# Week 15 - User Authentication

This material can be found in this [li nk](https://codeberg.org/kaduardo/shu-sad-javascript/src/branch/main/week08-security/).

## 1. Introduction

In this lab we are going to add authentication to our petshop application. We will be using token-based authentication with JSON Web Tokens (JWT). A good explanation about JWT can be found in its oficial [documentation](https://jwt.io/introduction).

These instructions consider that you have finished with the lab exercises about model association and software testing, have a front-end with the technology of your choice and a back-end with mongoose models to start with. In these instructions we’ll use the sample solution from model associations as starting point. Make sure to adapt the instruction to reflect your own code.

Make sure your application is working before starting.

* Mongodb database pointing to the correct dbpath.
* Back-end ready to go - nodemon start (don’t forget to run npm install if you are using one of the sample solutions).
* Front-end ready to go (React or vue).

Check that both the front-end and back-end are working. We recommend the use of an API testing tool (e.g., Postman) to test your back-end.

## 2. Protecting the back-end

The first thing to do is to install the necessary modules in your back-end. We need to add modules for JWT (jswonwebtoken) and bcrypt (bcryptjs).

In the main folder of your back-end project run the following command:

npm install jsonwebtoken bcryptjs

### 2.1. Adding password to user model

We need to modify our User model to include a field for password. Edit the file models/user.model.js according with the example below:

module.exports = mongoose => {  
 var User = mongoose.model(  
 "user",  
 mongoose.Schema({  
 username: {  
 type: String,  
 required: true,  
 lowercase: true,  
 unique: true  
 },  
 password: {  
 type: String  
 },  
 animals: [{  
 type: mongoose.Schema.Types.ObjectId,  
 ref: "animal"  
 }],  
 dateCreated:{  
 type: Date,  
 required: true,  
 default: Date.now }  
   
 })  
  
 );  
 return User;  
};

## 2.2. Creating the middlewares

jsonwebtoken requires a secret key for functions like verify() and sign().

Create a new file inside the config folder named auth.config.js and define the secret.

module.exports = {  
 secret: "sad-supersecret-key"  
};

Now we can create our middleware functions. We’ll use functions to check for duplicated username during signup and to check if a given token is valid or not after signin.

Inside your main back-end folder create a new folder for saving your middleware functions. The first file is used to help with signups and is named verifySignUp.js.

Use the content below for your middlewares/verifySignUp.js file:

const db = require("../models");  
const User = db.users;  
  
checkDuplicateUsername = (req, res, next) => {  
 // Username  
 User.findOne({  
 username: req.body.username  
 }).exec((err, user) => {  
 if (err) {  
 res.status(500).send({ message: err });  
 return;  
 }  
  
 if (user) {  
 res.status(400).send({ message: "Failed! Username is already in use!" });  
 return;  
 }  
  
 next();  
  
 });  
};  
  
const verifySignUp = {  
 checkDuplicateUsername  
};  
  
module.exports = verifySignUp;

Next create the middleware to help with the authentication process. Name this file middlewares/authjwt.js and use the content below:

const jwt = require("jsonwebtoken");  
const config = require("../config/auth.config.js");  
const db = require("../models");  
const User = db.users;  
  
verifyToken = (req, res, next) => {  
 let token = req.headers["x-access-token"];  
  
 if (!token) {  
 return res.status(403).send({ message: "No token provided!" });  
 }  
  
 jwt.verify(token, config.secret, (err, decoded) => {  
 if (err) {  
 return res.status(401).send({ message: "Unauthorised!" });  
 }  
 req.userId = decoded.id;  
 next();  
 });  
};  
  
const authjwt = {  
 verifyToken  
};  
module.exports = authjwt;

Finally, create a middlewares/index.js file, so we can access our middleware later on.

const authjwt = require("./authjwt");  
const verifySignUp = require("./verifySignUp");  
  
module.exports = {  
 authjwt,  
 verifySignUp  
};

## 2.3. Creating authentication controllers

Our back-end will have two new functions for dealing with authentication: signup and signin. These will be created in a new controller file auth.controller.js. Save it inside your controllers folder:

const config = require("../config/auth.config");  
const db = require("../models");  
const User = db.users;  
  
var jwt = require("jsonwebtoken");  
var bcrypt = require("bcryptjs");  
  
exports.signup = (req, res) => {  
 const user = new User({  
 username: req.body.username,  
 password: bcrypt.hashSync(req.body.password, 8)  
 });  
  
 user  
 .save()  
 .then(data => {  
 console.log("Signup User saved in the database");  
 res.send({ message: "User was registered successfully!" });  
 })  
 .catch(err => {  
 res.status(500).send({   
 message: err || "Some error during signup"});  
 });  
};  
  
exports.signin = (req, res) => {  
 User.findOne({  
 username: req.body.username  
 })  
 .exec((err, user) => {  
 if (err) {  
 res.status(500).send({ message: err });  
 return;  
 }  
  
 if (!user) {  
 return res.status(404).send({ message: "User Not found." });  
 }  
  
 var passwordIsValid = bcrypt.compareSync(  
 req.body.password,  
 user.password  
 );  
  
 if (!passwordIsValid) {  
 return res.status(401).send({  
 accessToken: null,  
 message: "Invalid Password!"  
 });  
 }  
  
 var token = jwt.sign({ id: user.id }, config.secret, {  
 expiresIn: 86400 // 24 hours  
 });  
  
 res.status(200).send({  
 id: user.\_id,  
 username: user.username,   
 accessToken: token  
 });  
 });  
};

Notice how we include password hashing function in our controller.

## 2.4. Creating new routes for dealing with authentication

Our current implementation has routes for both animals (/petshop/animals) and users (/petshop/users). We’ll now create new routes for authentication:

* POST /petshop/auth/signup
* POST /petshop/auth/signin

Create a new file inside your routes directory: auth.routes.js.

const { verifySignUp } = require("../middlewares");  
const controller = require("../controllers/auth.controller");  
  
module.exports = function(app) {  
 app.use(function(req, res, next) {  
 res.header(  
 "Access-Control-Allow-Headers",  
 "x-access-token, Origin, Content-Type, Accept"  
 );  
 next();  
 });  
  
 app.post(  
 "/petshop/auth/signup",  
 [  
 verifySignUp.checkDuplicateUsername  
 ],  
 controller.signup  
 );  
  
 app.post("/petshop/auth/signin", controller.signin);  
};

## 2.5. Protecting resources

Now we will create new controllers and routes to demonstrate the use of our authentication middlewares. For this exercise we’ll use two new routes:

* /petshop/security/public for public access
* /petshop/security/protected for logged in users

We start by creating a new controller named security.controller.js with the following content:

exports.publicContent = (req, res) => {  
 res.status(200).send("Public Content.");  
};  
  
exports.protectedContent = (req, res) => {  
 res.status(200).send("User Protected Content.");  
};

Now we create the routes for our new test endpoints. Create a new file named routes/security.routes.js with the following content:

const { authjwt } = require("../middlewares");  
const controller = require("../controllers/security.controller");  
  
module.exports = function(app) {  
 app.use(function(req, res, next) {  
 res.header(  
 "Access-Control-Allow-Headers",  
 "x-access-token, Origin, Content-Type, Accept"  
 );  
 next();  
 });  
  
 app.get("/petshop/security/public", controller.publicContent);  
  
 app.get("/petshop/security/protected", [authjwt.verifyToken], controller.protectedContent);  
  
};

Notice how the middleware is used in the /petshop/security/protected endpoint.

Finally, we add these routes to the main app.js file. Make sure you include those after the main app object has been created.

...  
// new routes for authentication  
require('./routes/auth.routes')(app);  
require('./routes/security.routes')(app);  
...

## 2.6. Testing your back-end

Now let’s test our back-end to make sure everything is working before proceeding to the front-end. These instructions assume you’re using Postman to run your tests. Make sure your database and back-end are running.

### Register some users

The first step is to register some users using our new /petshop/auth/signup endpoint.

Here are two users that can be used for testing.

//User #01  
{  
 "username": "user01",  
 "password": "pass01"  
}  
  
//User #02  
{  
 "username": "user02",  
 "password": "pass02"  
}

You have to create a POST request to the http://localhost:3050/petshop/auth/signup endpoint passing a user as the request body.

Check if the user has been created in your database.

### Access protected resources (with negative result)

In order to see the effects of protecting an endpoint we are going to perform two GET requests. One to a non-protected endpoint (/petshop/security/public) and one to a protected one (/petshop/security/protected).

You should receive an http 200 response from the public endpoint. The protected endpoint should respond with an http 403 and the message “No token provided!”.

### Login an account

Now let’s try to login with one of our users to recover a token.

This must be a POST request to the http://localhost:3050/petshop/auth/signin endpoint with a User object as the body.

Start by providing the wrong password

{  
 "username": "user01",  
 "password": "wrongpass01"  
}

You should receive a response with code 401, a null token and a message of “invalid password”.

Now create a new POST request to the same endpoint but this time passing in the correct password.

This time you should receive an http 200 response with a json object similar to the one bellow:

{  
 "id": "61ee84de10ff52dfeed036db",  
 "username": "user01",  
 "accessToken": "  
 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYzNjNkMjIzOGJmYThiYjYzMTM1MGRmMSIsImlhdCI6MTY2NzQ4NjMyNywiZXhwIjoxNjY3NTcyNzI3fQ.2GezNT8zAMq3GRdgMkCbliXOLhIXvWCrDGjbx0fgGcw"  
}

Notice how you have access to the user id, username and access token.

### Access protected resources (with valid token)

Now that you have a valid token you must use it in all your http requests to protected resources.

Create a new GET request to the protected resource http://localhost:3050/petshop/security/protected including the x-access-token header using the received access token as value. Based on the example from above your http request to the protected endpoint must include the following http header:

x-access-token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYzNjNkMjIzOGJmYThiYjYzMTM1MGRmMSIsImlhdCI6MTY2NzQ4NjMyNywiZXhwIjoxNjY3NTcyNzI3fQ.2GezNT8zAMq3GRdgMkCbliXOLhIXvWCrDGjbx0fgGcw

**Remember to replace the actual token with the response you got from the signin call**.

Now you should receive a http 200 response with User Protected Content..

You can also try with a wrong token to see what kind of response you get. Create a new request modify the token value. You should receive a http 401 response with a Unauthorised message.

A Postman Collection has been saved inside the solutions folder of the repository. The collection also include the automatic extraction of the token from the correct signin call.

## 3. Protecting the front-end

Now that you have a back-end protected by authentication using JWT it is time to modify your front-end to make use of this new functionality. We’ll be using vuex to store the access token.

These instructions use the solutions from model association exercise as starting point. Remember to adapt the codes and examples to your particular implementation.

### 3.1. Install npm modules

The first step is to install all the necessary modules in your front-end. Type the following commands from the main directory of your Vue project.

npm update  
npm install vue-router@4  
npm install vuex@4  
npm install vee-validate@4 yup  
npm install axios  
npm install bootstrap@4 jquery popper.js  
npm install @fortawesome/fontawesome-svg-core @fortawesome/free-solid-svg-icons @fortawesome/vue-fontawesome@prerelease

### 3.2. Create Services

We will add three new services to the src/services directory.

The authentication service provides methods for login, logout and register. Create a new file named auth.service.js with the following content:

import axios from 'axios';  
  
const API\_URL = 'http://localhost:3050/petshop/auth/';  
  
class AuthService {  
 login(user) {  
 return axios  
 .post(API\_URL + 'signin', {  
 username: user.username,  
 password: user.password  
 })  
 .then(response => {  
 if (response.data.accessToken) {  
 localStorage.setItem('user', JSON.stringify(response.data));  
 }  
  
 return response.data;  
 });  
 }  
  
 logout() {  
 localStorage.removeItem('user');  
 }  
  
 register(user) {  
 return axios.post(API\_URL + 'signup', {  
 username: user.username,  
 password: user.password  
 });  
 }  
}  
  
export default new AuthService();

Now let’s create a helper function to deal with the authorisation header. Inside the src/services folder create a new file named auth-header.js with the following content:

export default function authHeader() {  
 let user = JSON.parse(localStorage.getItem('user'));  
  
 if (user && user.accessToken) {  
 return { 'x-access-token': user.accessToken };  
 } else {  
 return {};  
 }  
}

Now we can create a service for accessing our public and protected endpoints. Inside the src/services folder create a new file named SecurityDataService.js with the following content:

import axios from 'axios';  
import authHeader from './auth-header';  
  
const API\_URL = 'http://localhost:3050/petshop/security/';  
  
class SecurityDataService {  
 getPublicContent() {  
 return axios.get(API\_URL + 'public');  
 }  
  
 getProtectedContent() {  
 return axios.get(API\_URL + 'protected',   
 { headers: authHeader() });  
 }  
  
}  
  
export default new SecurityDataService();

Notice how we can add the necessary http header to the request by using the authHeader() function to access the protected resource.

### 3.3. Create the local store

We’ll be using Vuex to deal with authentication and token storage. Inside the src folder create a new folder named store.

We need two files in this folder: the main authentication module (auth.module.js) and an index.js file that will import the main module.

Use the following content for the src/store/auth.module.js file:

import AuthService from '../services/auth.service';  
  
const user = JSON.parse(localStorage.getItem('user'));  
const initialState = user  
 ? { status: { loggedIn: true }, user }  
 : { status: { loggedIn: false }, user: null };  
  
export const auth = {  
 namespaced: true,  
 state: initialState,  
 actions: {  
 login({ commit }, user) {  
 return AuthService.login(user).then(  
 user => {  
 commit('loginSuccess', user);  
 return Promise.resolve(user);  
 },  
 error => {  
 commit('loginFailure');  
 return Promise.reject(error);  
 }  
 );  
 },  
 logout({ commit }) {  
 AuthService.logout();  
 commit('logout');  
 },  
 register({ commit }, user) {  
 return AuthService.register(user).then(  
 response => {  
 commit('registerSuccess');  
 return Promise.resolve(response.data);  
 },  
 error => {  
 commit('registerFailure');  
 return Promise.reject(error);  
 }  
 );  
 }  
 },  
 mutations: {  
 loginSuccess(state, user) {  
 state.status.loggedIn = true;  
 state.user = user;  
 },  
 loginFailure(state) {  
 state.status.loggedIn = false;  
 state.user = null;  
 },  
 logout(state) {  
 state.status.loggedIn = false;  
 state.user = null;  
 },  
 registerSuccess(state) {  
 state.status.loggedIn = false;  
 },  
 registerFailure(state) {  
 state.status.loggedIn = false;  
 }  
 }  
};

Now create the src/store/index.js file with the content below:

import { createStore } from 'vuex';  
import { auth } from './auth.module';  
  
const store = createStore({  
 modules: {  
 auth  
 }  
});  
  
export default store;

### 3.4. Create Login page

Now we can create our Vue pages. Inside the src/components folder create a file named LoginPage.vue for our Login page. Use the content below for this file:

<template>  
 <div class="col-md-12">  
 <div class="card card-container">  
 <img  
 id="profile-img"  
 src="//ssl.gstatic.com/accounts/ui/avatar\_2x.png"  
 class="profile-img-card"  
 />  
 <Form @submit="handleLogin" :validation-schema="schema">  
 <div class="form-group">  
 <label for="username">Username</label>  
 <Field name="username" type="text" class="form-control" />  
 <ErrorMessage name="username" class="error-feedback" />  
 </div>  
 <div class="form-group">  
 <label for="password">Password</label>  
 <Field name="password" type="password" class="form-control" />  
 <ErrorMessage name="password" class="error-feedback" />  
 </div>  
  
 <div class="form-group">  
 <button class="btn btn-primary btn-block" :disabled="loading">  
 <span  
 v-show="loading"  
 class="spinner-border spinner-border-sm"  
 ></span>  
 <span>Login</span>  
 </button>  
 </div>  
  
 <div class="form-group">  
 <div v-if="message" class="alert alert-danger" role="alert">  
 {{ message }}  
 </div>  
 </div>  
 </Form>  
 </div>  
 </div>  
</template>  
  
<script>  
import { Form, Field, ErrorMessage } from "vee-validate";  
import \* as yup from "yup";  
  
export default {  
 name: "Login",  
 components: {  
 Form,  
 Field,  
 ErrorMessage,  
 },  
 data() {  
 const schema = yup.object().shape({  
 username: yup.string().required("Username is required!"),  
 password: yup.string().required("Password is required!"),  
 });  
  
 return {  
 loading: false,  
 message: "",  
 schema,  
 };  
 },  
 computed: {  
 loggedIn() {  
 return this.$store.state.auth.status.loggedIn;  
 },  
 },  
 created() {  
 if (this.loggedIn) {  
 this.$router.push("/profile");  
 }  
 },  
 methods: {  
 handleLogin(user) {  
 this.loading = true;  
  
 this.$store.dispatch("auth/login", user).then(  
 () => {  
 this.$router.push("/profile");  
 },  
 (error) => {  
 this.loading = false;  
 this.message =  
 (error.response &&  
 error.response.data &&  
 error.response.data.message) ||  
 error.message ||  
 error.toString();  
 }  
 );  
 },  
 },  
};  
</script>  
  
<style scoped>  
label {  
 display: block;  
 margin-top: 10px;  
}  
  
.card-container.card {  
 max-width: 350px !important;  
 padding: 40px 40px;  
}  
  
.card {  
 background-color: #f7f7f7;  
 padding: 20px 25px 30px;  
 margin: 0 auto 25px;  
 margin-top: 50px;  
 -moz-border-radius: 2px;  
 -webkit-border-radius: 2px;  
 border-radius: 2px;  
 -moz-box-shadow: 0px 2px 2px rgba(0, 0, 0, 0.3);  
 -webkit-box-shadow: 0px 2px 2px rgba(0, 0, 0, 0.3);  
 box-shadow: 0px 2px 2px rgba(0, 0, 0, 0.3);  
}  
  
.profile-img-card {  
 width: 96px;  
 height: 96px;  
 margin: 0 auto 10px;  
 display: block;  
 -moz-border-radius: 50%;  
 -webkit-border-radius: 50%;  
 border-radius: 50%;  
}  
  
.error-feedback {  
 color: red;  
}  
</style>

### 3.5. Create Register page

Next we create our registration page. Create a new file inside components named RegisterPage.vue with the following content:

<template>  
 <div class="col-md-12">  
 <div class="card card-container">  
 <img  
 id="profile-img"  
 src="//ssl.gstatic.com/accounts/ui/avatar\_2x.png"  
 class="profile-img-card"  
 />  
 <Form @submit="handleRegister" :validation-schema="schema">  
 <div v-if="!successful">  
 <div class="form-group">  
 <label for="username">Username</label>  
 <Field name="username" type="text" class="form-control" />  
 <ErrorMessage name="username" class="error-feedback" />  
 </div>  
 <div class="form-group">  
 <label for="email">Email</label>  
 <Field name="email" type="email" class="form-control" />  
 <ErrorMessage name="email" class="error-feedback" />  
 </div>  
 <div class="form-group">  
 <label for="password">Password</label>  
 <Field name="password" type="password" class="form-control" />  
 <ErrorMessage name="password" class="error-feedback" />  
 </div>  
  
 <div class="form-group">  
 <button class="btn btn-primary btn-block" :disabled="loading">  
 <span  
 v-show="loading"  
 class="spinner-border spinner-border-sm"  
 ></span>  
 Sign Up  
 </button>  
 </div>  
 </div>  
 </Form>  
  
 <div  
 v-if="message"  
 class="alert"  
 :class="successful ? 'alert-success' : 'alert-danger'"  
 >  
 {{ message }}  
 </div>  
 </div>  
 </div>  
</template>  
  
<script>  
import { Form, Field, ErrorMessage } from "vee-validate";  
import \* as yup from "yup";  
  
export default {  
 name: "Register",  
 components: {  
 Form,  
 Field,  
 ErrorMessage,  
 },  
 data() {  
 const schema = yup.object().shape({  
 username: yup  
 .string()  
 .required("Username is required!")  
 .min(3, "Must be at least 3 characters!")  
 .max(20, "Must be maximum 20 characters!"),  
 email: yup  
 .string()  
 .required("Email is required!")  
 .email("Email is invalid!")  
 .max(50, "Must be maximum 50 characters!"),  
 password: yup  
 .string()  
 .required("Password is required!")  
 .min(6, "Must be at least 6 characters!")  
 .max(40, "Must be maximum 40 characters!"),  
 });  
  
 return {  
 successful: false,  
 loading: false,  
 message: "",  
 schema,  
 };  
 },  
 computed: {  
 loggedIn() {  
 return this.$store.state.auth.status.loggedIn;  
 },  
 },  
 mounted() {  
 if (this.loggedIn) {  
 this.$router.push("/profile");  
 }  
 },  
 methods: {  
 handleRegister(user) {  
 this.message = "";  
 this.successful = false;  
 this.loading = true;  
  
 this.$store.dispatch("auth/register", user).then(  
 (data) => {  
 this.message = data.message;  
 this.successful = true;  
 this.loading = false;  
 },  
 (error) => {  
 this.message =  
 (error.response &&  
 error.response.data &&  
 error.response.data.message) ||  
 error.message ||  
 error.toString();  
 this.successful = false;  
 this.loading = false;  
 }  
 );  
 },  
 },  
};  
</script>  
  
<style scoped>  
label {  
 display: block;  
 margin-top: 10px;  
}  
  
.card-container.card {  
 max-width: 350px !important;  
 padding: 40px 40px;  
}  
  
.card {  
 background-color: #f7f7f7;  
 padding: 20px 25px 30px;  
 margin: 0 auto 25px;  
 margin-top: 50px;  
 -moz-border-radius: 2px;  
 -webkit-border-radius: 2px;  
 border-radius: 2px;  
 -moz-box-shadow: 0px 2px 2px rgba(0, 0, 0, 0.3);  
 -webkit-box-shadow: 0px 2px 2px rgba(0, 0, 0, 0.3);  
 box-shadow: 0px 2px 2px rgba(0, 0, 0, 0.3);  
}  
  
.profile-img-card {  
 width: 96px;  
 height: 96px;  
 margin: 0 auto 10px;  
 display: block;  
 -moz-border-radius: 50%;  
 -webkit-border-radius: 50%;  
 border-radius: 50%;  
}  
  
.error-feedback {  
 color: red;  
}  
</style>

### 3.6. Create Profile page

We also need a page to show details about the current logged user. Create a ProfilePage.vue file inside src/components folder.

<template>  
 <div class="container">  
 <header class="jumbotron">  
 <h3>  
 <strong>{{currentUser.username}}</strong> Profile  
 </h3>  
 </header>  
 <p>  
 <strong>Token:</strong>  
 {{currentUser.accessToken.substring(0, 20)}} ... {{currentUser.accessToken.substr(currentUser.accessToken.length - 20)}}  
 </p>  
 <p>  
 <strong>Id:</strong>  
 {{currentUser.id}}  
 </p>  
 </div>  
</template>  
  
<script>  
export default {  
 name: 'ProfilePage',  
 computed: {  
 currentUser() {  
 return this.$store.state.auth.user;  
 }  
 },  
 mounted() {  
 if (!this.currentUser) {  
 this.$router.push('/login');  
 }  
 }  
};  
</script>

### 3.7. Create Public page

Now we can create Vue components for accessing public and protected resources. We start with a page to access unprotected resources.

Create a new file named PublicPage.vue insider your src/components folder.

<template>  
 <div class="container">  
 <header class="jumbotron">  
 <h3>{{content}}</h3>  
 </header>  
 </div>  
</template>  
  
<script>  
import SecurityDataService from '../services/SecurityDataService';  
  
export default {  
 name: 'PublicPage',  
 data() {  
 return {  
 content: ''  
 };  
 },  
 mounted() {  
 SecurityDataService.getPublicContent().then(  
 response => {  
 this.content = response.data;  
 },  
 error => {  
 this.content =  
 (error.response &&   
 error.response.data &&  
 error.response.data.message ) ||  
 error.message ||  
 error.toString();  
 }  
 );  
 }  
};  
</script>

### 3.8. Create Protected page

Finally, we have a page to show protected resources. Create a new file named ProtectedContent.vue inside your src/components folder.

<template>  
 <div class="container">  
 <header class="jumbotron">  
 <h3>{{content}}</h3>  
 </header>  
 </div>  
</template>  
  
<script>  
import SecurityDataService from '../services/SecurityDataService';  
  
export default {  
 name: 'ProtectedContent',  
 data() {  
 return {  
 content: ''  
 };  
 },  
 mounted() {  
 SecurityDataService.getProtectedContent().then(  
 response => {  
 this.content = response.data;  
 },  
 error => {  
 this.content =  
 (error.response &&   
 error.response.data &&  
 error.response.data.message ) ||  
 error.message ||  
 error.toString();  
 }  
 );  
 }  
};  
</script>

### 3.9. Define routes

We now put it all together by defining the routes for our Vue front-end. Our current front-end contains routes for managing animals and users. We need to add routes for the new components we created.

Add calls to import the new components and then use these new components in your router.js.

//Existing import commands  
//...  
  
//Existing routes  
//...  
  
//new routes  
 {  
 path: '/public',  
 name: 'public-page',  
 component: () => import("./components/PublicPage")  
 },  
 {  
 path: '/login',  
 name: 'login',  
 component: () => import("./components/LoginPage")  
 },  
 {  
 path: '/register',  
 name: 'register',  
 component: () => import("./components/RegisterPage")  
 },  
 {  
 path: '/profile',  
 name: 'profile',  
 component: () => import("./components/ProfilePage")  
 },  
 {  
 path: '/protected',  
 name: 'protected',  
 component: () => import("./components/ProtectedContent")  
 }  
//...

### 3.10. Add Navigation Bar

Now that the routes have been defined we can use them to update our navigation bar inside the main App.vue file.

Include new navbar items inside the navbar div:

//Existing code  
//...  
//New navbar items  
 <li class="nav-item">  
 <router-link to="/public" class="nav-link">  
 <font-awesome-icon icon="home" />Public  
 </router-link>  
 </li>  
 <li v-if="protectedContent" class="nav-item">  
 <router-link to="/protected" class="nav-link">Protected</router-link>  
 </li>  
//end new navbar items

We also need to create links for the registration and login pages. These will only appear if the user is not already logged into the system. Still in the App.vue file, create a new <div> inside the <nav> tags:

//Existing code for navbar (inside <nav> tag)  
//...  
 <div v-if="!currentUser" class="navbar-nav ml-auto">  
 <li class="nav-item">  
 <router-link to="/register" class="nav-link">  
 <font-awesome-icon icon="user-plus" />Sign Up  
 </router-link>  
 </li>  
 <li class="nav-item">  
 <router-link to="/login" class="nav-link">  
 <font-awesome-icon icon="sign-in-alt" />Login  
 </router-link>  
 </li>  
 </div>  
  
 <div v-if="currentUser" class="navbar-nav ml-auto">  
 <li class="nav-item">  
 <router-link to="/profile" class="nav-link">  
 <font-awesome-icon icon="user" />  
 {{ currentUser.username }}  
 </router-link>  
 </li>  
 <li class="nav-item">  
 <a class="nav-link" href @click.prevent="logOut">  
 <font-awesome-icon icon="sign-out-alt" />LogOut  
 </a>  
 </li>  
 </div>  
 </nav> //closing the nav tag  
//...

We also need to modify the <script> tag with code for handling the access token. Use the following code as reference for that:

<script>  
export default {  
 name: 'App',  
 computed: {  
 currentUser() {  
 return this.$store.state.auth.user;  
 },  
 protectedContent() {  
 return (this.currentUser?true:false);   
 }  
 },  
 methods: {  
 logOut() {  
 this.$store.dispatch('auth/logout');  
 this.$router.push('/login');  
 }  
 }  
};  
</script>

Noticed how we use the protectedContent() function to check whether the user is logged in.

### 3.11. All together now

Modify main.js to include all the new elements.

import Vue from 'vue';  
import App from './App.vue';  
import router from './router';  
import store from './store';  
import 'bootstrap';  
import 'bootstrap/dist/css/bootstrap.min.css';  
import VeeValidate from 'vee-validate';  
import { library } from '@fortawesome/fontawesome-svg-core';  
import { FontAwesomeIcon } from '@fortawesome/vue-fontawesome';  
import {  
 faHome,  
 faUser,  
 faUserPlus,  
 faSignInAlt,  
 faSignOutAlt  
} from '@fortawesome/free-solid-svg-icons';  
  
library.add(faHome, faUser, faUserPlus, faSignInAlt, faSignOutAlt);  
  
Vue.config.productionTip = false;  
  
Vue.use(VeeValidate);  
Vue.component('font-awesome-icon', FontAwesomeIcon);  
  
new Vue({  
 router,  
 store,  
 render: h => h(App)  
}).$mount('#app');

Save everything and have a go with your new front-end application. You should be able to replicate the tests performed with Postman:

* Signup with new users;
* Access public resource;
* Try to access a protected resource without authentication;
* Try to sign in with a user using the wrong password;
* Sign in with a user using the correct password;
* Access a protected resource and the profile page for the current user;
* Access the existing pages (e.g., List Animals).

### 3.12. Deal with unauthorised access

You can modify your code to always verify for authentication every time a navigation action is performed. This can be done by adding a router.beforeEach() function in your src/router.js:

router.beforeEach((to, from, next) => {  
 const publicPages = ['/login', '/register', '/public'];  
 const authRequired = !publicPages.includes(to.path);  
 const loggedIn = localStorage.getItem('user');  
  
 // trying to access a restricted page + not logged in  
 // redirect to login page  
 if (authRequired && !loggedIn) {  
 next('/login');  
 } else {  
 next();  
 }  
});

Repeat all the tests from the previous step. Also, what happens when you try to access some of the pages we have created in our previous exercises, e.g., list all animals in the database?

## 4. Exercises

Based on the references [1, 2] below, try to include different roles for users. - vet role allows manipulation of animals but not of users; - manager role allows full CRUD management of both Users and Animals.

You have to plan your approach. Start by considering what changes are needed in the back-end and front-end before actual code implementation.

## References and extra reading

Main references:

* [1] Bezkoder - Node.js + MongoDB: User Authentication & Authorization with JWT - <https://www.bezkoder.com/node-js-mongodb-auth-jwt/>
* [2] Bezkoder - Vue 3 Authentication with JWT, Vuex, Axios and Vue Router - <https://www.bezkoder.com/vue-3-authentication-jwt/>

Extra material:

* Bezkoder - React + Node.js Express: User Authentication with JWT example - <https://www.bezkoder.com/react-express-authentication-jwt/>
* Bezkoder - JWT tutorial: In-depth Introduction to JSON Web Token - <https://www.bezkoder.com/jwt-json-web-token/>