TP3 – 8INF138

MARA22050108 – PIET08110102

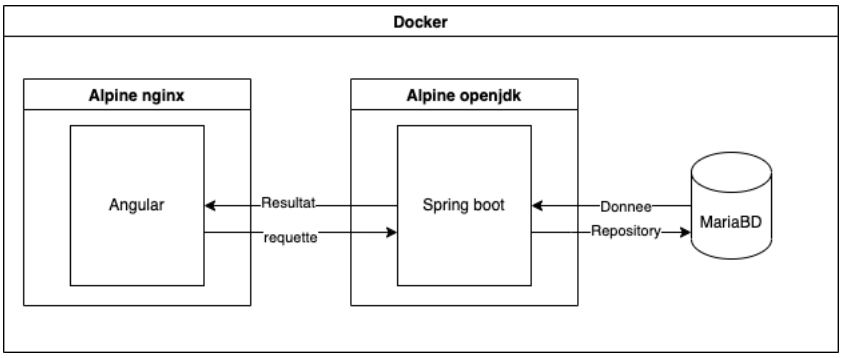
Table des matières

I. Introduction

Ce présent document est le rapport du TP3, Pour réaliser ce TP nous avons décidé d'utiliser les deux framework suivant : Spring Boot ainsi que Angular CLI. Le but de ce TP était la mise en place d'un système d'authentification sécurisé sur un site internet. Pour cela, nous allons dans un premier temps présenter l'architecture du système, ce qui nous amènera à parler de la gestion des accès ainsi que de l'authentification, pour enchaîner sur une partie sur la gestion des sessions. Le tout sera reconsidéré en conclusion.

II. Architecture du système

Ce système est un système distribue de 3tiers, une partie client avec Angular, une partie serveur avec Spring Boot et une partie Base de données avec MariaDB. Pour faire tourner l'ensemble de ces systèmes nous utiliseront docker.



III. Gestion des accès

L'accès aux données est protégé par deux sécurités, la première est le module de Auth d'angular qui permet de limiter l'accès à certaines URL via des conditions, par exemple la présence d'un token valide, la deuxième couche se trouve du côté de spring boot avec le module WebSecurity qui permet de configurer les autorisation pour accéder au différentes requêtes de l'API.

A. Role et token

Nous avons trois tables de la BDD qui entrent en jeux dans la partie de la gestion des acces, la table des rôles, la table des User et la table des Tokens.

1 Diagramme de la BDD

Nous pouvons observer ici que chaque token est relié à un rôle via la table User, c'est ainsi que la gestion d'accès par rôle fonctionne.

B. Spring Boot

Dans un premier temps nous allons observer cela du point de vue de l'API; quand l'api reçoit une requête, elle sera deviée et devra respecter plusieurs critères, le premier étant que l'URL demandé soit valide et ne possède pas d'instruction permettant de compromettre les données.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.cors().and() HttpSecurity
            .sessionManagement() SessionManagementConfigurer<HttpSecurity>
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            .and() HttpSecurity
            .exceptionHandling() ExceptionHandlingConfigurer<HttpSecurity>
            .and() HttpSecurity
            .addFilterBefore(new FiltreAuthentification(tokenService), AnonymousAuthenticationFilter.class)
            .authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .antMatchers(urlAllDispo) ExpressionUrlAuthorizationConfigurer<...>.AuthorizedUrl
            .permitAll() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .antMatchers(urlAdminConnected) ExpressionUrlAuthorizationConfigurer<...>.AuthorizedUrl
            .hasAuthority("admin") ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .antMatchers(urlResidentielConnected) ExpressionUrlAuthorizationConfigurer<...>.AuthorizedUrl
            .hasAnyAuthority( ...authorities: "residentiel", "admin") ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .antMatchers(urlAffaireConnected) ExpressionUrlAuthorizationConfigurer<...>.AuthorizedUrl
            .hasAnyAuthority( ...authorities: "affaire", "admin") ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .anyRequest().denyAll()
            .and() HttpSecurity
            .csrf().disable()
            .formLogin().disable()
            .httpBasic().disable()
            .logout().disable();
}
```

2 Filtrage des requête reçu

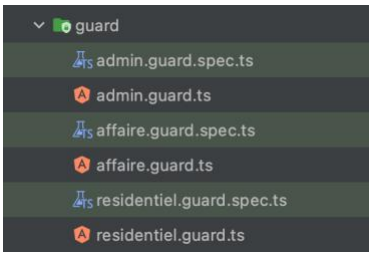
Dans un deuxième temps, elle va analyser la nature de cette est demandée, et selon son type, elle va vérifier si elle a besoin d'un contrôle de sécurité. Si l'URL demande est permise pour tout le monde alors elle peut renvoyer traiter la requête dans n'importe quel cas, si elle n'est permise que pour un rôle alors elle doit vérifier que le header de la requête http possède un token et que ce token est relié au rôle permettant d'accéder à l'URL, mais aussi que sa date d'expiration ne soit pas passée. Si ce n'est pas le cas, elle renverra une erreur 403.

```java
@Override
public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)
        throws IOException, ServletException {
    try {
        HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;
        String authToken = httpServletRequest.getHeader( s: "AUTHORIZATION");
        authorityList = new ArrayList<>();
        if (authToken != null) {
            authToken = StringUtils.removeStart(authToken,  remove: "Bearer").trim();

            Optional<Token> tokenOptional = tokenService.checkTokenUnique(authToken);

            if (tokenOptional.isPresent()) {

                Token token = tokenOptional.get();
                authorityList.add(new SimpleGrantedAuthority(token.getUser().getRole().getTagRole()));
                UsernamePasswordAuthenticationToken tokens = new UsernamePasswordAuthenticationToken( principal: "",  credentials: "",
                SecurityContextHolder.getContext().setAuthentication(tokens);
            }
        }
        filterChain.doFilter(servletRequest, servletResponse);   Marc, Today • Init of project BACK
    } catch (Exception ex) {
        throw new RuntimeException(ex);
    }
}
```

3 Verification que le token est valide et qu'il correspond au role demander

C. Angular

Pour angular, nous cherchons à empêcher l'accès à certaines URL, mais seulement au niveau du client. Pour cela, nous allons utiliser un Guard par rôle. Ce Guard va vérifier qu'il est possible de faire une requête avec un Token enregistré en mémoire.



4 Les Guard du projet

```typescript
canActivate(
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
  let returnStatement = false

  if (localStorage.getItem( key: 'token') == null) {
    return false;
  }
  return this.loginService.checkAdmin().toPromise().then(value => {
    return value;
  }).catch(error => {                          //Error callback
    console.error('Vous avez pas les droits');
    return false
    //throw error;   //You can also throw the error to a global error handler
  });
}
```

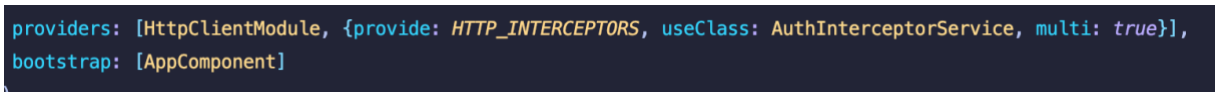
5 Guard de l'admin

On peut vérifier dans le Guard de l'admin que s'il y n'y a pas de token ou bien si la requête pour check la validité du token retourne une erreur, alors l'accès est refusé.

6 Gestion des path

Comme on le voit sur la figure 6, on doit rajouter pour chaque path que l'on souhaite souhaite protéger un canActivate qui pointe vers le Guard correspondant à la limitation d'accès voulu.

7 Intercepteur des requête http/https pour rajouter le token au header



8 Intégration de l'intercepteur au ngModule

Sur la figure 7, on voit le processus d'interception d'une requête afin de rajouter le token au header, tandis que sur la figure 8, on voit son intégration primordiale pour son fonctionnement.

IV. Authentification

A. Spring boot

L'authentification se passe surtout dans le SpringBoot.
Dans un premier temps, on reçoit via une méthode post un courriel ainsi qu'un password.

```java
@CrossOrigin
@PostMapping("login")
public Map<String, Object> login(@RequestBody Map<String, String> mapInformations) throws NoSuchAlgorithmException {
    String password = mapInformations.get("password");
    String login = mapInformations.get("login");
    return userService.login( login, password);
}
```

9 Appel du controleur

Ensuite, l'userService prend le relais.

```java
public Map<String, Object> login(String login, String password){   Marc, Today • Init of project BACK
    String passwordCrypt = "";
    User salt = userRepository.findByLogin(login);
    try {
        if(salt != null)
            passwordCrypt = Hasher.hashPassword(salt.getSalt(), password, iterations: 1000);
        System.out.println(passwordCrypt);
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
```

10 Methode de verification et l'existance de l'email et de hash du mot de passe via un salt

Dans un premier temps, il va vérifier si l'email envoyé est bien connu en base de données; si c'est le cas, il récupère le salt associé à l'email pour hash le password envoyé depuis le client.

```java
byte[] bSalt = Base64.getDecoder().decode(salt);
MessageDigest md = MessageDigest.getInstance("SHA-512");
md.update(bSalt);
byte[] hashedPassword = new byte[0];
hashedPassword = md.digest(password.getBytes(StandardCharsets.UTF_8));
for (int i = 0; i < iterations; i++)
    hashedPassword = md.digest(hashedPassword);
return Base64.getEncoder().encodeToString(hashedPassword);
```

11 Hash du mot de passe

La fonction de hash est assez simple mais efficace, on transforme le salt en byte pour ensuite configurer la méthode sur le SHA-512. Après cela, on hash le password le nombre de fois voulues, puis on le revoit sous forme de string encoder sous base64.

```
User user = userRepository.findByPasswordAndLogin(passwordCrypt,login);
String tokenConnexion = null;
Map<String, Object> mapInformations = null;
if (user!=null && user.getTentative() <= 3){
    mapInformations = new HashMap<>();
    tokenConnexion = tokenService.generateToken(user);
    mapInformations.put("token", tokenConnexion);
    mapInformations.put("role", user.getRole());
} else if (user!=null && user.getTentative() >= 3){
    mapInformations = new HashMap<>();
    mapInformations.put("error", "Trop de tentative, contacter un administrateur");
}
else if(salt!=null) {
    salt.setTentative(salt.getTentative() + 1);
    mapInformations = new HashMap<>();
    mapInformations.put("error", "Mauvais mots de passe ou login");
} else {
    mapInformations = new HashMap<>();
    mapInformations.put("error", "Mauvais mots de passe ou login");
}
return mapInformations;
```

12 Verification de la validite du mot de passe

Ensuite, nous faisons une requête à la base de données afin de vérifier si l'email envoyé et le login correspondent bien.

- Si l'utilisateur existe et que le nombre de tentative n'est pas dépassé, alors on crée une hash map qui renvoie le token ainsi que le rôle de l'utilisateur.
- Si l'utilisateur existe avec la combinaison du mot de passe et de l'email mais que le nombre de tentative est dépassé alors on renvoie un message d'erreur.
- Si l'utilisateur existe mais que le mot de passe est mauvais alors on augmente son nombre de tentative.

Suite à cela, tout est renvoyé au client Angular.

B. Angular

13 Processus Angular d'authentification

Pour le processus d'authentification du côté d'Angular, tout est géré dans une méthode ; si l'on reçoit un code d'erreur de la part de l'api, on l'affiche à l'écran en l'enregistrant dans une variable, et si le data reçu contient un token et un rôle, alors on enregistre le token dans le localStorage, puis on fait un switch sur le tag du rôle pour rediriger vers la bonne URL.

C. Point fort et points faible

Le principal point faible de cette méthode est la possibilité de se faire voler un token de connexion : en effet, comme le token est enregistré sur le localStorage, une faille dans un module permettant de récupérer le contenu du localStorage serait assez fatal.

Un autre point faible est la non-sécurisation du mot de passe quand nous ne sommes pas en https, il navigue librement sans entre hash au préalable.

Le point fort de cette méthode est son efficacité, on peut absolument tout paramétrer pour autoriser le moins de choses possible, on pourrait par exemple configurer le Cors pour autoriser les requêtes ne venant que du client.

V. Gestion des sessions

Nous utilisons une API Rest, donc une api qui ne prend pas en charge les session (Stateless). C'est donc pricipalement Angular qui va prendre en charge la gestion de la session et cela se fait via le token. Le token renvoyé par l'api peut représenter un id de session assez primaire.

Une fois le token expiré, la session doit être renouvellée, on pourrait imaginer une pop-up qui demande l'augmentation de la durée de validité du token un peu comme sur Moodle.

Le principal défaut de cette solution est le potentiel vol de session en récupérant le token, on pourrait ajouter une protection en associant le token à une adresse ip unique ce qui rendrais le vol plus compliqué.

Le principal avantage est la facilité à sa mise en place, en effet, toutes les informations utilisateur sont reliées à ce token et il est très simple de récupérer le rôle ou une autre information via ce dernier et ainsi limiter la transmission et le stockage de données sensibles.

VI. Conclusion

Nous avons pu voir via ce TP une autre manière de gérer l'authentification et la gestion des sessions via une API rest Stateless, nous avons fait le choix de cette méthode parmi plusieurs disponible, des points sont encore à améliorer comme la configuration du cors pour limiter les appels à l'API à une seul source, la configuration d'un anti DDOS qui peut se faire simplement via spring boot ou encore augmenter la sécurité des sessions via l'enregistrement de l'adresse IP ou d'un identifiant unique relié au navigateur du client

Nous sommes satisfait du résultat de ce travail malgré tout cela, car configurer une méthode d'authentification est toujours une étape critique dans le processus de création d'un site internet.