SPRING SECURITY

JWT

Présentation

INTRODUCTION (1)

Spring Security est un framework open-source développé par la communauté Spring pour offrir une sécurité robuste aux applications Java.

Il fournit des fonctionnalités telles que l'authentification, l'autorisation, la protection, la gestion de sessions, etc.

Il est hautement configurable et s'intègre parfaitement avec d'autres composants de l'écosystème Spring, comme le développement WEB avec Spring Boot.

INTRODUCTION (2)

- **Authentification** : Spring Security offre diverses options pour l'authentification, y compris l'authentification basée sur les formulaires ou l'authentification basée sur les jetons (token-based).
- **Autorisation** : Il permet de configurer des règles d'autorisation pour définir qui peut accéder à quelles ressources de l'application. Soit par un système de rôle, soit par un système d'accès.
- **Protection contre les attaques** : Spring Security fournit des protections contre les attaques courantes telles que CSRF, XSS, injections SQL, etc.
- **Gestion des sessions** : Il offre des fonctionnalités pour gérer les sessions utilisateur, y compris la gestion des sessions HTTP et la prise en charge de l'authentification à deux facteurs.
- **Intégration transparente** : Il s'intègre facilement avec d'autres composants Spring et s'appuie sur les fonctionnalités de gestion de la sécurité fournis par les conteneurs de servlet Java EE.

CSRF

Dans une attaque CSRF (ou **Cross-Site Request Forgery)**, un attaquant exploite la confiance que le site accorde aux utilisateurs authentifiés.

L'attaquant incite l'utilisateur à effectuer une action non intentionnelle sur un site web auquel il est authentifié.

Par exemple, l'utilisateur peut être incité à cliquer sur un lien malveillant qui envoie une requête à un site web en arrière-plan, utilisant les informations d'authentification stockées dans les cookies de l'utilisateur.

Pour se protéger contre les attaques CSRF, les développeurs utilisent souvent des tokens CSRF ou des stratégies telles que SameSite pour les cookies.

XSS

Dans une attaque XSS (ou **Cross-Site Scripting**), un attaquant insère du code JavaScript malveillant dans une page web, souvent via des champs de formulaire ou des URL.

Lorsque d'autres utilisateurs visitent cette page web, le code JavaScript est exécuté dans leur navigateur, permettant à l'attaquant de voler des informations sensibles, de modifier le contenu de la page, ou même de prendre le contrôle de la session utilisateur.

Pour se protéger contre les attaques XSS, il est recommandé d'échapper correctement les données entrantes et d'utiliser des en-têtes HTTP tels que Content Security Policy (CSP).

INJECTION SQL

Dans une attaque par injection SQL, un attaquant insère du code SQL malveillant dans les entrées d'un formulaire ou d'une URL, dans le but de manipuler la base de données sous-jacente.

Cela peut permettre à l'attaquant d'extraire des informations sensibles, de modifier ou supprimer des données, voire d'exécuter des commandes système sur le serveur.

Pour prévenir les attaques par injection SQL, il est essentiel d'utiliser des requêtes paramétrées ou des ORM (Object-Relational Mapping) sécurisés.

CLICKJACKING

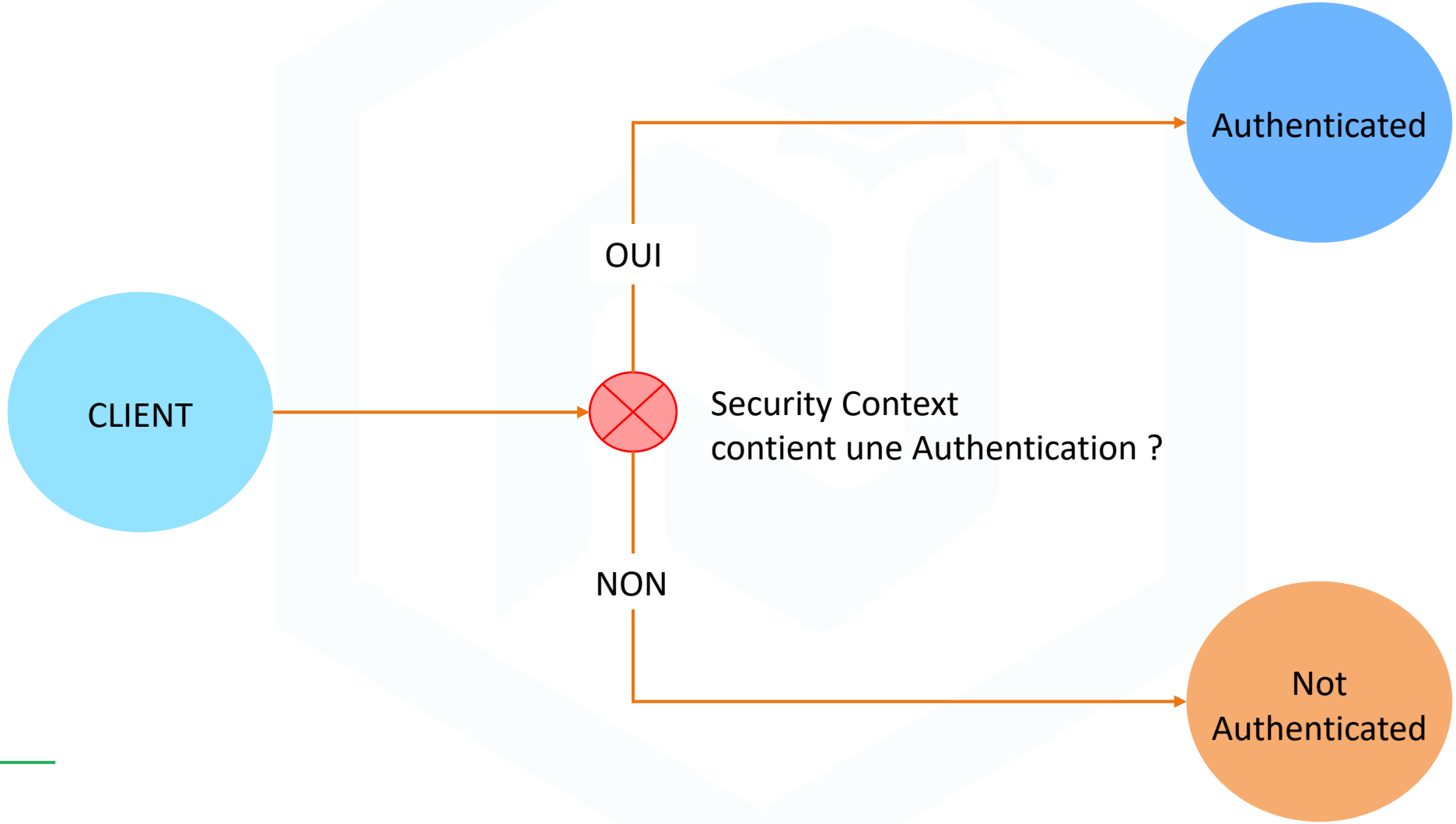
L'attaque par clickjacking consiste à tromper un utilisateur en cliquant sur un élément d'une page web à son insu.

Cela est généralement réalisé en superposant une page web transparente ou opaque sur une autre page web légitime, en incitant l'utilisateur à cliquer sur des éléments qu'il ne voit pas.

Les attaquants peuvent ainsi inciter les utilisateurs à effectuer des actions non intentionnelles, telles que l'installation de logiciels malveillants ou le partage d'informations sensibles.

Les moyens de prévention incluent l'utilisation de l'en-tête HTTP X-Frame-Options ou le Content Security Policy (CSP).

SECURITY CONTEXT (1)



SECURITY CONTEXT (2)

Security Context est un objet utilisé en spring security qui contient les informations de sécurité spécifique de l'utilisateur actuellement authentifié (tel que username et rôles).

Cet objet n'a de vie que dans une exécution d'une requête HTTP. C'est-à-dire que le client, s'il veut accéder à des ressources sécurisées, devra prouver son authentification à chaque fois qu'il appelle le server.

Spring automatise le traitement de non-autorisation (envoie d'une erreur au client), mais, en plus de pouvoir modifier ce comportement, on pourra récupérer à tout moment ces informations dans l'application.
Utile notamment pour savoir qui fait la requête pour tracer son exécution ou dans le traitement d'une requête pour accéder à une ressource personnelle.

JWT

INTRODUCTION

JWT ou Json Web Token est un standard (RFC 7519) qui définit un moyen compact et autonome pour représenter de manière sécurisé les informations entre deux parties sous forme d'objet JSON.

Ces tokens (ou jetons en Fr) sont auto-suffisants, ce qui signifie qu'ils portent toutes les informations nécessaires à la vérification de l'authenticité et de l'intégrité des données qu'ils transportent.

Leur format compact les rend faciles à transmettre via des en-têtes HTTP ou des paramètres d'URL.

COMPOSITION

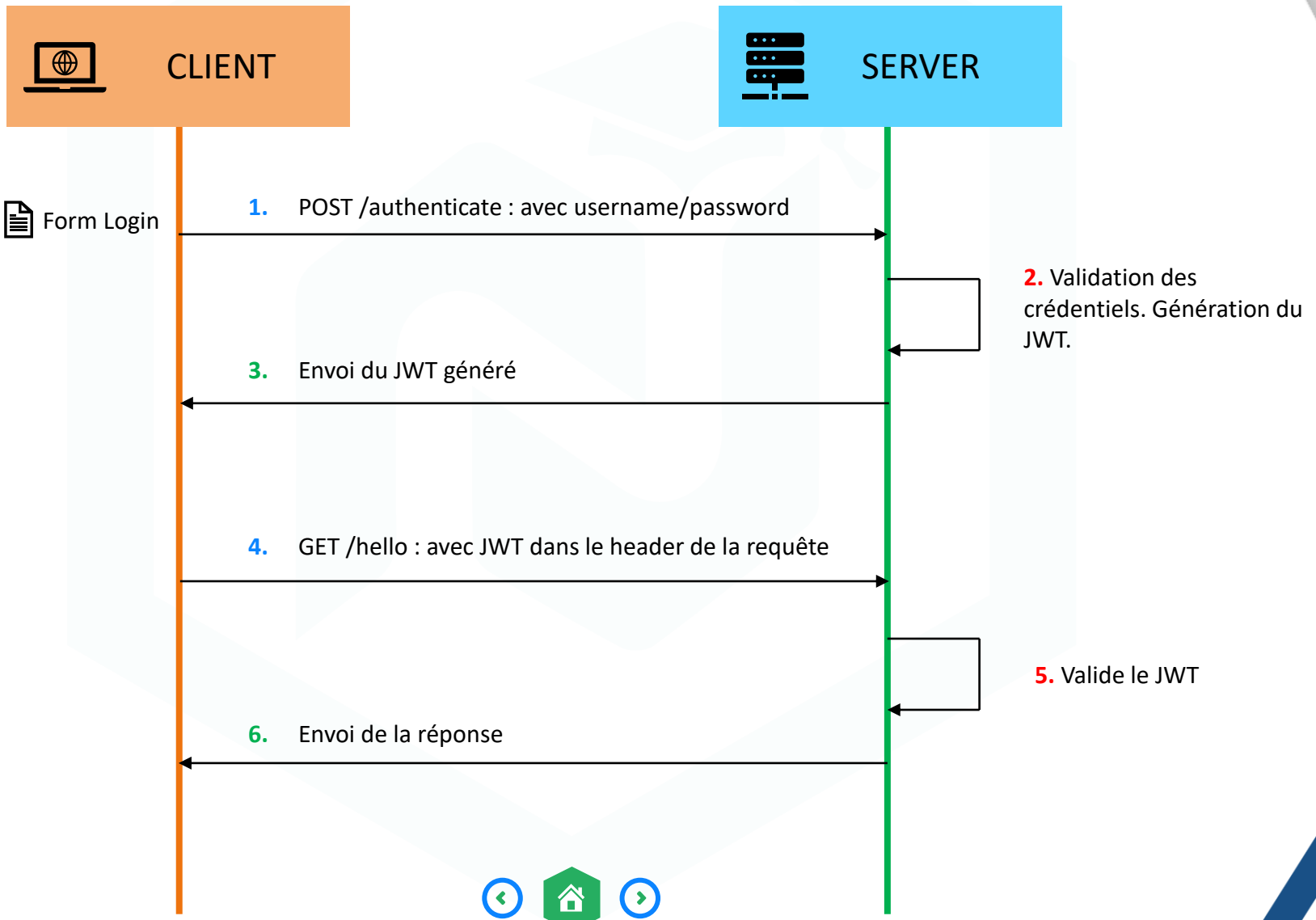
```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

- **Header** : Contient généralement deux parties, le type de token et l'algorithme de hachage utilisé pour signé le token.

- **Payload** : Contient des données (claims) que le token JWT transporte. Il peut s'agir de données d'identité de l'utilisateur (comme l'id, le nom, le rôle, etc.) ou d'autres informations pertinentes pour l'application. Le payload n'est pas crypté et n'est pas censé transporté des données sensibles.

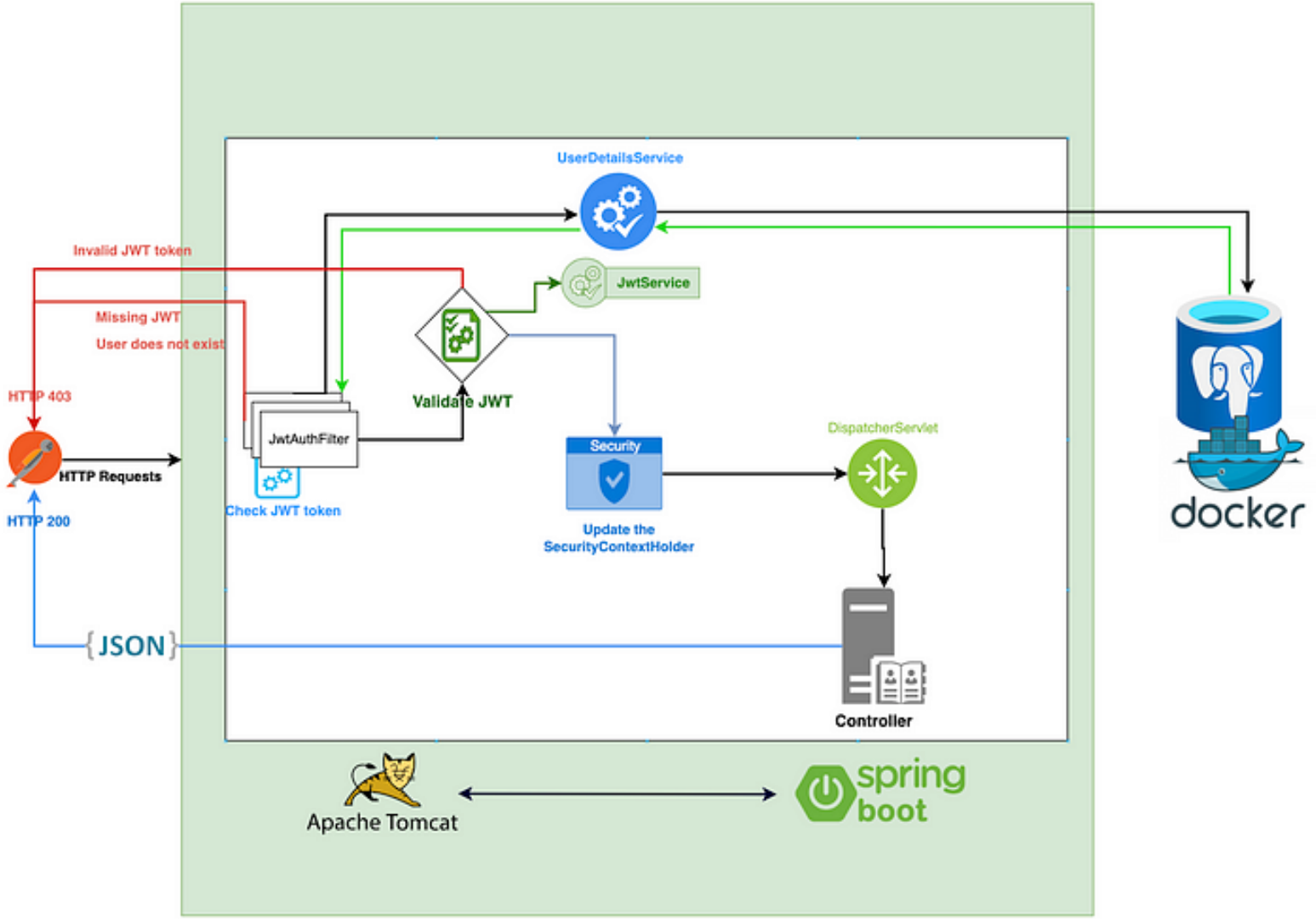
- **Signature** : Est utilisée pour vérifier que le JWT n'a pas été modifié en cours de route et qu'il provient d'une source fiable. La signature est calculée en utilisant le header encodé, le payload encodé, un secret ou clé privée, et un algorithme de hachage spécifié en header.

THÉORIE


CLIENT
SERVER
Form Login
1. POST /authenticate : avec username/password
2. Validation des crédentiels. Génération du JWT.
3. Envoi du JWT généré
4. GET /hello : avec JWT dans le header de la requête
5. Valide le JWT
6. Envoi de la réponse

Pratique

ARCHITECTURE (1)



ARCHITECTURE (2)

- **JwtAuthFilter** : Filtre personnalisé interceptant toutes les requêtes HTTP entrante pour vérifier et extraire les tokens JWT. Son rôle principal est de gérer l'authentification basée sur les JWT. On pourra y incorporer tous les checks personnalisés sur le JWT et l'authentification de l'utilisateur.

- **UserDetailsService** : Interface fournit par spring-security. Elle est utilisée pour récupérer un utilisateur à partir d'une source de donnée. Cette interface devra être implémentée pour redéfinir la méthode: *loadUserByUsername*.

- **JwtService** : Il s'agit d'une classe permettant de créer des méthodes de création de token, de récupération de claims, de validations de token JWT, etc.

- **SecurityContextHolder** : Classe fournit par Spring Security pour stocker les informations de l'utilisateur actuel de la requête.

- **DispatcherServlet** : Le « front controller » permettant de redirigé vers le controller que la requête essaye d'appeler.

POM.XML

On doit ajouter la dépendance de spring boot starter security dans le pom.xml.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Si on lance le projet on peut remarquer que la dépendance a automatiquement sécurisé l'application:

```
Using generated security password: 84e617e2-5cb3-4831-9fb5-dcdfe1b633b8

This generated password is for development use only. Your security configuration must be updated before running your application in production.
```

RUN

Au run de l'application Spring Security s'auto-configure pour implémenter une base de sécurité sur l'application, rendant tous les endpoints privés qui auront besoin d'une authentification pour pouvoir y accéder.

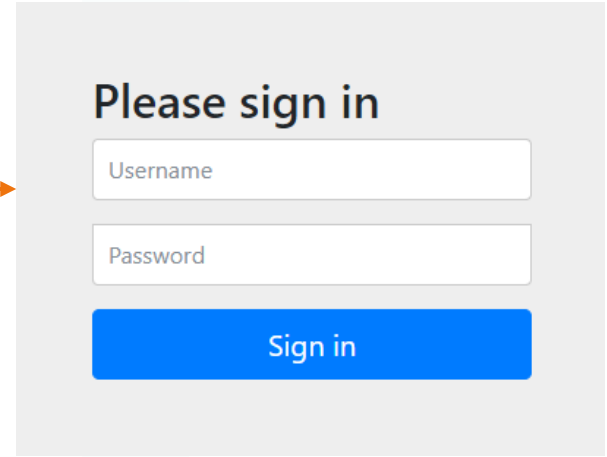
```java
@RestController
public class MainController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello World";
    }

}
```

Navigateur : http://localhost:8080/hello

Redirection automatique sur /login



Si on utilise le username **user** et le mot de passe affiché dans la console. On est connecté et on a accès à notre requête */hello*. On pourra aller sur */logout* pour se déconnecter.

CRÉATION DES CLASSES (1)

SECURITYCONFIGURATION

Dans son propre package, on créera une classe *SecurityConfiguration* qui permettra de reconfigurer la configuration de spring-security pour implémenter la nôtre.

- `@Configuration` : Marque une classe comme source de définition de beans (objet géré par IoC)
- `@EnableWebSecurity` : Active la configuration de la sécurité web.

```
/**
 * Classe utilisée pour configurer la sécurité
 * de l'application selon les besoins.
 */
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfiguration {

}
```

CRÉATION DES CLASSES (2)

JWTAUTHFILTER

On créera une classe *SecurityConfiguration* qui permettra de reconfigurer la configuration de spring-security pour implémenter la nôtre.

- `@Component` : Marque une classe comme étant un bean.
- `OncePerRequestFilter` : Classe abstraite utilisé pour créer des filtres de requête. Elle s'exécute une fois par requête.
- `doFilterInternal` : Méthode de la classe OncePerRequestFilter qui s'activera à chaque requête.

```java
/**
 * Classe utilisée pour filtrer les requêtes
 * et vérifier si l'utilisateur est authentifié.
 */
@Component
@RequiredArgsConstructor
public class JwtAuthFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(
            HttpServletRequest request,
            HttpServletResponse response,
            FilterChain filterChain
    ) throws ServletException, IOException {
        //todo: implement checking for jwt token
    }

}
```

CRÉATION DES CLASSES (3)

JWTSERVICE

On crée une classe *JwtService* qui nous permettra de créer nos méthodes de traitement de token, tel que:

- **generateToken** : méthode permettant de créer un token JWT correspondant à un utilisateur.
- **getUserFromToken** : récupérer le user dans un token JWT stipulé.
- **validateToken** : envoi un boolean spécifiant si le token est valide ou non.
- **isTokenExpired** : envoi un boolean spécifiant si le token est expiré.

```
/**
 * Classe utilisée pour gérer les tokens JWT.
 */
@Service
public class JwtService {

}
```

CRÉATION DES CLASSES (4)

USERDETAILSSERVICEIMPL (1) : CLASSE

On crée une classe *UserDetailsServiceImpl* permettant de créer une méthode pour récupérer un user de la BDD.

- `UserDetailsService` : Interface provenant de Spring Security.
- `loadUserByUsername` : Méthode à redéfinir provenant de *UserDetailsService* qui permettra de récupérer un *UserDetail*.

```java
/**
 * Classe utilisée pour récupérer les informations de l'utilisateur
 * depuis la base de données pour l'authentification.
 */
@Service
@RequiredArgsConstructor
public class UserDetailsServiceImpl implements UserDetailsService {

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        //todo: get user from database
        return null;
    }

}
```

CRÉATION DES CLASSES (4)

USERDETAILSSERVICEIMPL (2) : BEAN

Alternativement, depuis les versions récentes de Spring Security, on peut définir directement un bean permettant de redéfinir *UserDetailsService*.

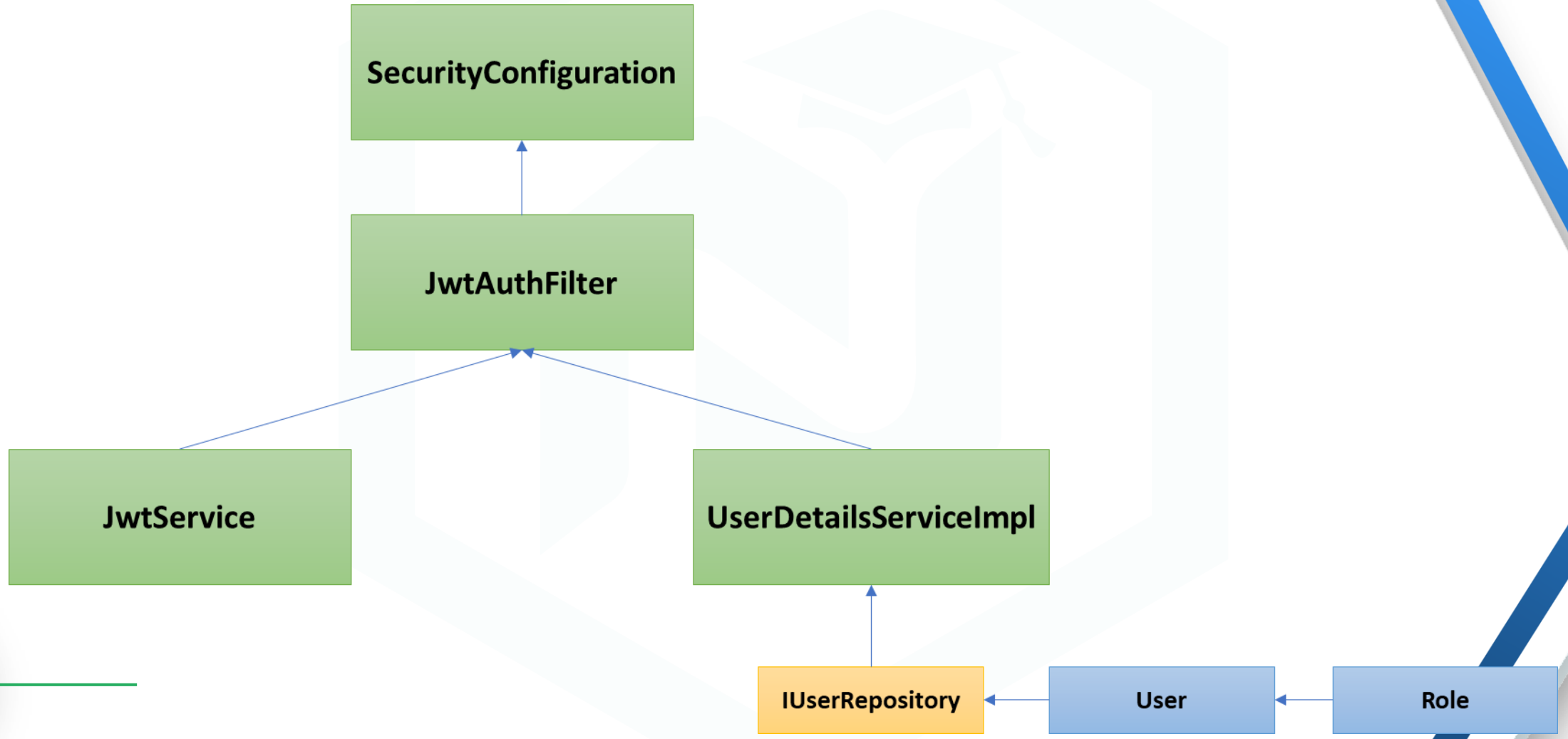
```
@Configuration
public class AppConfig {

    @Bean
    public UserDetailsService userDetailsService() {
        return username -> {
            // todo: get user from database
            return null;
        };
    }

}
```

CRÉATION DES CLASSES (5)

ARCHITECTURE



USERDETAILS

UserDetails est une interface provenant de Spring Security, utilisée pour représenter les détails d'un utilisateur dans le système de sécurité. Elle contient:

- *Username*
- *Password*
- Collection de *GrantedAuthority*
- Autres:
  - Compte expiré
  - Compte fermé
  - Compte désactivé
  - Crédentiel expiré

Cette interface est utilisée pour encapsuler les détails d'authentification d'un utilisateur dans un objet qui peut être facilement intégré dans le système de gestion d'authentification de Spring Security, ainsi qu'être implémenté dans une Entité pour la BDD.

ENTITÉ USER

```java
@Entity
@Table(name = "users")
public class User implements UserDetails {

    @Id
    private String username;
    private String password;

    @Override
    public Collection<Role> getAuthorities() {
        return new ArrayList<>();
    }

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public String getUsername() {
        return username;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

Pour stocker l'utilisateur en BDD on crée l'entité *User* et pour aider à son utilisation dans spring security on implémente *UserDetails*.

Il nous faut:

- *Username* : permettant de stocker un identifiant unique çà l'utilisateur. Peut-être un nom, un email, un code référence, etc. (peut également être l'ID).

- *Password*: permettant de stocké le mot de passe en BDD qui sera hashé.

- Autres: n'importe quel attribut permettant de gestionner le user (email, activé, téléphone, jeton d'activation ou de reset mot de passe, etc.)

- Les getters overridé de *UserDetails* **obligatoire** (important de bien faire attention aux getters override, et mettre un *return* approprié si on ne les utilise pas)

JwtService

POM.XML

Pour pouvoir gérer le token JWT, on aura besoin de dépendances jsonwebtoken à ajouter dans le pom.xml

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.12.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.12.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.12.5</version>
</dependency>
```

PROPERTIES

On a besoin de certaines propriétés dans l'application:

- **Secret** : qui servira de clé de signature pour notre token. Permettant de savoir si un token a été créé par notre application. Pour cela il est conseillé de générer un secret long et compliqué en production (que seul le(s) responsable(s) du server a(ont) accès)

- **Expiration** : La durée de vie du token en millisecondes

```yaml
application:
  security:
    jwt:
      secret: mysecret # secret key for JWT, should be a long, random string
      expiration: 86400000 # 24 hours
```

On pourra les importer dans *JwtService* grâce à l'annotation de Spring *@Value*

```java
@Value("${application.security.jwt.secret}")
private String secret;

@Value("${application.security.jwt.expiration}")
private long expiration;
```

RÉCUPÉRATION DES CLAIMS

Grâce à la librairie de *jsonwebtoken* on a des fonctionnalités toutes simple pour récupérer les claims.

Pour cela il nous faut la secret key du token.

En cas d'erreur on pourra throw une exception qui pourra être gérer dans un *ExceptionHandler*.

```
public Claims getAllClaimsFromToken(String token) {
    return Jwts
        .parser()
        .verifyWith(getSecretKey())
        .build()
        .parseSignedClaims(token)
        .getPayload();

}

private Key getSignInKey() {
    return Keys.hmacShaKeyFor(secret.getBytes(StandardCharsets.UTF_8));
}
```

RÉCUPÉRATION DU USERNAME

Dans le claims, le username est mis sous le nom *subject*, on le récupère simplement avec le getter.

```
public String getUsernameFromToken(String token) {
    final Claims claims = getAllClaimsFromToken(token);
    return claims.getSubject();
}
```

CHECK DE L'EXPIRATION

En usant du secret key pour vérifier la signature du token en même temps, on récupère l'expiration du token, et on vérifie qu'elle est après la date à l'instant T.

```
public boolean isTokenExpired(String token) {
    final Date claimExpiration = getAllClaimsFromToken(token).getExpiration();

    return claimExpiration.before(new Date());
}
```

CHECK DE L'EXPIRATION

Pour vérifier le token on vérifiera:

1. Username dans le token correspond au username d'un user récupérer de la BDD
2. Token n'est pas expiré

```
public boolean validateToken(String token, UserDetails userDetails) {
    final String username = getUsernameFromToken(token);
    final boolean isUsernameValid = username.equals(userDetails.getUsername());
    final boolean isTokenExpired = isTokenExpired(token);
    return isUsernameValid && !isTokenExpired;
}
```

UserDetailsService

CLASSE

Grâce à une méthode préalablement créé dans le *UserRepository* on récupère l'utilisateur grâce à son username et on le retourne si existant, ou retourne null sinon (**important** de ne pas throw une erreur ici).

```java
/**
 * Classe utilisée pour récupérer les informations de l'utilisateur depuis la base de données pour l'authentification.
 */
@Service
@RequiredArgsConstructor
public class UserDetailsServiceImpl implements UserDetailsService {

    private final IUserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        return userRepository.findByUsername(username).orElse(null);
    }

}
```

BEAN

Comme vu précédemment, on peut créer un bean pour implémenter *UserDetailsService*.

```
@Configuration
@RequiredArgsConstructor
public class AppConfig {

    private final IUserRepository userRepository;

    @Bean
    public UserDetailsService userDetailsService() {
        return username -> userRepository.findByUsername(username).orElse(null);
    }

}
```

JwtAuthFilter

JWTAUTHFILTER (1)

JwtAuthFilter va nous permettre de définir comment on valide l'authentification de l'utilisateur via son token.
Donc pour cela il nous faudra valider les choses suivantes :

- Token présent dans la requête
  - Le token doit être dans les headers
  - Le header se nomme « Authorization »
  - La valeur commence par « Bearer »
- Username présent dans le payload du token
- Vérifier que l'utilisateur est en BDD
- Vérifier la validité du token

Une fois cela fait, on pourra créer l'authentification.

JWTAUTHFILTER (2)

Dans la méthode *doFilterInternal* on commence à mettre nos checks:

1. On récupère le header nommé *Authorization*

2. Usant de lang3 pour simplifier les checks, on vérifie que la valeur:
    1. Existe
    2. N'est pas vide
    3. Commence par *Bearer et un espace après*

3. Si ce n'est pas le cas on arrête le filtrage sans authentifier l'utilisateur

```java
final String bearerToken = request.getHeader("Authorization");
if (StringUtils.isBlank(bearerToken) || !bearerToken.startsWith("Bearer ")) {
    filterChain.doFilter(request, response);
    return;
}
```

JWTAUTHFILTER (3)

1. On récupère le token dans la valeur, en enlevant les 7 premiers caractères correspondant à «Bearer »

2. On vérifie que la valeur:
    1. Existe
    2. N'est pas vide

3. Si ce n'est pas le cas on arrête le filtrage sans authentifier l'utilisateur

```java
final String token = bearerToken.substring(7);
if (StringUtils.isBlank(token)) {
    filterChain.doFilter(request, response);
    return;
}
```

JWTAUTHFILTER (4)

1. On récupère le username dans le token grâce à la méthode prévue dans *JwtService*

2. On vérifie que la valeur:
   1. Existe
   2. N'est pas vide

3. Si ce n'est pas le cas on arrête le filtrage sans authentifier l'utilisateur

```java
final String username = jwtService.getUsernameFromToken(token);
if (StringUtils.isBlank(username)) {
    filterChain.doFilter(request, response);
    return;
}
```

JWTAUTHFILTER (5)

1. On récupère le username dans le token grâce à la méthode prévue dans *JwtService*

2. On vérifie que la valeur:
    1. Existe
    2. N'est pas vide

3. Si ce n'est pas le cas on arrête le filtrage sans authentifier l'utilisateur

```java
final String username = jwtService.getUsernameFromToken(token);
if (StringUtils.isBlank(username)) {
    filterChain.doFilter(request, response);
    return;
}
```

JWTAUTHFILTER (6)

1. On récupère le User de la BDD
2. On vérifie que la valeur n'est pas nulle
3. Si ce n'est pas le cas on arrête le filtrage sans authentifier l'utilisateur

```
final UserDetails userDetails = userDetailsService.loadUserByUsername(username);
if (userDetails == null) {
    filterChain.doFilter(request, response);
    return;
}
```

JWTAUTHFILTER (7)

1. Grâce à la méthode prévu dans JwtService, on valide le token
2. On vérifie que la valeur n'est pas FALSE
3. Si ce n'est pas le cas on arrête le filtrage sans authentifier l'utilisateur

```
final boolean isTokenValid = jwtService.validateToken(token, userDetails);
if (Boolean.FALSE.equals(isTokenValid)) {
    filterChain.doFilter(request, response);
    return;
}
```

JWTAUTHFILTER (8)

1. On crée l'objet contenant les informations d'authentification, avec les informations du user et ses rôles
2. On les sets dans le *SecurityContext*
3. On arrête le filtrage après avoir authentifié l'utilisateur

```
1 final UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(
        userDetails,
        null,
        userDetails.getAuthorities()
);
2 SecurityContextHolder.getContext().setAuthentication(authenticationToken);
3 filterChain.doFilter(request, response);
```

RÉSUMÉ

```java
@Component
@RequiredArgsConstructor
public class JwtAuthFilter extends OncePerRequestFilter {

    private final JwtService jwtService;
    private final UserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(
            HttpServletRequest request,
            HttpServletResponse response,
            FilterChain filterChain
    ) throws ServletException, IOException {
        final String bearerToken = request.getHeader("Authorization");
        if (StringUtils.isBlank(bearerToken) || !bearerToken.startsWith("Bearer ")) {
            filterChain.doFilter(request, response);
            return;
        }

        final String token = bearerToken.substring(7);
        if (StringUtils.isBlank(token)) {
            filterChain.doFilter(request, response);
            return;
        }

        final String username = jwtService.getUsernameFromToken(token);
        if (StringUtils.isBlank(username)) {
            filterChain.doFilter(request, response);
            return;
        }

        final UserDetails userDetails = userDetailsService.loadUserByUsername(username);
        if (userDetails == null) {
            filterChain.doFilter(request, response);
            return;
        }

        final boolean isTokenValid = jwtService.validateToken(token, userDetails);
        if (Boolean.FALSE.equals(isTokenValid)) {
            filterChain.doFilter(request, response);
            return;
        }

        final UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(
                userDetails,
                null,
                userDetails.getAuthorities()
        );
        SecurityContextHolder.getContext().setAuthentication(authenticationToken);
        filterChain.doFilter(request, response);
    }

}
```

SecurityConfig

INTRODUCTION

SecurityConfiguration nous permettra de configurer de manière personnalisée (ici avec la norme JWT) spring security.

On va créer un bean de sécurité pour *SecurityFilterChain*. Celui-ci sera la base que l'on personnalisera après.

```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfiguration {

    private final JwtAuthFilter jwtAuthFilter;
    private final AuthenticationProvider authenticationProvider;


    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.build();
    }

}
```

BEANS (1)

Avant de commencer, on a besoin de plusieurs beans :

- **PasswordEncoder** : Le mot de passe étant haché en BDD, il nous faut créer un bean pour la méthode de hachage de celui-ci.

- **AuthenticationProvider** : Venant de SpringSecurity, est utilisé pour implémenté la logique d'authentification personnalisée.

BEANS (2)

PASSWORDENCODER

La méthode classique pour définir notre password encoder, est d'utiliser Bcrypt.

Il s'agit d'un algorithme de hachage spécialement conçu pour la sécurisation des mots de passes dans une application.

- Il est très résistant aux attaques brute force
- Il gère de « salage » des mots de passe: morceau de donnée aléatoire rendant tout hachage unique même avec un mot de passe identique.
- Il est personnalisable pour augmenter le niveau de hachage

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

BEANS (3)

AUTHENTICATIONPROVIDER

On personnalise l'AuthenticationProvider avec :

- *userDetailsService()* : Notre bean implémentant UserDetailsService
- *passwordEncoder()* : Notre bean définissant notre encodeur de mots de passe

Ce bean nous sera utile pour laisser SpringSecurity s'occuper de l'authentification de l'utilisateur avec nos paramétrages personnalisés.

```
@Bean
public AuthenticationProvider authenticationProvider() {
    DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
    authProvider.setUserDetailsService(userDetailsService());
    authProvider.setPasswordEncoder(passwordEncoder());
    return authProvider;
}
```

BEANS (4)

AUTHENTICATIONMANAGER

On définit le bean basique pour *AuthenticationManager q*ui nous permettra notamment la méthode *authenticate* dans le controller de login.

```
@Bean
public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exception {
    return config.getAuthenticationManager();
}
```

BEANS (4)

AUTHENTICATIONMANAGER

On définit le bean basique pour *AuthenticationManager q*ui nous permettra notamment la méthode *authenticate* dans le controller de login.

```
@Bean
public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exception {
    return config.getAuthenticationManager();
}
```

SECURITYFILTERCHAIN

Usant du standard JWT, on **doit** suivre celui-ci avec notamment:

- Session en *STATELESS* pour ne pas créer de session d'authentification côté serveur puisque à chaque requête client il faut prouver notre authentification.
- Désactiver CSRF (car pas de stockage de session côté serveur)

On ajoute également:

- Notre Bean AuthenticationProvider
- Notre filtre JwtAuthFilter mis avant le filtre intégré dans Spring Security
- Une liste des règles d'autorisation pour les requêtes HTTPs dans l'application. Ici toutes ont besoin d'authentification, excepté */register* et */authenticate* qui sont logiquement en public.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    return http
            .csrf(AbstractHttpConfigurer::disable)
            .sessionManagement(sessionManagement ->
                    sessionManagement.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            )
            .authorizeHttpRequests(authorize ->
                    authorize
                            .requestMatchers("/register").permitAll()
                            .requestMatchers("/authenticate").permitAll()
                            .anyRequest().authenticated()
            )
            .authenticationProvider(authenticationProvider)
            .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class)
            .build();
}
```

Register

INTRODUCTION

Maintenant que tout est configuré, il nous reste à créer les endpoints pour l'utilisateur.
Notamment créer le endpoint */register* permettant à un utilisateur de se créer un compte sur l'application.
(on n'oublie pas que ce endpoint a été mis en public dans les configurations)

On aura donc besoin:

- Un objet pour récupérer la requête
- Un controller
- Un service qui utilise l'encodeur de mot de passe

AUTHREQUEST

Pour contenir les données d'inscription, on créer un DTO spécifique pour cela contenant les informations nécessaires pour qu'un utilisateur non connecté puisse créer son compte.

Ici on n'a que le username et le mot de passe. Mais on pourrait ajouter email, téléphone, nom, prénom, etc.

Il ne s'agit **que** des informations, on ne laisse pas l'utilisateur setter des données qu'il n'est pas censé créer (comme son rôle, son état de compte (actif, inactif, banni, etc.), etc.)

```
public record AuthRequest(
        String username,
        String password
) {
}
```

SERVICE

On crée une création de compte simple en ne settant que le username et en encodant le password avec la méthode ***.encode***.

En temps normal il faudra également vérifier si le username n'est pas déjà en BDD, et auquel cas envoyer une erreur au client.

```java
@Service
@Transactional
@RequiredArgsConstructor
public class UserServiceImpl implements IUserService {

    private final IUserRepository userRepository;
    private final PasswordEncoder passwordEncoder;

    @Override
    public void register(RegisterRequest registerRequest) {
        User user = new User();
        user.setUsername(registerRequest.username());
        final String encodedPassword = passwordEncoder.encode(registerRequest.password());
        user.setPassword(encodedPassword);
        userRepository.save(user);
    }
}
```

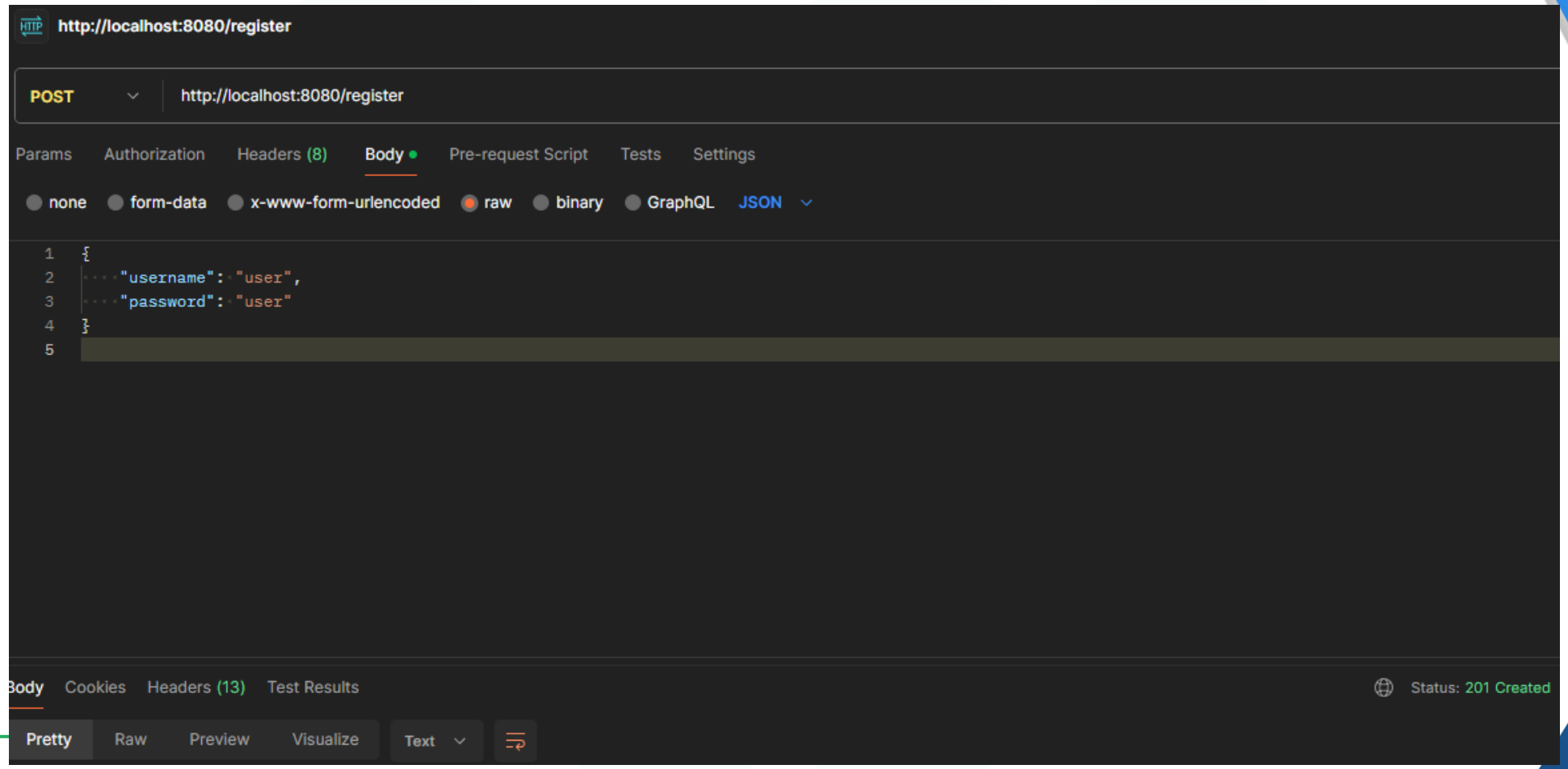
CONTROLLER

On crée le controller **POST: /register** permettant d'envoyer le DTO et d'user du service pour créer l'utilisateur. Puis envoyer un simple statut 201.

On pourrait envoyé les informations du compte également.

```
@PostMapping("/register")
public ResponseEntity<Void> register(@RequestBody RegisterRequest registerRequest) {
    userService.register(registerRequest);
    return ResponseEntity.status(HttpStatus.CREATED).build();
}
```

POSTMAN



Authenticate

INTRODUCTION

Il ne nous reste plus qu'à avoir le controller pour l'authentification qui sera sur le endpoint */authenticate*.

On aura donc besoin:

- Un objet pour récupérer la requête
- Un controller
- Utiliser l'authentification Spring Security
- Créer une méthode création de Token JWT
- Un objet pour la réponse

DTOS

```java
public record AuthRequest(
        String username,
        String password
) {
}
```

On crée un objet pour la requête contenant username et mot de passe, et pouvant contenir plus par exemple le boolean *rememberMe*.

```java
public record AuthResponse(
        String accessToken
) {
}
```

Dans l'objet de réponse on met le token JWT, et possiblement plus tard le token refresh et autres infos utilisateur dont le client pourrait avoir besoin.

GENERATE TOKEN

La librairie précédemment installée a un builder tout fait pour pouvoir générer le token, il nous faut:

- **Subject** : qui représentera le propriétaire du token, généralement le username, mais peut aussi être un autre champs unique relié à l'utilisateur email, un code de référence, etc.
- **Date d'expiration** : permettant de limiter la durée de vie du token pour éviter d'en avoir un éternel.
- **Signature du token** avec l'algorithme utilisé (ici HS256 demandant donc une secret key de 256 bits minimale)
- Autre : On peut également rajouter d'autres claims, comme les rôles, email, ou autre info user non confidentiels.

```
public String generateToken(String username) {
    return Jwts.builder()
            .setSubject(username)
            .setExpiration(new Date(System.currentTimeMillis() + expiration))
            .signWith(getSignInKey(), SignatureAlgorithm.HS256)
            .compact();
}
```

CONTROLLER

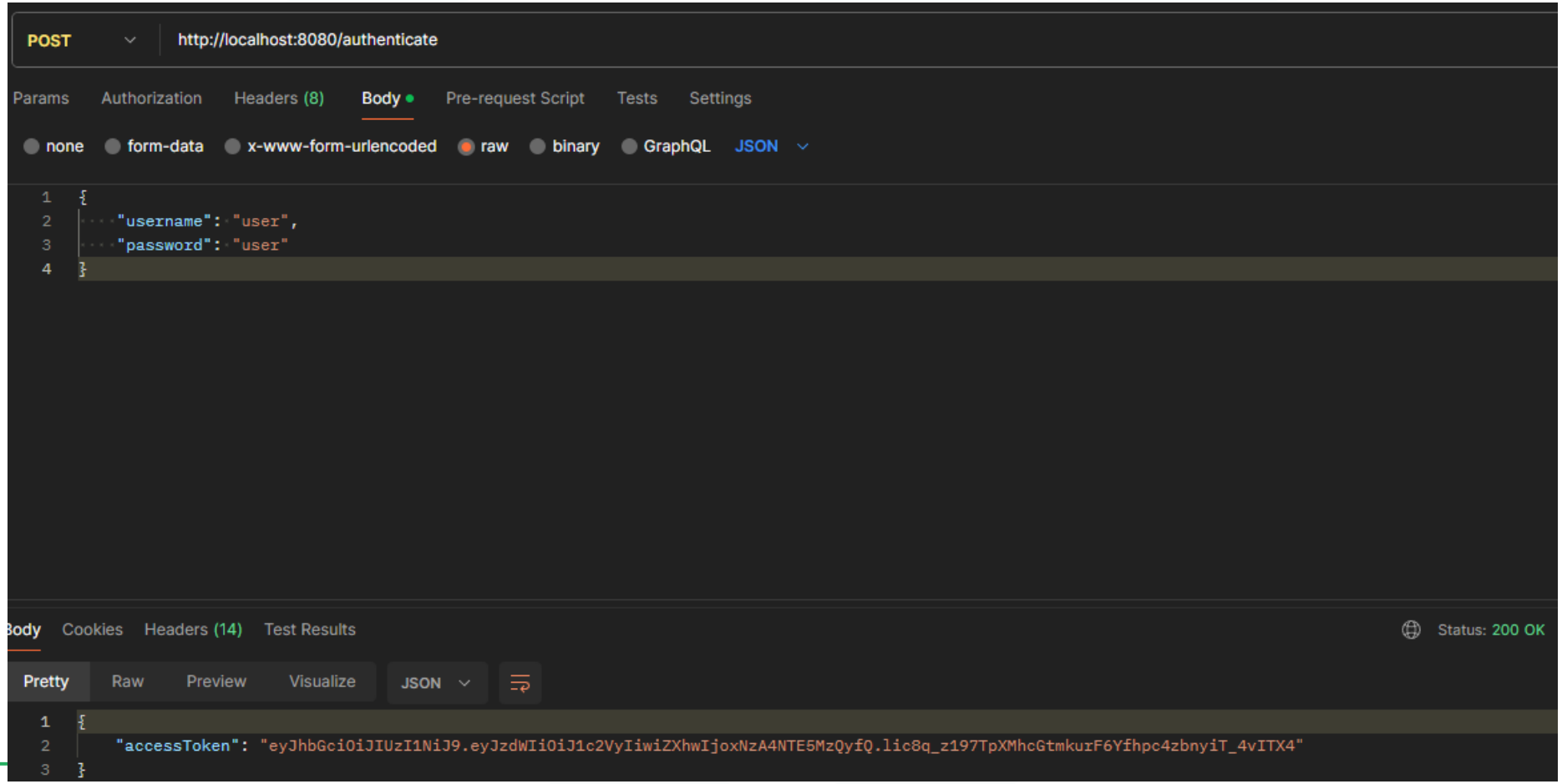
Pour authentifier l'utilisateur, on utilise la méthode *authenticate* de l'*AuthenticationManager* qui aura besoin d'un objet *UsernamePasswordAuthenticationToken*.

Cette méthode va automatiquement vérifier que le username et mot de passe données correspondent à un utilisateur et mot de passe haché en BDD.

On pourra catcher l'erreur, et personnaliser la réponse selon celle-ci.

```
@PostMapping("/authenticate")
public ResponseEntity<AuthResponse> authenticate(@RequestBody AuthRequest request) {
    try {
        final UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(
                request.username(),
                request.password()
        );
        authenticationManager.authenticate(authenticationToken);
        final String token = jwtService.generateToken(request.username());
        final AuthResponse response = new AuthResponse(token);
        return ResponseEntity.ok(response);
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
    }
}
```

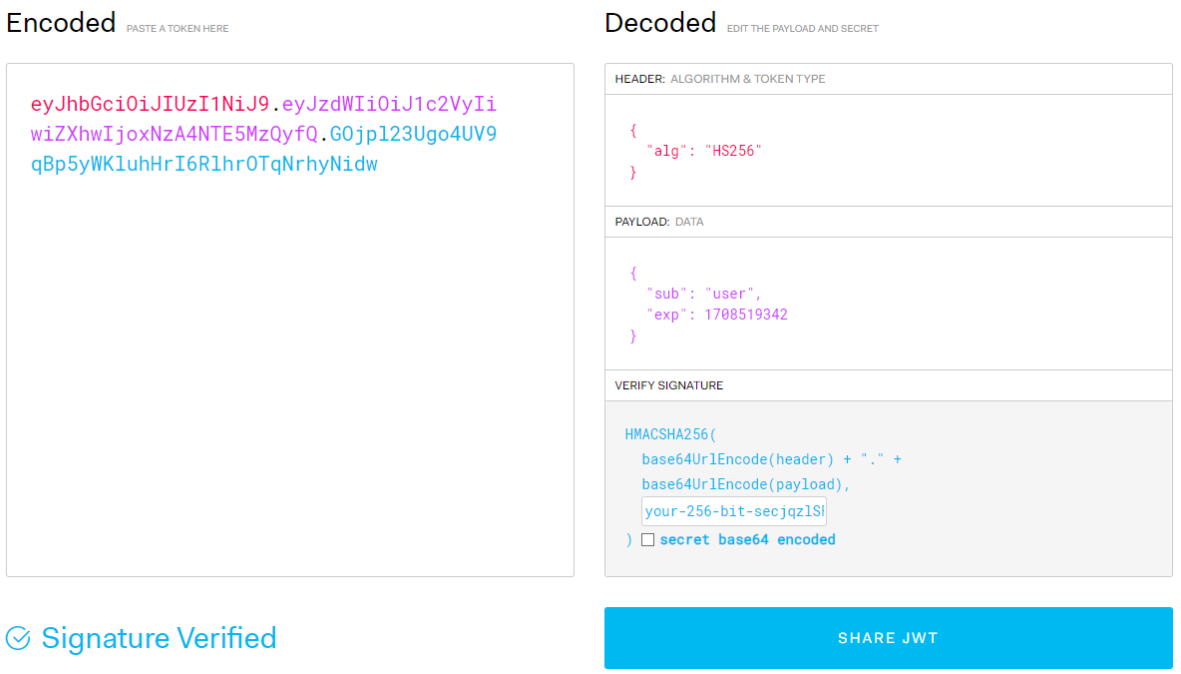
POSTMAN


POST
http://localhost:8080/authenticate
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
{
"username": "user",
"password": "user"
}
Body
Cookies
Headers (14)
Test Results
Status: 200 OK
Pretty
Raw
Preview
Visualize
JSON
{
"accessToken": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1c2VyIiwiZXhwIjoxNzA4NTE5MzQyfQ.lic8q_z197TpXMhcGtmkurF6Yfhpc4zbnyiT_4vITX4"
}

JWT.IO

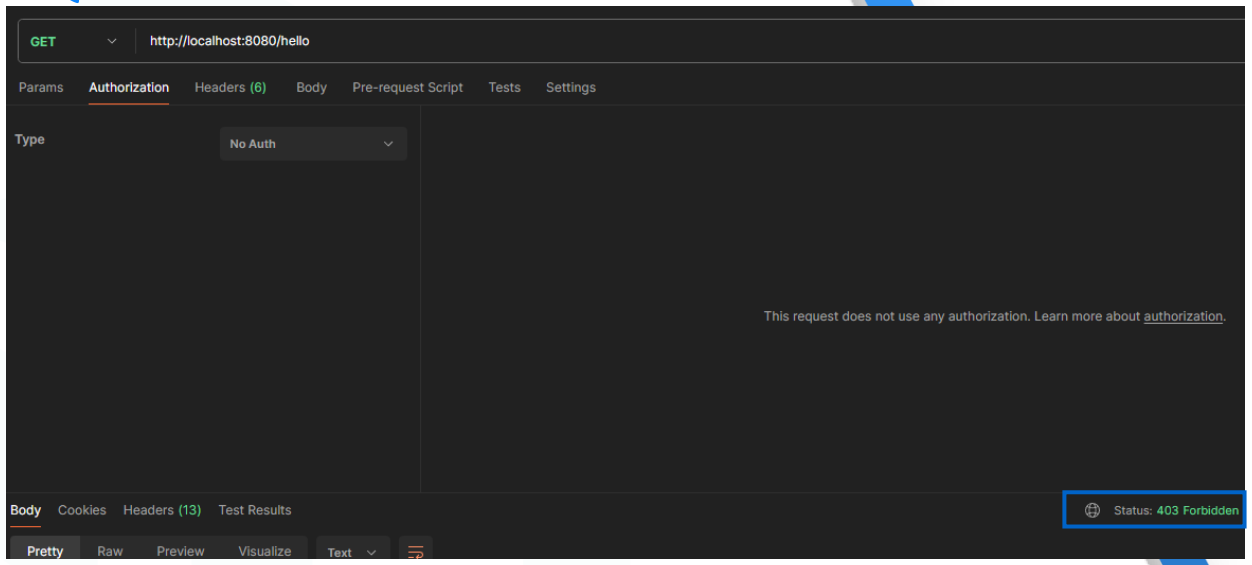
JWT a un site permettant notamment de checker le token: https://jwt.io/

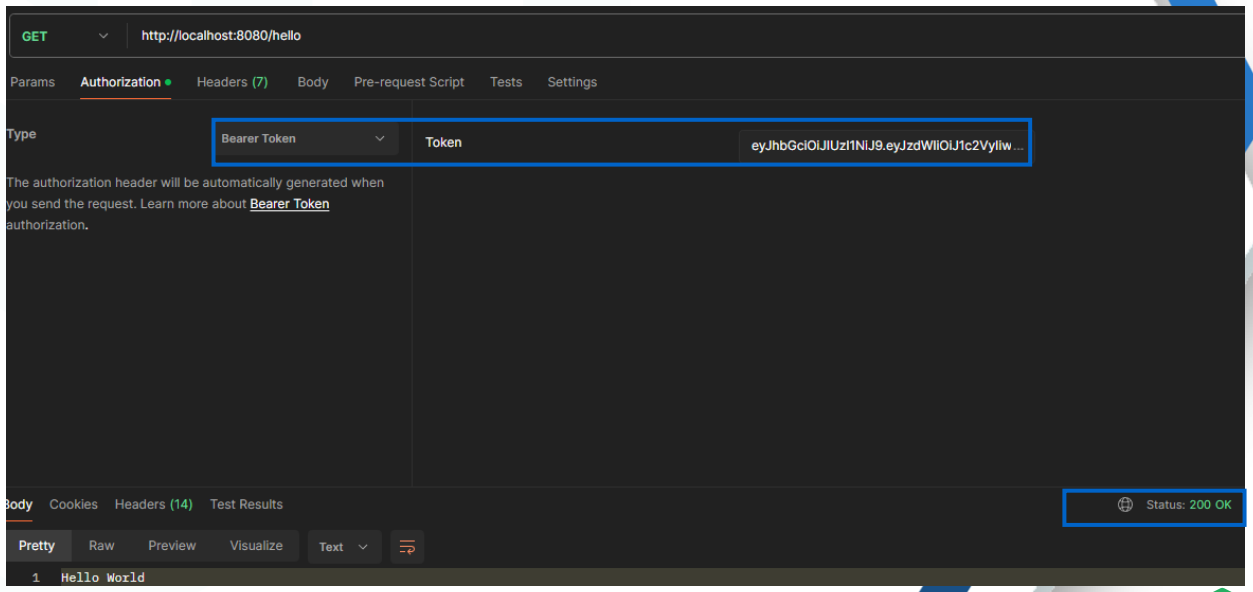
Comme on peut bien voir dessus, on peut récupérer le payload sans la secret key. D'où l'importance de ne pas mettre de donnée sensible dedans.



TESTER UNE REQUÊTE PRIVÉE

```java
@GetMapping("/hello")
public String hello() {
    return "Hello World";
}
```





En utilisant un controller créer et non mis dans la white liste des requêtes publiques dans SecurityConfiguration, on peut tester la requête sans être connecté et en étant connecté pour en voir la différence.

Gestion de rôle

INTRODUCTION

En plus de permettre la gestion de compte, Spring Security permet également de gérer les rôles/autorités de ceux-ci:

- **Rôle** : La gestion de rôle consiste à donner un niveau d'accès à l'utilisateur. Par exemple: admin, manager, utilisateur, etc. Ces rôles auront des droits spécifiques dans l'application selon les règles de gestion du projet. Un utilisateur pouvant possiblement avoir plusieurs rôles.

- **Autorité** : La gestion par autorité consiste à créer une autorité pour chaque fonctionnalité de l'application (création produit, update produit, manage user, etc.) et donner à l'utilisateur la liste de ses accès disponible. Dans l'interface visuelle on peut simplifier cela en créant des rôles qui sont des regroupements d'autorité.

ENTITÉ ROLE

UserDetail ayant besoin d'une liste de *GrantedAuthority*, il nous faut une entité pour cela. *GrantedAuthority* est une interface permettant de géré les accès de l'utilisateur (par rôle ou autorité).

Pour cela on crée une entité *Role* implémentant *GrantedAuthority*. Pour laquelle il nous faudra overrider la méthode *getAuthority* censé être un getter pour récupérer en string le nom de l'autorité (ou ici le rôle pour nous).

Etant donné que l'attribut *name* est censé être unique en BDD et qu'il s'agit du seul attribut de la classe, on peut en faire l'ID de l'entité.

```java
@Entity
@Table(name = "roles")
public class Role implements GrantedAuthority {

    @Id
    private String name;

    @Override
    public String getAuthority() {
        return name;
    }

}
```

ENTITÉ USER

On met à jour l'entité User, en y ajoutant la relation avec Role.
Ici on prend la décision qu'un utilisateur n'a qu'un seul rôle, mais UserDetails ayant besoin du getter getAuthorities on devra l'overrider pour envoyer une collection.

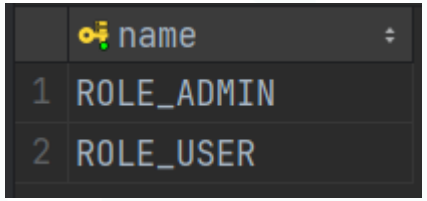
Également, on aura besoin de spécifier que la relation est en fetch eager, puisqu'il faut récupérer l'utilisateur et son rôle en une seule fois. Obligatoire avec l'utilisation de Spring Security.

```
@ManyToOne(fetch = FetchType.EAGER)
private Role role;

@Override
public Collection<Role> getAuthorities() {
    return Collections.singletonList(role);
}
```

AJOUT RÔLES EN BDD

Il nous faut ajouter les rôles de base dans la BDD. On peut utiliser différentes façons de faire, insertion en SLQ au run de l'application, algorithme insertion avec JPA au run, utilisation de manager de BDD comme Liquibase, ou simplement le mettre à la main dans la BDD.



	name
1	ROLE_ADMIN
2	ROLE_USER

REGISTER

On modifie le register pour y ajouter le set du rôle de l'utilisateur.

On le récupère bien de la BDD, et on le set si présent. Dans la méthode ici si le rôle n'existe pas on crée le compte sans rôle. Dans un vrai projet il y aura des règles de gestion pour préciser ce qu'on fait si pas existant (notamment envoyer une erreur).

Important : dans un register on ne laisse pas (ou très rarement, et seulement certains) l'utilisateur setter lui-même son rôle.

```
@Override
public void register(RegisterRequest registerRequest) {
    User user = new User();
    user.setUsername(registerRequest.username());
    final String encodedPassword = passwordEncoder.encode(registerRequest.password());
    user.setPassword(encodedPassword);
    roleRepository.findByName("ROLE_USER").ifPresent(user::setRole);
    userRepository.save(user);
}
```

TOKEN

Généralement on ajoute également le rôle de l'utilisateur dans le token, ce qui permettra au client de le récupérer et créer des restrictions.

Ici comme on a qu'un seul rôle, et qu'il peut être null, on ajoute le nom de celui-ci ou ANONYMOUS s'il s'agit d'un compte sans rôle. Cela est encore une fois arbitraire dans l'application et est customisable selon les besoins du projet.

```
public String generateToken(UserDetails user) {
    String role = user.getAuthorities()
            .stream()
            .findFirst()
            .map(GrantedAuthority::getAuthority)
            .orElse("ANONYMOUS");
    final Map<String, Object> claims = Map.of("role", role);
    return Jwts.builder()
            .setSubject(user.getUsername())
            .setExpiration(new Date(System.currentTimeMillis() + expiration))
            .addClaims(claims)
            .signWith(getSignInKey(), SignatureAlgorithm.HS256)
            .compact();
}
```

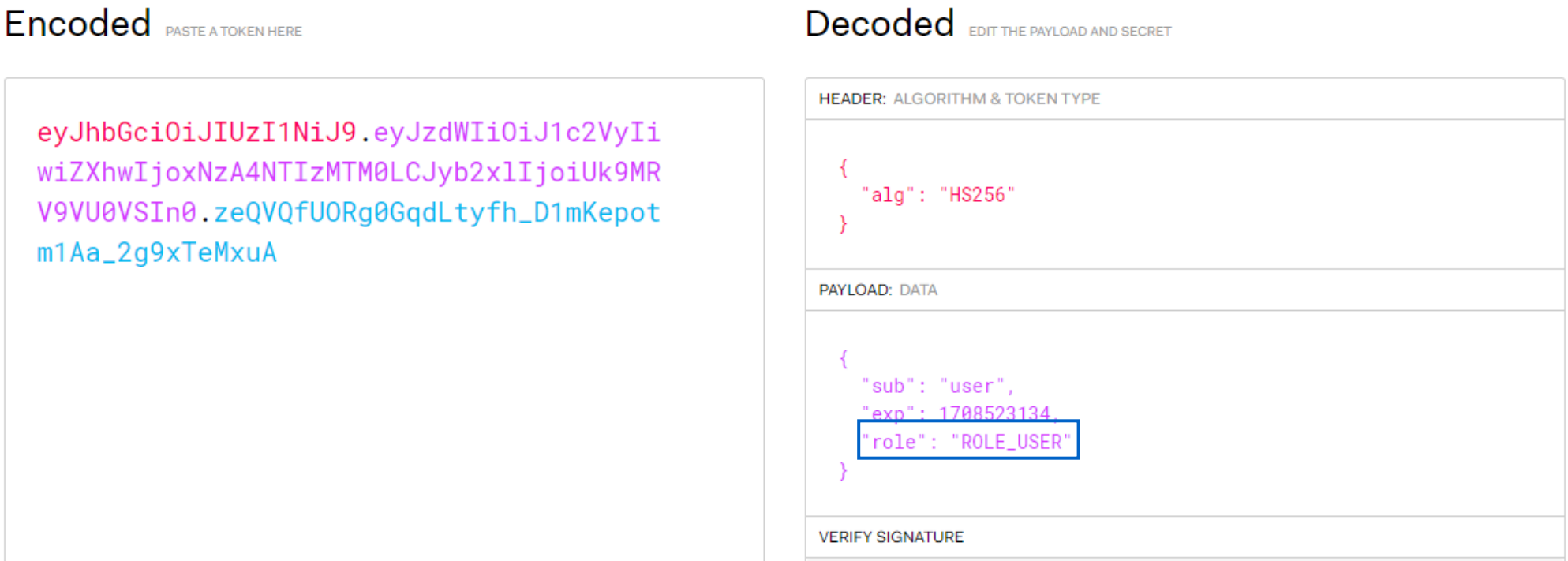
CONTROLLER

Etant donné qu'on ajoute le rôle dans le token, on a besoin de récupérer l'utilisateur en BDD. Donc on modifie un peut le controller.

```java
@PostMapping("/authenticate")
public ResponseEntity<AuthResponse> authenticate(@RequestBody AuthRequest request) {
    try {
        final UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(
                request.username(),
                request.password()
        );
        authenticationManager.authenticate(authenticationToken);
        final User user = userService.findByUsername(request.username())
                .orElseThrow(() -> new UsernameNotFoundException("User not found"));
        final String token = jwtService.generateToken(user);
        final AuthResponse response = new AuthResponse(token);
        return ResponseEntity.ok(response);
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
    }
}
```

TESTER

En testant le */register* et le */authenticate* on vérifie que le token récupéré fournit bien le rôle.



SÉCURISER LES ENDPOINTS

REQUESTMATCHERS

Si on veut sécuriser des endpoints selon le rôle de l'utilisateur, il existe plusieurs manières.

On peut rajouter un *matcher* dans la *securityFilterChain* comme on a fait pour rendre des endpoints publiques.

L'avantage est que tout est rassembler au même endroit, le désavantage est quand lisant un controller on ne sait pas s'il est restreint par rôle.

```
.requestMatchers("/admin/**").hasAuthority("ROLE_ADMIN")
```

SÉCURISER LES ENDPOINTS

@SECURED

Un autre moyen est d'utiliser @Secured sur les controllers qu'on veut sécuriser.

Il faudra au préalable activer cette fonctionnalité grâce à l'annotation `@EnableMethodSecurity` à mettre sur la classe *SecurityConfiguration*

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(securedEnabled = true)
@RequiredArgsConstructor
public class SecurityConfiguration {
```

```
@Secured("ROLE_ADMIN")
@GetMapping("/admin")
public String admin() {
    return "Hello Admin";
}

@Secured({"ROLE_ADMIN", "ROLE_USER"})
@GetMapping("/user")
public String user() {
    return "Hello User";
}
```

On peut maintenant user de l'annotation `@Secured` à chaque fois qu'on veut restreindre un controller ave un ou plusieurs rôles.

L'avantage c'est qu'en lisant un controller on sait comment il est restreint, le désavantage c'est qu'on n'a pas un regroupement global de ces règles.

Conclusion

CONCLUSION

Voilà qui conclut les bases de Spring Security avec une authentification en JWT simple.

On peut également avoir d'autres façon de customiser notre sécurité:

- Refresh token : implémenter en plus du access token principale, un token de refresh à plus longue durée de vie pour récupérer automatiquement un nouveau access token à chaque fois que celui-ci est périmé.
- OAuth2 : implémenter une authentification par application tier (ex: Google, Facebook, GitHub, etc.)
- Plus de filtres : Selon les besoins, on peut rajouter autant de filtre de requêtes permettant de valider les requêtes
- Validation : validation de compte par email
- Reset password : implémenter deux controllers de reset password avec envoi d'email
- Stockage des tokens : on peut également stocker les token en BDD pour garder en mémoire ceux qui sont actif. Utile pour compétemment logout un utilisateur sur tous ses clients.
- etc.