TP 2 : Sécurisation d'une Application Web avec Spring Security & JWT

Auteur : Saoudi Haythem
Matière : JEE2
Classe : 5ème Année Génie Logiciel

Table des matières

1 Le Problème : Gérer l'Identité d'un Utilisateur

Pour sécuriser une application, un serveur doit savoir qui est l'utilisateur.
Il existe deux philosophies pour gérer cela.

1.1 Approche Stateful : Le Serveur "se souvient"

C'est l'approche traditionnelle.

1. L'utilisateur s'identifie (login/mot de passe).
2. Le serveur crée une **session** pour cet utilisateur, stocke des informations le concernant et renvoie un identifiant de session (Session ID) au client, généralement via un **cookie**.
3. À chaque requête suivante, le navigateur envoie automatiquement ce cookie, permettant au serveur de retrouver la session et "d'identifier" l'utilisateur.

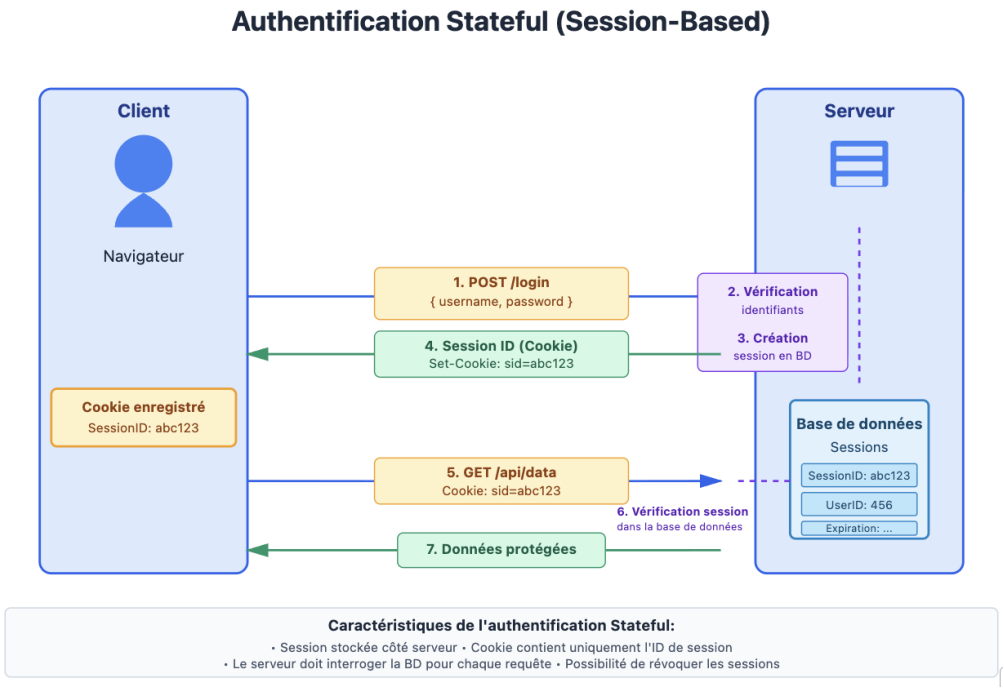

FIGURE 1 – Flux d'une authentification Stateful.

Cette méthode est simple mais présente des limites, notamment dans les architectures modernes. De plus, elle est particulièrement vulnérable à une attaque de type **CSRF (Cross-Site Request Forgery)**.

1.1.1 La vulnérabilité au CSRF

Le mécanisme même du web facilite le CSRF. Lorsqu'un utilisateur se connecte, le serveur place un **cookie de session** dans son navigateur. Ce cookie est ensuite **automatiquement envoyé** par le navigateur avec chaque requête vers le même domaine, que la requête provienne du site légitime ou d'un site malveillant.

Un attaquant peut donc forger une requête (par exemple, "supprimer le compte") sur son propre site. Si la victime authentifiée déclenche cette requête, son navigateur enverra la requête malveillante **avec le cookie de session**, authentifiant ainsi l'action non désirée à l'insu de l'utilisateur.

1.2 Approche Stateless : Le Serveur est "sans état"

L'approche sans état (stateless) délègue la gestion de l'état au client.

1. L'utilisateur s'identifie.

2. Le serveur valide les informations, mais ne stocke **aucune session**. Il génère à la place un **jeton (token)** auto-contenu qui encapsule l'identité de l'utilisateur.
3. Ce jeton est renvoyé au client, qui doit le conserver et le présenter manuellement (généralement dans l'en-tête 'Authorization') à chaque requête protégée.

Le serveur n'a plus besoin de "se souvenir" ; il lui suffit de valider le jeton à chaque appel.

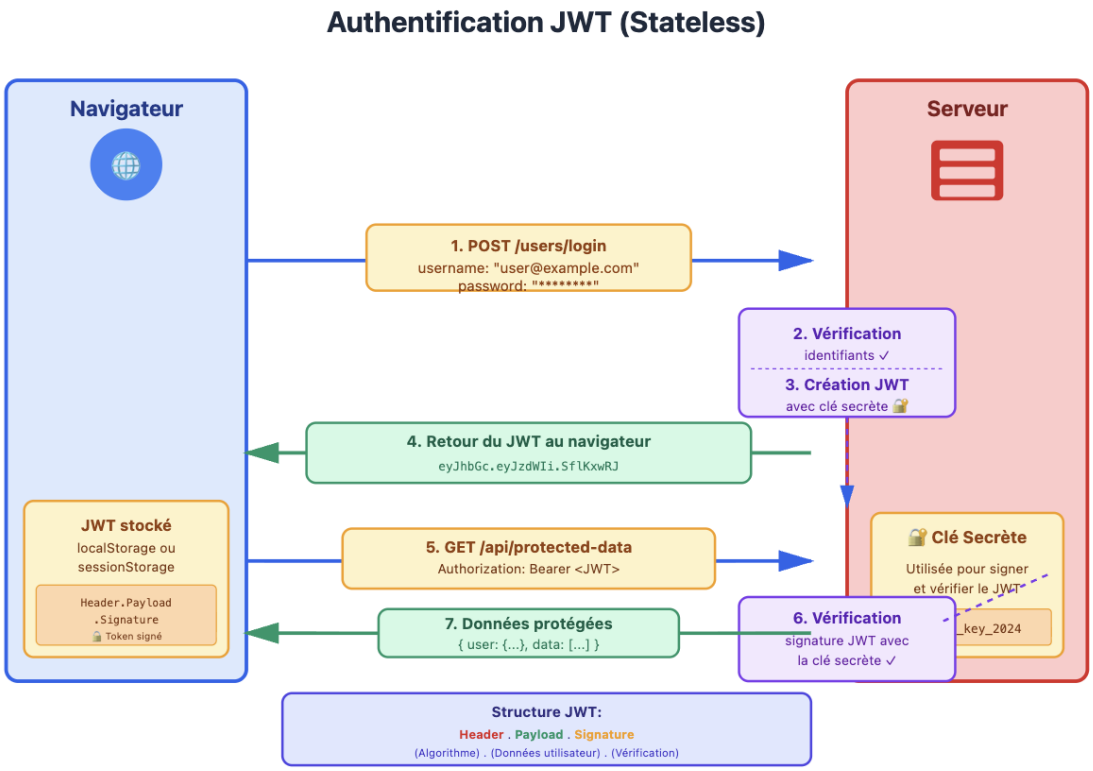

FIGURE 2 – Fonctionnement d'une authentification Stateless avec JWT.

[utf8]inputenc xcolor graphicx float [T1]fontenc
minted

2 La Solution Approfondie : Le JSON Web Token (JWT)

Le JSON Web Token (JWT) est un standard ouvert (RFC 7519) qui est devenu un **standard incontournable** pour la création de jetons d'accès compacts et autonomes. Ces jetons sont conçus pour échanger des informations de manière sécurisée entre différentes parties. La structure d'un JWT est une chaîne de caractères simple, composée de trois parties encodées en Base64Url et séparées par des points : `Header.Payload.Signature`.

- **Header (En-tête)** : Cette première partie contient les métadonnées du jeton. Il s'agit d'un objet JSON qui inclut généralement deux champs :
  - `typ` (Type) : Définit le type du jeton, qui est systématiquement "'JWT'".
  - `alg` (Algorithme) : Spécifie l'algorithme cryptographique utilisé pour la signature. Les plus courants sont :
    - `HS256` **(HMAC + SHA-256)** : Un algorithme **symétrique**. La même clé secrète est utilisée pour créer la signature et pour la vérifier.
    - `RS256` **(RSA + SHA-256)** : Un algorithme **asymétrique**. Il utilise une *clé privée* pour signer le jeton et une *clé publique* pour vérifier la signature.
- **Payload (Charge Utile)** : Le cœur du jeton. C'est un objet JSON qui contient les **claims** (revendications). On distingue trois types de claims :

— **Registered Claims (Revendications Enregistrées)** : Ce sont des revendications standardisées (définies dans la RFC 7519), non obligatoires mais fortement recommandées pour garantir l'interopérabilité.
  — `iss` (Issuer) : L'émetteur du jeton.
  — `sub` (Subject) : Le sujet du jeton (l'ID de l'utilisateur).
  — `aud` (Audience) : Le ou les destinataires du jeton.
  — `exp` (Expiration Time) : La date d'expiration (timestamp).
  — `nbf` (Not Before) : La date avant laquelle le jeton n'est pas valide.
  — `iat` (Issued At) : La date d'émission du jeton.
  — `jti` (JWT ID) : Un identifiant unique pour le jeton.
— **Public Claims (Revendications Publiques)** : Des informations personnalisées conçues pour être comprises par différentes applications. Il est recommandé d'utiliser des noms uniques (par ex. des URI) pour éviter les collisions.

```
{
  "name": "Jeanne Dupont",
  "email": "jeanne.dupont@example.com",
  "https://example.com/is_premium_user": true
}
```

— **Private Claims (Revendications Privées)** : Revendications personnalisées dont la signification n'est connue que par les parties qui les échangent.

```
{
  "department_id": 42,
  "permissions": ["read", "write"]
}
```

Exemple de Payload Complet :

```
{
  "iss": "https://api.monentreprise.com",
  "sub": "user-12345",
  "exp": 1735689600,
  "name": "Jean Martin",
  "https://monentreprise.com/roles": [ "editor" ],
  "department_id": 7
}
```

Note importante : Le payload est simplement encodé, pas chiffré. Il ne faut **jamais** y stocker d'informations sensibles (mots de passe, etc.).

— **Signature** : C'est le sceau numérique qui garantit l'intégrité (le jeton n'a pas été modifié) et l'authenticité (il provient bien de l'émetteur attendu). Son calcul combine :

1. L'en-tête (Header) encodé en Base64Url.
2. La charge utile (Payload) encodée en Base64Url.
3. Une clé secrète (pour HS256) ou une clé privée (pour RS256).

La formule peut être résumée ainsi :

```
Signature = Algo(base64Url(Header) + "." + base64Url(Payload), Clé)
```

Toute modification du header ou du payload invaliderait la signature.

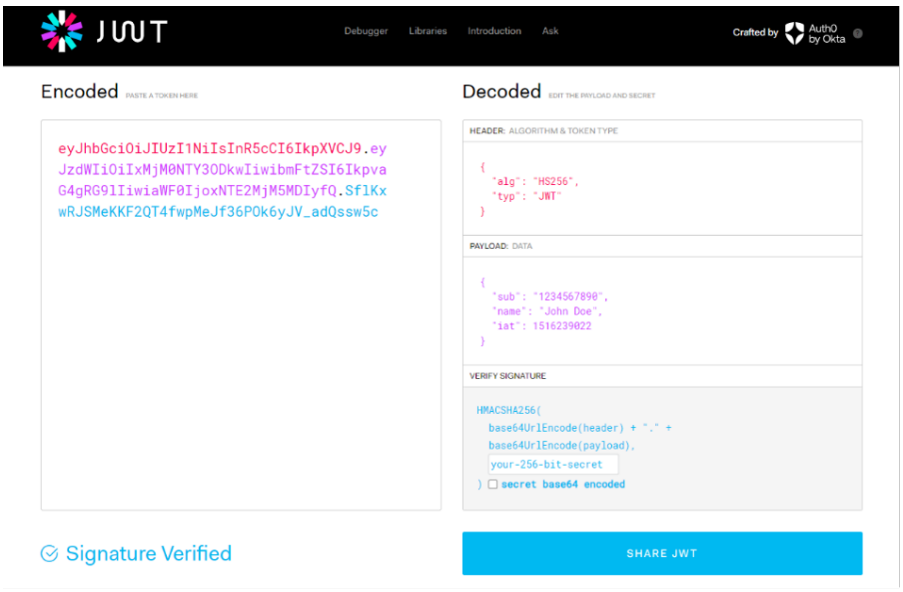

FIGURE 3 – Décodage d'un JWT sur le site jwt.io, illustrant les trois parties.

3 Spring Security en Action

Spring Security protège une application en établissant une **chaîne de filtres de sécurité**. Chaque requête doit traverser cette chaîne avant d'atteindre sa destination (le contrôleur).

3.1 La Chaîne de Filtres de Sécurité (Security Filter Chain)

Il faut imaginer cette chaîne comme une série de postes de contrôle ou de gardes de sécurité. Chaque filtre a une mission précise et, une fois sa tâche accomplie, il passe la requête au filtre suivant. L'ordre de ces filtres est crucial.

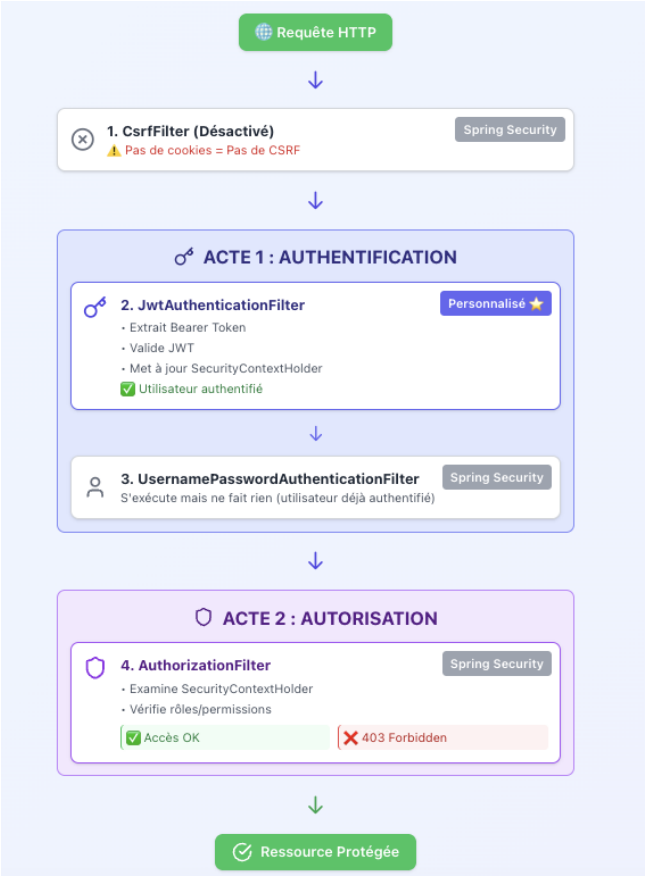

FIGURE 4 – Chaîne de Filtres Spring Security avec JWT.

Dans notre configuration avec JWT, voici les filtres les plus importants et leur rôle :

— **CsrfFilter :** Ce filtre protège contre les attaques CSRF. Dans une API stateless où nous n'utilisons pas de cookies de session, cette protection n'est pas nécessaire. C'est pourquoi l'une des premières étapes de notre configuration est de le **désactiver**.

— **JwtAuthenticationFilter (Notre filtre personnalisé) :** C'est le cœur de notre logique. Nous l'insérons **avant** les filtres d'authentification standards de Spring. Sa mission est d'inspecter chaque requête, de chercher un en-tête 'Authorization' avec un Bearer Token, de valider ce token et, si la validation réussit, d'authentifier l'utilisateur en mettant à jour le 'SecurityContextHolder'.

— **UsernamePasswordAuthenticationFilter :** Le filtre par défaut de Spring pour gérer les soumissions de formulaires de login. Comme notre filtre JWT s'exécute avant, ce filtre sera généralement ignoré pour les requêtes API authentifiées par token.

— **AuthorizationFilter (ou FilterSecurityInterceptor) :** Ce filtre se trouve vers la fin de la chaîne. Son rôle est de vérifier les **autorisations**. Il examine le 'SecurityContextHolder' (que notre filtre JWT a préalablement rempli) et vérifie si l'utilisateur authentifié a les bons rôles ou permissions (ex : 'ADMIN') pour accéder à la ressource demandée. Si ce n'est pas le cas, il rejette la requête (souvent avec une erreur 403 Forbidden).

Le processus de sécurité se déroule donc en deux actes principaux, orchestrés par cette chaîne.

3.2 Acte I : L'Échange - Obtenir un Jeton sur `/login`

L'objectif est d'échanger des identifiants (username/password) contre un JWT.

1. L'utilisateur envoie une requête POST sur `/login`. Comme cet endpoint est public, la requête traverse la chaîne de filtres sans être bloquée.
2. Notre contrôleur reçoit la requête et déclenche l'**AuthenticationManager**.

3. L'AuthenticationManager délègue la validation à l'**AuthenticationProvider** qui, à l'aide du **UserDetailsService** et du **PasswordEncoder**, vérifie les identifiants.
4. Si l'authentification réussit, notre code utilise un **JwtService** pour générer et signer le jeton JWT.
5. Le jeton est renvoyé au client.

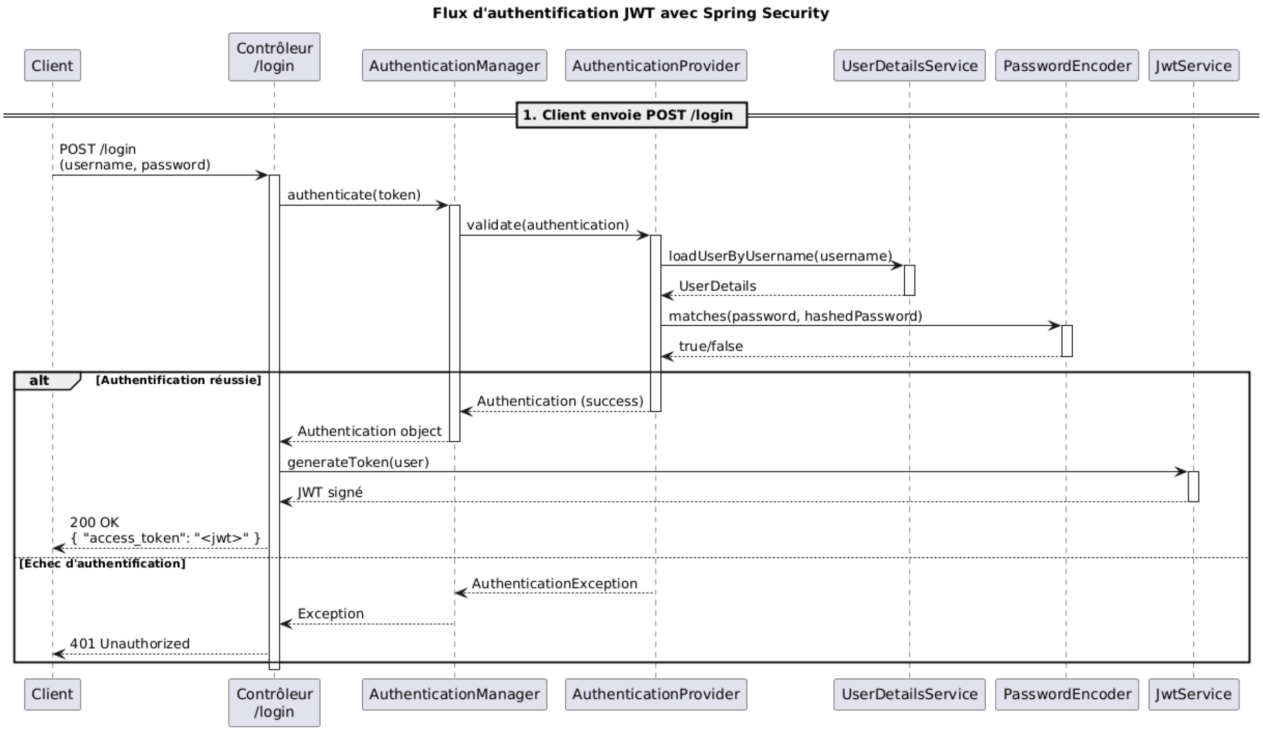

FIGURE 5 – Flux d'authentification pour la génération du JWT.

3.3 Acte II : Le Laissez-passer - Accéder à une Ressource Protégée

Le client utilise son jeton comme un badge d'accès.

1. Le client envoie une requête (ex : GET sur `/api/produits`) en incluant le jeton dans l'en-tête.
2. Notre **JwtAuthenticationFilter** intercepte la requête, valide le jeton et met à jour le **SecurityContextHolder**.
3. La requête continue sa route. Lorsqu'elle atteint l'**AuthorizationFilter**, celui-ci vérifie que l'utilisateur (maintenant présent dans le contexte) a les permissions requises.
4. Si tout est en ordre, la requête atteint finalement le contrôleur.

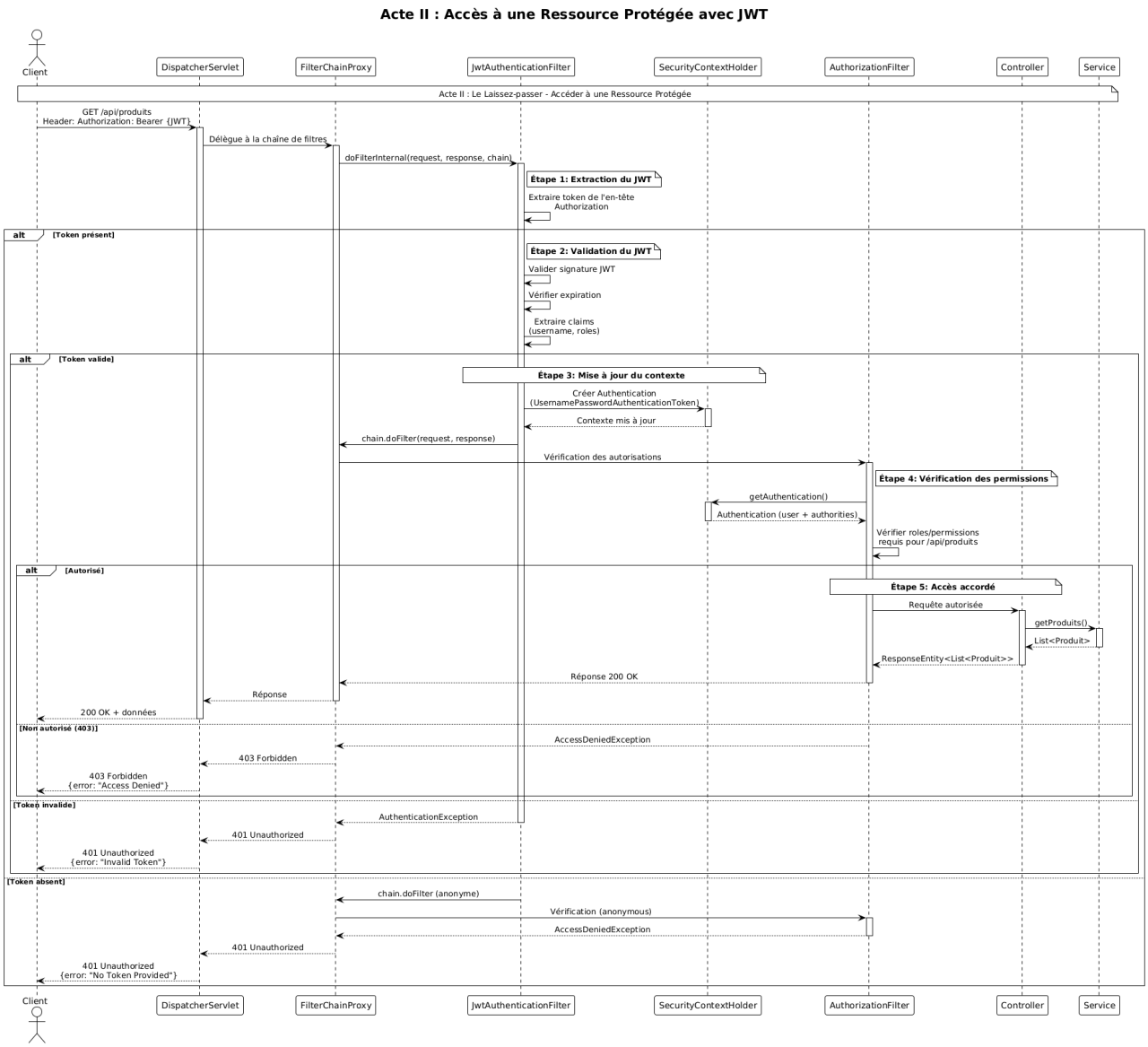

FIGURE 6 – Processus de validation du JWT par le filtre pour l'accès à une ressource.

Note Pédagogique : Pourquoi recharger l'utilisateur ?

On pourrait se demander pourquoi le filtre doit appeler le 'UserDetailsService' à chaque requête. C'est une mesure de sécurité essentielle :

- **Permissions à jour :** Si les droits d'un utilisateur ont été modifiés, recharger son profil depuis la base de données garantit que nous utilisons toujours ses permissions actuelles, et non celles figées dans le jeton.
- **Bannissement d'utilisateur :** Si un utilisateur a été désactivé, cette étape empêchera son jeton (même valide) de fonctionner.

4 Résumé des Acteurs Clés

- **SecurityFilterChain** : La scène où se joue la sécurité, une chaîne de filtres que chaque requête traverse.
- **JwtAuthenticationFilter** : Notre acteur principal, un filtre personnalisé qui examine les jetons à l'entrée.
- **AuthenticationManager** : Le metteur en scène, qui orchestre le processus de validation des

identifiants.
— **UserDetailsService** : L'archiviste, qui fournit le "dossier" d'un utilisateur depuis la base de données.
— **PasswordEncoder** : L'expert en cryptographie pour les mots de passe.
— **SecurityContextHolder** : Le "tableau d'affichage" qui indique quel utilisateur est actuellement authentifié.

5 Implémentation : Sécurisation de l'API de Gestion de Produits

Dans cette partie, nous allons repartir de notre projet de gestion de produits (TP1) et y intégrer une couche de sécurité robuste. L'objectif est de mettre en place une authentification stateless basée sur JWT et de définir des règles d'autorisation pour les rôles `USER` et `ADMIN`.

5.1 Étape 1 : Préparation du Projet

La première étape consiste à ajouter les dépendances nécessaires à Spring Security et JWT dans notre projet. Ouvrez votre fichier `pom.xml` et ajoutez les blocs `<dependency>` suivants :

```
<!-- Spring Boot Starter Security -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<!-- Dependances pour JSON Web Tokens (JJWT) -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
```

Listing 1 – Dépendances de sécurité dans pom.xml

N'oubliez pas de recharger votre projet Maven pour que ces nouvelles dépendances soient téléchargées.

5.2 Étape 2 : Modélisation des Utilisateurs et des Rôles

Nous définissons comment nos utilisateurs et leurs rôles seront représentés en base de données avec deux entités JPA : `AppUser` et `AppRole`, ainsi que leurs Repositories.

```
package com.poly.gestioncatalogue.entities;

import jakarta.persistence.*;
import lombok.*;
import java.util.List;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
```

```java
public class AppUser {
    @Id
    private String id;
    @Column(unique = true)
    private String username;
    private String password;
    private String email;
    // Charger les roles immediatement avec l'utilisateur
    @ManyToMany(fetch = FetchType.EAGER)
    private List<AppRole> roles;
}
```

Listing 2 – Entité AppUser.java

```java
package com.poly.gestioncatalogue.entities;
// ... imports ...

@Entity @Data @NoArgsConstructor @AllArgsConstructor @Builder
public class AppRole {
    @Id
    private String nom;
}
```

Listing 3 – Entité AppRole.java

```java
// Dans le package repository

public interface UserRepository extends JpaRepository<AppUser, String> {
    // Methode pour trouver un utilisateur par son nom
    AppUser findByUsername(String username);
}

public interface RoleRepository extends JpaRepository<AppRole, String> {
}
```

Listing 4 – UserRepository.java et RoleRepository.java

5.3 Étape 3 : Initialisation des Données avec Mots de Passe Hachés

Pour pouvoir tester notre application, nous avons besoin d'utilisateurs dans notre base de données. Nous allons les insérer au démarrage de l'application.

Une règle fondamentale en sécurité est de **ne jamais stocker de mots de passe en clair**. Nous devons toujours les "hacher". Spring Security nous facilite la tâche avec l'interface `PasswordEncoder` et son implémentation la plus robuste, `BCryptPasswordEncoder`.

5.3.1 Définition du Bean PasswordEncoder

Avant de créer nos utilisateurs, nous devons d'abord dire à Spring comment il doit hacher les mots de passe. Pour cela, nous déclarons un Bean de type `PasswordEncoder`. Vous pouvez ajouter cette méthode dans votre classe de configuration des beans (`BeanConfig.java`) ou, pour simplifier, directement dans votre classe d'application principale.

```java
import org.springframework.context.annotation.Bean;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

// ... a l'interieur de votre classe @Configuration ou @SpringBootApplication

@Bean
public PasswordEncoder passwordEncoder() {
    // Utilisation de l'algorithme BCrypt, le standard recommande
    return new BCryptPasswordEncoder();
}
```

Listing 5 – Déclaration du Bean PasswordEncoder

5.3.2 Création des utilisateurs au démarrage

Maintenant que Spring sait comment obtenir un `PasswordEncoder`, nous pouvons l'injecter dans un `CommandLineRunner` pour créer nos utilisateurs de test avec des mots de passe sécurisés.

Ajoutez ce Bean dans votre classe principale. Spring injectera automatiquement le `PasswordEncoder` que nous venons de définir.

```
@Bean
CommandLineRunner commandLineRunner(UserRepository userRepository,
                                    RoleRepository roleRepository,
                                    PasswordEncoder passwordEncoder) { // Injection du Bean
    return args -> {
        // Creer les roles s'ils n'existent pas
        if (roleRepository.count() == 0) {
            roleRepository.save(AppRole.builder().nom("USER").build());
            roleRepository.save(AppRole.builder().nom("ADMIN").build());
        }

        AppRole roleUser = roleRepository.findById("USER").get();
        AppRole roleAdmin = roleRepository.findById("ADMIN").get();

        // Creer les utilisateurs si la base est vide
        if (userRepository.count() == 0) {
            // Utilisateur standard
            userRepository.save(AppUser.builder()
                    .id(UUID.randomUUID().toString())
                    .username("user")
                    // Utilisation du PasswordEncoder pour hacher le mot de passe
                    .password(passwordEncoder.encode("1234"))
                    .email("user@example.com")
                    .roles(List.of(roleUser))
                    .build());

            // Administrateur
            userRepository.save(AppUser.builder()
                    .id(UUID.randomUUID().toString())
                    .username("admin")
                    .password(passwordEncoder.encode("1234"))
                    .email("admin@example.com")
                    // L'admin a les deux roles
                    .roles(List.of(roleUser, roleAdmin))
                    .build());

            System.out.println(">>> Utilisateurs et Roles initialises !");
        }
    };
}
```

Listing 6 – Insertion des utilisateurs au démarrage

5.4 Étape 4 : Configuration Stratégique de la Sécurité

Après avoir posé les fondations en ajoutant la dépendance Spring Security et en préparant nos utilisateurs et leurs rôles, il est temps de bâtir notre stratégie de sécurité. Par défaut, Spring Security applique une politique de verrouillage complet ; nous allons donc maintenant définir nos propres règles du jeu. C'est ici qu'intervient la classe SecurityConfig, qui agit comme un chef d'orchestre pour définir la chaîne de filtres de sécurité (Security Filter Chain). C'est elle qui dictera comment les requêtes sont interceptées, validées et autorisées.

```java
package com.poly.gestioncatalogue.security;

import lombok.AllArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.config.annotation.method.configuration.
    EnableMethodSecurity;
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;


@Configuration
@EnableWebSecurity // (A) Active la configuration de sécurité web de Spring.
@EnableMethodSecurity // (B) Active la sécurité au niveau des méthodes (ex:
    @PreAuthorize).
@AllArgsConstructor
public class SecurityConfig {

    private final JwtAuthenticationFilter jwtAuthFilter;
    private final AuthenticationProvider authenticationProvider;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
    Exception {
        return http

                .csrf(csrf -> csrf.disable())
                .sessionManagement(sess -> sess.sessionCreationPolicy(
    SessionCreationPolicy.STATELESS))


                .authorizeHttpRequests(auth -> auth
            .requestMatchers("/api/auth/**").permitAll()
                    .anyRequest().authenticated()
                )

                .authenticationProvider(authenticationProvider)

                .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.
    class)

                .build();
    }
}
```

Listing 7 – Classe SecurityConfig.java : la tour de contrôle de la sécurité

Analyse de la Configuration

La configuration ci-dessus n'est pas une simple liste d'options ; elle raconte une histoire et établit une stratégie claire. Décomposons-la en trois actes logiques :

1. **Fondation : Une API REST *Stateless*.** La première décision clé est d'adopter une politique sans état.
   — `.csrf(csrf -> csrf.disable())` : La protection CSRF repose sur la synchronisation de jetons stockés en session. Puisque nous n'utilisons pas de sessions, cette protection devient non seulement inutile mais aussi contre-productive.

— `.sessionManagement(...STATELESS)` : Cette ligne est la déclaration formelle de notre approche. Nous demandons à Spring Security de **ne jamais créer ou utiliser de** `HttpSession`. Chaque requête doit être atomique et se suffire à elle-même en contenant ses propres informations d'authentification (le token JWT).

2. **Règles du jeu : Le Contrôle d'Accès.** Une fois la nature de notre API établie, nous définissons qui a le droit d'accéder à quoi.
   — `.authorizeHttpRequests(...)` : Ce bloc est le portier de notre application. Nous adoptons une approche sécurisée par défaut : tout est protégé (`.anyRequest().authenticated()`) sauf ce qui est explicitement autorisé.
   — `.requestMatchers("/api/auth/**").permitAll()` : Nous créons une "brèche" contrôlée et publique pour permettre aux utilisateurs de s'enregistrer et de se connecter, processus nécessaires pour obtenir le précieux token JWT.
3. **Mécanique : Intégration de notre logique JWT.** C'est ici que nous connectons nos composants personnalisés à la mécanique interne de Spring Security.
   — `.authenticationProvider(...)` : Nous indiquons à Spring Security quel *fournisseur* utiliser pour traiter une demande d'authentification (par exemple, lors d'un appel à `/api/auth/login`). C'est lui qui sait comment valider un couple utilisateur/mot de passe contre notre base de données.
   — `.addFilterBefore(jwtAuthFilter, ...)` : C'est l'étape la plus cruciale pour JWT. Nous insérons notre filtre `jwtAuthFilter` **avant** le filtre de base qui gère l'authentification par formulaire. Ainsi, pour chaque requête entrante sur une route protégée, notre filtre a la priorité. Il extraira le token de l'en-tête `Authorization`, le validera et, en cas de succès, chargera l'identité de l'utilisateur dans le contexte de sécurité, rendant la requête authentifiée pour la suite du traitement.

5.5 Étape 4.1 : Configuration des Beans de Sécurité (`BeanConfig.java`)

Pour garder notre configuration principale (`SecurityConfig`) propre et focalisée sur la chaîne de filtres, nous déplaçons la création de tous les "outils" (les Beans) dont Spring Security a besoin dans une classe dédiée. Cette classe, annotée avec `@Configuration`, agit comme une usine qui produit et configure les composants essentiels de notre logique d'authentification.

L'annotation `@AllArgsConstructor` de Lombok nous permet d'injecter facilement le `UserRepository` dont nous aurons besoin pour accéder à nos données utilisateur.

5.5.1 Le `UserDetailsService` : La carte d'identité de l'utilisateur

Ce Bean a une mission unique et cruciale : être le pont entre notre base de données (avec nos entités `AppUser` et `AppRole`) et le monde de Spring Security (qui a besoin d'un objet de type `UserDetails`).

```
@Bean
public UserDetailsService userDetailsService() {
    // On retourne une implémentation de l'interface fonctionnelle
    return username -> {
        // 1. Chercher l'utilisateur dans notre base de données
        AppUser appUser = userRepository.findByUsername(username);
        if (appUser == null) {
            // Si l'utilisateur n'est pas trouvé, on doit lever cette exception
            throw new UsernameNotFoundException("Utilisateur non trouvé");
        }

        // 2. Transformer notre AppUser en UserDetails de Spring Security
        return new User(
            appUser.getUsername(),
            appUser.getPassword(),
            // On transforme notre liste de AppRole en une collection de
    GrantedAuthority
            // C'est le format que Spring Security comprend pour les rôles.
            appUser.getRoles().stream()
```

```
                .map(role -> new SimpleGrantedAuthority(role.getNom()))
                .toList()
        );
    };
}
```

Listing 8 – Bean UserDetailsService

5.5.2 Le `PasswordEncoder` : Le gardien des mots de passe

Ce Bean définit l'algorithme que nous utiliserons pour hacher les mots de passe. C'est une règle de sécurité fondamentale : **on ne stocke jamais les mots de passe en clair**. En exposant ce Bean, nous permettons à Spring de l'utiliser partout où une comparaison de mot de passe est nécessaire.

```
@Bean
public PasswordEncoder passwordEncoder() {
    // On utilise BCrypt, l'algorithme standard de l'industrie pour le hachage
    // Il est sécurisé car il est adaptatif et intègre un "sel" (salt)
    automatiquement.
    return new BCryptPasswordEncoder();
}
```

Listing 9 – Bean PasswordEncoder

5.5.3 L'`AuthenticationProvider` : L'expert en validation

C'est le composant qui assemble les pièces. Il prend le `UserDetailsService` et le `PasswordEncoder` pour effectuer la validation réelle des identifiants. Son travail consiste à :

1. Utiliser le `UserDetailsService` pour charger les détails de l'utilisateur.
2. Utiliser le `PasswordEncoder` pour comparer le mot de passe fourni par l'utilisateur avec celui, haché, qui est stocké en base de données.

```
@Bean
public AuthenticationProvider authenticationProvider() {
    // DaoAuthenticationProvider est l'implémentation standard pour l'
    authentification
    // basée sur une base de données.
    DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();

    // On lui fournit notre UserDetailsService pour qu'il sache où trouver les
    utilisateurs.
    authProvider.setUserDetailsService(userDetailsService());

    // On lui fournit notre PasswordEncoder pour qu'il sache comment vérifier les
    mots de passe.
    authProvider.setPasswordEncoder(passwordEncoder());

    return authProvider;
}
```

Listing 10 – Bean AuthenticationProvider

5.5.4 L'`AuthenticationManager` : Le coordinateur de l'authentification

C'est le gestionnaire principal, le point d'entrée que nous appellerons depuis notre contrôleur pour déclencher le processus d'authentification. Il ne fait pas le travail lui-même ; il délègue la tâche à l'`AuthenticationProvider` que nous venons de configurer.

```
@Bean
public AuthenticationManager authenticationManager(AuthenticationConfiguration config
    )
```

```
                                                    throws Exception {
    // Cette méthode, fournie par Spring, expose le gestionnaire d'authentification
    // configuré par défaut, qui utilisera notre AuthenticationProvider.
    return config.getAuthenticationManager();
}
```

Listing 11 – Bean AuthenticationManager

5.6 Étape 5 : Construction du Gardien de l'API : Le Service et le Filtre JWT

Cette étape est cruciale. Nous allons créer deux composants essentiels :

— `JwtService` : Le "cerveau" de notre gestion de tokens. Il saura comment créer, signer, valider et déchiffrer les jetons JWT.

— `JwtAuthenticationFilter` : Le "gardien" de notre API. Il interceptera chaque requête pour vérifier la présence et la validité d'un token avant de laisser la requête continuer.

5.6.1 Configuration des propriétés JWT

Avant de coder, nous devons définir deux paramètres importants dans notre fichier `src/main/resources/applic`

1. Une **clé secrète** pour signer nos tokens. Elle doit être longue, complexe et gardée secrète. Nous utiliserons un encodage en Base64.
2. Une **durée d'expiration** pour les tokens, définie en millisecondes.

```
# ======================================================
# =            CONFIGURATION JSON WEB TOKEN            =
# ======================================================
# Clé secrète (encodée en Base64).
# A generer avec un outil en ligne pour plus de securite.
 spring.app.secretkey =NDY4RTU3NjVKNDlPNDJBNkZIN040M0EyRjZBNkZIN0EyRjZBMkY2QjM1NzY=

# Duree de validite du token en millisecondes (ici, 24 heures)
spring.app.expiration=86400000
```

Listing 12 – Ajouts dans application.properties

5.6.2 Création du JwtService

Nous créons maintenant le composant central de notre infrastructure JWT : le JwtService. Cette classe est le "coffre-fort" de notre application. Elle détient la connaissance de la clé secrète et centralise toute la logique de création, d'analyse et de validation des tokens. La version que nous allons implémenter a une particularité importante : elle intègre directement les rôles (autorisations) de l'utilisateur dans le payload du token. Cette approche rend chaque token autonome et évite des appels superflus à la base de données à chaque requête.

```
package com.poly.gestioncatalogue.security;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.io.Decoders;
import io.jsonwebtoken.security.Keys;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Service;

import java.security.Key;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Collectors;
```

```java

/**
 * Service pour la gestion des JSON Web Tokens (JWT).
 * Cette classe centralise toutes les operations liees aux JWT :
 * - Generation du token a partir des informations de l'utilisateur.
 * - Extraction des informations (claims) contenues dans le token.
 * - Validation de la conformite et de l'integrite du token.
 * L'annotation @Service indique a Spring que cette classe est un composant de la couche de service.
 */
@Service
public class JwtService {

    // Cle secrete utilisee pour signer et verifier les tokens.
    // Elle est injectee depuis les proprietes de l'application (ex: application.properties).
    // Il est crucial que cette cle soit longue, complexe et gardee secrete.
    @Value("${spring.app.secretkey}")
    private String secretKey;

    // Duree de validite du token en millisecondes.
    // Egalement injectee depuis les proprietes de l'application.
    @Value("${spring.app.expiration}")
    private long expiration;

    /**
     * Extrait le nom d'utilisateur (le "sujet" du token) a partir d'un token JWT.
     * Le sujet est une des revendications (claims) standard d'un JWT et identifie de maniere unique l'entite concernee par le token.
     *
     * @param token Le JWT sous forme de chaine de caracteres.
     * @return Le nom d'utilisateur extrait du token.
     */
    public String extractUsername(String token) {
        return extractAllClaims(token).getSubject();
    }

    /**
     * Genere un nouveau token JWT pour un utilisateur authentifie.
     * Cette methode est typiquement appelee apres une connexion reussie.
     *
     * @param userDetails Les details de l'utilisateur (fournis par Spring Security) pour qui le token doit etre genere.
     * @return Une chaine de caracteres representant le JWT compacte et signe.
     */
    public String generateToken(UserDetails userDetails) {
        Map<String, Object> claims = new HashMap<>();

        // Ajout des roles (autorites) de l'utilisateur comme une revendication personnalisee ("claim").
        // Cela permet de creer un token "autonome" : le serveur peut verifier les autorisations
        // de l'utilisateur en se basant uniquement sur le contenu du token, sans avoir a interroger
        // la base de donnees a chaque requete. C'est une optimisation de performance et de conception.
        claims.put("roles", userDetails.getAuthorities().stream()
                .map(GrantedAuthority::getAuthority)
                .collect(Collectors.toList()));

        return createToken(claims, userDetails.getUsername());
    }

    /**
     * Valide un token JWT.
     * La validation se fait en deux etapes cruciales pour la securite :
```

```
75      * 1. On verifie que le nom d'utilisateur dans le token correspond a celui de l'
    utilisateur en cours de traitement.
76      * 2. On s'assure que le token n'a pas expire.
77      *
78      * @param token Le JWT a valider.
79      * @param userDetails Les details de l'utilisateur a comparer avec les
    informations du token.
80      * @return true si le token est valide, false sinon.
81      */
82     public boolean isTokenValid(String token, UserDetails userDetails) {
83         final String username = extractUsername(token);
84         return (username.equals(userDetails.getUsername())) && !isTokenExpired(token)
    ;
85     }
86
87     // --- Methodes privees utilitaires ---
88
89     /**
90      * Verifie si un token a expire en comparant sa date d'expiration avec la date
    actuelle.
91      *
92      * @param token Le JWT a verifier.
93      * @return true si le token a expire, false sinon.
94      */
95     private boolean isTokenExpired(String token) {
96         return extractExpiration(token).before(new Date());
97     }
98
99     /**
100     * Extrait la date d'expiration d'un token JWT.
101     *
102     * @param token Le JWT.
103     * @return La date d'expiration (java.util.Date).
104     */
105    private Date extractExpiration(String token) {
106        return extractAllClaims(token).getExpiration();
107    }
108
109    /**
110     * Methode centrale pour la creation du token.
111     * Elle utilise un "builder" pour assembler les differentes parties du JWT :
112     * - Les revendications (claims), incluant les claims personnalises et le sujet.
113     * - La date d'emission (Issued At).
114     * - La date d'expiration (Expiration).
115     * - La signature avec l'algorithme et la cle secrete.
116     *
117     * @param claims Les revendications (payload) a inclure dans le token.
118     * @param subject Le sujet du token (generalement le nom d'utilisateur).
119     * @return Le JWT final, serialise sous forme de chaine compacte.
120     */
121    private String createToken(Map<String, Object> claims, String subject) {
122        return Jwts.builder()
123                .setClaims(claims)
124                .setSubject(subject)
125                .setIssuedAt(new Date(System.currentTimeMillis()))
126                .setExpiration(new Date(System.currentTimeMillis() + expiration))
127                .signWith(getSigningKey()) // Signature du token
128                .compact(); // Construction et serialisation du token
129    }
130
131    /**
132     * Extrait toutes les revendications (claims) d'un token.
133     * C'est une operation critique : avant d'extraire le corps (payload), la
    bibliotheque `jjwt`
134     * va d'abord verifier la signature du token en utilisant la cle secrete.
135     * Si la signature est invalide (le token a ete altere), une exception sera levee
```

```
    ,
     * garantissant ainsi l'integrite des donnees.
     *
     * @param token Le JWT a parser.
     * @return Un objet Claims contenant toutes les informations du payload.
     */
    private Claims extractAllClaims(String token) {
        return Jwts.parserBuilder()
                .setSigningKey(getSigningKey())
                .build()
                .parseClaimsJws(token)
                .getBody();
    }

    /**
     * Genere la cle de signature a partir de la cle secrete encodee en Base64.
     * Cette methode decode d'abord la cle secrete, puis cree une instance de `Key`
     * specifique a l'algorithme HMAC-SHA.
     *
     * @return L'objet Key pret a etre utilise pour la signature/verification.
     */
    private Key getSigningKey() {
        byte[] keyBytes = Decoders.BASE64.decode(secretKey);
        return Keys.hmacShaKeyFor(keyBytes);
    }
}
```

Listing 13 – JwtService.java

5.6.3 Création du JwtAuthenticationFilter

Maintenant que notre `JwtService` est capable de gérer les tokens, nous devons créer le composant qui l'utilisera à chaque requête entrante. C'est le rôle du `JwtAuthenticationFilter`.

Ce filtre, véritable "gardien" de notre API, a pour mission d'intercepter **chaque** requête (sauf celles explicitement autorisées comme `/login`), d'en extraire le token JWT, de le valider et, si tout est correct, d'authentifier l'utilisateur pour la durée de la requête.

Nous étendons la classe `OncePerRequestFilter` fournie par Spring. C'est une bonne pratique qui garantit que notre filtre ne s'exécutera qu'**une seule fois** par requête, même en cas de redirections internes au sein du serveur.

Créez la classe `JwtAuthenticationFilter` dans votre package `security`.

```
package com.poly.gestioncatalogue.security;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;
import org.springframework.lang.NonNull;
import org.springframework.security.authentication.
    UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource
    ;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import java.io.IOException;

@Component
@AllArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {

```

```
22      private  JwtService jwtService;
23      private  UserDetailsService userDetailsService;
24
25      @Override
26      protected void doFilterInternal(
27              @NonNull HttpServletRequest request,
28              @NonNull HttpServletResponse response,
29              @NonNull FilterChain filterChain
30      ) throws ServletException, IOException {
31
32
33          String authHeader = request.getHeader("Authorization");
34          String jwt;
35          String username;
36
37
38          if (authHeader == null || !authHeader.startsWith("Bearer ")) {
39              filterChain.doFilter(request, response);
40              return;
41          }
42
43
44          jwt = authHeader.substring(7);
45
46
47          username = jwtService.extractUsername(jwt);
48
49
50          if (username != null && SecurityContextHolder.getContext().getAuthentication
    () == null) {
51
52              UserDetails userDetails = this.userDetailsService.loadUserByUsername(
    username);
53
54
55              if (jwtService.isTokenValid(jwt, userDetails)) {
56
57                  UsernamePasswordAuthenticationToken authToken = new
    UsernamePasswordAuthenticationToken(
58                          userDetails,
59                          null, // On n'a pas besoin des credentials ici
60                          userDetails.getAuthorities()
61                  );
62
63                  authToken.setDetails(
64                          new WebAuthenticationDetailsSource().buildDetails(request)
65                  );
66
67                  SecurityContextHolder.getContext().setAuthentication(authToken);
68              }
69          }
70
71          filterChain.doFilter(request, response);
72      }
73  }
```

Listing 14 – Implémentation du filtre d'authentification JWT

Analyse du fonctionnement du filtre Le code ci-dessus suit une logique de sécurité rigoureuse :

- **Étape 1 & 2 :** Il vérifie la présence d'un token JWT dans le header `Authorization`. Si aucun token n'est présenté, il ne fait rien et laisse les autres filtres de sécurité de Spring décider si la ressource demandée est publique ou non.
- **Étape 3 :** Il délègue l'extraction du `username` au `JwtService`. C'est une bonne séparation des responsabilités.

— **Étape 4 :** C'est le cœur de la validation. Le filtre vérifie que l'utilisateur n'est pas déjà authentifié dans le contexte de sécurité actuel. Ensuite, il charge les données de l'utilisateur (y compris les rôles/autorisations) depuis la base de données via le `UserDetailsService`. Cette étape est cruciale car elle garantit que les permissions de l'utilisateur sont toujours à jour. Enfin, il utilise le `JwtService` pour valider la signature et la date d'expiration du token par rapport aux données de l'utilisateur.
— **Mise à jour du Contexte de Sécurité :** Si tout est valide, le filtre crée un objet `UsernamePasswordAuthenticationToken` et le place dans le `SecurityContextHolder`. C'est l'action qui dit officiellement à Spring Security : « Pour cette requête, cet utilisateur est authentifié et voici ses autorisations ».
— **Étape 5 :** Quoi qu'il arrive, la requête est finalement passée au filtre suivant dans la chaîne, qui pourra alors utiliser le contexte de sécurité que nous venons (ou non) de mettre en place.

5.7 Étape 6 : Création de l'Endpoint d'Authentification

Pour permettre aux utilisateurs de s'identifier, nous devons exposer un endpoint public qui acceptera un nom d'utilisateur et un mot de passe, et qui retournera un token JWT en cas de succès.

5.7.1 Définition du Contrat d'API (DTO)

D'abord, nous créons une classe `AuthenticationRequest` qui définit la structure JSON attendue. C'est une pratique essentielle qui rend notre API robuste et claire.

```java
package com.poly.gestioncatalogue.model;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

/**
 * DTO (Data Transfer Object) représentant les données attendues
 * pour une requête d'authentification.
 */
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class AuthenticationRequest {
    private String username;
    private String password;
}
```

Listing 15 – DTO pour la requête d'authentification (AuthenticationRequest.java)

5.7.2 Implémentation du Contrôleur

Ensuite, nous créons le contrôleur qui utilise ce DTO pour traiter la requête de login.

```java
package com.poly.gestioncatalogue.controller;

import com.poly.gestioncatalogue.model.AuthenticationRequest;
import com.poly.gestioncatalogue.security.JwtService;
import lombok.AllArgsConstructor;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.
    UsernamePasswordAuthenticationToken;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
```

```java
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/auth")
@AllArgsConstructor
public class AuthController {

    private final AuthenticationManager authenticationManager;
    private final JwtService jwtService;
    private final UserDetailsService userDetailsService;

    @PostMapping("/login")
    public String login(@RequestBody AuthenticationRequest request) {
        // Valide les identifiants en utilisant le manager de Spring Security
        authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(
                request.getUsername(),
                request.getPassword()
            )
        );

        // Si l'authentification réussit, on charge les détails de l'utilisateur
        final UserDetails user = userDetailsService.loadUserByUsername(request.
    getUsername());

        // On génère et retourne le token JWT
        return jwtService.generateToken(user);
    }
}
```

Listing 16 – Contrôleur d'authentification (AuthController.java)

5.8 Étape 7 : Application de l'Autorisation par Rôles

Maintenant, nous restreignons l'accès aux fonctionnalités de modification des produits aux seuls administrateurs en utilisant l'annotation `@PreAuthorize`.

```java
package com.poly.gestioncatalogue.controller;

import com.poly.gestioncatalogue.entities.Produit;
import com.poly.gestioncatalogue.service.IProduitService;
import lombok.AllArgsConstructor;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping("/api")
@CrossOrigin("*")
@AllArgsConstructor
public class ProduitController {

    private final IProduitService produitService;

    @GetMapping("/products")
    public List<Produit> getAll() {
        // Accessible a tous les utilisateurs authentifies
        return produitService.getAllProduits();
    }

    @PostMapping("/products/add")
    @PreAuthorize("hasAuthority('ADMIN')") // Seul un ADMIN peut executer
    public Produit addProduit(@RequestBody Produit produit) {
        return produitService.saveProduit(produit);
    }
```

```
29
30      @PutMapping("/products/update")
31      @PreAuthorize("hasAuthority('ADMIN')")
32      public Produit updateProduit(@RequestBody Produit produit) {
33          return produitService.updateProduit(produit);
34      }
35
36      @DeleteMapping("/products/{id}")
37      @PreAuthorize("hasAuthority('ADMIN')")
38      public void deleteProduit(@PathVariable Long id) {
39          produitService.deleteProduitById(id);
40      }
41
42      // ... les autres methodes GET restent accessibles a tous les users authentifies
43  }
```

Listing 17 – Mise à jour du ProduitController.java

5.9 Étape 8 : Test de la Solution Complète avec Swagger UI

La configuration est terminée. Il est temps de vérifier que l'ensemble de notre système de sécurité fonctionne comme attendu. Nous utiliserons Swagger UI, l'interface de documentation interactive générée par Springdoc, pour simuler un client API.

5.9.1 Configuration Requise

Pour que Swagger UI puisse interagir avec notre API sécurisée, deux configurations sont nécessaires.

1. Activer la Sécurité dans Swagger UI Nous devons informer Swagger de notre schéma de sécurité "Bearer Token". Cela ajoutera un bouton "Authorize" global à l'interface, permettant aux développeurs de s'authentifier. Ajoutez ce Bean dans votre classe de configuration (par exemple, `BeanConfig.java` ou votre classe principale).

```
1  import io.swagger.v3.oas.models.info.Info;
2  import io.swagger.v3.oas.models.OpenAPI;
3  import io.swagger.v3.oas.models.security.SecurityRequirement;
4  import io.swagger.v3.oas.models.security.SecurityScheme;
5  import org.springframework.context.annotation.Bean;
6  import org.springframework.context.annotation.Configuration;
7
8  @Configuration
9  public class OpenApiConfig { // ou ajoutez le Bean dans une classe existante
10
11      @Bean
12      public OpenAPI customOpenAPI() {
13          final String securitySchemeName = "bearerAuth";
14          return new OpenAPI()
15
16                  .info(new Info().title("Catalogue API").version("1.0"))
17
18                  .addSecurityItem(new SecurityRequirement().addList(securitySchemeName
        ))
19
20                  .components(new io.swagger.v3.oas.models.Components()
21                          .addSecuritySchemes(securitySchemeName,
22                                  new SecurityScheme()
23                                          .name(securitySchemeName)
24                                          .type(SecurityScheme.Type.HTTP)
25                                          .scheme("bearer")
26                                          .bearerFormat("JWT")
27                          )
28                  );
29      }
```

```
}
```

Listing 18 – Configuration OpenAPI pour la sécurité JWT

2. Autoriser l'Accès Public à Swagger UI Notre `SecurityConfig` bloque tout par défaut. Nous devons explicitement autoriser l'accès aux ressources de Swagger UI, sinon la page de documentation elle-même serait inaccessible.

Modifiez votre `SecurityFilterChain` dans `SecurityConfig.java` :

```
// ... dans la méthode securityFilterChain(HttpSecurity http)

.authorizeHttpRequests(auth -> auth

    .requestMatchers("/api/auth/**").permitAll()

    // On autorise l'accès à la documentation Swagger
    .requestMatchers(
        "/v3/api-docs/**",
        "/swagger-ui/**",
        "/swagger-ui.html"
    ).permitAll()


    .anyRequest().authenticated()
)

// ... reste de la configuration
```

Listing 19 – Mise à jour de SecurityConfig pour autoriser Swagger

5.9.2 Scénario 1 : Authentification et obtention du Token (Admin)

Maintenant que tout est configuré, accédez à l'URL de Swagger UI dans votre navigateur : `http://localhost:80`

1. Cliquez sur la ligne `POST /api/auth/login`.
2. Cliquez sur le bouton **"Try it out"**.
3. Dans le champ "Request body", modifiez le JSON pour entrer les identifiants de l'admin :

```
{
    "username": "admin",
    "password": "1234"
}
```

4. Cliquez sur le bouton bleu **"Execute"**. Vous devriez recevoir une réponse avec un code **200** et le jeton JWT dans le corps de la réponse. **Copiez l'intégralité de ce jeton**.

5.9.3 Scénario 2 : Accès Autorisé (Action Admin)

Utilisons maintenant ce token pour autoriser toutes nos futures requêtes dans Swagger.

1. En haut à droite de la page Swagger, cliquez sur le bouton **"Authorize"**.
2. Une fenêtre pop-up "Available authorizations" apparaît. Dans le champ "Value" pour `bearerAuth`, **collez le jeton JWT** de l'admin que vous venez de copier.
3. Cliquez sur **"Authorize"**, puis fermez la fenêtre. Le cadenas du bouton est maintenant fermé.
4. Naviguez jusqu'à l'endpoint protégé, par exemple le `product-controller`, et développez la requête `DELETE /api/products/{id}`.

5. Cliquez sur **"Try it out"**, entrez un ID de produit existant (ex : 1), et cliquez sur **"Execute"**.
6. La requête doit réussir avec un statut **200 OK**, confirmant que l'admin a bien le droit de supprimer un produit.

5.9.4 Scénario 3 : Accès Refusé (Action Admin avec un Token User)

Vérifions maintenant que notre protection basée sur les rôles est efficace.

1. Répétez le **Scénario 1** avec les identifiants de l'utilisateur standard (`"username": "user"`). Copiez le nouveau token obtenu.
2. Cliquez à nouveau sur le bouton **"Authorize"** en haut à droite.
3. Cliquez d'abord sur **"Logout"** pour effacer l'ancien token, puis collez le **nouveau token de l'utilisateur "user"** dans le champ "Value" et cliquez sur "Authorize".
4. Retournez à la requête `DELETE /api/products/{id}` et exécutez-la de nouveau.
5. Comme attendu, la requête doit échouer avec un statut **403 Forbidden**. La réponse indique que l'accès est refusé, confirmant que notre sécurité a bien bloqué l'utilisateur qui n'a pas l'autorité `'ADMIN'`.