tuto Angular + Spring JWT

Ce tuto est un résumé du tuto aux adresses :

<https://www.javainuse.com/spring/ang7-login>

<https://www.javainuse.com/spring/ang7-basic>

<https://www.javainuse.com/spring/ang7-basic-interceptor>

https://www.javainuse.com/spring/ang7-jwt

[1 – Spring](#_r72948s8n8c1)

[1.1 User Model](#_lmduhe2gbf9e)

[1.1.1 Models -> UserAdmin.java](#_9s170lfcdln3)

[dao-> UserDao.java](#_agcnh0evbsp7)

[1.2 Ajout de Spring Secu+ JWT](#_tqpmpt2lo60i)

[1.3 Controleur pour l’authentification](#_c7n76zwmnima)

[Rest -> JwtAuthenticationController.java](#_hykamsdyemq4)

[Auth -> JwtTokenUtil.java](#_r3p3uvdz46gb)

[Auth -> JwtUserDetailsService.java](#_3b4aith7tuor)

[1.4 Gestion des tokens](#_7n89s13kyg62)

[models -> JwtResponse.java](#_swgp8dhws15j)

[models -> JwtRequest.java](#_r04iec4u7h1b)

[Auth -> JwtAuthenticationEntryPoint.java](#_q23ll7vrpht)

[1.5 WebSecurity](#_g8cvky6419d3)

[Auth -> JwtRequestFilter.java](#_2n9ul1778703)

[Auth -> WebSecurityConfig.java](#_d5jjb3dnxuip)

[Cette classe paramètre la sécurité mise en place : quels urls sont contrôlés, quels filtres sont appliqués, quels headers sont acceptés etc…](#_27u8khhylwjj)

[2 – Angular](#_i445ww8ffs8r)

[1.1 Authentication Service](#_ulxp7p1etqgw)

[1.2 Login Component](#_xl0bmwi2aa8p)

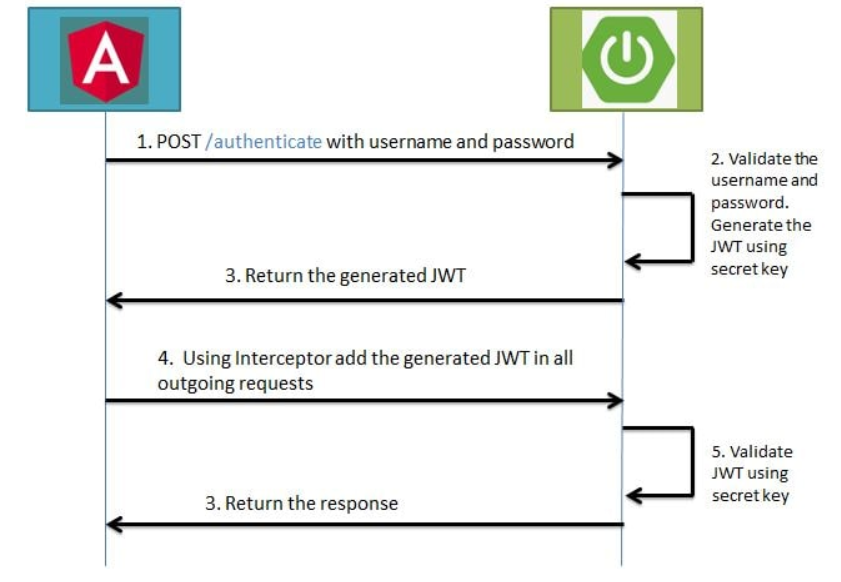
[1.3 Logout Component](#_f0o3vvgrje74)

[1.4 Modification du header](#_877grcowm199)

[1.4 Ajout de la Guard](#_dv6jx838axaq)

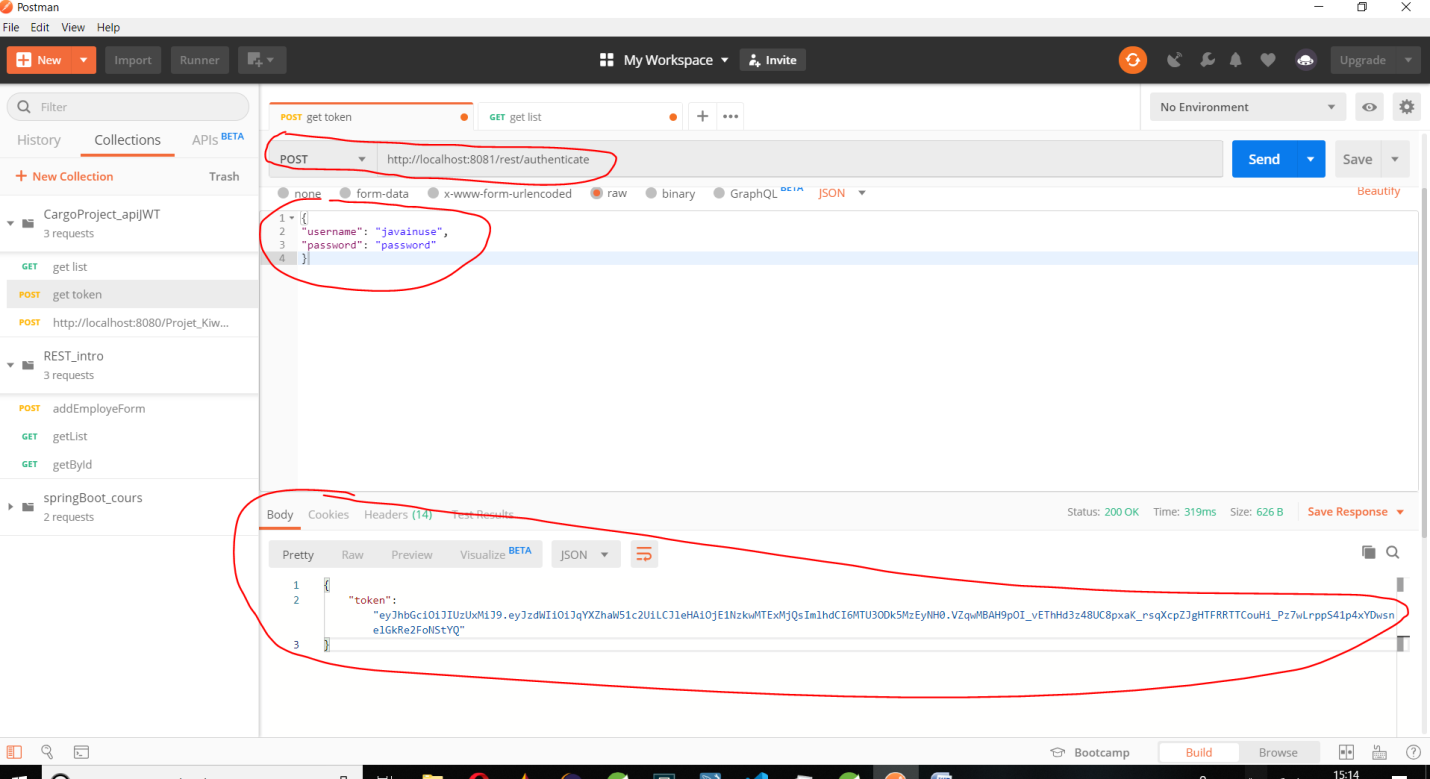
[1.5 Ajout d’un intercepteur HTTP](#_igs9sjp4pbpe)

Principe de l’identification par Token :



* Le Client (appli Angular) veut acceder aux ressource du web service RESTful (Spring).
* Pour s’identifier le Client envoie au serveur ses identifiants (ici username + password).
* Le serveur, plutôt que de s’embêter à vérifier à chaque requête si les identifiants sont corrects, va générer un *token* (une clef d’identification) qu’il renvoie au client. Ce token est généré à partir d’une autre clef secrète codée au préalable dans le serveur et est unique.
* Pour toutes les requêtes suivantes, le client ajoute uniquement dans le body de ses requetes un param “Authorization” dans lequel il stoque le token.

Exemple d’authentification avec Postman :



Dans cet exemple le serveur renvoie un token si username et password sont corrects.

Résumé des étapes d’implémentation :

Java :

* Ajouter des utilisateurs
* Ajouter JWT pour gérer des tokens
* Ajouter Spring Secu pour restraindre l’accès aux endpoints de l’api

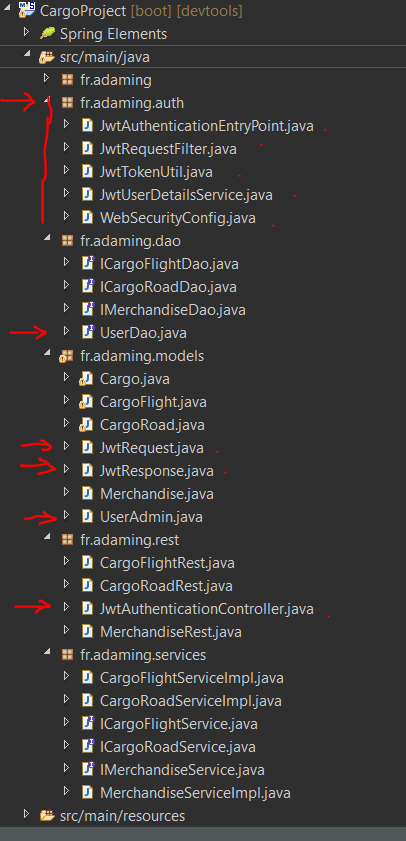
Angular :

* Ajouter un service d’identification avec le serveur
* Ajouter un composant login/logout
* Restreindre les routes (guards)
* Intercepter les requetes sortantes pour y ajouter le token reçu

# 

# 1 – Spring

L’architecture coté Spring va nécessiter d’ajouter les classes et interfaces suivantes :



## 1.1 User Model

Première étape : On va créer une classe « User » qui représente un utilisateur qui veut se logger :

### 1.1.1 Models -> UserAdmin.java

|  |
| --- |
| package fr.adaming.models;  import javax.persistence.Column; import javax.persistence.Entity; import javax.persistence.GeneratedValue; import javax.persistence.GenerationType; import javax.persistence.Id; import javax.persistence.Table;  @Entity @Table(name="users") public class UserAdmin {   @Id  @GeneratedValue(strategy=GenerationType.IDENTITY)  @Column(name = "id\_u")  private String username;  private String password;     public UserAdmin() {  super();  }   public UserAdmin(String username, String password) {  super();  this.username = username;  this.password = password;  }   public String getUsername() {  return username;  }   public void setUsername(String username) {  this.username = username;  }   public String getPassword() {  return password;  }   public void setPassword(String password) {  this.password = password;  }   } |

Cette classe est une entity, cela permettra de d’enregistrer les utilisateurs dans la base de données ; Dans l’exemple décris ici les valeurs sont codées en dur mais on pourrait utiliser des utilisateurs enregistrés dans la base de données.

Il faut donc une DAO associée a cette Entity (soit une interface qui hérite simplement de JpaRepository) :

### dao-> UserDao.java

|  |
| --- |
| package fr.adaming.dao;  import org.springframework.data.jpa.repository.JpaRepository; import org.springframework.stereotype.Repository;  import fr.adaming.models.UserAdmin;  @Repository public interface UserDao extends JpaRepository<UserAdmin, Integer> {   } |

## 1.2 Ajout de Spring Secu+ JWT

On peut alors ajouter Spring Secu & le module de gestion des tokens dans le POM.xml (penser à update le projet hein)

|  |
| --- |
| **<!-- SPRING SECU -->**  **<dependency>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-starter-security</artifactId>**  **</dependency>**  **<!-- Java Web Token (JWT)-->**  **<dependency>**  **<groupId>io.jsonwebtoken</groupId>**  **<artifactId>jjwt</artifactId>**  **<version>0.9.1</version>**  **</dependency>** |

## 1.3 Controleur pour l’authentification

### Rest -> JwtAuthenticationController.java

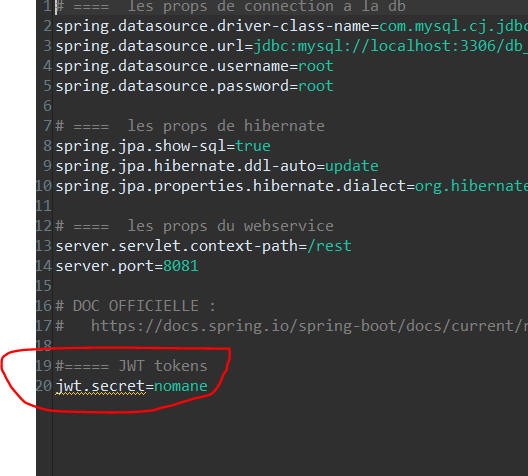
On ajoute un contrôleur qui permettra de récupérer et de gérer les connections des clients web. Les composants @Autowired sont explicité après ; JwtUserDetailsService permet de gérer les infos des utilisateurs et JwtTokenUtil s’occupe de générer les tokens.

|  |
| --- |
| package fr.adaming.rest;  import org.springframework.beans.factory.annotation.Autowired; import org.springframework.http.ResponseEntity; import org.springframework.security.authentication.AuthenticationManager; import org.springframework.security.authentication.BadCredentialsException; import org.springframework.security.authentication.DisabledException; import org.springframework.security.authentication.UsernamePasswordAuthenticationToken; import org.springframework.security.core.userdetails.UserDetails; import org.springframework.web.bind.annotation.CrossOrigin; import org.springframework.web.bind.annotation.RequestBody; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RequestMethod; import org.springframework.web.bind.annotation.RestController;  import fr.adaming.auth.JwtTokenUtil; import fr.adaming.auth.JwtUserDetailsService; import fr.adaming.models.JwtRequest; import fr.adaming.models.JwtResponse;  @RestController @CrossOrigin public class JwtAuthenticationController {  @Autowired  private AuthenticationManager authenticationManager;  @Autowired  private JwtTokenUtil jwtTokenUtil;  @Autowired  private JwtUserDetailsService userDetailsService;   @RequestMapping(value = "/authenticate", method = RequestMethod.POST)  public ResponseEntity<?> createAuthenticationToken(@RequestBody JwtRequest authenticationRequest) throws Exception {  authenticate(authenticationRequest.getUsername(), authenticationRequest.getPassword());  final UserDetails userDetails = userDetailsService.loadUserByUsername(authenticationRequest.getUsername());  final String token = jwtTokenUtil.generateToken(userDetails);  return ResponseEntity.ok(new JwtResponse(token));  }   private void authenticate(String username, String password) throws Exception {  try {  authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(username, password));  } catch (DisabledException e) {  throw new Exception("USER\_DISABLED", e);  } catch (BadCredentialsException e) {  throw new Exception("INVALID\_CREDENTIALS", e);  }  } } |

### Auth -> JwtTokenUtil.java

Cette classe permet de générer des tokens. Elle se base sur une variable “jwt.secret”, ecrit en dur dans un fichier de conf (voir ci apres) pour la garder secrete,pour générer des token. Un token est une clef unique qui a une durée de validitée.

|  |
| --- |
| package fr.adaming.auth;  import java.io.Serializable; import java.util.Date; import java.util.HashMap; import java.util.Map; import java.util.function.Function; import org.springframework.beans.factory.annotation.Value; import org.springframework.security.core.userdetails.UserDetails; import org.springframework.stereotype.Component; import io.jsonwebtoken.Claims; import io.jsonwebtoken.Jwts; import io.jsonwebtoken.SignatureAlgorithm;  @Component public class JwtTokenUtil implements Serializable {  private static final long serialVersionUID = -2550185165626007488L;  public static final long JWT\_TOKEN\_VALIDITY = 5 \* 60 \* 60;  // L’annotation @Value permet de recuperer une variable ecrite dans un fichier de conf  // Ce secret est une valeur sur laquelle se base JWT pour calculer un token  @Value("${jwt.secret}")  private String secret;   //retrieve username from jwt token  public String getUsernameFromToken(String token) {  return getClaimFromToken(token, Claims::getSubject);  }   //retrieve expiration date from jwt token  public Date getExpirationDateFromToken(String token) {  return getClaimFromToken(token, Claims::getExpiration);  }   public <T> T getClaimFromToken(String token, Function<Claims, T> claimsResolver) {  final Claims claims = getAllClaimsFromToken(token);  return claimsResolver.apply(claims);  }   // for retrieveing any information from token we will need the secret key  private Claims getAllClaimsFromToken(String token) {  return Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody();  }  //check if the token has expired  private Boolean isTokenExpired(String token) {  final Date expiration = getExpirationDateFromToken(token);  return expiration.before(new Date());  }  //generate token for user  public String generateToken(UserDetails userDetails) {  Map<String, Object> claims = new HashMap<>();  return doGenerateToken(claims, userDetails.getUsername());  }  //while creating the token - //1. Define claims of the token, like Issuer, Expiration, Subject, and the ID //2. Sign the JWT using the HS512 algorithm and secret key. //3. According to JWS Compact Serialization(https://tools.ietf.org/html/draft-ietf-jose-json-web-signature-41#section-3.1) // compaction of the JWT to a URL-safe string   private String doGenerateToken(Map<String, Object> claims, String subject) {  return Jwts.builder().setClaims(claims).setSubject(subject).setIssuedAt(new Date(System.currentTimeMillis()))  .setExpiration(new Date(System.currentTimeMillis() + JWT\_TOKEN\_VALIDITY \* 1000))  .signWith(SignatureAlgorithm.HS512, secret).compact();  }  //validate token  public Boolean validateToken(String token, UserDetails userDetails) {  final String username = getUsernameFromToken(token);  return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));  } } |
|  |

Pour ajouter dans Spring un mot qui servira à la génération de token, il faut l’ajouter dans le fichier *application.properties* de notre application :

### Auth -> JwtUserDetailsService.java

Cette classe fait la traduction pour l’authentification d’un potentiel client. Elle simule la base de donnée avec un client “javainuse”/”password” en renvoyant le hash correspondant a son mot de passe. Les hash des mots de passe seront par la suite comparés pour identifier le client. Elle permet aussi d’enregistrer un nouveau client (UserAdmin ici).

|  |
| --- |
| package fr.adaming.auth;   import java.util.ArrayList;  import org.springframework.beans.factory.annotation.Autowired; import org.springframework.security.core.userdetails.User; import org.springframework.security.core.userdetails.UserDetails; import org.springframework.security.core.userdetails.UserDetailsService; import org.springframework.security.core.userdetails.UsernameNotFoundException; import org.springframework.security.crypto.password.PasswordEncoder; import org.springframework.stereotype.Service;  import fr.adaming.dao.UserDao;  @Service public class JwtUserDetailsService implements UserDetailsService {   @Autowired  private UserDao userDao;   @Autowired  private PasswordEncoder bcryptEncoder;  // Cette methode simule la base de donnee :  // Si le username correspond a javainuse (ie il est present dans la base) alors  // elle retourne un objet user avec en password le hash correspondant a notre mot de passe  // “password” ecrit en dur dans l’appli  @Override  public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  if ("javainuse".equals(username)) {  return new User("javainuse", "$2a$10$slYQmyNdGzTn7ZLBXBChFOC9f6kFjAqPhccnP6DxlWXx2lPk1C3G6",  new ArrayList<>());  } else {  throw new UsernameNotFoundException("User not found with username: " + username);  }  }   public UserAdmin save(UserAdmin user) {  UserAdmin newUser = new UserAdmin();  newUser.setUsername(user.getUsername());  newUser.setPassword(bcryptEncoder.encode(user.getPassword()));  return userDao.save(newUser);  }  } |

A l’issu de ces étapes on peut récupérer les identifiants d’un client, calculer le hash correspondant a son mot de passe, vérifier qu’il est correct et (presque) envoyer un token au client en retour.

## 

## 1.4 Gestion des tokens

Bon la on rentre dans la partie obscure qui gère les tokens d’authentification. Afin de générer un token on a besoin de plusieurs classes outils.

### models -> JwtResponse.java

Cette classe sert de modele pour le body de la reponse http

|  |
| --- |
| package fr.adaming.models;  import java.io.Serializable;  public class JwtResponse implements Serializable {  private static final long serialVersionUID = -8091879091924046844L;  private final String jwttoken;   public JwtResponse(String jwttoken) {  this.jwttoken = jwttoken;  }   public String getToken() {  return this.jwttoken;  } } |

### models -> JwtRequest.java

Cette classe sert de modele pour la requete http reçue

|  |
| --- |
| package fr.adaming.models;  import java.io.Serializable;  public class JwtRequest implements Serializable {  private static final long serialVersionUID = 5926468583005150707L;  private String username;  private String password;  //need default constructor for JSON Parsing  public JwtRequest() {  }   public JwtRequest(String username, String password) {  this.setUsername(username);  this.setPassword(password);  }   public String getUsername() {  return this.username;  }   public void setUsername(String username) {  this.username = username;  }   public String getPassword() {  return this.password;  }   public void setPassword(String password) {  this.password = password;  } } |

### 

### 

### 

### 

### Auth -> JwtAuthenticationEntryPoint.java

Cette classe est appelée dans la classe WebSecurityConfig (voir ci-après) pour setup le traitement des requetes.

|  |
| --- |
| package fr.adaming.auth;  import java.io.IOException; import java.io.Serializable; import javax.servlet.http.HttpServletRequest; import javax.servlet.http.HttpServletResponse; import org.springframework.security.core.AuthenticationException; import org.springframework.security.web.AuthenticationEntryPoint; import org.springframework.stereotype.Component;  @Component public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint, Serializable {  private static final long serialVersionUID = -7858869558953243875L;   @Override  public void commence(HttpServletRequest request, HttpServletResponse response,  AuthenticationException authException) throws IOException {  response.sendError(HttpServletResponse.SC\_UNAUTHORIZED, "Unauthorized");  } } |

## 1.5 WebSecurity

Maintenant que l’on peut récupérer les infos des client et générer proprement des tokens, il faut mettre en place la sécurité consistant à vérifier pour chaque client si il possède un token valide et a lui en générer un si besoin.

### Auth -> JwtRequestFilter.java

Cette Classe filtre toutes les requetes entrantes pour vérifier le token.

|  |
| --- |
| package fr.adaming.auth;  import java.io.IOException; import javax.servlet.FilterChain; import javax.servlet.ServletException; import javax.servlet.http.HttpServletRequest; import javax.servlet.http.HttpServletResponse; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.security.authentication.UsernamePasswordAuthenticationToken; import org.springframework.security.core.context.SecurityContextHolder; import org.springframework.security.core.userdetails.UserDetails; import org.springframework.security.web.authentication.WebAuthenticationDetailsSource; import org.springframework.stereotype.Component; import org.springframework.web.filter.OncePerRequestFilter; import io.jsonwebtoken.ExpiredJwtException;   import fr.adaming.auth.JwtUserDetailsService; import fr.adaming.auth.JwtTokenUtil;   @Component public class JwtRequestFilter extends OncePerRequestFilter {  @Autowired  private JwtUserDetailsService jwtUserDetailsService;  @Autowired  private JwtTokenUtil jwtTokenUtil;   @Override  protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)  throws ServletException, IOException {  final String requestTokenHeader = request.getHeader("Authorization");  String username = null;  String jwtToken = null; // JWT Token is in the form "Bearer token". Remove Bearer word and get // only the Token  if (requestTokenHeader != null && requestTokenHeader.startsWith("Bearer ")) {  jwtToken = requestTokenHeader.substring(7);  try {  username = jwtTokenUtil.getUsernameFromToken(jwtToken);  } catch (IllegalArgumentException e) {  System.out.println("Unable to get JWT Token");  } catch (ExpiredJwtException e) {  System.out.println("JWT Token has expired");  }  } else {  logger.warn("JWT Token does not begin with Bearer String");  } // Once we get the token validate it.  if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {  UserDetails userDetails = this.jwtUserDetailsService.loadUserByUsername(username); // if token is valid configure Spring Security to manually set // authentication  if (jwtTokenUtil.validateToken(jwtToken, userDetails)) {  UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new UsernamePasswordAuthenticationToken(  userDetails, null, userDetails.getAuthorities());  usernamePasswordAuthenticationToken  .setDetails(new WebAuthenticationDetailsSource().buildDetails(request)); // After setting the Authentication in the context, we specify // that the current user is authenticated. So it passes the // Spring Security Configurations successfully.  SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);  }  }  chain.doFilter(request, response);  } } |

### 

### 

### Auth -> WebSecurityConfig.java

### Cette classe paramètre la sécurité mise en place : quels urls sont contrôlés, quels filtres sont appliqués, quels headers sont acceptés etc…

|  |
| --- |
| package fr.adaming.auth;  import org.springframework.beans.factory.annotation.Autowired; import org.springframework.context.annotation.Bean; import org.springframework.context.annotation.Configuration; import org.springframework.http.HttpMethod; import org.springframework.security.authentication.AuthenticationManager; import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder; import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity; import org.springframework.security.config.annotation.web.builders.HttpSecurity; import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity; import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter; import org.springframework.security.config.http.SessionCreationPolicy; import org.springframework.security.core.userdetails.UserDetailsService; import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder; import org.springframework.security.crypto.password.PasswordEncoder; import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;  @Configuration @EnableWebSecurity @EnableGlobalMethodSecurity(prePostEnabled = true) public class WebSecurityConfig extends WebSecurityConfigurerAdapter {  @Autowired  private JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint;  @Autowired  private UserDetailsService jwtUserDetailsService;  @Autowired  private JwtRequestFilter jwtRequestFilter;   @Autowired  public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception { // configure AuthenticationManager so that it knows from where to load // user for matching credentials // Use BCryptPasswordEncoder  auth.userDetailsService(jwtUserDetailsService).passwordEncoder(passwordEncoder());  }   @Bean  public PasswordEncoder passwordEncoder() {  return new BCryptPasswordEncoder();  }   @Bean  @Override  public AuthenticationManager authenticationManagerBean() throws Exception {  return super.authenticationManagerBean();  }   @Override  protected void configure(HttpSecurity httpSecurity) throws Exception { // We don't need CSRF for this example  httpSecurity.csrf().disable() // dont authenticate this particular request  .authorizeRequests().antMatchers("/authenticate").permitAll() // Authorize ANGULAR OPTIONS CALLS  .antMatchers(HttpMethod.OPTIONS, "/\*\*").permitAll() // all other requests need to be authenticated  .anyRequest().authenticated().and(). // make sure we use stateless session; session won't be used to // store user's state.  exceptionHandling().authenticationEntryPoint(jwtAuthenticationEntryPoint).and().sessionManagement()  .sessionCreationPolicy(SessionCreationPolicy.STATELESS); // Add a filter to validate the tokens with every request  httpSecurity.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);  } } |

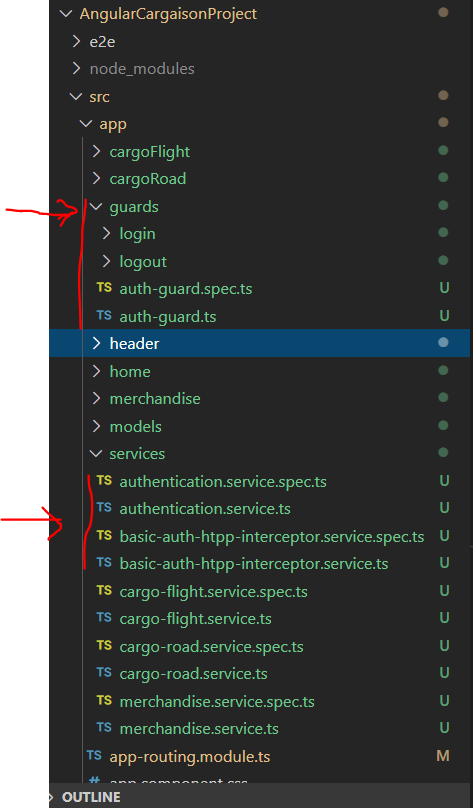
# 

# 2 – Angular

Côté Angular, on a besoin d’ajouter des composants de login/logout qui utiliseront un service qui communiquera avec le serveur pour s’authentifier et récupérer un token. Il faudra aussi ajouter une guard qui restreindra les routes et un httpInterceptor pour ajouter le token dans les requetes.

/!\ Il faut bien penser à déclarer les modules etc dans *app.module*, et dans les contructeurs de certains modules quand c’est nécessaire.

Les fichiers ajoutés vont ressembler à ça :

****

## 1.1 Authentication Service

En angular il est possible de stocker des attributs clef/valeur dans la session du navigateur web avec sessionStorage. Ces données sont perdues lorsque le navigateur est fermé ou lorsque la session est close. On a donc besoin de récupérer username/password/token au besoin.

* authenticate() Authenticate the username and password
* isUserLoggedIn() -checks the session storage if user name exists. If it does then return true
* logout()- This method clears the session storage of user name

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { map } from 'rxjs/operators';

export class User{

constructor(

public status:string,

) {}

}

export class JwtResponse{

constructor(

public jwttoken:string,

) {}

}

@Injectable({

providedIn: 'root'

})

export class AuthenticationService {

constructor(private httpClient:HttpClient) { }

authenticate(username, password) {

console.log("Authenticate : " + username +":"+password);

return this.httpClient.post<any>('http://localhost:8081/rest/authenticate',

{username:username,password:password}).subscribe(

userData => {

sessionStorage.setItem('username',username);

let tokenStr= 'Bearer '+userData.token;

sessionStorage.setItem('token', tokenStr);

return userData;

}

);

}

isUserLoggedIn() {

let user = sessionStorage.getItem('username')

// console.log("username :" , sessionStorage.getItem('username'));

// console.log("token :" , sessionStorage.getItem('token'));

return !(user === null)

}

logOut() {

sessionStorage.removeItem('username')

}

}

## 1.2 Login Component

Ce composant récupère le username + password entré par un utilisateur en utilise authenticationService pour essayer de récupérer un token.

**1.2.1 TypeScript**

Code TS :

import { Component, OnInit } from '@angular/core';

import { AuthenticationService } from 'src/app/services/authentication.service';

import { HttpClient } from '@angular/common/http';

import { Router } from '@angular/router';

@Component({

selector: 'app-login',

templateUrl: './login.component.html',

styleUrls: ['./login.component.css']

})

export class LoginComponent {

username = 'javainuse'

password = ''

invalidLogin = false

constructor(private router: Router,

private loginservice: AuthenticationService) { }

ngOnInit() {

}

checkLogin() {

if (this.loginservice.authenticate(this.username, this.password)

) {

this.router.navigate([''])

this.invalidLogin = false

} else

this.invalidLogin = true

}

}

**1.2.2 HTML**

Code HTML :

**<div class="container">**

**<div>**

**User Name : <input type="text" name="username" [(ngModel)]="username"> Password : <input type="password" name="password" [(ngModel)]="password">**

**</div>**

**<button (click)=checkLogin() class="btn btn-success">**

**Login**

**</button>**

**</div>**

## 1.3 Logout Component

Invalide la session.

**1.3.1 TypeScript**

import { Component, OnInit } from '@angular/core';

import { AuthenticationService } from 'src/app/services/authentication.service';

import { Router } from '@angular/router';

@Component({

selector: 'app-logout',

templateUrl: './logout.component.html',

styleUrls: ['./logout.component.css']

})

export class LogoutComponent implements OnInit {

constructor(

private authentocationService: AuthenticationService,

private router: Router) {

}

ngOnInit() {

this.authentocationService.logOut();

this.router.navigate(['login']);

}

}

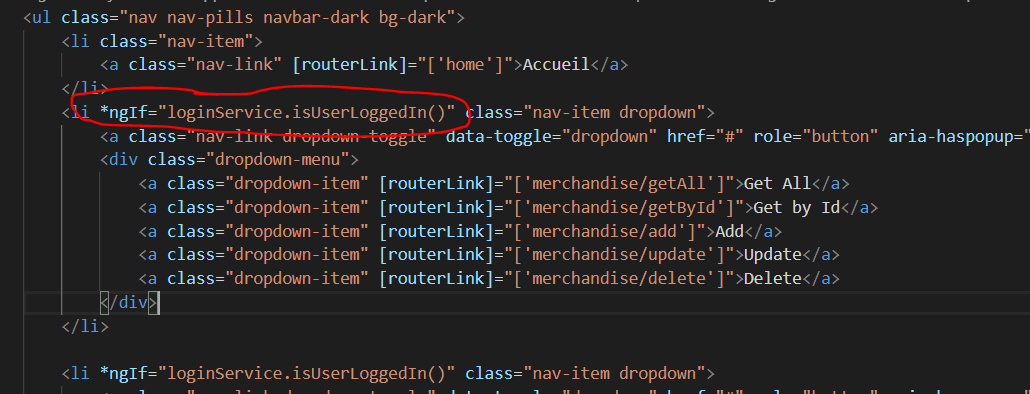
**1.3.2 HTML**

Y’en a pas lol

## 1.4 Modification du header

On ajoute dans le header.html des directives \*ngIf pour n’afficher les onglets que lorsque l’utilisateur est connecté.

Exemple :



Et il faut donc ajouter le *loginService* dans le TS du header :

export class HeaderComponent implements OnInit {

constructor(private loginService:AuthenticationService) { }

ngOnInit() {

}

}

## 1.4 Ajout de la Guard

>> ng g service guards/authGuard

Une guard est un filtre appelé lorsque l’utilisateur veut accéder a certaines routes. On peut par exemple vérifier si l’utilisateur est connecté avant de le laisser acceder au pages persos.

import {CanActivate, Router, ActivatedRouteSnapshot, RouterStateSnapshot} from "@angular/router";

import { Injectable } from '@angular/core';

import { AuthenticationService } from '../services/authentication.service';

@Injectable({

providedIn: 'root'

})

export class AuthGuard implements CanActivate {

constructor(private router: Router,

private authService: AuthenticationService) { }

canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {

if (this.authService.isUserLoggedIn())

return true;

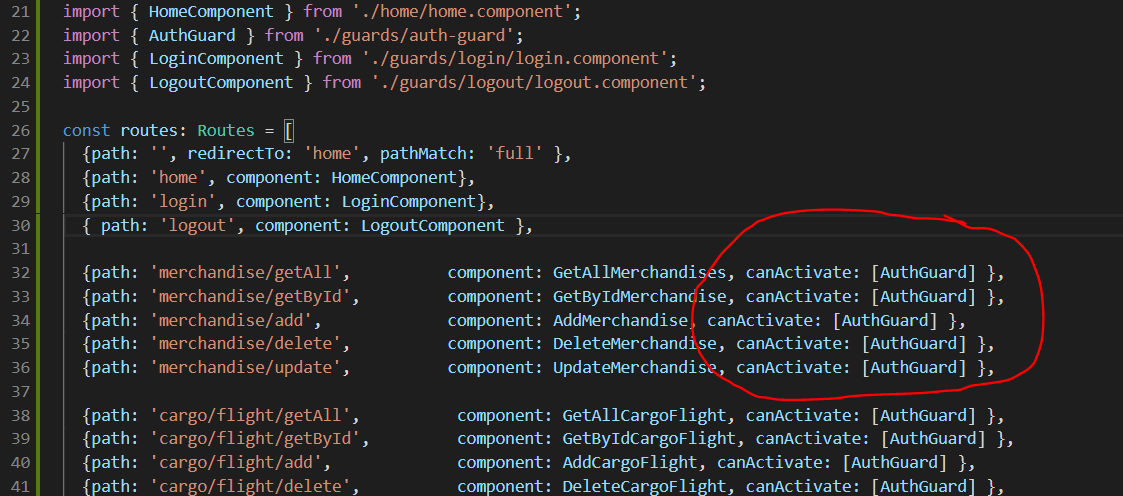
this.router.navigate(['login']);

return false;

}

}

Il faut ensuite indiquer a Angular pour quelles routes appeler la guard :



## 1.5 Ajout d’un intercepteur HTTP

L’intercepteur http BasicAuthInterceptor va vérifier les valeurs enregistrées en session en va rajouter des headers aux requetes sortantes (par exemple le token).

>> ng g service services/BasicAuthHtppInterceptor

TypeScript:

import { Injectable } from '@angular/core';

import { HttpInterceptor, HttpHandler, HttpRequest } from '@angular/common/http';

@Injectable({

providedIn: 'root'

})

export class BasicAuthHtppInterceptorService implements HttpInterceptor {

constructor() { }

intercept(req: HttpRequest<any>, next: HttpHandler) {

if (sessionStorage.getItem('username') && sessionStorage.getItem('token')) {

req = req.clone({

setHeaders: {

Authorization: sessionStorage.getItem('token')

}

})

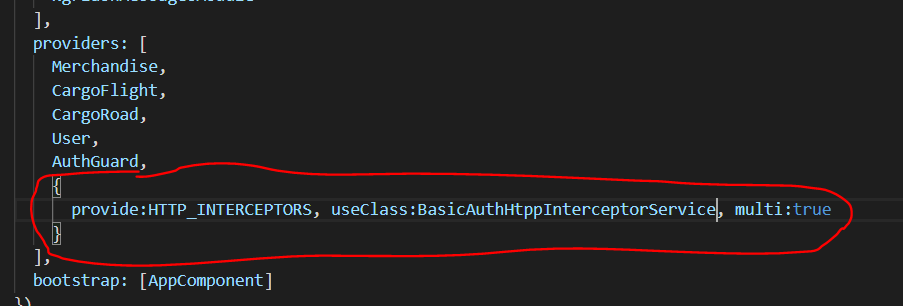
}

return next.handle(req);

}

}

Il faut aussi enregistrer l’intercepteur dans *app.module* :



**3 – Annexes**

Mon app.module & mon app-routing.module de mon projet si besoin :

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { FormsModule } from '@angular/forms';

import { NgFlashMessagesModule } from 'ng-flash-messages';

{…}

import { HomeComponent } from './home/home.component';

import { HeaderComponent } from './header/header.component';

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

import { HttpClientModule, HTTP\_INTERCEPTORS } from '@angular/common/http';

import { Merchandise } from './models/merchandise';

import { CargoFlight } from './models/cargo-flight';

import { CargoRoad } from './models/cargo-road';

import { AuthGuard } from './guards/auth-guard';

import { LoginComponent } from './guards/login/login.component';

import { User } from './models/user';

import { LogoutComponent } from './guards/logout/logout.component';

import { BasicAuthHtppInterceptorService } from './services/basic-auth-htpp-interceptor.service';

@NgModule({

declarations: [

GetAllMerchandises,

GetByIdMerchandise,

AddMerchandise,

DeleteMerchandise,

UpdateMerchandise,

GetAllCargoFlight,

GetByIdCargoFlight,

AddCargoFlight,

DeleteCargoFlight,

UpdateCargoFlight,

GetAllCargoRoad,

GetByIdCargoRoad,

AddCargoRoad,

DeleteCargoRoad,

UpdateCargoRoad,

HomeComponent,

HeaderComponent,

AppComponent,

LoginComponent,

LogoutComponent,

],

imports: [

BrowserModule,

AppRoutingModule,

HttpClientModule,

FormsModule,

NgFlashMessagesModule

],

providers: [

Merchandise,

CargoFlight,

CargoRoad,

User,

AuthGuard,

{

provide:HTTP\_INTERCEPTORS, useClass:BasicAuthHtppInterceptorService, multi:true

}

],

bootstrap: [AppComponent]

})

export class AppModule { }

const routes: Routes = [

{path: '', redirectTo: 'home', pathMatch: 'full' },

{path: 'home', component: HomeComponent},

{path: 'login', component: LoginComponent},

{ path: 'logout', component: LogoutComponent },

{path: 'merchandise/getAll', component: GetAllMerchandises, canActivate: [AuthGuard] },

{path: 'merchandise/getById', component: GetByIdMerchandise, canActivate: [AuthGuard] },

{path: 'merchandise/add', component: AddMerchandise, canActivate: [AuthGuard] },

{path: 'merchandise/delete', component: DeleteMerchandise, canActivate: [AuthGuard] },

{path: 'merchandise/update', component: UpdateMerchandise, canActivate: [AuthGuard] },

{path: 'cargo/flight/getAll', component: GetAllCargoFlight, canActivate: [AuthGuard] },

{path: 'cargo/flight/getById', component: GetByIdCargoFlight, canActivate: [AuthGuard] },

{path: 'cargo/flight/add', component: AddCargoFlight, canActivate: [AuthGuard] },

{path: 'cargo/flight/delete', component: DeleteCargoFlight, canActivate: [AuthGuard] },

{path: 'cargo/flight/update', component: UpdateCargoFlight, canActivate: [AuthGuard] },

{path: 'cargo/road/getAll', component: GetAllCargoRoad, canActivate: [AuthGuard] },

{path: 'cargo/road/getById', component: GetByIdCargoRoad, canActivate: [AuthGuard] },

{path: 'cargo/road/add', component: AddCargoRoad, canActivate: [AuthGuard] },

{path: 'cargo/road/delete', component: DeleteCargoRoad, canActivate: [AuthGuard] },

{path: 'cargo/road/update', component: UpdateCargoRoad, canActivate: [AuthGuard] },

{path: '\*\*', component: HomeComponent}

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }