



VUE - Login, Rutas privadas, Rutas protegidas, mostrar nombre.

Instalando las dependencias necesarias

Para la autenticación de una persona usuaria en la aplicación necesitaremos instalar JWT y JWT-Decode:

- `npm i --save jwt-decode`
- `npm install jsonwebtoken`



Creando la autenticación

Siempre asumiremos que tienes una carpeta llamada "api" en src, donde tendrás supuestamente tu archivo "api.js" donde has creado todos tus endpoints, conexión a la bbdd, puerto de la API y demás.

Dentro de la carpeta `api`, crea un nuevo archivo llamado `config.js` y escribe la siguiente línea:



```
module.exports = {  
  llave: 'llavesecreta'  
}
```

Esta es la "clave secreta" que se añade al token de seguridad cuando un usuario/a hace login.

A continuación, dirígete al archivo `api.js`, donde has declarado todas las constantes del cors, body-parser y demás, y añade las dos dependencias que has instalado anteriormente:



```
// TOKEN  
const config = require('./config')  
  
const jwt = require('jsonwebtoken')
```

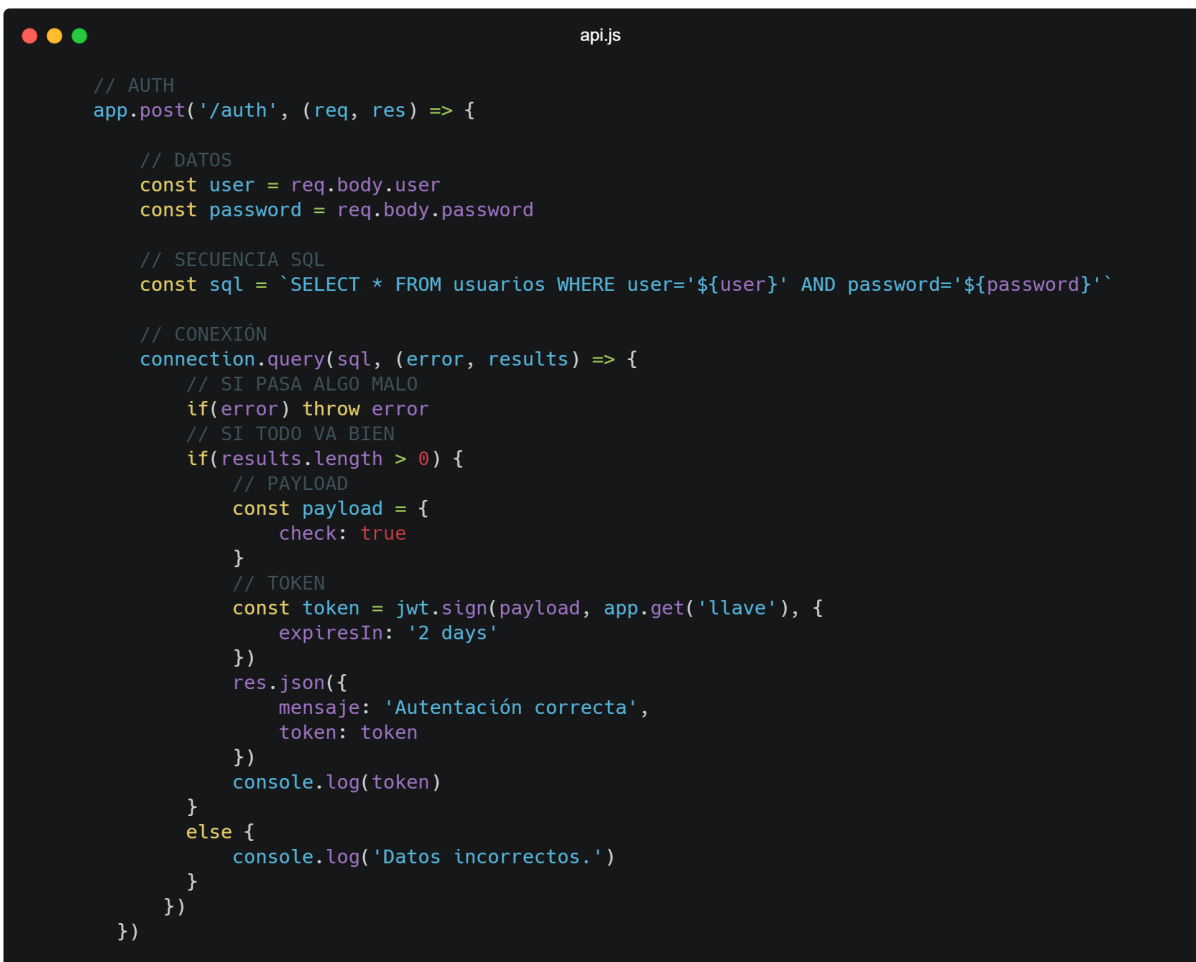
Más abajo, donde indicaste los `app.use()blabla`, añade también la siguiente línea:



```
api.js

app.set('llave', config.llave)
```

A continuación, crea la función que recoge el user y el password desde la vista y las envía a la base de datos para ver si existe. A esta función también añadirás el jsonwebtoken:



```
api.js

// AUTH
app.post('/auth', (req, res) => {

  // DATOS
  const user = req.body.user
  const password = req.body.password

  // SECUENCIA SQL
  const sql = `SELECT * FROM usuarios WHERE user='${user}' AND password='${password}'`

  // CONEXIÓN
  connection.query(sql, (error, results) => {
    // SI PASA ALGO MALO
    if(error) throw error
    // SI TODO VA BIEN
    if(results.length > 0) {
      // PAYLOAD
      const payload = {
        check: true
      }
      // TOKEN
      const token = jwt.sign(payload, app.get('llave'), {
        expiresIn: '2 days'
      })
      res.json({
        mensaje: 'Autenticación correcta',
        token: token
      })
      console.log(token)
    }
    else {
      console.log('Datos incorrectos.')
    }
  })
})
})
```

Ahora que la función de la API ya está creada, es necesario crear una función que permita a la vista comunicarse con la API.

Crea un nuevo archivo dentro de la carpeta api llamado `utils.js`. Este archivo nos servirá para guardar las funciones más "pesadas" del tema de autenticación y protección de rutas para tenerlas concentradas en un mismo archivo y no desperdigadas por el proyecto.

En este archivo `utils.js`, primero importa la dependencia del `jwt-decode` y de `axios`:



```
import jwt from 'jwt-decode'  
import axios from 'axios'
```

Seguidamente, crea una const de endpoint para no tener que escribir la dirección de autenticación de la API una y otra vez:



```
const ENDPOINT = 'http://localhost:3050'
```

Y a continuación, crea la función que se comunicará desde la vista con la función de la API (es un post de AXIOS normal y corriente, que envía el user y password del usuario/a a la API):



```
utils.js

export function login(user, password) {
  try {
    axios.post(`${ENDPOINT}/auth`, {
      user: user,
      password: password
    })
    .then(function(response){
      console.log(response)
    })
  } catch(error) {
    console.log('Error: ' + error)
  }
}
```

Le colocamos "export function" delante del nombre porque necesitamos exportarla para ir usándola en las vistas que veamos convenientes.

Finalmente dirígete a tu vista de Login.vue, donde deberías tener ya tu formulario de login con un input para el nombre de usuario y otro input para el password, junto a un botón de "Login".

Dentro del `Login.vue` crea un objeto `data()` como siempre, y declara 2 variables: **user** y **password**, que serán strings.

```
Login.vue

data() {
  return {
    user: '',
    password: ''
  }
}
```

Enlaza estas variables a tus inputs de forma correspondiente con un v-model:

```
<input v-model="user" placeholder="Your user">
<input v-model="password" placeholder="Your password">
<button>
  Login
</button>
```

Ahora que nuestras variables ya están enlazadas a los inputs, es momento de importar nuestra función de login desde `utils.js`, para eso la importaremos tal cual, como si se tratase de cualquier otro componente o módulo:

```
Login.vue

import { login } from '../api/utils';
```

Ahora sólo queda crear una función en la vista que haga alguna comprobación (por ejemplo, en caso de que los campos estén vacíos), y si no lo están, llame a la función de login. Esta función ha de llamarse diferente de la función principal de `login()`, por ejemplo, `loginUser()`:

```
Login.vue

methods: {
  // FUNCIÓN DE LOGIN DE LA VISTA
  loginUser(){
    // COMPROBANDO QUE NO HAY CAMPOS VACÍOS
    if(this.user !== '' && this.password !== '') {
      // LLAMANDO AL LOGIN DE UTILS.JS
      login(this.user, this.password)
      // ENVIANDO AL USER LOGUEADO A LA HOME
      this.$router.push('/home')
    }
    else {
      alert('No puedes dejar campos vacíos.')
    }
  }
}
```

Finalmente, enlazamos esta función al botón de Login en el HTML con un `@click`:

```
Login.vue

<input v-model="user" placeholder="Your user">
<input v-model="password" placeholder="Your password">
<button @click="loginUser()">
  Login
</button>
```

Y ahora, ¡intenta hacer login!

Si introduces datos incorrectos y aún así te lleva al home, no te preocupes. Aún no hemos cubierto esa fase.

Guardando el token en local storage

Para securizar y proteger las rutas de nuestro proyecto, es necesario guardar el token.

Crearemos una función llamada `setAuthToken` en el archivo de `utils.js` para hacer esto.

```
utils.js

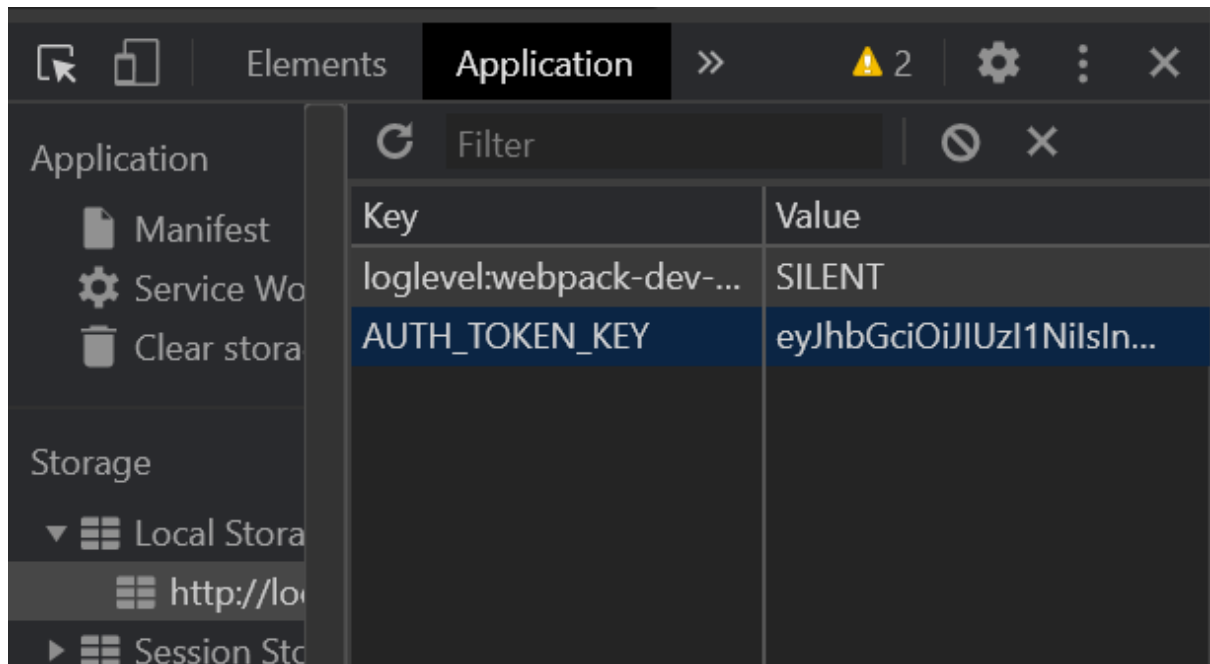
export function setAuthToken(token) {
  axios.defaults.headers.common['Authorization'] = `Bearer ${token}`
  localStorage.setItem('AUTH_TOKEN_KEY', token)
}
```


Llamaremos a esta función dentro de nuestra función principal de login(), también situada en `utils.js`.

A screenshot of a code editor window titled 'utils.js'. The code is written in JavaScript and defines an export function 'login' that takes 'user' and 'password' as arguments. It uses 'axios.post' to send a request to an endpoint defined by a variable '{ENDPOINT}'. The request body contains the user and password. The response is logged to the console, and the token is stored in local storage using 'setAuthToken'. Error handling is implemented with a 'catch' block that logs and alerts the error, then returns it.

```
export function login(user, password){
  try {
    axios.post(`${ENDPOINT}/auth`, {
      user: user,
      password: password
    })
      .then(function(response){
        console.log(response)
        // GUARDANDO EL TOKEN EN LOCALSTORAGE
        setAuthToken(response.data.token)
      })
  } catch(err){
    console.log('Error: ' + err)
    alert('Error: ' + err)
    return err;
  }
}
```

Prueba a hacer login. Una vez lo hayas hecho correctamente, inspecciona la página, métete en la pestañita de Application y fíjate que debería haberse guardado tu token ahí:



Haciendo logout

Para hacer logout es necesario borrar el token del usuario/a. Para ello, crearemos una función de logout en `utils.js`, la llamaremos `clearToken()`:

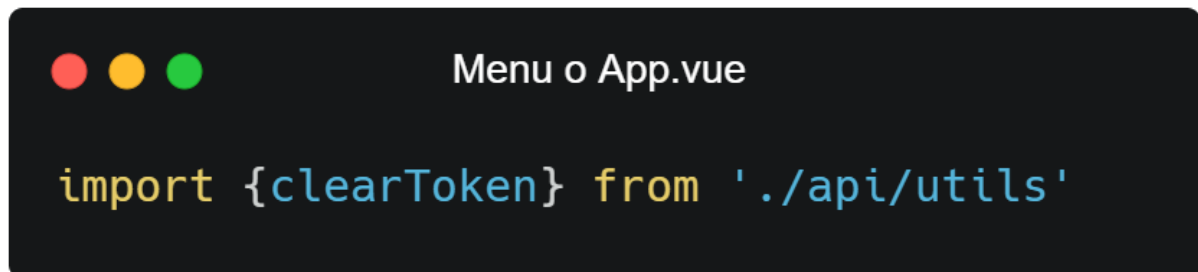
```
utils.js

export function clearToken() {
  axios.defaults.headers.common['Authorization'] = ''
  localStorage.removeItem('AUTH_TOKEN_KEY')
}
```

Dirígete al componente menú, o donde sea que ahora mismo está tu menú.

Crea un botón de logout en tu HTML.

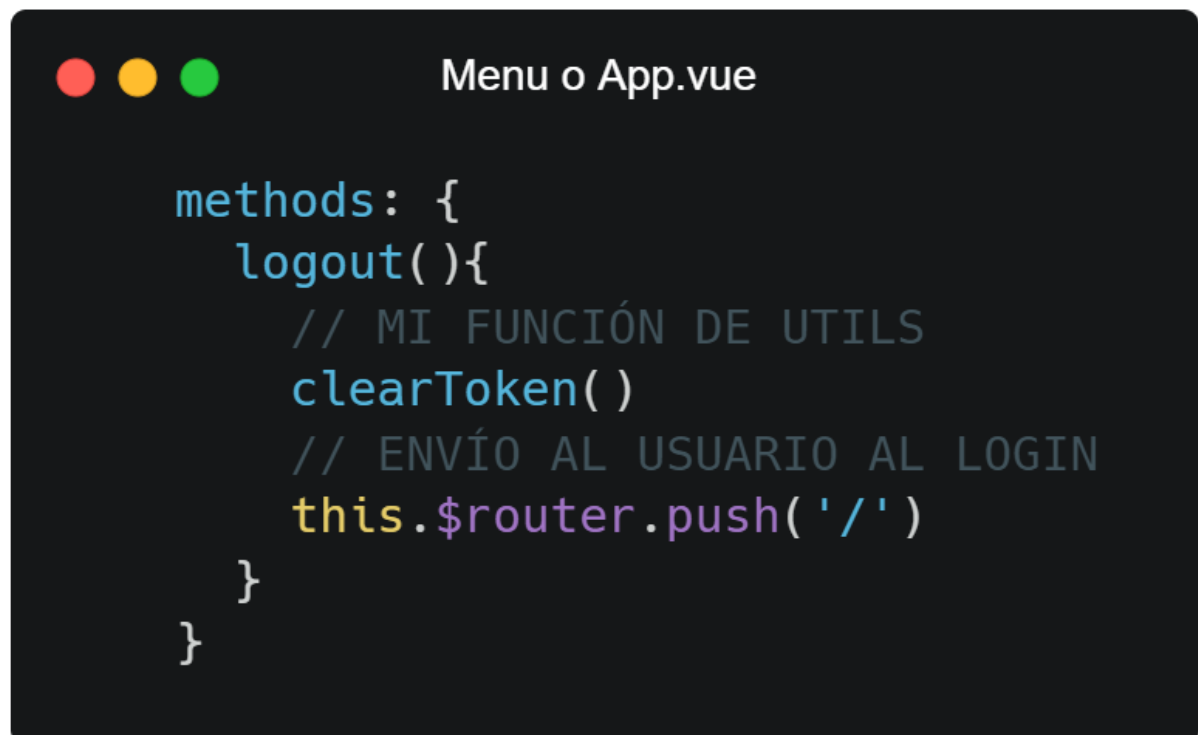
Es momento de importar nuestra función `clearToken()` desde `utils` en nuestro menú:



```
import {clearToken} from './api/utils'
```

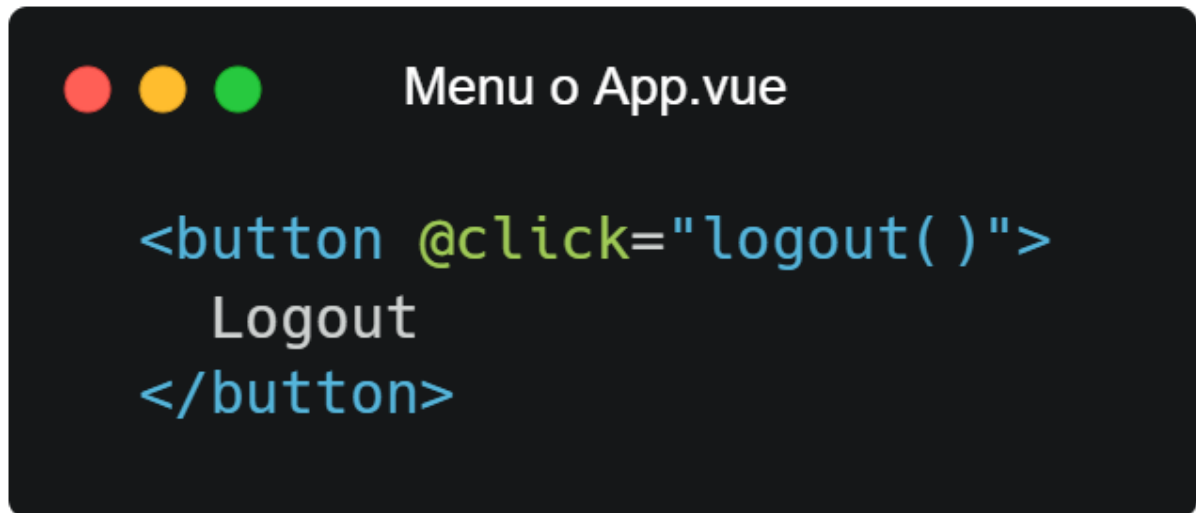
Recuerda revisar si la ruta (en este caso `./api/utils`) es correcta, puede no serlo en tu proyecto.

A continuación, crea tu función `logout()` en los `methods`:



```
methods: {  
  logout(){  
    // MI FUNCIÓN DE UTILS  
    clearToken()  
    // ENVÍO AL USUARIO AL LOGIN  
    this.$router.push('/')  
  }  
}
```

Y, finalmente, enlace esta función con el botón de Logout que he creado en el HTML:



```
<button @click="logout()">
  Logout
</button>
```

Ahora comprueba que si al clicar Logout, tu token se borra de localStorage y la función te lleva de vuelta al login.

Protegiendo rutas

Protegeremos rutas de la siguiente manera:

1. Recuperaremos el token del usuario.
2. Recuperaremos la fecha del token y la encodearemos.
3. Comprobaremos que no esté expirado (caducado).
4. Finalmente, comprobaremos que el usuario está logueado.

Primero, creemos en `utils.js` la función para recuperar el token desde el `localStorage`:

```
utils.js

export function getAuthToken() {
  return localStorage.getItem('AUTH_TOKEN_KEY')
}
```

Luego, crearemos una función para conseguir la fecha de caducidad del token:

```
utils.js

export function tokenExpiration(encodedToken) {
  let token = jwt(encodedToken)
  if(!token.exp){
    return null
  }
  let date = new Date(0)
  date.setUTCSeconds(token.exp)
  return date
}
```

Ahora crearemos una función que recibe la fecha del token y comprueba con la fecha actual si la fecha del token está caducada o en vigor:

```
utils.js

export function isExpired(token){
  let expirationDate = tokenExpiration(token)
  return expirationDate < new Date()
}
```

Finalmente, crearemos una función que compruebe si un usuario está logueado comprobando si tiene token y si este está caducado:

```
utils.js

export function isLoggedIn(){
  let authToken = getAuthToken()
  return !!authToken && !isExpired(authToken)
}
```

Ahora nos dirigiremos al `index.js` del router. Vamos a proteger finalmente las rutas.

El router de Vue tiene 2 funcionalidades muy interesantes: 1. permite ejecutar funciones y comprobaciones justo antes de que la persona acceda a la URL, y 2. se pueden proteger rutas con el objeto `meta`.

Primero marcaremos cada ruta según queramos que sea privada o pública. Por ejemplo imaginemos que tenemos algunas rutas privadas, como: Clientes, Productos y Registro de clientes. Y otras rutas públicas como: Home y About.

Las definiremos como públicas o privadas dentro del objeto meta, con una variable que llamaremos "*allowAnon*" (permitir anónimos). Si vale true, cualquier persona anónima no logueada podrá ver la página; si vale false, sólo personas logueadas podrán verla. Esta variable podría llamarse de cualquier otra manera, es sólo un boolean.

Ejemplo:

```
index.js (router)

{
  path: '/home',
  name: 'Home',
  component: Home,
  meta: {
    // RUTA HABILITADA SÓLO PARA USUARIOS LOGED
    allowAnon: false
  }
},
{
  path: '/',
  name: 'login',
  component: () => import('../views/Login.vue'),
  meta: {
    // RUTA HABILITADA PARA ANÓNIMOS Y LOGUEADOS
    allowAnon: true
  }
}
}
```

Ahora, importaremos desde `utils.js` nuestra función `isLoggedIn()`, ya que la necesitaremos para ejecutarla justo antes de que la persona acceda a cada URL y comprobar si la persona está logueada y si su token es válido.



index.js (router)

```
import {isLoggedIn} from '../api/utils.js'
```

Finalmente, usaremos un nav-guard llamado `beforeEach`, lo que hace este navguard dentro del router es ejecutarse antes de que accedamos a la ruta, por lo tanto es el sitio perfecto para comprobar si la persona que intenta entrar a la ruta está logueada o no, o si su token está caducado o no lo está.

Justo antes del export router, crearemos este nav-guard, donde comprobaremos qué clase de ruta es la ruta a la que intentamos acceder (si es privada o pública), y si la persona está logueada o no.



```
router.beforeEach( (to, from, next) => {  
  if(!to.meta.allowAnon && !isLoggedIn()) {  
    next ( {  
      path: '/',  
      query: { redirect: to.fullPath }  
    } )  
  } else {  
    next()  
  }  
})
```




Proteger rutas únicamente para usuarios

Es momento de proteger rutas.


Lo normal en un proyecto web, sobre todo en webs sobre gestión y fuentes información interna de la empresa, es que haya dos tipos de usuarios a nivel protección: usuarios y administradores.

Recuerda para gestionar los roles debes habilitar un campo en tu base de datos para hacer las comprobaciones. En este caso crearás un campo llamado "isAdmin" en la base de datos, y valdrá 0 si no es admin, y 1 si es admin.

Vamos a recuperar ese dato desde el proyecto y vamos a gestionarlo para ir protegiendo rutas.


Primero, dirígete directamente a tu archivo `api.js`, ya que este dato debes guardarlo directamente desde el login.

Dentro de la función de login, justo antes de configurar el token añadiremos lo siguiente:



```
// RECOGIENDO Y GUARDANDO EL VALOR DE ADMIN
let admin = null
if(results[0].isAdmin === 1) {
    admin = true
} else {
    admin = false
}
```

Y dentro del res.json, también en la función de login, añadiré una línea más para enviar el valor del rol de administrador en la respuesta json:



```
res.json({
  mensaje: 'Autenticación correcta',
  token: token,
  admin: admin
})
```

La función completa de login, ahora quedaría así:

```

app.post('/auth', (req, res) => {

  // DATOS
  const user = req.body.user
  const password = req.body.password

  // SECUENCIA SQL
  const sql = `SELECT * FROM usuarios WHERE user='${user}' AND password='${password}'`

  // CONEXIÓN
  connection.query(sql, (error, results) => {
    // SI PASA ALGO MALO
    if(error) throw error
    // SI TODO VA BIEN
    if(results.length > 0) {
      // PAYLOAD
      const payload = {
        check: true
      }
      // RECOGIENDO Y GUARDANDO EL VALOR DE ADMIN
      let admin = null
      if(results[0].isAdmin === 1) {
        admin = true
      } else {
        admin = false
      }
      // TOKEN
      const token = jwt.sign(payload, app.get('llave'), {
        expiresIn: '2 days'
      })
      res.json({
        mensaje: 'Autenticación correcta',
        token: token,
        admin: admin
      })
      console.log(token)
      return res;
    }
    else {
      console.log('Datos incorrectos.')
    }
  })
})
})

```

Dirígete ahora a `utils.js`, donde vamos a crear varias funciones que gestionen y comprueben si el usuario logueado es admin o no.

Primero, necesitamos guardar el estado de administrador del usuario en el `localStorage`. Con la siguiente función lo conseguiremos:



```
// GUARDO EL ROL EN LOCALSTORAGE
export function setIsAdmin(admin) {
  localStorage.setItem('ROLE', admin)
}
```

A continuación, crearemos una función para recoger ese estado del localStorage. Esto lo utilizaremos cuando tengamos que comprobar si la persona tiene un rol u otro.



```
// RECUPERO EL ROL DEL LOCALSTORAGE
export function getIsAdmin(){
  return localStorage.getItem('ROLE')
}
```

Y, finalmente, crearé una función para comprobar el rol y devolverlo:

```
utils.js

// COMPROBAR SI EL USUARIO LOGUEADO ES ADMIN O NO
export function checkIsAdmin() {

    let role = null
    // RECOJO EL ESTADO DE ADMIN
    let admin = getIsAdmin()

    // COMPRUEBO EL ESTADO DE ADMIN
    if(admin === 'true') {
        // SI ES ADMIN, ES TRUE
        role = true
    } else {
        // SI NO ES ADMIN, ES FALSE
        role = false
    }

    // DEVUELVE TRUE O FALSE
    return role
}
```

Esta es por ahora la lógica que necesitamos, pero ahora debemos colocarla donde toca para que cada función haga lo que debe.

Para guardar el rol de admin que nos llega desde la API, aún en `utils.js` dirígete a la función de login y llama ahí la función que guarda el rol del usuario en localStorage, llamada aquí `setIsAdmin()`.

```
utils.js / login()

.then(function(response) {
  console.log(response)
  // GUARDANDO EL TOKEN EN LOCALSTORAGE
  setAuthToken(response.data.token)
  // GUARDANDO EL ROL EN LOCALSTORAGE
  setIsAdmin(response.data.admin)
})
```

Dirígete ahora al `index.js` del router. Importa desde `utils.js` la función de `checkIsAdmin()` para poder llamarla cada vez que la persona entra a una ruta protegida para admins.

```
index.js

import {checkIsAdmin} from '../api/utils.js'
```

Anteriormente usaste la etiqueta meta para prohibir algunas rutas para quienes no están logueados, ahora también indicaremos qué rutas son únicamente para admins. Utilizaremos un nuevo boolean llamado 'onlyAdmin'. Si 'onlyAdmin' es true, esa página sólo podrán verla administradores.

Por ahora sólo marcaremos la ruta `/addclients` como privada para admins:

```
index.js

{
  path: '/addclients',
  name: 'AddClients',
  component: () => import('../views/AddClients.vue'),
  meta: {
    allowAnon: false,
    // PÁGINA SOLO PARA ADMINS
    onlyAdmin: true
  }
}
```

Dentro de la propia ruta añadiremos el nav-guard `beforeEnter`, que incide en el acceso de la ruta antes de que se complete. Con esto, la propia ruta comprobará cada vez que su URL se intenta acceder, si la persona es admin o no:

```
index.js

meta: {
  allowAnon: false,
  onlyAdmin: true
},
beforeEnter: (to, from, next) => {
  if(to.meta.onlyAdmin === true && !checkIsAdmin()){
    next({
      path: '/home',
      query: { redirect: to.fullPath }
    })
  }
  else {
    next()
  }
}
```

Arreglando el error del navigation guard

Para arreglar el error del navigation guard, colocaremos un setTimeout al router-push para que la petición tarde un momento en llegar al router.

```
Login.vue (función loginUser)

loginUser(){
  if(this.user === '' || this.password === '') {
    alert('Te faltan datos.')
  } else {
    login(this.user, this.password)
  }
  // AÑADIMOS UN SET TIME OUT PARA EL PUSH ROUTER
  // Y PARA HACER UN RELOAD DE LA PANTALLA.
  setTimeout(() => {
    this.$router.push('/home')
    location.reload()
  }, 1000);
}
```

Guardando el nombre de usuario

Necesitaremos guardar el nombre de la persona cuando nos venga de la bbdd, en una variable. Para ello, en la función de la llamada de la API a /auth, añadiremos, justo debajo de donde guardamos si la persona es admin o no:


```
api.js

//GUARDANDO EL NOMBRE
let user = null
user = results[0].user
```

También lo añadimos en la respuesta del res.json:

```
api.js

res.json({
  mensaje: 'Autenticación completada con éxito',
  token: token,
  admin: admin,
  user: user // AÑADIMOS EL USER
})
```

**en este ejemplo se tiene en cuenta que el nombre de usuario en la bbdd está guardado en el campo "user".*

Luego, en `utils.js`, crearemos 2 funciones adicionales: una para guardar el nombre en localStorage, y otra para recoger el nombre guardado en localStorage.



utils.js

```
// FUNCIÓN PARA GUARDAR NOMBRE DEL USER EN  
LOCALSTORAGE  
export function setName(user) {  
  localStorage.setItem('NAME', user)  
}  
  
// FUNCIÓN PARA RECUPERAR EL NOMBRE DE  
LOCALSTORAGE  
export function getName() {  
  return localStorage.getItem('NAME')  
}
```

Todavía en `utils.js`, nos falta añadir la función que acabamos de crear, `setName`, en el login, para que guarde, junto al token y al rol del usuario, el nombre:



utils.js

```
// GUARDO EL NOMBRE  
setName(response.data.user)
```

Y añadiremos el `removeItem` del `localStorage` en la función de `logout`, para que al hacer `logout` se borren el token, el rol y el nombre de usuario:

```
utils.js

// FUNCIÓN DE LOGOUT
export function logout(){
  axios.defaults.headers.common['Authorization'] = ''
  localStorage.removeItem('AUTH_TOKEN_KEY')
  localStorage.removeItem('ROLE')
  localStorage.removeItem('NAME')
}
```

A continuación, iremos al componente del Menú o al `App.vue`, según dónde tengas tu menú, y primero importaremos la función que recoge el nombre del `localStorage`:

```
Menu o App.vue

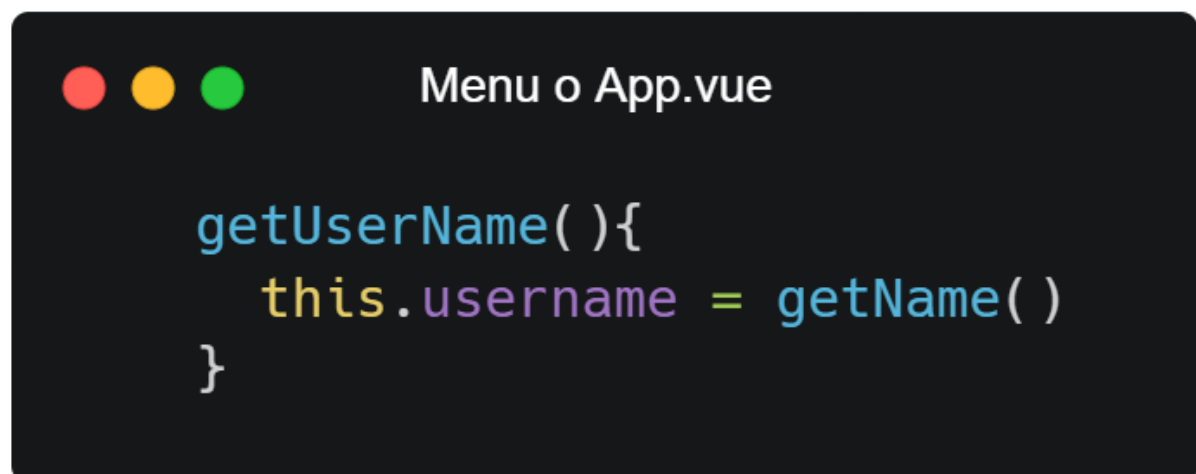
import { getName } from '../api/utils'
```

Luego en `data`, crearemos la variable `username`.



```
data( ){  
  return {  
    username: ''  
  }  
}
```

Finalmente en el objeto `methods`, crearemos una función que guarde el nombre del localStorage dentro de nuestra variable `username`:



```
getUserUsername( ){  
  this.username = getName( )  
}
```

Y llamaremos a esta función dentro del hook `created`, justo después del objeto `methods`:


```
Menu o App.vue

created( ){
  this.getUserName( )
}
```

Finalmente, interpolaremos dicha variable en el menú:

```
Menú o App.vue

<p>
  Hola, {{username}}
</p>
```

 **Haciendo que algunos links del menú se vean o no se vean según si la persona está logueada o no**

Primero, iremos de nuevo a nuestro menú o App.vue (o donde sea que tienes tu menú). Crea una variable llamada `logged` en `data()`, valiendo `false`.

```
data(){  
  return {  
    logged: false  
  }  
}
```

A continuación, importa la función `isLoggedIn()` desde `utils`:

```
import { isLoggedIn } from './api/utils'
```

En el HTML, indica con un `v-show="logged"` aquellas rutas que se podrán ver únicamente si estás logueado.

Por ejemplo:

```
Menu o App.vue

<!-- ESTA CON V-SHOW, NO SE VERÁ SI NO ESTÁS LOGUEADO -->
<router-link v-show="logged" :to="{ name: 'AddClients' }">
  Formulario
</router-link>
<!-- ESTA SIN V-SHOW, SIEMPRE SERÁ VISIBLE -->
<router-link :to="{ name: 'Login' }">
  Login
</router-link>
```

Dentro de `methods`, crea una función que iguale la variable `logged` al valor que salga de la función `isLoggedIn`:

```
Menu o App.vue

getLogin(){
  this.logged = isLoggedIn()
}
```

Y finalmente, llama a esta función dentro del hook `created`.



Menu o App.vue

```
created( ){  
  this.getLogin( )  
}
```