Intro to Authentication

In a modern web application, it's likely that you'll need to make parts of your app private so that not just anyone on the internet can access everything in your app. But if you want some users to be able to access those restricted parts, this requires implementing some form of user authentication to your app, so that your users can create an account, and later log back into that account, to access your protected content.

In Vue Mastery's Token-Based Authentication course, we walk through the process of creating a front-end authentication solution for a Vue.js app using JSON Web Tokens.

While there are many options for authenticating users in a Vue app, we've chosen to teach JWT because it offers a straightforward solution that doesn't rely on any specific third-party services. If you end up using a different option, many of the concepts we are teaching here will still be quite helpful when you need to implement authentication in your app.

How Token-Based Authentication works

Token-based authentication means that our app will allow users to log into it. But we can't log just anyone in. Our users need to be authenticated, which means when they type their username and password into our app, we'll send that info to our server so it can authenticate it. If everything is good to go, the user is logged in and the server returns a token.

We then store that token in our browser's local storage and use it when we need to, such as when we need to make an API call for some private data. At this point, we make a copy of the token we've stored, and send it along with our API request. The server then decrypts the token, making sure our user has the proper permission to access the data that's being requested. If approved, the server sends the requested private data to the client.

Whenever the user logs out, the token is cleared from local storage. We can also set an expiration on the token so it automatically clears after a set amount of time.

How a JWT is Structured

The specific type of tokens we'll be using are JSON Web Tokens (JWTs). You can think of a JWT like a key that can unlock the parts of your app that are private.

The structure of a JWT consists of three encoded strings: the header, payload and signature. ****Each hashed string is separated by a dot, like so: `xxxxx.yyyyy.zzzzz`.

The **header** is contains the *type* of token (in our case: JWT) and the hashing algorithm being used.

The **payload** contains the information we are transmitting (usually info about the user) along with some optional "claims" such as who issued the token, its expiration date, and if the user is an admin, etc.

The **signature** is a hash of the header + the payload + the secret. The secret lives on the server and is used to decrypt the token as well as to sign new tokens.

You can read more about the structure of a JWT here.

How we'll use JWT

In this course, we'll use JWT to build out a front-end authentication solution into a simple Vue app. This means we'll be creating an interface for users to create an account that they can log in and out of. Both actions of signing up and logging in will provide the user with a token.

When logged in, the user will be able to access a protected route, which is our dashboard. When the dashboard loads, it requests some private data from the server by making an API call that includes a copy of our JWT token.

Upon logging out, we'll make sure the token is cleared from local storage. Additionally, we'll learn how to handle authentication errors.

Learning Outcomes

We'll cover a lot in this course, but we won't be teaching how to build out the backend. There are many options for this, from Node.js to Rails or PHP, so we'll let you decide what is the best option for you or your team.

This course focuses on creating a front-end authentication solution that can be paired with your backend of choice. By the end of the course, you'll understand how to combine JWT with Vue Router, Vuex and Axios to craft a straightforward user authentication interface for your Vue app.

Project Structure

In this lesson we'll be looking at the starting code for the example app we'll be using for this course. We'll understand the steps that need to be taken to add authentication to it, which we'll do lesson by lesson throughout the course. We will be using Vue Router, Vuex, and Axios, so if you are not yet familiar with those topics, please take our Real World Vue and Mastering Vuex courses, then meet me back here.

Exploring the Starting Code

If you'd like to code along during this course, you can download the starting code in the Lesson Resources located on this page. Once you're ready, we'll start in the **main.js** file.

main.js

```
import Vue from 'vue'
import App from './App.vue'
import router from './router'
import store from './store'

Vue.config.productionTip = false

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

There's nothing new here, but note that we're using Vue Router and Vuex (`router` and `store`).

Now let's look at the **App.vue** file.

App.vue

```
<template>
  <div id="app">
    <app-nav />
    <router-view class="page" />
  </div>
</template>
<script>
import AppNav from './components/AppNav'
export default {
  components: { AppNav }
}
</script>
<style lang="scss">
@import './assets/styles/global.scss';
.page {
  display: flex;
  justify-content: center;
  flex-direction: column;
  align-items: center;
  min-height: calc(100vh - 56px);
}
</style>
```

In the `style` section you can see we're importing a global stylesheet from our **assets** directory and have imported the **AppNav** component, which we're using in the template.

Now let's look at the **AppNav** component.

src/components/AppNav.vue

```
<template>
  <div id="nav">
    <router-link to="/">
      Home
    </router-link>
    <router-link to="/dashboard">
      Dashboard
    </router-link>
  </div>
</template>

<script>
export default {}
</script>

<style lang="scss" scoped>
#nav {
  display: flex;
  align-items: center;
  min-height: 50px;
  padding: 0.2em 1em;
  background: linear-gradient(to right, #16c0b0, #84cf6a);
}
.nav-welcome {
  margin-left: auto;
  margin-right: 1rem;
  color: white;
}
a {
  font-weight: bold;
  color: #2c3e50;
  margin: auto 0.8em auto 0.4em;
  text-decoration: none;
  border-top: 2px solid transparent;
  border-bottom: 2px solid transparent;
}
.router-link-exact-active {
  color: white;
  border-bottom: 2px solid #fff;
}
button,
.button {
  margin-left: auto;
  background: white;
  text-decoration: none;
  color: #2c3e50;
  &.router-link-active {
    color: #2c3e50;
  }
}
.logoutButton {
  cursor: pointer;
}
.nav-welcome + button {
  margin-left: 0;
}
</style>
```

This file simply has two `router-links` to our **views**, and some scoped styles.

Also in the **components** directory we have the **EventCard.vue** file, which we've seen in previous courses.

src/components/EventCard.vue

```
<template>
  <div class="event-card">
    <span>@{{ event.time }} on {{ event.date }}</span>
    <h4>{{ event.title }}</h4>
  </div>
</template>
<script>
export default {
  name: 'EventCard',
  props: {
    event: {
      type: Object,
      default: () => {
        return {}
      }
    }
  }
}
</script>
<style scoped>
.event-card {
  width: 13em;
  margin: 1em auto 1em auto;
  padding: 1em;
  border: solid 1px #2c3e50;
  cursor: pointer;
  transition: all 0.2s linear;
}
.event-card:hover {
  transform: scale(1.01);
  box-shadow: 0 3px 12px 0 rgba(0, 0, 0, 0.2), 0 1px 15px 0 rgba(0, 0, 0,
0.19);
}
.event-card h4 {
  font-size: 1.4em;
  margin-top: 0.5em;
  margin-bottom: 0.3em;
}
</style>
```

This component expects an `event` prop, which it displays in its template.

Now let's head into our **views** directory, where we have the **Dashboard.vue** and **Home.vue** files.

src/views/Home.vue

```
<template>
  <div class="home">
    <h1>Welcome to the App!</h1>
  </div>
</template>

<script>
export default {}
</script>
```

The **Home** view is currently very simple, with just an `h1`. The **Dashboard** is where the action is happening.

src/views/Dashboard.vue

```
<template>
  <div>
    <h1>Dashboard</h1>
    <template v-if="!isLoading">
      <EventCard v-for="event in events" :key="event.id" :event="event"
/>
    </template>
    <p v-else>
      Loading events
    </p>
  </div>
</template>

<script>
import axios from 'axios'
import EventCard from '../components/EventCard'
export default {
  components: { EventCard },
  data () {
    return {
      isLoading: true,
      events: []
    }
  },
  created () {
    axios.get('//localhost:3000/dashboard').then(({ data }) => {
      this.events = data.events.events
      this.isLoading = false
    })
  }
}
</script>
```

Here, we are importing axios, and using it when the component is `created` to make a call out to our api, which returns a list of events. We then set the **Dashboard**'s component data equal to the response. We are also changing `isLoading` to `false`. We use the boolean value of `isLoading` in the template to determine whether to `v-for` through an **EventCard** for each event in our `events` data or to display a loading message instead.

If we look at our **router** file, we can see we're importing both **Home** and **Dashboard** and have a route for each of them, respectively.

router.js

```
import Vue from 'vue'
import Router from 'vue-router'
import Home from './views/Home.vue'
import Dashboard from './views/Dashboard.vue'
Vue.use(Router)
const router = new Router({
  mode: 'history',
  base: process.env.BASE_URL,
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home
    },
    {
      path: '/dashboard',
      name: 'dashboard',
      component: Dashboard
    }
  ]
})
export default router
```

Peeking into our **store** file, we'll see it's currently blank.

store.js

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

Vue.use(Vuex)

export default new Vuex.Store({
  state: {},
  mutations: {},
  actions: {}
})
```

We'll be adding to Vuex as we build out the app.

Also, notice how we have a **server.js** file that makes use of the two files in our **db** directory. Our api will get events from the **events.json**, and we'll use the **user.json** file to register and log in users. Please know that this server-side code is a simple solution meant only for this course. The backend code is not meant for a real production-level application. This course focuses on how to develop an Vue.js authentication *front-end*, which should work with your backend solution of choice.

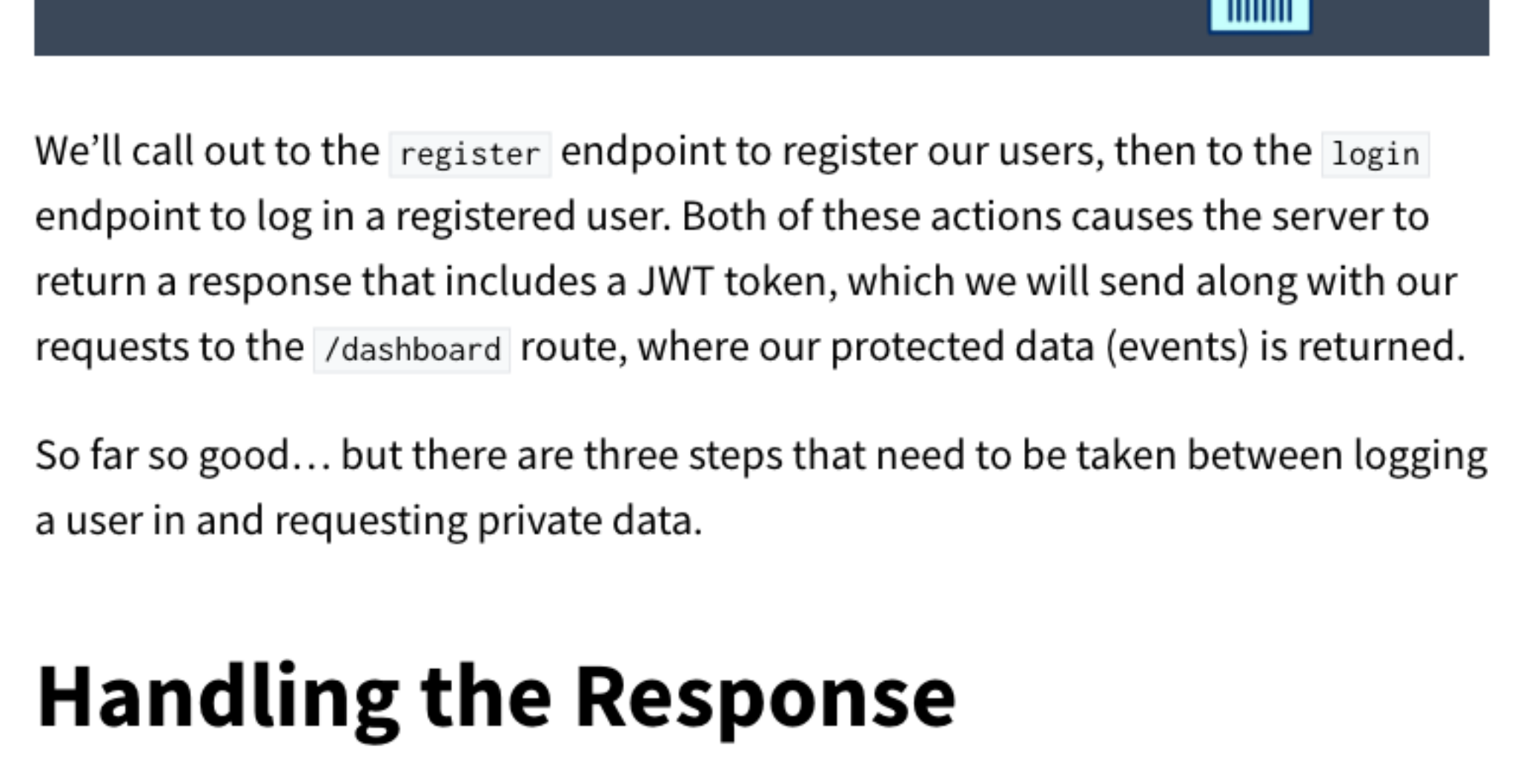
Finally, in the **package.json**, you'll see I've added a `start` script, which runs our server and builds our app. We'll type `npm run start` in the terminal to get our project up and running.

Understanding the Tasks Ahead

Now that we have explored the app, we can start adding authentication to it. But we need to take a step back and understand what steps we'll be taking to make that happen.

Client/Server Communication

We'll need to understand the communication that should happen between the client and our server. Our server has three api endpoints: `/register`, `/login`, and `/dashboard`.



We'll call out to the `register` endpoint to register our users, then to the `login` endpoint to log in a registered user. Both of these actions causes the server to return a response that includes a JWT token, which we will send along with our requests to the `/dashboard` route, where our protected data (events) is returned.

So far so good… but there are three steps that need to be taken between logging a user in and requesting private data.

Handling the Response

When a user registers or logs in, our server will return a response, which includes a JWT token along with the user's email and name.

We'll be using Vuex to do three things with that user data:

1. Store userData in Vuex State
2. Store userData in local storage (to persist it in case of browser refresh)
3. Add token to Axios header

We'll also need to be logging out our user, which will reverse these steps.

What's next?

Now that we have a foundational understanding of what token-based authentication is and the steps we'll take to implement it, we are ready to start building. In the next lesson, we'll add the ability to register users.