Vue Authentication And Route Handling Using Vue-router

scotch.io/tutorials/vue-authentication-and-route-handling-using-vue-router

Vue is a progressive Javascript framework that makes building frontend applications easy. Coupled with vue-router, we can build high performance applications with complete dynamic routes. Vue-router is an efficient tool and can handle authentication in our Vue application seamlessly. In this tutorial, we will look at using vue-router to handle authentication and access control for different parts of our application.

Getting Started

To begin, install Vue cli and create a vue application with it:

```
$ npm install -g @vue/cli
$ npm install -g @vue/cli-init
$ vue init webpack vue-router-auth
```

Follow the setup prompt and complete the installation of this application. If you are not sure of an option, simply click the return key (enter key) to continue with the default option. When asked to install vue-router, accept the option, because we need vue-router for this application.

Setup Node.js Server

Next, we will setup a Node.js server that would handle authentication for us. For our Node.js server, we will use SQLite as the database of choice. Run the following command to install SQLite driver:

```
$ npm install --save sqlite3
```

Because we are dealing with passwords, we need a way to hash passwords. We will use bcrypt to hash all our passwords. Run the following command to install it:

```
$ npm install --save bcrypt
```

We also want a way to confirm the user's we authenticate when they try to make a request to a secured part of our application. For this, we will use JWT. Run the following command to install the JWT package we will use:

```
$ npm install jsonwebtoken --save
```

To read `json` data we will send to our server, we need `body-parser`. Run the following command to install it:

```
$ npm install --save body-parser
```

Now that it's all set, let us create a simple nodejs server that would handle user authentication. Create a new directory named **server**. This is where we will store everything we will use to make our node backend. In the server directory, create a file and save it as

`app.js` . Add the following to it:

Essential Reading: Learn React from Scratch! (2019 Edition)

```
"use strict";
const express = require('express');
const DB = require('./db');
const config = require('./config');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const bodyParser = require('body-parser');

const db = new DB("sqlitedb")
const app = express();
const router = express.Router();

router.use(bodyParser.urlencoded({ extended: false }));
router.use(bodyParser.json());
```

We have required all the packages we need for our application, defined the database, created an express server and an express router.

Now, let's define CORS middleware, to ensure we do not run into any cross origin resource errors:

```
const allowCrossDomain = function(req, res, next) {
    res.header('Access-Control-Allow-Origin', '*');
    res.header('Access-Control-Allow-Methods', '*');
    res.header('Access-Control-Allow-Headers', '*');
    next();
}

app.use(allowCrossDomain)
```

Many people would use a CORS package here, but we do not have any complicated configurations, so this is fine.

Let's define the route for registering a new user:

```
router.post('/register', function(req, res) {
    db.insert([
        req.body.name,
        req.body.email,
        bcrypt.hashSync(req.body.password, 8)
    ],
    function (err) {
        if (err) return res.status(500).send("There was a problem registering the user.")
        db.selectByEmail(req.body.email, (err,user) => {
            if (err) return res.status(500).send("There was a problem getting user")
            let token = jwt.sign({ id: user.id }, config.secret, {expiresIn: 86400
            });
            res.status(200).send({ auth: true, token: token, user: user });
        });
    });
```

```
});
```

There are a few things happening here. First, we pass the request body to a database method (which we will define later), and pass a callback function that to handle the response from the database operation. As expected, we have defined error checks to ensure we provide accurate information ot users.

When a user is successfully registered, we select the user data by email and create an authentication token for the user with the `jwt` package we had imported earlier. We use a secret key in our config file (which we will create later) to sign the auth credentials. This way, we can verify a token sent to our server and a user cannot fake an identity.

Now, define the route for registering an administrator and logging in, which are similar to register:

```
router.post('/register-admin', function(req, res) {
    db.insertAdmin([
        req.body.name,
        req.body.email,
        bcrypt.hashSync(req.body.password, 8),
        1
    ],
    function (err) {
        if (err) return res.status(500).send("There was a problem registering the user.")
        db.selectByEmail(req.body.email, (err,user) => {
            if (err) return res.status(500).send("There was a problem getting user")
            let token = jwt.sign({ id: user.id }, config.secret, { expiresIn: 86400
            });
            res.status(200).send({ auth: true, token: token, user: user });
        });
    });
});

router.post('/login', (req, res) => {
    db.selectByEmail(req.body.email, (err, user) => {
        if (err) return res.status(500).send('Error on the server.');
        if (!user) return res.status(404).send('No user found.');
        let passwordIsValid = bcrypt.compareSync(req.body.password, user.user_pass);
        if (!passwordIsValid) return res.status(401).send({ auth: false, token: null });
        let token = jwt.sign({ id: user.id }, config.secret, { expiresIn: 86400
        });
        res.status(200).send({ auth: true, token: token, user: user });
    });
```

```
})
```

For login, we use bcrypt to compare our hashed password with the user supplied password. If they are the same, we log the user in. If not, well, feel free to respond to the user how you please.

Now, let's use the express server to make our application accessible:

```
app.use(router)

let port = process.env.PORT || 3000;

let server = app.listen(port, function() {
   console.log('Express server listening on port ' + port)
});
```

We created a server on `port: 3000` or any dynamically generated port by our system (heroku generates dynamic ports).

> If you find any of the above confusing, you can take an introductory course on Node.js

Then, create another file `config.js` in the same directory and add the following to it:

```
module.exports = {
   'secret': 'supersecret'
};
```

Finally, create another file `db.js` and add the following to it:

```
"use strict";
const sqlite3 = require('sqlite3').verbose();

class Db {
    constructor(file) {
        this.db = new sqlite3.Database(file);
        this.createTable()
    }

    createTable() {
        const sql = `
            CREATE TABLE IF NOT EXISTS user (
                id integer PRIMARY KEY,
                name text,
                email text UNIQUE,
                user_pass text,
                is_admin integer)`
        return this.db.run(sql);
    }

    selectByEmail(email, callback) {
        return this.db.get(
            `SELECT * FROM user WHERE email = ?`,
            [email],function(err,row){
                callback(err,row)
            })
    }

    insertAdmin(user, callback) {
        return this.db.run(
            'INSERT INTO user (name,email,user_pass,is_admin) VALUES (?,?,?,?)',
            user, (err) => {
                callback(err)
            })
    }

    selectAll(callback) {
        return this.db.all(`SELECT * FROM user`, function(err,rows){
```

```
            callback(err,rows)
        })
    }

    insert(user, callback) {
        return this.db.run(
            'INSERT INTO user (name,email,user_pass) VALUES (?,?,?)',
            user, (err) => {
                callback(err)
            })
    }
}

module.exports = Db
```

We created a class for our database to abstract the basic functions we need. You may want to use more generic and reusable methods here for database operations and likely use a promise to make it more efficient. This will allow you have a repository you can use with all other classes you define (especially if your application uses MVC architecture and has controllers).

All done and looking good, let's make the vue application now.

Updating The Vue-router File

The vue-router file can be found in `./src/router/` directory. In the `index.js` file, we will define all the routes we want our application to have. This is different from what we did with our server and should not be confused.

Open the file and add the following:

```
import Vue from 'vue'
import Router from 'vue-router'
import HelloWorld from '@/components/HelloWorld'
import Login from '@/components/Login'
import Register from '@/components/Register'
import UserBoard from '@/components/UserBoard'
import Admin from '@/components/Admin'

Vue.use(Router)
```

We have imported all the components our application will use. We will create the components below.

Now, let's define the routes for our application:

```
let router = new Router({
    mode: 'history',
    routes: [
        {
            path: '/',
            name: 'HelloWorld',
            component: HelloWorld
        },
        {
            path: '/login',
            name: 'login',
            component: Login,
            meta: {
                guest: true
            }
        },
        {
            path: '/register',
            name: 'register',
            component: Register,
            meta: {
                guest: true
            }
        },
        {
            path: '/dashboard',
            name: 'userboard',
            component: UserBoard,
            meta: {
                requiresAuth: true
            }
        },
        {
            path: '/admin',
```

```
        name: 'admin',
        component: Admin,
        meta: {
          requiresAuth: true,
          is_admin : true
        }
      },
  ]
})
```

Vue router allows use define a meta on our routes so we can specify additional behaviour. In our case above, we have defined some routes as guest (which means only users not authenticated should see it), some to require authentication (which means only authenticated users should see it) and the last one to be only accessible to admin users.

Now, let's handle requests to these routes based on the meta specification:

```
router.beforeEach((to, from, next) => {
    if(to.matched.some(record => record.meta.requiresAuth)) {
        if (localStorage.getItem('jwt') == null) {
            next({
                path: '/login',
                params: { nextUrl: to.fullPath }
            })
        } else {
            let user = JSON.parse(localStorage.getItem('user'))
            if(to.matched.some(record => record.meta.is_admin)) {
                if(user.is_admin == 1){
                    next()
                }
                else{
                    next({ name: 'userboard'})
                }
            }else {
                next()
            }
        }
    } else if(to.matched.some(record => record.meta.guest)) {
```

```
      if(localStorage.getItem('jwt') == null){
        next()
      }
      else{
        next({ name: 'userboard'})
      }
    }else {
      next()
    }
})

export default router
```

Vue-router has a beforeEach method that is called before each route is processed. This is where we can define our checking condition and restrict user access. The method takes three parameters — to, from and next. `to` is where the user wishes to go, `from` is where the user is coming from, `next` is a callback function that continues the processing of the user request. Our check is on the `to` object.

We check a few things:

- if route `requiresAuth`, check for a `jwt` token showing the user is logged in.
- if route `requiresAuth` and is only for admin users, check for auth and check if the user is an admin
- if route requires `guest`, check if the user is logged in

We redirect the user based on what we are checking for. We use the name of the route to redirect, so check to be sure you are using this for your application.

> **IMPORTANT:** Always ensure you have `next()` called at the end of every condition you are checking. This is to prevent your application from failing in the event that there is a condition you forgot to check.

Define Some Components

To test out what we have built, let's define a few components. In the `./src/components/` directory, open the `HelloWorld.vue` file and add the following:

```
<template>
  <div class="hello">
    <h1>This is homepage</h1>
    <h2>{{msg}}</h2>
  </div>
</template>

<script>
  export default {
    data () {
      return {
        msg: 'Hello World!'
      }
    }
  }
</script>
  <!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>
  h1, h2 {
    font-weight: normal;
  }
  ul {
    list-style-type: none;
    padding: 0;
  }
  li {
    display: inline-block;
    margin: 0 10px;
  }
  a {
    color: #42b983;
  }
</style>
```

Create a new file `Login.vue` in the same directory and add the following:

```
<template>
   <div>
      <h4>Login</h4>
      <form>
         <label for="email" >E-Mail Address</label>
```

```
        <div>
          <input id="email" type="email" v-model="email" required autofocus>
        </div>
        <div>
          <label for="password" >Password</label>
          <div>
            <input id="password" type="password" v-model="password" required>
          </div>
        </div>
        <div>
          <button type="submit" @click="handleSubmit">
            Login
          </button>
        </div>
      </form>
    </div>
</template>
```

That is for the HTML template. Now, let's define the script handling login:

```
<script>
   export default {
      data(){
         return {
            email : "",
            password : ""
         }
      },
      methods : {
         handleSubmit(e){
            e.preventDefault()
            if (this.password.length > 0) {
               this.$http.post('http://localhost:3000/login', {
                  email: this.email,
                  password: this.password
               })
               .then(response => {

               })
               .catch(function (error) {
                  console.error(error.response);
               });
            }
         }
      }
   }
</script>
```

At this point, we have the `email` and `password` data attributes bound to the form fields to collect user input. We made a request to the server to authenticate the credentials the user supplies.

Now, let's use the response from the server:

```
[...]
methods : {
        handleSubmit(e){
            [...]
                .then(response => {
                    let is_admin = response.data.user.is_admin
                    localStorage.setItem('user',JSON.stringify(response.data.user))
                    localStorage.setItem('jwt',response.data.token)

                    if (localStorage.getItem('jwt') != null){
                        this.$emit('loggedIn')
                        if(this.$route.params.nextUrl != null){
                            this.$router.push(this.$route.params.nextUrl)
                        }
                        else {
                            if(is_admin== 1){
                                this.$router.push('admin')
                            }
                            else {
                                this.$router.push('dashboard')
                            }
                        }
                    }
                })
                [...]
            }
        }
    }
}
```

We store the `jwt` token and `user` information in localStorage so we can access it from all parts of our application. Of course, we redirect the user to whichever part of our application they tried to access before being redirected to login. If they came to login directory, we redirect them based on the user type.

If you are not familiar with composing vue components, you can take this Getting Started With Vue course.

Next, create a `Register.vue` file and add the following to it:

```
<template>
  <div>
    <h4>Register</h4>
    <form>
      <label for="name">Name</label>
      <div>
        <input id="name" type="text" v-model="name" required autofocus>
      </div>

      <label for="email" >E-Mail Address</label>
      <div>
        <input id="email" type="email" v-model="email" required>
      </div>

      <label for="password">Password</label>
      <div>
        <input id="password" type="password" v-model="password" required>
      </div>

      <label for="password-confirm">Confirm Password</label>
      <div>
        <input id="password-confirm" type="password" v-model="password_confirmation" required>
      </div>

      <label for="password-confirm">Is this an administrator account?</label>
```

```
        <div>
          <select v-model="is_admin">
            <option value=1>Yes</option>
            <option value=0>No</option>
          </select>
        </div>

        <div>
          <button type="submit" @click="handleSubmit">
            Register
          </button>
        </div>
      </form>
    </div>
</template>
```

Now, define the script handling registration:

```
<script>
   export default {
      props : ["nextUrl"],
      data(){
         return {
            name : "",
            email : "",
            password : "",
            password_confirmation : "",
            is_admin : null
         }
      },
      methods : {
         handleSubmit(e) {
            e.preventDefault()

            if (this.password === this.password_confirmation && this.password.length > 0)
            {
               let url = "http://localhost:3000/register"
               if(this.is_admin != null || this.is_admin == 1) url = "http://localhost:3000/register-admin"
               this.$http.post(url, {
                  name: this.name,
                  email: this.email,
                  password: this.password,
                  is_admin: this.is_admin
               })
               .then(response => {
                  localStorage.setItem('user',JSON.stringify(response.data.user))
                  localStorage.setItem('jwt',response.data.token)

                  if (localStorage.getItem('jwt') != null){
                     this.$emit('loggedIn')
                     if(this.$route.params.nextUrl != null){
                        this.$router.push(this.$route.params.nextUrl)
                     }
                     else{
                        this.$router.push('/')
                     }
                  }
```

```
            })
            .catch(error => {
                console.error(error);
            });
        } else {
            this.password = ""
            this.passwordConfirm = ""

            return alert("Passwords do not match")
        }
      }
    }
  }
</script>
```

This is similar in structure to the `Login.vue` file. It creates the register component and accompanying method to handle user submission of the registration form.

Now, create the file `Admin.vue` and add the following:

```
<template>
    <div class="hello">
        <h1>Welcome to administrator page</h1>
        <h2>{{msg}}</h2>
    </div>
</template>

<script>
    export default {
        data () {
            return {
                msg: 'The superheros'
            }
        }
    }
</script>
<style scoped>
    h1, h2 {
        font-weight: normal;
    }
```

```
  ul {
    list-style-type: none;
    padding: 0;
  }
  li {
    display: inline-block;
    margin: 0 10px;
  }
  a {
    color: #42b983;
  }
</style>
```

This is the component we will mount when a user visits the admin page.

Finally, create the file `UserBoard.vue` and add the following:

```
<template>
   <div class="hello">
      <h1>Welcome to regular users page</h1>
      <h2>{{msg}}</h2>
   </div>
</template>

<script>
   export default {
      data () {
         return {
            msg: 'The commoners'
         }
      }
   }
</script>

<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>
   h1, h2 {
```

```
    font-weight: normal;
  }
  ul {
    list-style-type: none;
    padding: 0;
  }
  li {
    display: inline-block;
    margin: 0 10px;
  }
  a {
    color: #42b983;
  }
</style>
```

This is the file we will see when a user visits the dashboard page.

And that's it for components.

Setting Up Axios Globally

For all our server requests, we will use axios. Axios is a promise based HTTP client for the browser and node.js. Run the following command to install axios:

```
$ npm install --save axios
```

To make it accessible across all our components, open the `./src/main.js` file and add the following:

```
import Vue from 'vue'
import App from './App'
import router from './router'
import Axios from 'axios'

Vue.prototype.$http = Axios;

Vue.config.productionTip = false

new Vue({
    el: '#app',
    router,
    components: { App },
    template: '<App/>'
})
```

By defining `Vue.prototype.$http = Axios` we have modified the vue engine and added axios. We can now use axios in all our components like `this.$http` .

Running The Application

Now that we are done with the application, we need to build all our assets and run it. Because we have a node js server along with our vue application, we will be both of them for our application to work.

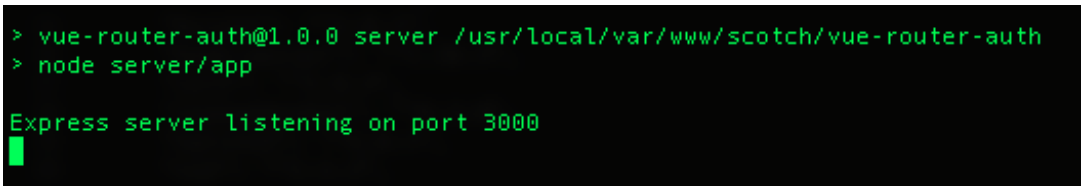
Let's add a script that will help us run our node server. Open the `package.json` file and add the following:

```
[...]
"scripts": {
    "dev": "webpack-dev-server --inline --progress --config build/webpack.dev.conf.js",
    "start": "npm run dev",
    "server": "node server/app",
    "build": "node build/build.js"
  },
[...]
```

We added the `server` script to help us start up the node server. Now, run the following command to start the server:

```
$ npm run server
```

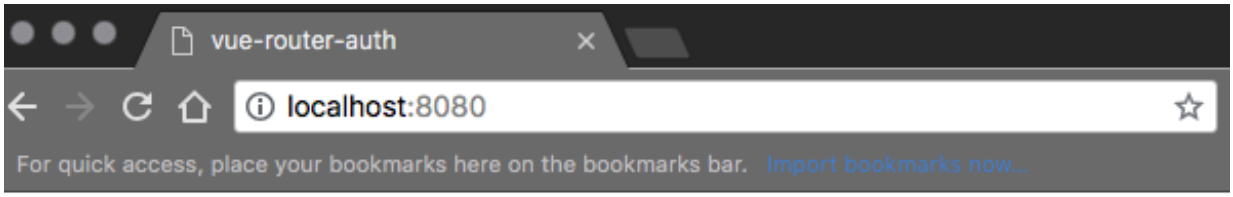
You should see something like this:



Then create another terminal instance and run the vue app like this:

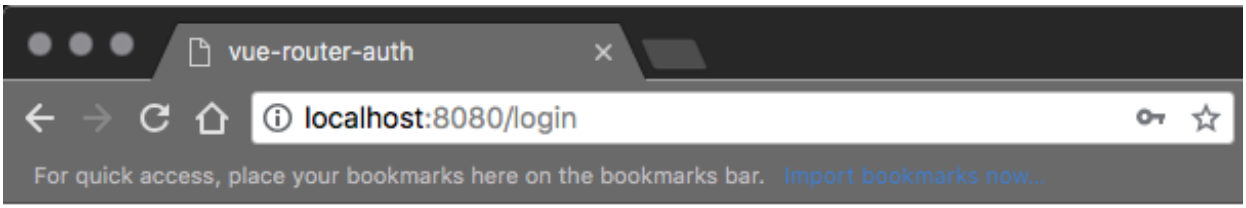
```
$ npm run dev
```

This will build all the assets and st art the application. You can open the link it shows you to see the application



This is homepage

Hello World!



Login

E-Mail Address

Password

Login

Conclusion

In this guide, we have seen how to use vue-router to define checks on our routes and prevent users from accessing certain routes. We also saw how to redirect users to different parts of our application based on the authentication state. We also built a mini server with Node.js to handle user authentication.

What we did is an example of how access control is designed in frameworks like Laravel. You can checkout out vue-router and see what other cool things you can do with it.