# Spring Security et JWT

# Démonstration 2 du module 6

|  |
| --- |
| Les objectifs de cette démonstration sont   * Ajout des libriairies pour JWT * Mise en place de JWT avec Spring Security |

Contexte

* Nous reprenons notre application représentant une partie du cœur de métier de l’ENI Ecole.
  + Il s’agit de l’application demo-eni-ecole
  + Si votre application n’est pas complète, vous pouvez reprendre le code Java dans les ressources
  + Attention, le paramétrage du projet Spring Boot n’est pas inclus
    - Il vous faut reprendre les démonstrations des modules 1, 2 et 4 pour valider les starters de Spring Boot
* Dans cette itération, nous voulons ajouter l’authentification d’un utilisateur et ses habilitations

Pour la sécurité, nous voulons déclarer 2 rôles différents

* Rôle « EMPLOYE » - a les droits en lecture sur l’application (GET)
* Rôle « ADMIN » - a tous les droits du rôle « EMPLOYE » et les droits de créations, mise à jour et suppression (POST, PUT, PATCH, DELETE)

Déroulement

Nous allons nous appuyer sur les 2 schémas du cours pour réaliser les étapes de la démonstration

* Cycle d’authentification

Une image contenant texte, capture d’écran, Police, diagramme

Description générée automatiquement

* Cycle d’autorisation

Une image contenant texte, Police, capture d’écran, diagramme

Description générée automatiquement

# Etape 1 : Librairies JJWT :

Il faut ajouter les lib librairies JJWT API et JJWT Impl pour permettre de manipuler le jeton JWT

* Dans build.gradle ; ajouter les librairies pour JWT

// https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-api

**implementation** 'io.jsonwebtoken:jjwt-api:0.11.5'

// https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-impl

**runtimeOnly** 'io.jsonwebtoken:jjwt-impl:0.11.5'

// https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-jackson

**runtimeOnly** 'io.jsonwebtoken:jjwt-jackson:0.11.5'

* Faire un Gradle Refresh Project

# Etape 2 : Création de la classe UserInfo et en faire une entité JPA

Dans les 2 schémas, il faut manipuler les données d’un utilisateur depuis la base. Pour cela, il faut une entité et un Repository associé

* L’entité correspond à la table « users » déjà existante

Une image contenant texte, Police, capture d’écran, ligne

Description générée automatiquement

**package** fr.eni.demo.security.jwt;

**import** java.util.\*;

**import** jakarta.persistence.\*;

**import** lombok.\*;

@Data

@AllArgsConstructor

@NoArgsConstructor

@Entity

@Table(name="users")

**public** **class** UserInfo {

@Id

@Column(length = 250)

**private** String pseudo;

@Column(length = 68, nullable = **false**)

**private** String password;

@Column(length = 15, nullable = **false**)

**private** String authority;

}

* Création de UserInfoRepository

**package** fr.eni.demo.security.jwt;

**import** org.springframework.data.jpa.repository.JpaRepository;

**public** **interface** UserInfoRepository **extends** JpaRepository<UserInfo, String>{

}

# Etape 3 : La classe UserInfo doit implémenter l’interface UserDetails

Pour manipuler cette entité dans le contexte de Spring Security, il faut que l’entité implémente UserDetails

* Cette interface appartient à Spring Security
  + Elle représente l’utilisateur a connecté pour Spring Security
* Pour comprendre cette classe et ce qu’elle implique
  + Il faut aller explorer la classe : org.springframework.security.core.userdetails.UserDetails
  + https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/userdetails/UserDetails.html

**package** fr.eni.demo.security.jwt;

**import** java.util.\*;

**import** org.springframework.security.core.GrantedAuthority;

**import** org.springframework.security.core.authority.SimpleGrantedAuthority;

**import** org.springframework.security.core.userdetails.UserDetails;

**import** jakarta.persistence.\*;

**import** lombok.\*;

@Data

@AllArgsConstructor

@NoArgsConstructor

@Entity

@Table(name="users")

**public** **class** UserInfo **implements** UserDetails{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

@Id

@Column(length = 250)

**private** String pseudo;

@Column(length = 68, nullable = **false**)

**private** String password;

@Column(length = 15, nullable = **false**)

**private** String authority;

//Correspond aux rôles de l'utilisateur

@Override

**public** Collection<? **extends** GrantedAuthority> getAuthorities() {

**return** Arrays.*asList*(**new** SimpleGrantedAuthority(authority));

}

// Correspond à l'élément unique d'authentification

@Override

**public** String getUsername() {

**return** pseudo;

}

// Etat du compte utilisateur – compte non expiré ?

@Override

**public** **boolean** isAccountNonExpired() {

**return** **true**;

}

// Etat du compte utilisateur – non verrouillé ?

@Override

**public** **boolean** isAccountNonLocked() {

**return** **true**;

}

// Indique si les informations d’identification sont non expirées ?

@Override

**public** **boolean** isCredentialsNonExpired() {

**return** **true**;

}

// Etat du compte utilisateur – actif ?

@Override

**public** **boolean** isEnabled() {

**return** **true**;

}

}

# Etape 4 : UserDetailsService

Pour manipuler UserInfo avec Spring Security ; il faut déclarer UserDetailsService

* UserDetailsService doit être déclaré dans une classe de configuration
  + Création d’une classe JwtAppConfig
  + Elle centralise la configuration de sécurité liée à l’entité UserInfo
  + Manipule la base de données au travers de UserInfoRepository
  + Et encapsule la sécurité : AuthenticationProvider et AuthenticationManager

**package** fr.eni.demo.security.jwt;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.context.annotation.\*;

**import** org.springframework.security.authentication.\*;

**import** org.springframework.security.authentication.dao.DaoAuthenticationProvider;

**import** org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;

**import** org.springframework.security.core.userdetails.\*;

@Configuration

**public** **class** JwtAppConfig {

/\*\*

\* Authentification de l'utilisateur depuis la base de données

\*/

@Autowired

**private** UserInfoRepository userInfoRepository;

@Bean

UserDetailsService userDetailsService() {

**return** username -> userInfoRepository.findById(username)

.orElseThrow(() -> **new** UsernameNotFoundException("User not found"));

}

@Bean

AuthenticationProvider authenticationProvider() {

DaoAuthenticationProvider authProvider = **new** DaoAuthenticationProvider();

authProvider.setUserDetailsService(userDetailsService());

**return** authProvider;

}

@Bean

AuthenticationManager authenticationManager(AuthenticationConfiguration config) **throws** Exception {

**return** config.getAuthenticationManager();

}

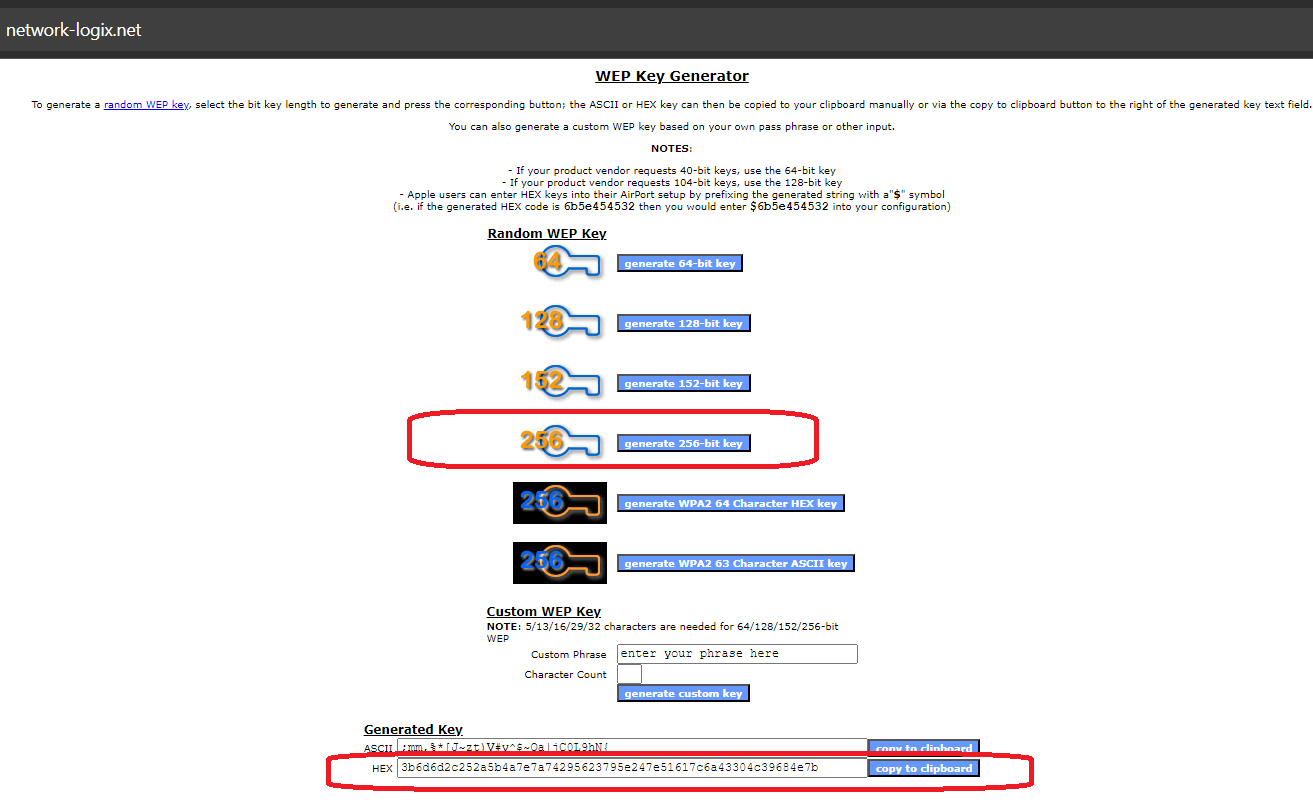
}

# Etape 5 : Créer JwtService

D’après les 2 schémas, JwtService va gérer le jeton JWT.

* C’est pour manipuler le jeton JWT, qu’il fallait ajouter les dépendances JJWT API et JJWT Impl
* JwtService permet de
  + Générer un jeton
    - Définir l'entête,
    - Les « claims » « champs standards »,
    - Le délai d'expiration
    - Et permet de signer avec la « secret key »
* Elle doit
  + Vérifier l'authenticité du jeton,
  + Obtenir les garanties de jeton
* Elle définit une méthode extractUsername(String token) pour retourner le pseudo contenu dans le jeton

1. Déclarer une clef de chiffrement sur 256 en héxadécimal
   * Utiliser le site : http://network-logix.net/



* + Cliquer sur « generate 256 bit key »
  + Copier la clef en héxadécimal
  + La déclarer en paramètre dans application.properties

#JWT

app.jwt.secret=2529355127…………

* + La récupérer dans la classe JwtService

**package** fr.eni.demo.security.jwt;

**import** java.security.Key;

**import** java.util.\*;

**import** java.util.function.Function;

**import** org.springframework.beans.factory.annotation.Value;

**import** org.springframework.security.core.userdetails.UserDetails;

**import** org.springframework.stereotype.Service;

**import** io.jsonwebtoken.\*;

**import** io.jsonwebtoken.io.Decoders;

**import** io.jsonwebtoken.security.Keys;

@Service

**public** **class** JwtService {

// Déclarer une clef de sécurité, en utilisant

@Value("${app.jwt.secret}")

**private** String SECRET\_KEY;

1. Il faut aussi créer la méthode getSigningKey
   * La clef de signature est utilisée pour signer numériquement le JWT
   * Une « secret key » est utilisée pour créer la partie « signature » du JWT
   * Permet de vérifier que l’expéditeur du JWT est celui qu’il prétend
   * Et être sûr que votre message n’a pas été modifié en route
   * La clef de signature est utilisée conjointement à l’algorithme de connexion spécifié dans l’entête
   * hmacShaKeyFor est l’un des algorithmes de connexion précédemment cités

…

//Signature transmise pour la création du jeton.

//Et chiffrer/déchiffrer les données du jeton

**private** Key getSignInKey() {

**byte**[] keyBytes = Decoders.***BASE64***.decode(SECRET\_KEY);

**return** Keys.*hmacShaKeyFor*(keyBytes);

}

1. Il faut des méthodes pour extraire du jeton tous les « claims » ou un particulier

// Extraire « claims » du jeton

**private** Claims extractAllClaims(String token) {

**return** Jwts.*parserBuilder*().setSigningKey(getSignInKey()).build().parseClaimsJws(token).getBody();

}

// Extraire 1 « claims » du jeton

**private** <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {

**final** Claims claims = extractAllClaims(token);

**return** claimsResolver.apply(claims);

}

1. Ajouter une méthode pour extraire le pseudo de l’utilisateur
   * C’est une donnée du « subject »

// Extraire le pseudo du jeton

**public** String extractUserName(String token) {

**return** extractClaim(token, Claims::getSubject);

}

1. Ajouter 2 méthodes pour générer le jeton JWT
   * La première permet de préciser une durée de vie, la clef de signature et le contexte utilisateur à conserver
   * La deuxième permet de créer le jeton avec les données vides

// Générer le jeton JWT

**public** String generateToken(Map<String, Object> extraClaims, UserDetails userDetails) {

**return** Jwts.*builder*().setClaims(extraClaims).setSubject(userDetails.getUsername())

.setIssuedAt(**new** Date(System.*currentTimeMillis*()))

.setExpiration(**new** Date(System.*currentTimeMillis*() + 1000 \* 60 \* 24))

.signWith(getSignInKey(), SignatureAlgorithm.***HS256***).compact();

}

**public** String generateToken(UserDetails userDetails) {

**return** generateToken(**new** HashMap<>(), userDetails);

}

1. Ajouter des méthodes pour gérer la validation du jeton

// Validation du jeton

**public** **boolean** isTokenValid(String token, UserDetails userDetails) {

**final** String username = extractUserName(token);

**return** (username.equals(userDetails.getUsername()) && !isTokenExpired(token));

}

**private** **boolean** isTokenExpired(String token) {

**return** extractExpiration(token).before(**new** Date());

}

**private** Date extractExpiration(String token) {

**return** extractClaim(token, Claims::getExpiration);

}

}

Nous avons terminé de créer la classe JwtService qui permet de créer, manipuler et valider le jeton

# Etape 6 : JWTAuthenticationFilter

Si on regarde notre schéma, l’élément qui intercepte les requêtes est JwtAuthenticationFilter

* Elle hérite d’OncePerRequestFilter
  + Comme son nom l’indique ; il sera utilisé à chaque appel
  + Regarder le code de cette classe depuis la JavaDoc
    - Elle hérite de la classe abstraite GenericFilterBean
* Elle doit manipuler 2 services
  + JwtService 🡪 qui gèrera le jeton JWT
  + UserDetailsService est une interface de Spring Security qui permet de manipuler l’utilisateur remonté de la base de données

1. Création de JwtAuthFilter
   * Lui Injecter JwtService et UserDetailsService

**package** fr.eni.demo.configuration.security;

**import** java.io.IOException;

**import** org.springframework.lang.NonNull;

**import** org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

**import** org.springframework.security.core.context.SecurityContextHolder;

**import** org.springframework.security.core.userdetails.\*;

**import** org.springframework.security.web.authentication.WebAuthenticationDetailsSource;

**import** org.springframework.stereotype.Component;

**import** org.springframework.web.filter.OncePerRequestFilter;

**import** jakarta.servlet.\*;

**import** jakarta.servlet.http.\*;

//Doit être active dès qu'il y a une requête

//Doit devenir un bean pour spring

@Component

**public** **class** JwtAuthenticationFilter **extends** OncePerRequestFilter {

// Injection de la couche BLL pour gérer le token

**private** JwtService jwtService;

// Injection de la couche BLL pour gérer les données de la DB

**private** UserDetailsService userDetailsService;

**public** JwtAuthenticationFilter(JwtService jwtService, UserDetailsService userDetailsService) {

**this**.jwtService = jwtService;

**this**.userDetailsService = userDetailsService;

}

…

1. Implémenter la méthode de l’interface : doFilterInternal
   * HttpServletRequest permet de manipuler la requête http request
   * HttServletResponse permet de manipuler la réponse http
   * Le fait d’avoir accès au protocole http, cela nous permet d’intercepter chaque requête http et de manipuler le comportement
   * Il nous est possible de capturer des données, changer les en-têtes et renvoyer des données, …
   * FilterChain, représente la chaîne d’événements
   * Les 3 paramètres ne doivent pas être null
     + Il est possible d’ajouter l’annotation @NonNull
   * D’après le schéma, la première chose à vérifier dans JwtAuthFilter est le JWT Token
     + Le jeton JWT est passé dans l’entête de la requête
   * Sur la classe HttpRequest il existe la méthode getHeader pour récupérer les données
     + La clef dans l’entête s’appelle « Authorization »
     + Les jetons JWT commence par "Bearer "
     + Structure de l’entête : « Authorization: Bearer <jeton> »
     + Donc si dans la value de l’entête "Bearer " n’est pas présent c’est qu’il n’est pas correct

…

**protected** **void** doFilterInternal(@NonNull HttpServletRequest request,

@NonNull HttpServletResponse response,

@NonNull FilterChain filterChain)

**throws** ServletException, IOException {

// vérifier le jeton JWT – il est passé dans l’en-tête

**final** String authHeader = request.getHeader("Authorization");

**final** String jwt;

// vérifier qu’il y a une donnée dans l’entête qui correspond à Authorization

// l’entête contient Bearer <jeton> SINON erreur

// Sinon laissé le comportement suivre son cours

**if** (authHeader == **null** || !authHeader.startsWith("Bearer ")) {

filterChain.doFilter(request, response);

**return**;

}

// Il y a un JWT – il faut l’extraire

jwt = authHeader.substring(7);// 7 correspond à Bearer

}

}

1. Il nous faut terminer le processus de validation « Validate JWT » dans le filtre JwtAuthFilter
   * En appelant les méthodes de validation de JwtService
   * Et il faut aussi appeler la validation de DB, il faut pour cela utiliser UserDetailsService
   * Mise à jour du contexte de sécurité avec le jeton créé
   * Et pour finir la méthode permettre de transmettre le contexte du filtre

@Override

**protected** **void** doFilterInternal(@NonNull HttpServletRequest request,

@NonNull HttpServletResponse response,

@NonNull FilterChain filterChain)

**throws** ServletException, IOException {

// vérifier le jeton JWT

**final** String authHeader = request.getHeader("Authorization");

**final** String jwt;

**if** (authHeader == **null** || !authHeader.startsWith("Bearer ")) {

filterChain.doFilter(request, response);

**return**;

}

jwt = authHeader.substring(7);// 7 correspond à Bearer

// Vérification de l'utilisateur

**final** String userEmail = jwtService.extractUserName(jwt);// Extraire du jeton JWT

// Validation des données par rapport à la DB

**if** (userEmail != **null** && SecurityContextHolder.*getContext*().getAuthentication() == **null**) {

// Check in DB

UserDetails userDetails = **this**.userDetailsService.loadUserByUsername(userEmail);

// Validation du jeton JWT

**if** (jwtService.isTokenValid(jwt, userDetails)) {

// Gestion du contexte de sécurité de l’utilisateur

//Création d'un nouveau jeton avec les informations et les rôles de l'utilisateur

UsernamePasswordAuthenticationToken authToken = **new**

UsernamePasswordAuthenticationToken(userEmail, **null**,

userDetails.getAuthorities());

//Transmettre les détails de la demande d’origine

authToken.setDetails(**new**

WebAuthenticationDetailsSource().buildDetails(request));

//Mise à jour du contexte de sécurité

SecurityContextHolder.*getContext*().setAuthentication(authToken);

}

}

filterChain.doFilter(request, response);

}

}

# Etape 7 : Contexte de Spring Security

D’après nos schémas :

* Activer le JwtAuthenticationFilter comme premier filtre avec celui du contexte de sécurité
* Définir la solution de reconnaissance de l’utilisateur à connecter (JwtAppConfig)

1. Ajouter dans la classe EniSecurityConfig
   * Injection du filtre : JwtAuthenticationFilter
   * Injection d’AuthenticationProvider 🡪 il correspond au contexte de la classe JwtAppConfig

**package** fr.eni.demo.security;

**import** org.apache.commons.logging.\*;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.context.annotation.\*;

**import** org.springframework.http.HttpMethod;

**import** org.springframework.security.authentication.\*;

**import** org.springframework.security.config.annotation.web.builders.HttpSecurity;

**import** org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

**import** org.springframework.security.config.http.SessionCreationPolicy;

**import** org.springframework.security.web.SecurityFilterChain;

**import** org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

**import** jakarta.servlet.Filter;

@Configuration

@EnableWebSecurity

**public** **class** EniSecurityConfig {

**protected** **final** Log logger = LogFactory.*getLog*(getClass());

@Autowired

**private** Filter jwtAuthenticationFilter;

@Autowired

**private** AuthenticationProvider authenticationProvider;

…

}

1. Supprimer de la classe EniSecurityConfig ; la méthode userDetailsManager
   * Elle n’est plus utilisée
   * Elle est remplacée par AuthenticationManager et AuthenticationProvider
   * Il faut ajouter un URL racine qui sera valide pour tout le monde pour l’authentification

/\*\*

\* Restriction des URLs selon la connexion utilisateur et leurs rôles

\*/

@Bean

SecurityFilterChain filterChain(HttpSecurity http) **throws** Exception {

http.authorizeHttpRequests(auth -> {

auth

//Permettre l'accès à l'URL racine à tout le monde

.requestMatchers("/eniecole").permitAll()

.requestMatchers("/eniecole/auth/\*\*").permitAll()

1. Dans la méthode filterChain
   * Remplacer l’authentification « httpBasic »
   * Par l’appel de notre bean authentificationProvider
   * Activation du filtre JWT
   * Nous voulons que la session soit Stateless, donc nous le forçons dans le filtre

/\*\*

\* Restriction des URLs selon la connexion utilisateur et leurs rôles

\*/

@Bean

SecurityFilterChain filterChain(HttpSecurity http) **throws** Exception {

http.authorizeHttpRequests(auth -> {

auth

// Permettre l'accès à l'URL racine à tout le monde

.requestMatchers("/eniecole").permitAll()

.requestMatchers("/eniecole/auth/\*\*").permitAll()

// Permettre aux rôles EMPLOYE et ADMIN de manipuler les URLs en GET

.requestMatchers(HttpMethod.***GET***, "/eniecole/employes/\*\*").hasAnyRole("EMPLOYE", "ADMIN")

// Restreindre la manipulation des méthodes POST, PUT, PATCH et DELETE au rôle

// ADMIN

.requestMatchers(HttpMethod.***POST***, "/eniecole/employes").hasRole("ADMIN")

.requestMatchers(HttpMethod.***PUT***, "/eniecole/employes").hasRole("ADMIN")

.requestMatchers(HttpMethod.***PATCH***, "/eniecole/employes").hasRole("ADMIN")

.requestMatchers(HttpMethod.***DELETE***, "/eniecole/employes").hasRole("ADMIN")

// Toutes autres url et méthodes HTTP ne sont pas permises

.anyRequest().denyAll();

});

// Désactivé Cross Site Request Forgery

// Non préconisé pour les API REST en stateless. Sauf pour POST, PUT, PATCH et

// DELETE

http.csrf(csrf -> {

csrf.disable();

});

//Connexion de l'utilisateur

http.authenticationProvider(authenticationProvider);

//Activer le filtre JWT et l'authentication de l'utilisateur

http.addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.**class**);

// Session Stateless

http.sessionManagement(session -> {

session.sessionCreationPolicy(SessionCreationPolicy.***STATELESS***);

});

**return** http.build();

}

}

# Etape 8 : Contrôleur et service d’authentification

D’après notre schéma d’authentification, il nous reste à créer : AuthenticationController et AuthenticationService

* Le contrôleur AuthenticationController permettra de récupérer les données d’authentification
  + L’URL racine déclaré ; dans la classe EniSecurityConfig : /eniecole/auth/\*\*
  + Il faut qu’il manipule la base de données, au travers d’un service AuthenticationService
* Il déclare une méthode sur Post pour gérer l’authentification
* Pour cette méthode il faut déclarer
  + AuthenticationResponse 🡪 POJO contenant uniquement le jeton JWT à retourner
  + AuthenticationRequest 🡪 POJO représentant les données du formulaire de connexion

1. Créer AuthenticationResponse et AuthenticationRequest

**package** fr.eni.demo.security.jwt;

**import** lombok.\*;

@Data

@AllArgsConstructor

@NoArgsConstructor

@EqualsAndHashCode(of = "pseudo")

**public** **class** AuthenticationRequest {

**private** String pseudo;

**private** String password;

}

**package** fr.eni.demo.security.jwt;

**import** lombok.\*;

@Data

@AllArgsConstructor

@NoArgsConstructor

@EqualsAndHashCode

**public** **class** AuthenticationResponse {

**private** String token;

}

1. Créer AuthenticationService

* Il permet de manipuler les données en base
  + Pour cela, il faut injecter notre UserRepository pour récupérer
  + D’authentifier un utilisateur
    - Pour cela, il faut injecter [org](eclipse-javadoc:%E2%98%82=demo-jwt2/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/org.springframework.security%5C/spring-security-core%5C/6.0.3%5C/d3b35481b651c777fd7366b8a496a1d67098fed3%5C/spring-security-core-6.0.3.jar=/gradle_used_by_scope=/main,test=/%3Corg).[springframework](eclipse-javadoc:%E2%98%82=demo-jwt2/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/org.springframework.security%5C/spring-security-core%5C/6.0.3%5C/d3b35481b651c777fd7366b8a496a1d67098fed3%5C/spring-security-core-6.0.3.jar=/gradle_used_by_scope=/main,test=/%3Corg.springframework).[security](eclipse-javadoc:%E2%98%82=demo-jwt2/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/org.springframework.security%5C/spring-security-core%5C/6.0.3%5C/d3b35481b651c777fd7366b8a496a1d67098fed3%5C/spring-security-core-6.0.3.jar=/gradle_used_by_scope=/main,test=/%3Corg.springframework.security).[authentication](eclipse-javadoc:%E2%98%82=demo-jwt2/C:%5C/Users%5C/abaille%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/org.springframework.security%5C/spring-security-core%5C/6.0.3%5C/d3b35481b651c777fd7366b8a496a1d67098fed3%5C/spring-security-core-6.0.3.jar=/gradle_used_by_scope=/main,test=/%3Corg.springframework.security.authentication).AuthenticationManager
    - L’utiliser pour authentifier l’utilisateur avec son pseudo et mot de passe
* authenticate 🡪 cette méthode retourne une exception si l’authentification échoue
  + Si OK 🡪 Récupérer l’utilisateur depuis notre UserRepository
  + Transmettre une exception si non trouvé avec orElseThrow()
  + Si tout est OK 🡪 Création d’un jeton JWT associé
    - Injecter JwtService pour accéder à la création d’un jeton JWT
    - Et injecter ce jeton dans l’AuthenticationResponse

**package** fr.eni.demo.security.jwt;

**import** org.springframework.security.authentication.\*;

**import** org.springframework.stereotype.Service;

**import** lombok.AllArgsConstructor;

@AllArgsConstructor

@Service

**public** **class** AuthenticationService {

**private** UserInfoRepository userInfoRepository;

**private** AuthenticationManager authenticationManager;

**private** JwtService jwtService;

**public** AuthenticationResponse authenticate(AuthenticationRequest request) {

authenticationManager.authenticate(

**new** UsernamePasswordAuthenticationToken(request.getPseudo(), request.getPassword()));

UserInfo user = userInfoRepository.findById(request.getPseudo()).orElseThrow();

String jwtToken = jwtService.generateToken(user);

AuthenticationResponse authResponse = **new** AuthenticationResponse();

authResponse.setToken(jwtToken);

**return** authResponse;

}

}

1. Créer AuthenticationController

**package** fr.eni.demo.security.jwt;

**import** org.springframework.http.ResponseEntity;

**import** org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/eniecole/auth")

**public** **class** AuthenticationController {

**private** **final** AuthenticationService authenticationService;

**public** AuthentificationController(AuthenticationService authenticationService) {

**this**.authenticationService = authenticationService;

}

@PostMapping

**public** ResponseEntity<AuthenticationResponse> register(@RequestBody AuthenticationRequest request) {

**return** ResponseEntity.*ok*(authenticationService.authenticate(request));

}

}

Nous avons à présent l’ensemble des classes, beans de nos schémas

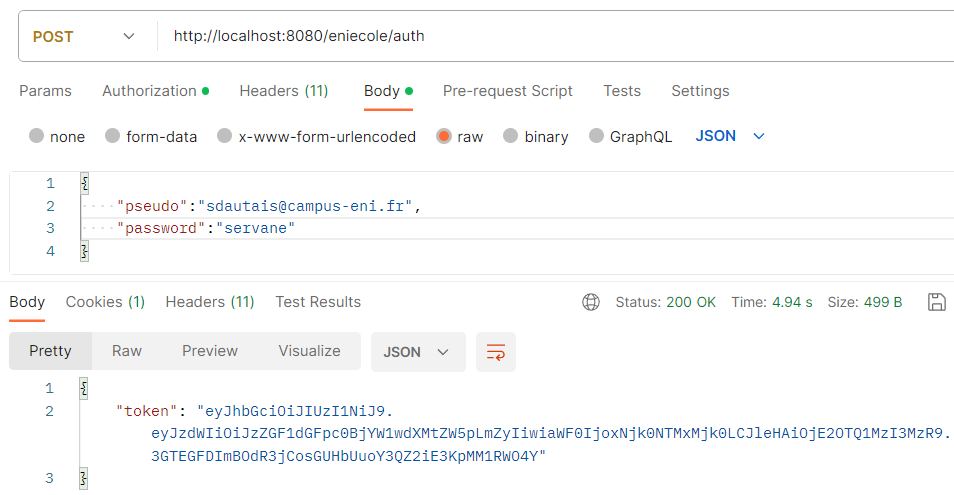
# Exécution :

Pour manipuler notre sécurité avec jeton JWT

1. Créer un jeton JWT en connectant un utilisateur
2. Copier le jeton JWT pour manipuler les URLs de l’application

## Exécution avec sdautais@campus-eni.fr et servane

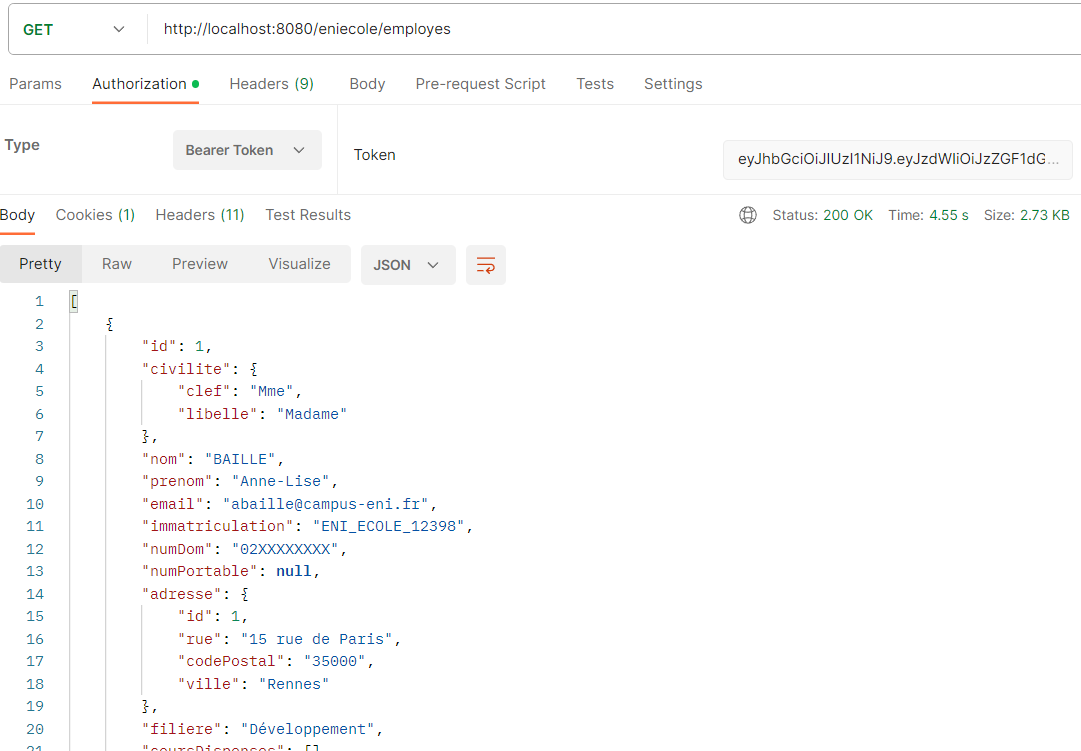
* L’URL d’authentification est http://localhost:8080/eniecole/auth
* Dans Postman, créer une nouvelle requête
  + POST avec un formulaire



* Nous avons un jeton JWT de généré

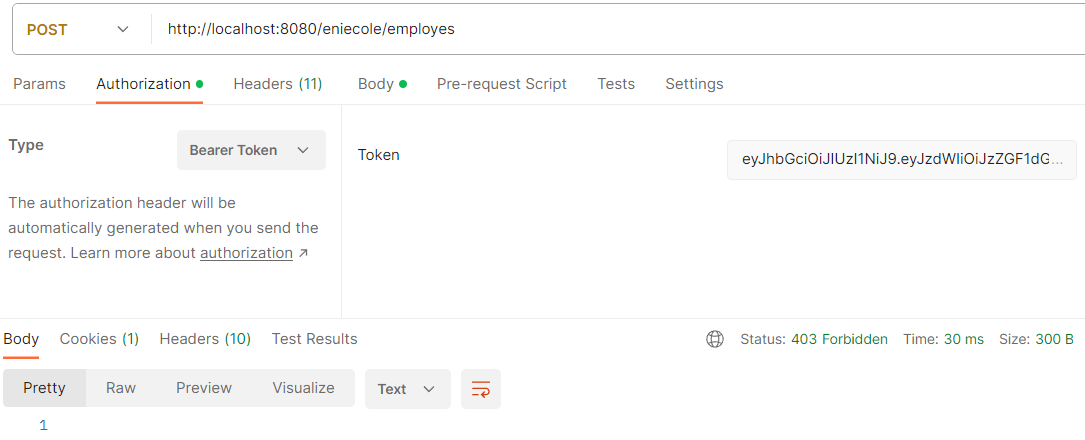
## Exécution du Get /eniecole/employes avec le jeton JWT de servane

* Création d’une nouvelle requête sous Postman
  + Ajouter une Authorization de type «Bearer Token »



* + Nous pouvons accéder aux données de l’application avec ce jeton

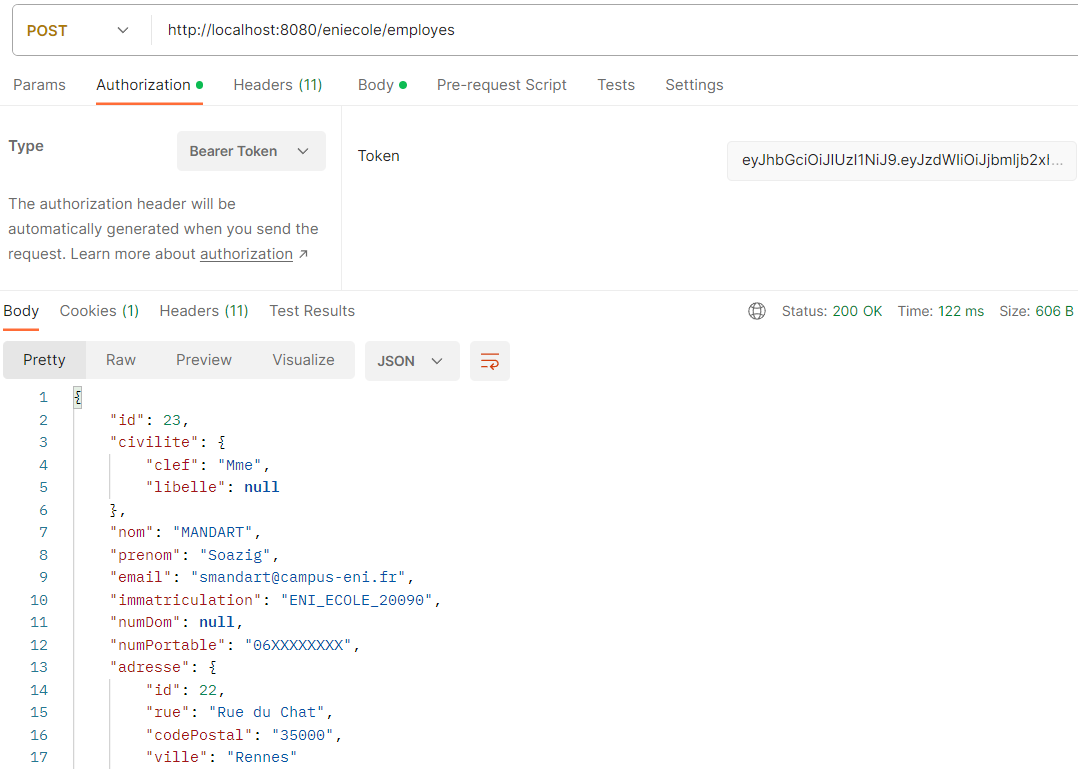
## Exécution avec le jeton de servane d’une sauvegarde d’Employe



Servane a le rôle EMPLOYE ; elle n’a toujours pas le droit d’accéder à ces URLs

## Création d’un jeton pour cnicolas@campus-eni.fr et cédric et tester la sauvegarde d’un Employe

* Il a un rôle ADMIN
* Donc en créant un jeton avec cet utilisateur
  + La sauvegarde est opérationnelle



Notre application gère les appels d’API par jeton JWT.

Vous pouvez évidemment tester les cas d’erreurs.