >= 5">Big number</p>
    <p v-else>Little number</p>
  </div>
</template>
```

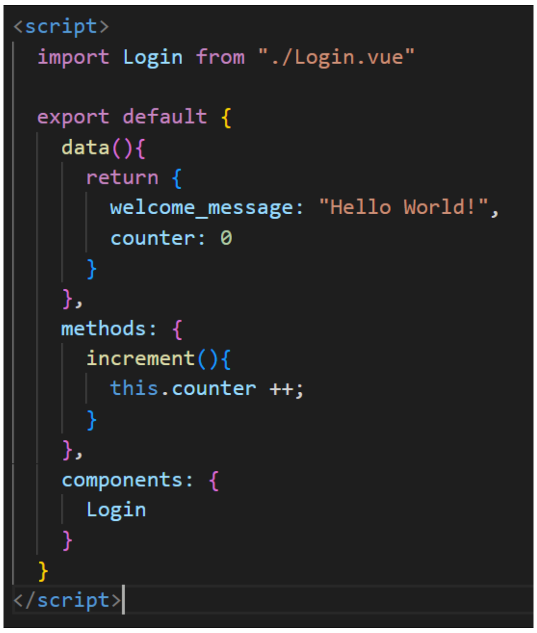
10. Test it in your browser.
11. There are a lot of Vue directives that you can use, and you can even build your own. The Vue documentation is extremely good, use the documentation to play around with the v-for directive: https://vuejs.org/api/built-in-directives.html

Exercise 3: Create a login form

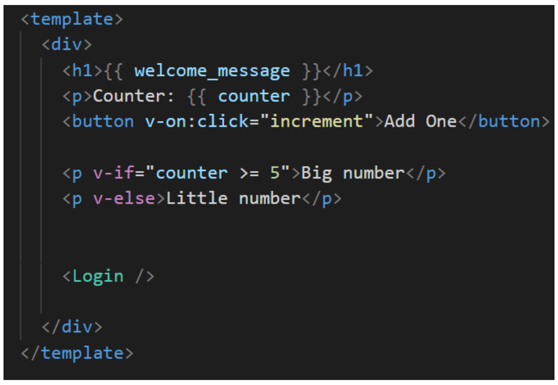
We've now played around with some of the basic concepts within Vue, but let's have a look at how they work with a more realistic example. Let's create a simple Login form.

Exercise 3.1: Form Basics

1. In the "src" directory, create a new file called "Login.vue"
2. Add a simple template that just contains a <h1> tag with the text "Login" (like how we started the previous exercise)
3. We can import this component into our App component. To do this, you need to import the component into the script section, and then add the component to the template section. We also need to make sure that we export the component in the script section as shown below:

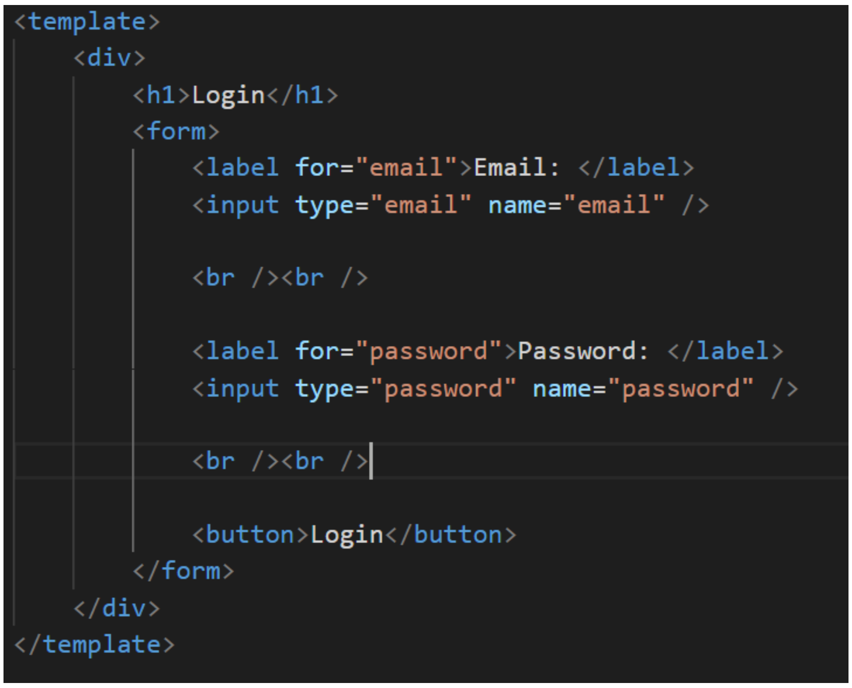


```
<script>
  import Login from "./Login.vue"

  export default {
    data(){
      return {
        welcome_message: "Hello World!",
        counter: 0
      }
    },
    methods: {
      increment(){
        this.counter ++;
      }
    },
    components: {
      Login
    }
  }
</script>
```



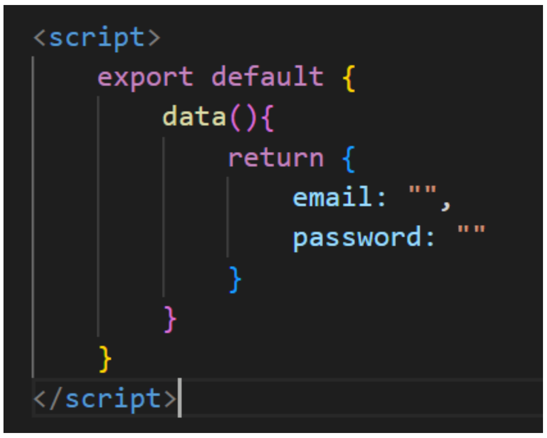
```
<template>
  <div>
    <h1>{{ welcome_message }}</h1>
    <p>Counter: {{ counter }}</p>
    <button v-on:click="increment">Add One</button>

    <p v-if="counter >= 5">Big number</p>
    <p v-else>Little number</p>


    <Login />

  </div>
</template>
```

4. Test in your browser, can you see the Login component imported into your App?
5. In the Login component, let’s create a simple HTML form:



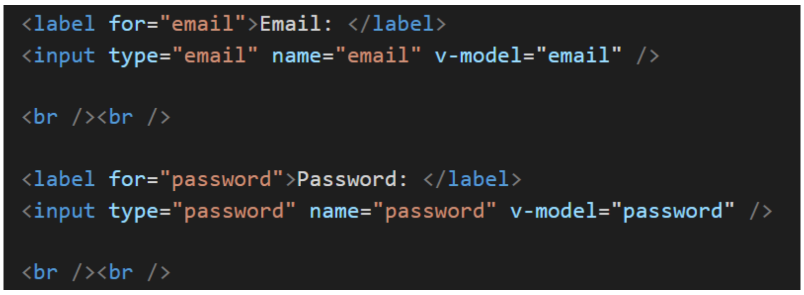
```
<template>
    <div>
        <h1>Login</h1>
        <form>
            <label for="email">Email: </label>
            <input type="email" name="email" />

            <br /><br />

            <label for="password">Password: </label>
            <input type="password" name="password" />

            <br /><br />

            <button>Login</button>
        </form>
    </div>
</template>
```

6. In the script portion of our component, we first want to be able to access the data being inputted. To do this, create some empty variables for storing the data:



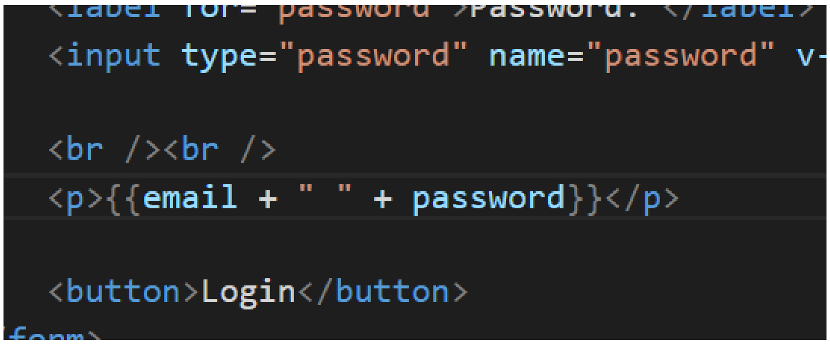
```
<script>
    export default {
        data(){
            return {
                email: "",
                password: ""
            }
        }
    }
</script>
```

7. Next, we can bind our inputs to these variables with the v-model directive:



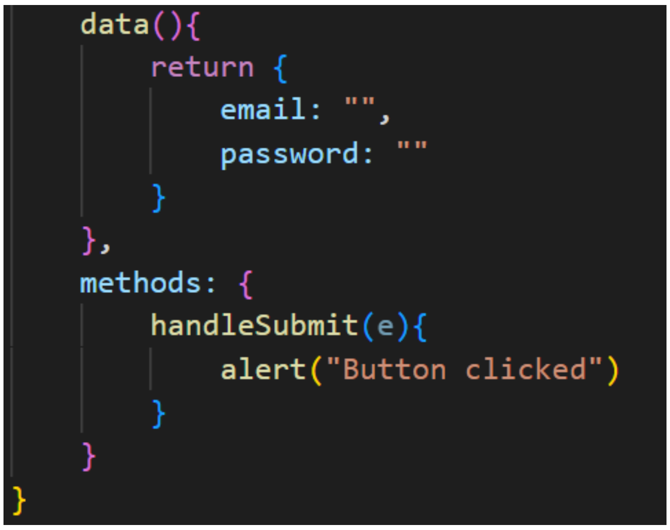
```
<label for="email">Email: </label>
<input type="email" name="email" v-model="email" />

<br /><br />

<label for="password">Password: </label>
<input type="password" name="password" v-model="password" />

<br /><br />
```

8. We can test this works by temporarily printing the email and password variable to the browser window:



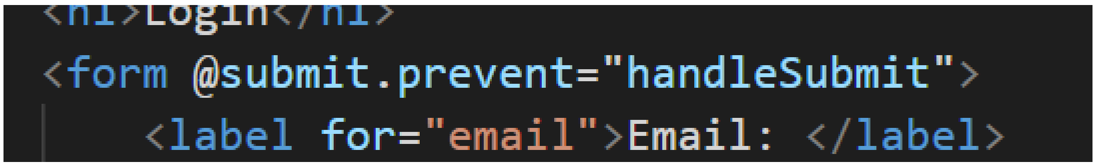
```
<label for="password">Password: </label>
<input type="password" name="password" v-

<br /><br />
<p>{{email + " " + password}}</p>

<button>Login</button>
</form>
```

9. When the Login button is clicked, we want to export a function. This function will handle all our validation, and then eventually handle sending the data to our back-end (we will cover API interactions in a later lab). Let's first add a methods section to the script portion to handle the button click:



```
    data(){
        return {
            email: "",
            password: ""
        }
    },
    methods: {
        handleSubmit(e){
            alert("Button clicked")
        }
    }
}
```

10. Then in the template, this line calls the function when the form is submitted:

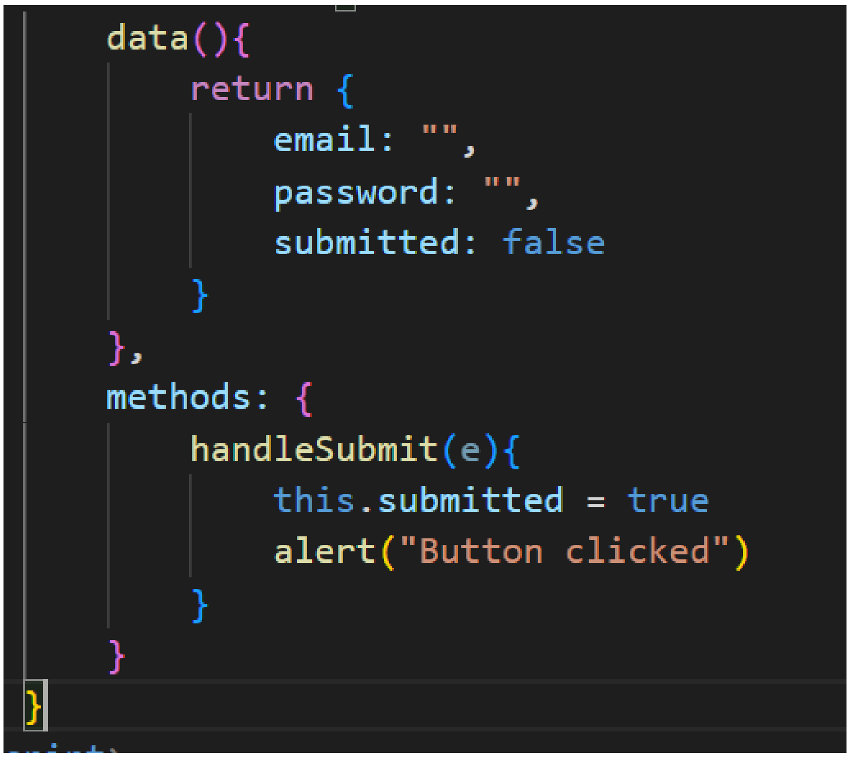


```
<h1>Login</h1>
<form @submit.prevent="handleSubmit">
    <label for="email">Email: </label>
```

11. Test your button click

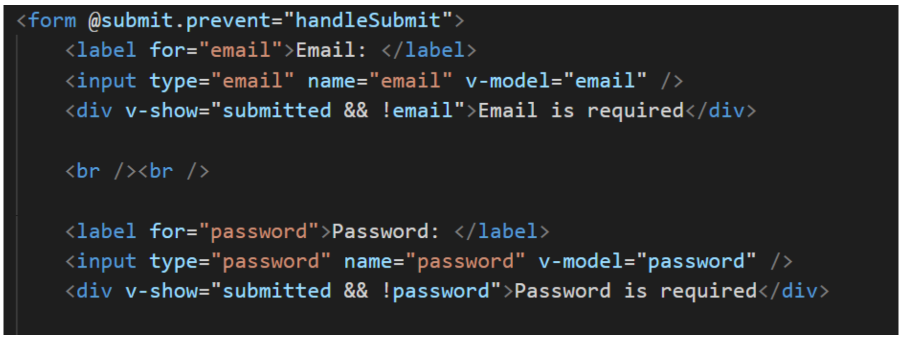
Exercise 3.2: Form validation

1. Now that we have out base form in place, we can add some validation to the inputs. The first step is to give some user feedback if they submit the form without entering all of the required fields. Start by created a variable called “submitted” which is initialised to false, and then changed to true when the button is clicked:



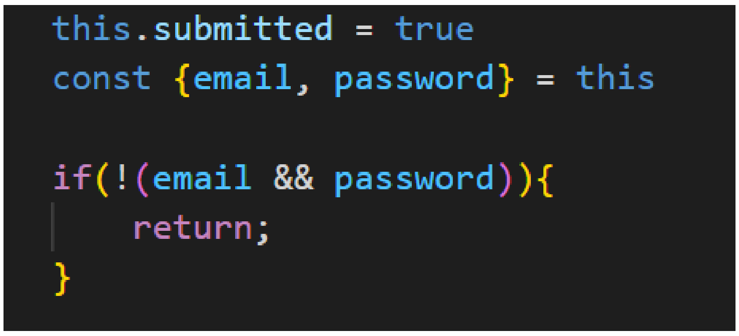
```
    data(){
        return {
            email: "",
            password: "",
            submitted: false
        }
    },
    methods: {
        handleSubmit(e){
            this.submitted = true
            alert("Button clicked")
        }
    }
}
```

2. Now, we can add divs to our template that only show if the form has been submitted but the field is empty (using v-show):



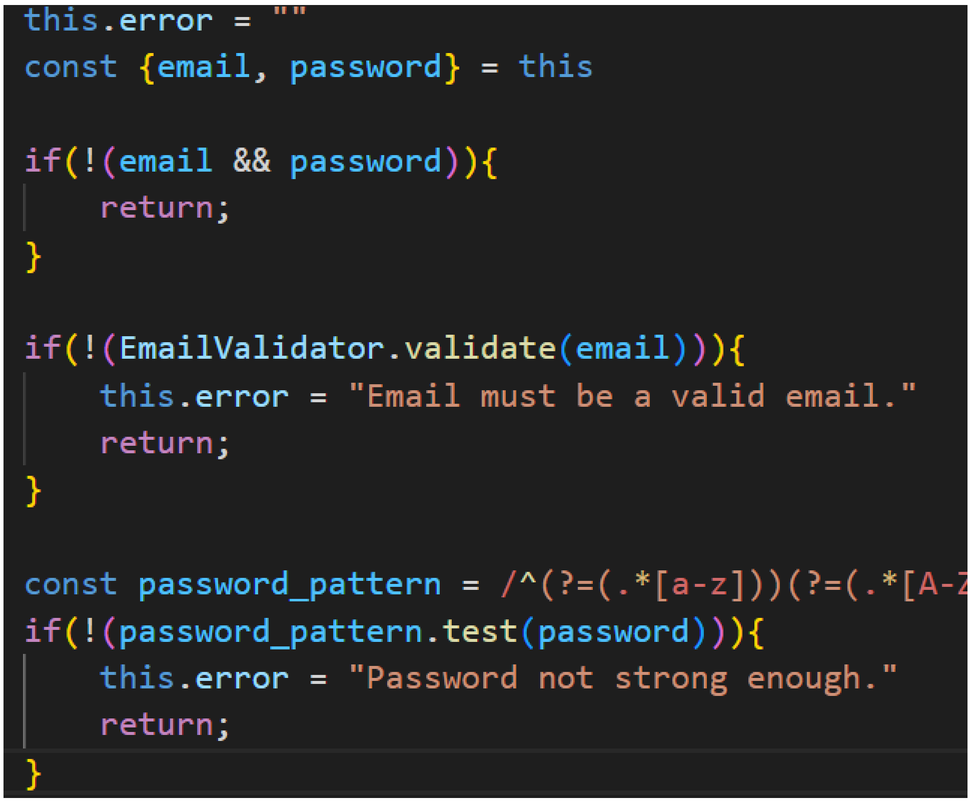
```
<form @submit.prevent="handleSubmit">
    <label for="email">Email: </label>
    <input type="email" name="email" v-model="email" />
    <div v-show="submitted && !email">Email is required</div>

    <br /><br />

    <label for="password">Password: </label>
    <input type="password" name="password" v-model="password" />
    <div v-show="submitted && !password">Password is required</div>
```

3. We can also check in the method that both the email and password have been submitted, and if not then simply return the form without doing anything (other than displaying the divs from the previous step:



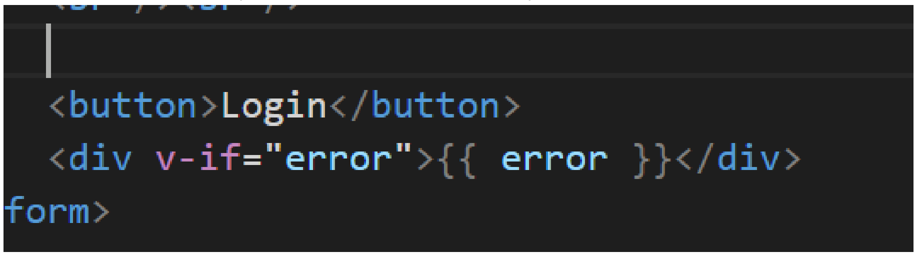
```
this.submitted = true
const {email, password} = this

if(!(email && password)){
    return;
}
```

4. Test your form.
5. If both an email and password have been submitted, lets check that they are both valid. We can do this using the same methods as from the back-end application. For me, I like to use the email-validator package for email checking, and a regular expression for passwords. I write any issues out to a “error” variable (**remember, the email validator library is external, so needs to be installed and imported in – read the documentation!**).



```
this.error = ""
const {email, password} = this

if(!(email && password)){
    return;
}

if(!(EmailValidator.validate(email))){
    this.error = "Email must be a valid email."
    return;
}

const password_pattern = /^(?=(.*[a-z]))(?=(.*[A-Z
if(!(password_pattern.test(password))){
    this.error = "Password not strong enough."
    return;
}
```

6. If there is an error, I can output it to a div in the template, like before!

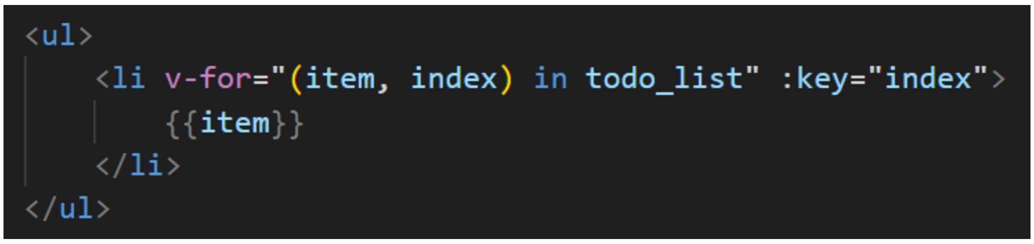


```
  <button>Login</button>
  <div v-if="error">{{ error }}</div>
form>
```

7. Test your form. If the user passes all validation without error, the next step would be to make the API request to the back-end (but we’ll look at that next week).

Exercise 4: Create a ToDo list component

1. Create a new component and import it into your App component as we have done in the previous exercise.
2. Create a basic form with a single text field and a button. This form will be used to add items to your todo list.
3. Create the script portion which should include a variable for the text field, an array of list items, and a method for handling the submit
4. When the button is clicked, your handleSubmit function should validate that the text field isn't empty, and then add the item to the list (with the .push() function).
5. Use the v-for directive to display the items in your component. The :key directive before uses the index of the item in the array to provide each element with a unique id:



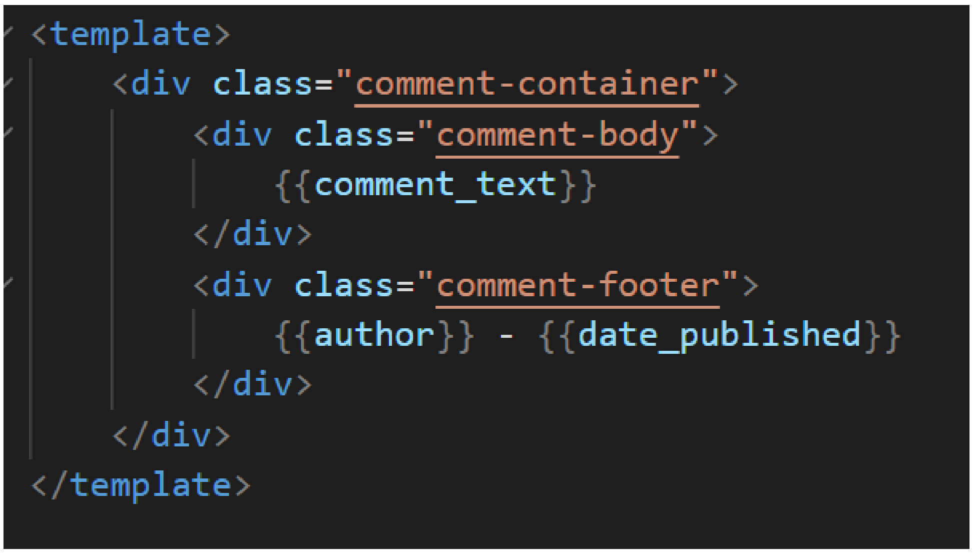
```
<ul>
    <li v-for="(item, index) in todo_list" :key="index">
        {{item}}
    </li>
</ul>
```

6. Can you add functions for deleting items from the list? What about editing?

Exercise 5: Reusable components and props

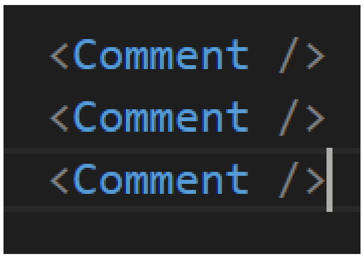
When we are creating component-based applications, we want to make as many reusable components as possible in order to make development quick and easy. Let's start with the humble "Comment" component. We will create a component to display a single comment, which we can style to make it look pretty. That way, all comments across our application will look consistent.

1. Start by creating a Comment component and giving it a simple template. Perhaps you want to style it, so it looks pretty? For now we can add some dummy data in the data() function:

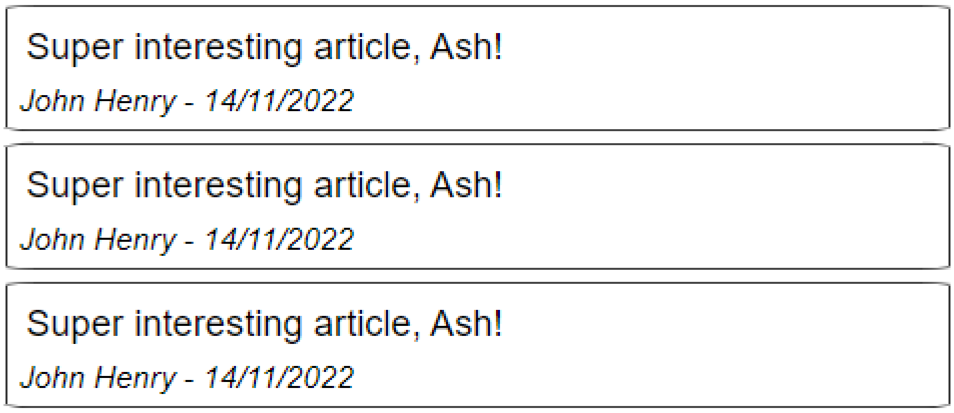


```
<template>
    <div class="comment-container">
        <div class="comment-body">
            {{comment_text}}
        </div>
        <div class="comment-footer">
            {{author}} - {{date_published}}
        </div>
    </div>
</template>
```

2. We can now import it into our App component. If we want to display two comments, we simply add the Comment component twice:



```
<Comment />
<Comment />
<Comment />
```

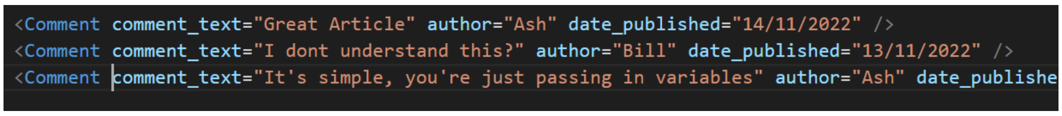


Super interesting article, Ash!
John Henry - 14/11/2022

Super interesting article, Ash!
John Henry - 14/11/2022

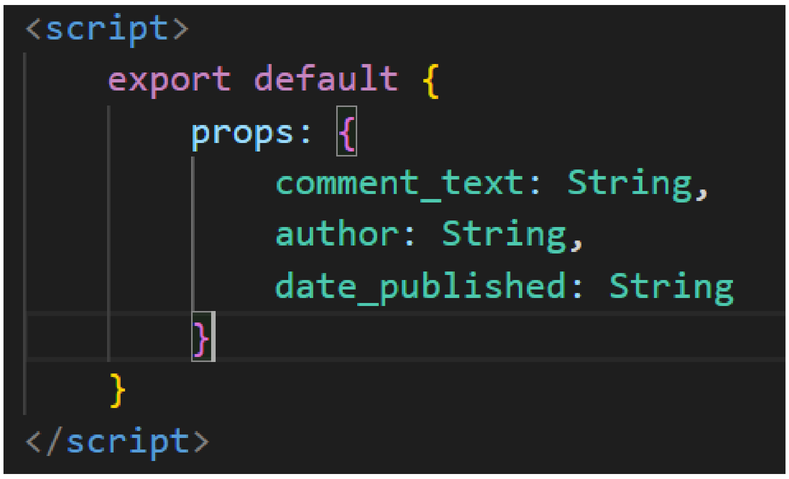
Super interesting article, Ash!
John Henry - 14/11/2022

3. All good, but the values are hardcoded. Meaning the data within the component will always be the same. Instead, we can pass data into the component as a set of properties ("props" for short):



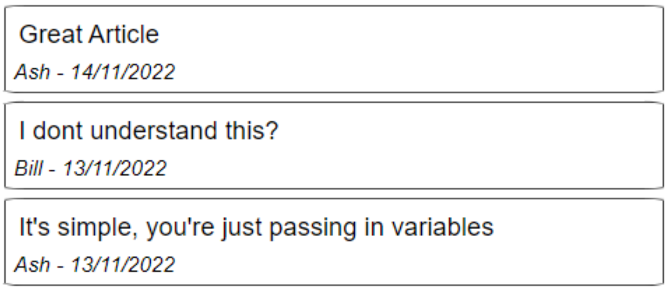
```
<Comment comment_text="Great Article" author="Ash" date_published="14/11/2022" />
<Comment comment_text="I dont understand this?" author="Bill" date_published="13/11/2022" />
<Comment comment_text="It's simple, you're just passing in variables" author="Ash" date_publishe
```

4. In our comment component, we need to replace the data() function with a props object. The probs object states the datatype of the incoming props:



```
<script>
    export default {
        props: {
            comment_text: String,
            author: String,
            date_published: String
        }
    }
</script>
```

5. Now we can tailor the data going into the comment and make the Comment component more dynamic:

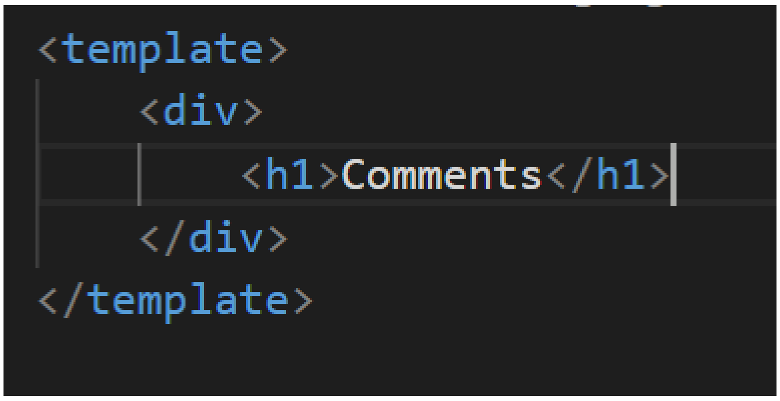


Great Article
Ash - 14/11/2022

I dont understand this?
Bill - 13/11/2022

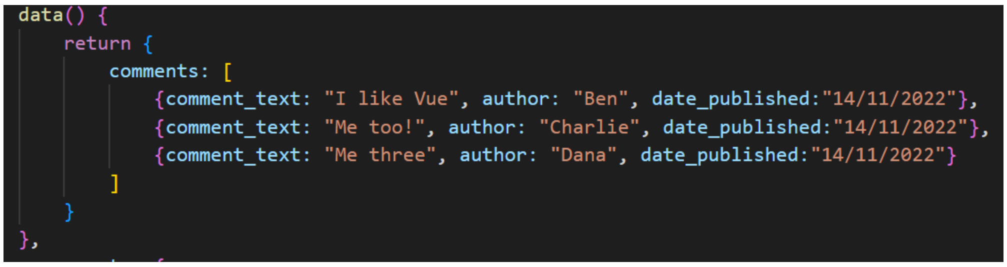
It's simple, you're just passing in variables
Ash - 13/11/2022

6. How about we next create a component that will display a list of comments (using the Comment component we have just made)? To begin with, create a CommentList component and import it into our App component:



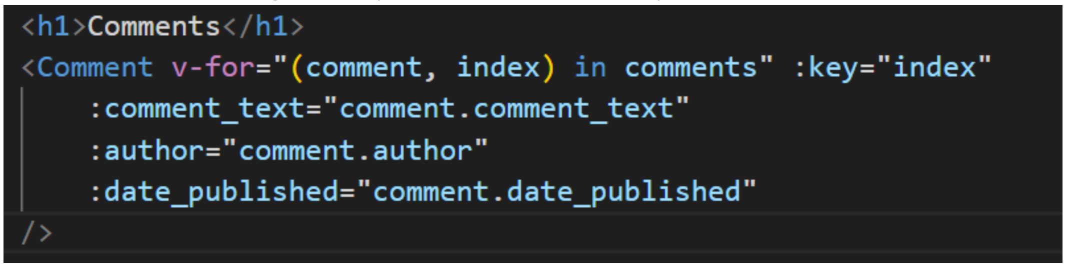
```
<template>
    <div>
        <h1>Comments</h1>
    </div>
</template>
```

7. In the data() function, create a list of comments (in the future, we will pull this data from the API):



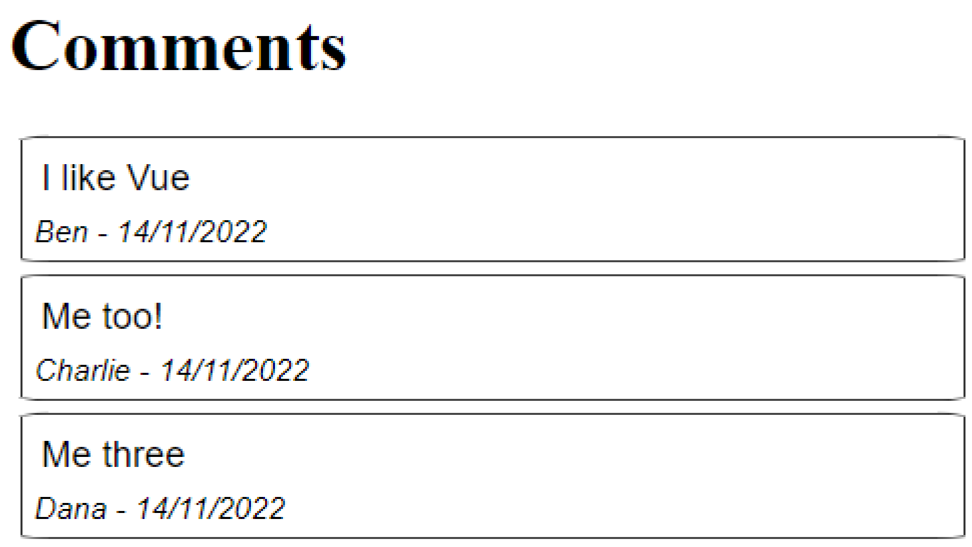
```
data() {
    return {
        comments: [
            {comment_text: "I like Vue", author: "Ben", date_published:"14/11/2022"},
            {comment_text: "Me too!", author: "Charlie", date_published:"14/11/2022"},
            {comment_text: "Me three", author: "Dana", date_published:"14/11/2022"}
        ]
    }
},
```

8. Now for each comment in comments, add a Comment component to the template, adding in the relevant props. The colon before the prop just tells the component that we are passing in a variable (don't forget to import the Comment component):



```
<h1>Comments</h1>
<Comment v-for="(comment, index) in comments" :key="index"
    :comment_text="comment.comment_text"
    :author="comment.author"
    :date_published="comment.date_published"
/>
```

9. Test it out in your browser. Imagine if the CommentList component took in an article_id as a prop, and then would automatically pull the comments for that article_id from the API! Can you see how component-based frameworks can be very powerful when done correctly?



Comments

I like Vue
Ben - 14/11/2022

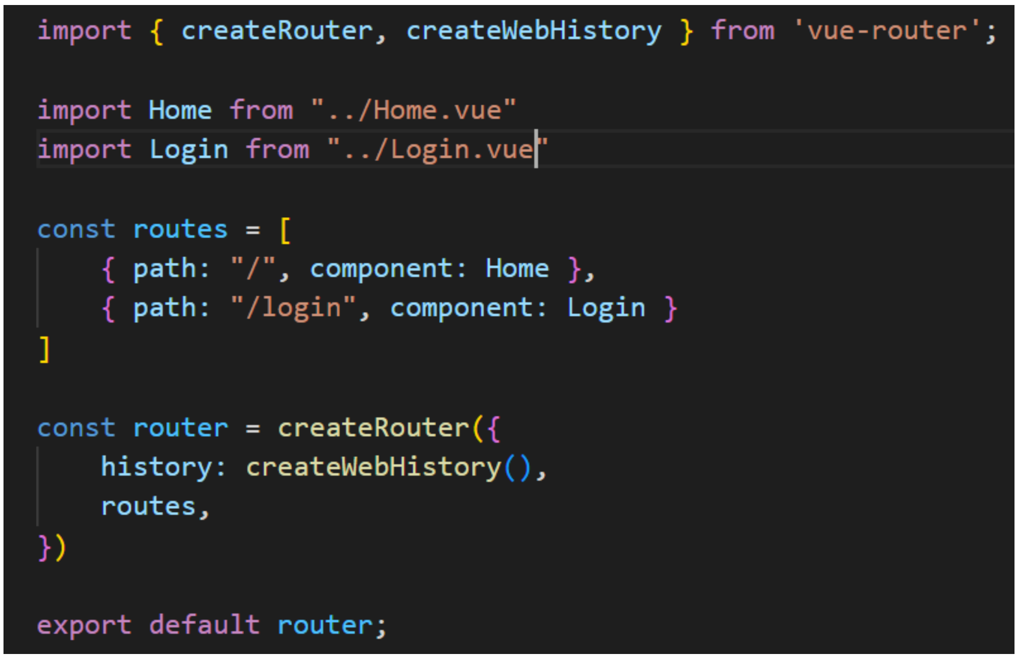
Me too!
Charlie - 14/11/2022

Me three
Dana - 14/11/2022

Exercise 6: Implement a Router

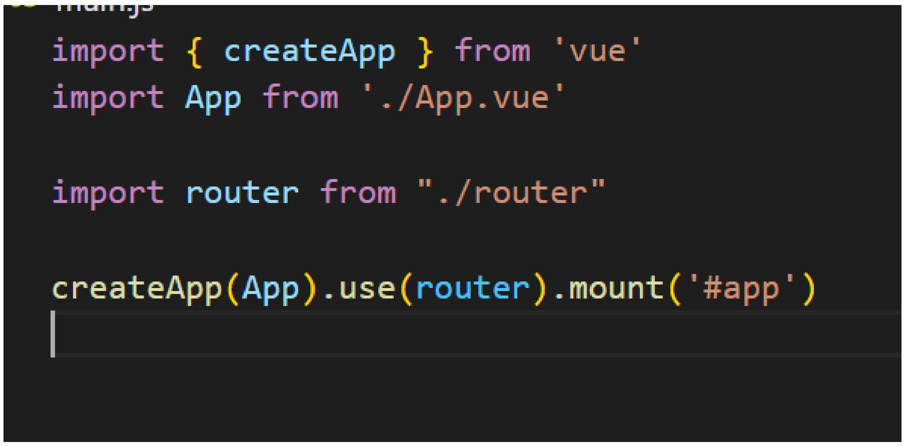
So far so good, but websites are not all located on one page. We want to have multiple pages and ways of navigating between them. We also want to block users from accessing certain webpages unless they are logged in. For all of this, we can use a router.

1. First install the vue-router into your application: "npm install --save vue-router"
2. In your "src" directory, create a directory called "router". Inside it, create a "index.js" file
3. Inside the index.js file, we need to:
    a. Import "createRouter" and "createWebHistory" from "vue-router"
    b. Create an array which contains all of our route paths and components
    c. Create the router with the "createRouter" function
    d. Export the router so that it is publicly available



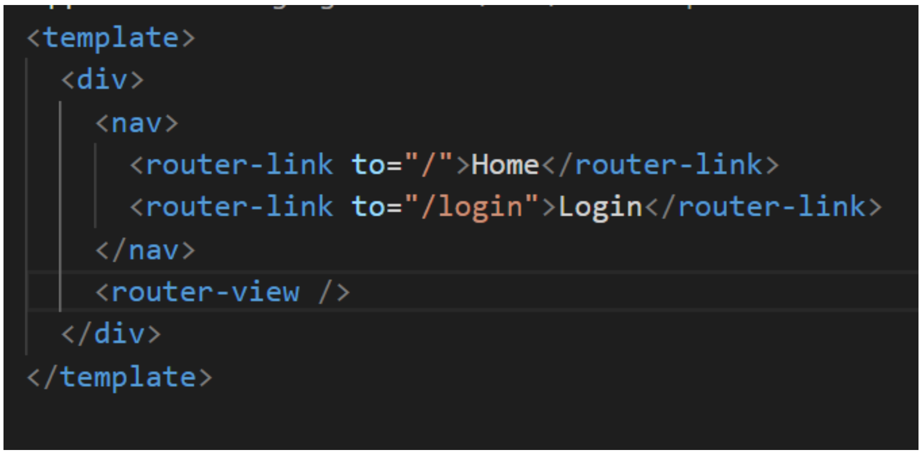
```
import { createRouter, createWebHistory } from 'vue-router';

import Home from "../Home.vue"
import Login from "../Login.vue"

const routes = [
    { path: "/", component: Home },
    { path: "/login", component: Login }
]

const router = createRouter({
    history: createWebHistory(),
    routes,
})

export default router;
```

4. In your main.js file, we now want to make sure that we are using the router that we have just created. Import it in, and then instruct your app to "use" it:



```
import { createApp } from 'vue'
import App from './App.vue'

import router from "./router"

createApp(App).use(router).mount('#app')
```

5. Finally, in your App component, you need to specify where on the screen the component will be visible. We can do this with the <router-view /> element. We can also navigate between components with the <router-link /> element as shown below:

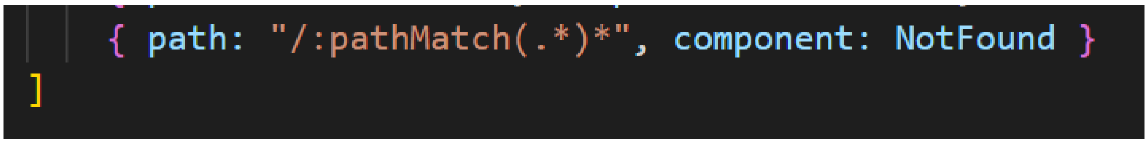


```
<template>
  <div>
    <nav>
      <router-link to="/">Home</router-link>
      <router-link to="/login">Login</router-link>
    </nav>
    <router-view />
  </div>
</template>
```

6. Try it out in your browser.

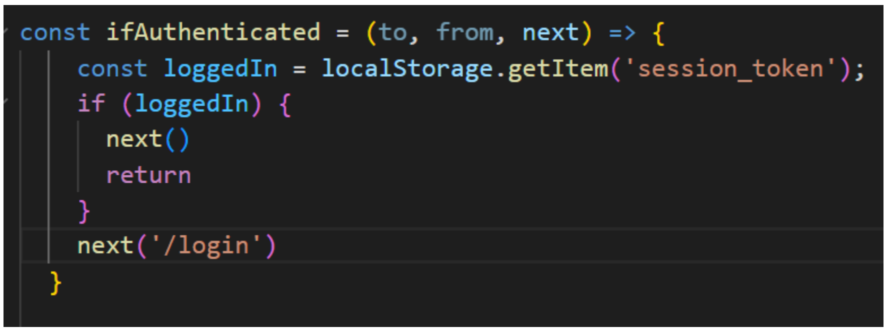
What else can we do with the router?

Now that we can implement a basic router, we can do some cool things with it. For example, if we want to add a default 404 page to our app, we can simply create the component, and then add the below to our array of routes. Routes are checked in order, so we need to make sure that it is the last element in the routes array:



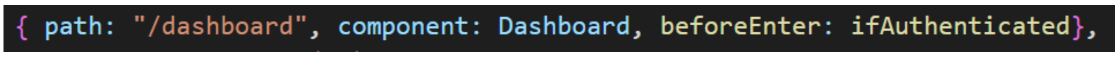
```
    { path: "/:pathMatch(.*)*", component: NotFound }
]
```

Also, we can handle authentication in the router. Imagine that a user has logged into your app, the session_token has been returned, and we have stored it in our localStorage (the browsers storage system). This function can then check if that token exists in localStorage, and if not it will redirect the user to the login path:



```
const ifAuthenticated = (to, from, next) => {
    const loggedIn = localStorage.getItem('session_token');
    if (loggedIn) {
      next()
      return
    }
    next('/login')
  }
```

Any routes that require you to be logged in can now include this check beforehand so that we only access pages that are allowed:



```
{ path: "/dashboard", component: Dashboard, beforeEnter: ifAuthenticated},
```

Perhaps not relevant to this weeks lab as you are just trying to get to grips with the basics this week, but this will definitely be something you want to come back to when you are working on your assignment router!

Exercise 7: Assignment

While you can't implement any API interactions following this lab, there is a lot that you can do on your assignment now:

1. Draw out your assignment front-end application on a piece of paper. What pages do you want in your app? How do they all fit together? Are you hitting every endpoint in an appropriate place?
2. Create your assignment application for the front end (Exercise 1)
3. Add a router to your app so that you can navigate between the homepage and the login (Exercise 6)
4. Build some reusable components, like for displaying comments, or articles (Exercise 5)
5. Create your forms and implement all of the validation on them (Exercise 3)