Role-based Access Control (Front-end + Back-end)

In this document, we will learn how we can secure our front-end application (implemented in Vue.js) by restricting access to its resources following the Role-based Access Control (RBAC) concept.

The method we will use is quite simple; we will encode and submit the user credentials to the backend for authentication, and use the encoded security token in every REST interaction. The security token will be kept in the local storage of our front-end, and will be removed when the user logs out.

Back-end (Spring Boot)

We will use the Spring Boot application that authenticates and authorizes its users through JPA, which we developed earlier as a starting point.

We start by performing several changes to our back-end App to allow it to work properly with our front-end.

You need to disable the `formLogin()`, remove the strikethrough line of the code, and add the line of code following it, which is highlighted with red color. If you did not disable the `formLogin()`, the Spring Boot default login form might be returned to you if you tried to access any of the protected resources.

You also need to add the `.httpBasic()`, as shown below, to enable user authentication through HTTP requests. When `httpBasic()` is called, we are telling Spring to authenticate the request using the values passed by the Authorization request header. If the request is not authenticated you will get a returned status of 401 and an error message of unauthorized.

```java
// SecurityConfiguration.java

@EnableWebSecurity
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter{

@Autowired
private UserDetailsService userDetailsService;

@override
protected void configure(AuthenticationManagerBuilder auth) throws Exception{
    auth.userDetailsService(userDetailsService)
        .passwordEncoder(getPasswordEncoder());
}
```
Modify SecController.java, as follows:

```java
@Bean(name = "passwordEncoder")
public PasswordEncoder getPasswordEncoder() {
    return NoOpPasswordEncoder.getInstance();
}

@Override
protected void configure(HttpSecurity http) throws Exception{
    http.cors()
    .and()
    .httpBasic()
    .and().cors()
    .and().csrf().disable()
    .authorizeRequests()
    .antMatchers("/admin").hasAuthority("ADMIN")
    .antMatchers("/user").hasAnyAuthority("ADMIN", "USER")
    .antMatchers("/auth").authenticated()
    .antMatchers("/").permitAll()
    .and().formLogin().permitAll();
    .and().formLogin().disable();
}
```

//SecController.java

```java
@CrossOrigin // you may need a CrossOrigin annotation
@RestController
public class SecController {

    @GetMapping("/authenticate")
    public List<String> authenticate() {

        // Obtains the currently authenticated principal
        Object principal = SecurityContextHolder
            .getContext()
            .getAuthentication().getPrincipal();
        List<String> roles = new ArrayList<>();
        if (principal instanceof MyUserDetails) {
            UserDetails details = (MyUserDetails) principal;
            for (GrantedAuthority authority: details.getAuthorities())
                roles.add(authority.getAuthority());
        }
```
The authenticate method would be invoked if the HTTP request includes a security token in the header, which we will do while developing our front-end app. With this token, Spring Boot security can determine the username and also query the database for the corresponding "authorities"/"roles", which will be returned as a list.

The SecurityContextHolder is a helper class that provides access to the security context. By default, it uses a ThreadLocal object to store security context, which means that the security context is always available to methods in the same thread of execution, even if you don't pass the SecurityContext object around.

More specifically, in order to get the current username (logged-in user), you first need a SecurityContext, which is obtained from SecurityContextHolder. The object returned by getContext() is an instance of the SecurityContext interface, which is the object that is stored in thread-local storage. The SecurityContext keeps the user details in an Authentication object, which can be obtained by calling the getAuthentication() method. The getPrincipal() method normally returns the UserDetails object in Spring Security, which contains all the details of the currently logged-in user.
Front-end (Vue.js)

We will create a simple login page that allows a user that can be authenticated to login by sending a security token to the back end that will, in turn, returns the role(s) that the user plays.

Setting up the Vue application: in a terminal window, you can run the following command to create a new Vue application.

```
vue create vue-sec
cd vue-sec
```

Note: do not forget to select the router component in the configuration menu during the installation.

We also need to install buefy for creating lightweight UI components for Vue and axios for REST interactions, which can be done in your terminal as follows:

```
npm install --save buefy axios
```

After the installation is concluded, open your project into your IDE (e.g. MS code), and run your application

```
npm run serve
```

Now, you need to import buefy and register into Vue by adding the following lines to the main.js, as follows:

```
// src/main.js

import Buefy from "buefy";
import "buefy/dist/buefy.css";

Vue.use(Buefy)
```
Adding a login page: add a Vue component to manage users’ login through the front-end application, i.e., create a file called `components/Login.vue` and copy the following code.

```html
// Login.vue

<template>
  <div>
    <div class="column is-half is-offset-one-quarter">
      <b-field label="Username">
        <b-input v-model="username"></b-input>
      </b-field>
      <b-field label="Password">
        <b-input type="password" v-model="password"></b-input>
      </b-field>
      <a class="button is-info is-fullwidth" @click="login">Log in</a>
    </div>
  </div>
</template>

<script>
  import auth from "./auth";
  export default {
    name: 'Login',
    data: function() {
      return {
        username: "",
        password: ""
      }
    },
    methods: {
      login: function() {
        console.log(`${this.username}, ${this.password}`);
        auth.login(this, this.username, this.password, "/");
      }
    }
  }
</script>
```

We are now ready to add the Javascript code for authentication for handling the authentication. Create a file called `components/auth.js` and copy the following code into it.

```javascript
// auth.js

import axios from "axios";
```
export default {
  user: { roles: "", username: "", authenticated: false },
  login: function(context, username, password, redirect) {
    let token = btoa(username + ":" + password);

    // do not forget to change the port to the right one, where your Spring Boot Application is running
    axios.get("http://localhost:8080/authenticate", { headers: { 'Authorization': 'Basic ' + token } })
      .then(response => {
        this.username = username;
        this.user.roles = response.data;
        this.authenticated = true;
        window.localStorage.setItem('token-' + this.username, token);

        console.log(`$ {this.username}, $ {this.user.roles}`);
        console.log(token);
        console.log(this.authenticated);
        console.log(localStorage.getItem('token-' + this.username));

        if (redirect)
          context.$router.push({ path: redirect });
      })
      .catch(error => {
        console.log(error);
      });
  },

  hasAnyOf: function(roles) {
    return this.user.roles.find(role => roles.includes(role));
  },

  logout: function() {
    window.localStorage.removeItem('token-' + this.username);
    this.user = { roles: "", username: "", authenticated: false };
  },

  authenticated: function() {
    return this.user.authenticated;
  },
}

The code above defines a property "user" that keeps the user session’s information: the username, the set of roles associated with the user, and a flag indicating if the user has been authenticated.

The login method uses a REST interaction with the back-end to retrieve the role associated with the user that is trying to login. If the credentials are not correct, the information about the user will be kept as they were (usually referring to a nonexisting session). However, if the authentication succeeds, we will store the information about the session and redirect to the path that was sent as a reference.

The statement btoa(username+":"+password) is the one that computes the Base64 encoding for the concatenation of the credentials. Note that we will copy that value into the headers as part of the REST interaction (see {headers: {'Authorization': `Basic ${token}`}}) and later into local storage for future interactions, i.e. window.localStorage.setItem('token-’+this.username, token).

The file also includes some other functions:

- **logout** cleans up the user session’s information (including local storage).
- **hasAnyOf** takes as input an array with roles as strings and checks if there is at least one matching with the roles of the currently logged-in user.
- **authorized** returns true, if the application has successfully checked the credentials of the user and his/her session is still valid (he/she has not logged out).

**Restricting the access to content:** With all the above, we have now the required infrastructure to implement the authentication process upon the backend. Since we have the information about the roles played by the logged-in user, we will now implement simple access control. Firstly, we will secure access to web pages using the router component from Vue. Please replace fully the content of the file router/index.js from your project with the following snippet.

```javascript
// router/index.js

import Vue from "vue";
import VueRouter from "vue-router";
import Home from ".../views/Home.vue";

import Login from "../components/Login.vue"
import auth from ".../components/auth.js"

Vue.use(VueRouter);

const routes = [
{
    path: "/",
    name: "home",
```
The key section in the code above is the use of the Vue-router guards\(^1\), which we can use to control access to certain pages or even invoke some actions when routing between pages. For example, we are using beforeResolve/ beforeEnter to control the access to the home, which will be accessible to a user if and only if the user has been authenticated. For the logout, the auth.logout() method is called and it will clear all the user’s information (token). It then redirects the application to the login page.

**Update the application’s navigation bar**: we need to add a reference to login/logout in the App.vue file, as follows:

\(^1\) For a comprehensive set of vue-router guards, you can refer to: https://itnext.io/yes-this-is-how-vue-router-guards-work-when-to-use-them-ed7e34946211
Now, we can use the function `auth.hasAnyOf()` to control the access to certain parts of the user interface. The following is just a very simple example, you can add components instead of the text, where such components will be shown to the users based on their roles.

```javascript
// Home vue
<template>
  <div class="home">
    <div v-if="checkRoles(['USER'])">
      Hello User
    </div>
    <div v-if="checkRoles(['ADMIN'])">
      Hello Admin
    </div>
  </div>
</template>

<script>
// @ is an alias to /src
import auth from '@/components/auth.js';

export default {
  name: "Home",
  components: {
  },
  methods: {
    checkRoles: function(roles) {
      return auth.hasAnyOf(roles);
    }
  }
};
</script>
```