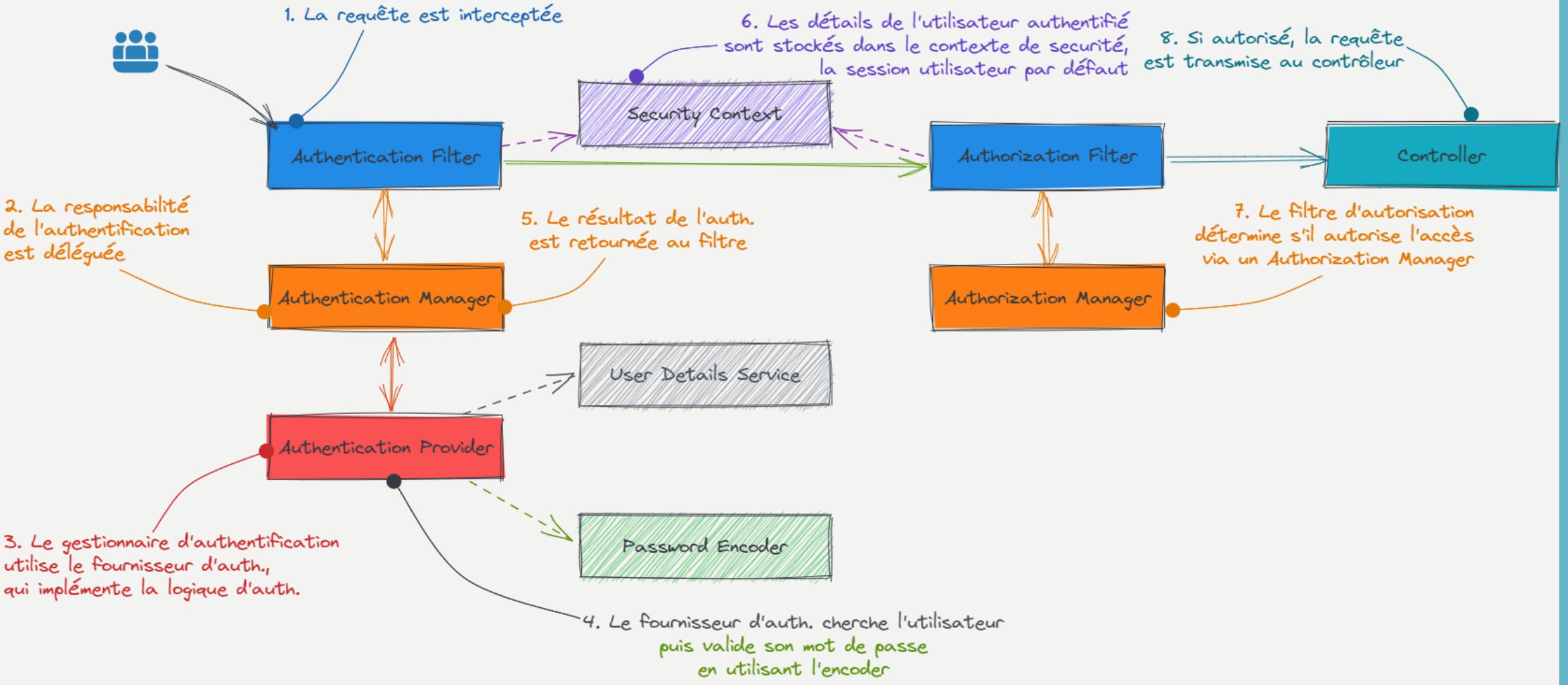
01/12/2022
Version 4

SPRING

INTRODUCTION

INTRODUCTION ET CONFIGURATION

INTRODUCTION



INTRODUCTION

- Spring Security a deux aspects
  - Authentification
    - Authentifie un utilisateur sur le système (connexion, session, jeton, etc.)
    - Si l'authentification échoue, le filtre des autorisations ne sera pas invoqué, un statut 401 sera retourné
  - Autorisation
    - Détermine si l'utilisateur connecté possède les privilèges suffisants pour accéder à la ressource demandée
    - Si l'autorisation est insuffisante, une statut 403 sera retourné

INTRODUCTION

- Les filtres
  - Sont appelés lorsqu'une requête HTTP est reçu
  - Font partie d'un ensemble de filtres, gérés dans une chaine de filtres **FilterChain**
    - Qui utilise le pattern *Chain of Responsibility*

CONFIGURATION

- Spring Security requiert 2 dépendances
  - **spring-security-web**
  - **spring-security-config**
- Spring Security (via Spring Boot) requiert un starter
  - spring-boot-starter-security

CONFIGURATION

- Hors Spring Boot, mise en place du filtre **DelegatingFilterProxy**
  - On active le filtre sur toutes les ressources (« /* »)
  - On sécurise l'ensemble des pages

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

CONFIGURATION

- Définition des URL autorisées

```
<security:http pattern="/assets/**" security="none" />

<security:http auto-config="true">
  <security:intercept-url pattern="/**" access="hasAnyRole('ADMIN', 'USER')" />
</security:http>
```

CONFIGURATION

- Définition des utilisateurs et de leur rôles
  - Avec le préfix {noop} pour préciser qu'on ne chiffre pas les mots de passe (**NoOpPasswordEncoder**)

```
<security:authentication-manager>
  <security:authentication-provider>
    <security:user-service>
      <security:user name="admin" password="{noop}admin123456" authorities="ROLE_ADMIN" />
      <security:user name="user" password="{noop}mdp123456" authorities="ROLE_USER" />
    </security:user-service>
  </security:authentication-provider>
</security:authentication-manager>
```

CONFIGURATION

- Configuration par classe

```
@Configuration
public class SecurityConfig {

}
```

NOTE : Depuis la version 5.7, l'héritage de `WebSecurityConfigurerAdapter` est déprécié. Elle a été supprimée en version 6.0.

CONFIGURATION

```java
@Bean
public UserDetailsService inMemory() {
  InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();

  manager.createUser(User.withUsername("user")
    .password("{noop}mdp123456")
    .roles("USER")
    .build()
  );

  manager.createUser(User.withUsername("admin")
    .password("{noop}admin123456")
    .roles("ADMIN")
    .build()
  );

  return manager;
}
```

CONFIGURATION

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  http
    .httpBasic()
    .and()
    .authorizeHttpRequests(authorize -> {
      authorize.requestMatchers("/**").authenticated();
    });

  return http.build();
}
```

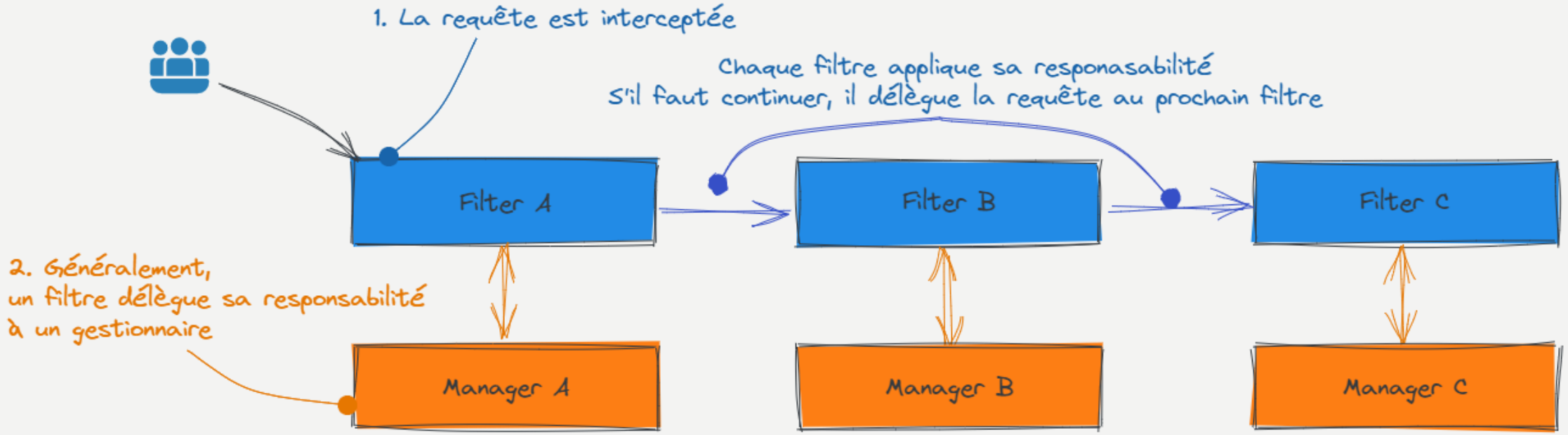
EXERCICE

- Créer un projet Spring Boot, MVC, sécurisé
  - /hello GET, retourne « hello world », accessible sans sécurité
  - /secured-hello GET, retourne « hello securisé », accessible si authentifié

FILTRES

L'ENCHAINEMENT DES FILTRES

FILTRES


1. La requête est interceptée
Chaque filtre applique sa responasabilité
S'il faut continuer, il délègue la requête au prochain filtre
Filter A
Filter B
Filter C
2. Généralement,
un filtre délègue sa responsabilité
à un gestionnaire
Manager A
Manager B
Manager C

FILTRES

- Dans la configuration précédente, utilisation d'un bean **SecurityFilterChain**
  - Ceux utilisés sont ceux fournis par Spring Security
  - Il est possible d'ajouter nos propres filtres
    - Avec une classe qui implémente `Filter`

```
public class DemoFilter implements Filter {
  @Override
  public void doFilter(ServletRequest req, ServletResponse resp, FilterChain filterChain) throws ... {
    // TODO

    filterChain.doFilter(req, resp); // On continue
  }
}
```

FILTRES

- On ajoute le filtre dans la configuration
  - Avant un autre filtre
  - Après un autre filtre
  - A la fin
  - Au même niveau qu'un autre filtre … (l'ordre entre 2 filtres n'est pas garanti)

```
http.addFilterBefore(new DemoFilter(), UsernamePasswordAuthenticationFilter.class);
```

https://docs.spring.io/spring-security/reference/6.0.0/servlet/architecture.html#servlet-security-filters

FILTRES

- Spring Security fourni des classes abstraites qui implémentent l'interface `Filter`
  - Aide au développement
  - Une couramment utilisée est `OncePerRequestFilter`
    - En effet, un filtre peut être appelé plusieurs fois dans une requête

```
public class DemoOncePerRequestFilter extends OncePerRequestFilter {
  @Override
  protected void doFilterInternal(HttpServletRequest req, HttpServletResponse resp, FilterChain fc) throws ... {
    // TODO ...
    fc.doFilter(request, response);
  }
}
```

EXERCICE

- Ajouter un nouveau filtre
  - N'autoriser que les requêtes qui ont dans l'en-tête « Authorization »
  - Ajouter le rôle ADMIN si Authorization = « admin »
    - Utiliser la classe `UsernamePasswordAuthenticationToken`

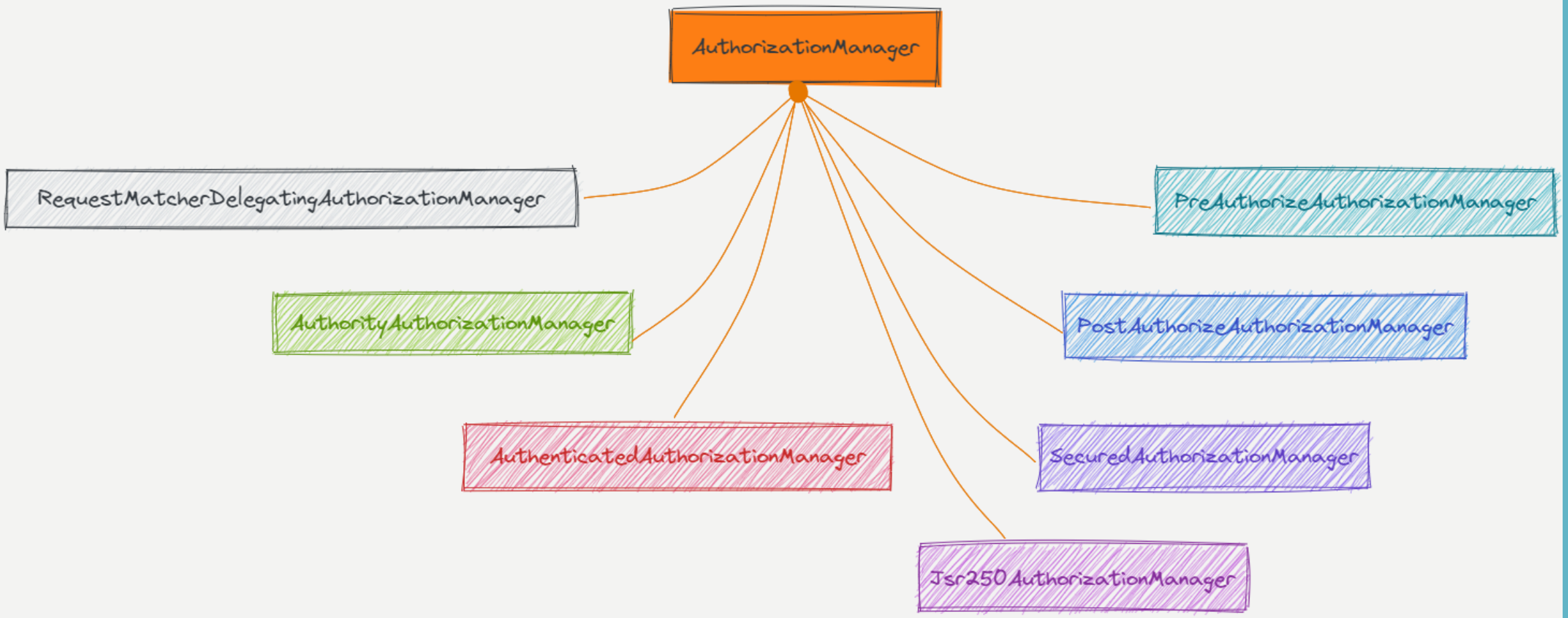
```
SecurityContextHolder.getContext().setAuthentication(...);
```

```
List.of(
  new SimpleGrantedAuthority("ROLE_ADMIN")
);
```

ACCESS

RESTRICTIONS D'ACCÈS PERSONNALISÉS

ACCESS


AuthorizationManager
RequestMatcherDelegatingAuthorizationManager
AuthorityAuthorizationManager
AuthenticatedAuthorizationManager
Jsr250AuthorizationManager
SecuredAuthorizationManager
PostAuthorizeAuthorizationManager
PreAuthorizeAuthorizationManager

ACCESS

- Spring Security fourni des gestionnaires d'accès
  - `AuthenticatedAuthorizationManager`
    - anonymous() l'utilisateur n'est pas authentifié
    - authenticated() l'utilisateur est authentifié
    - fullyAuthenticated() l'utilisateur est authentifié sans « remember me »
    - rememberMe() l'utilisateur est « remember me »
  - `AuthorityAuthorizationManager`
    - hasAuthority(..) l'utilisateur possède l'autorisation
    - hasAnyAuthority(..) l'utilisateur possède l'une des autorisations
    - hasRole(..) l'utilisateur possède le rôle
    - hasAnyRole(..) l'utilisateur possède l'un des rôles
  - permitAll() autoriser tout le monde
  - denyAll() refuser tout le monde
  - access() gestionnaire d'accès personnalisé

ACCESS

- Concernant la liste des autorisations
  - Un rôle est une autorisation préfixée de « ROLE_ »
  - Un utilisateur peut avoir autant d'autorisations que nécessaire

ACCESS

- Pour ajouter son propre gestionnaire
  - Une classe qui implémente l'interface `AuthorizationManager`
    - `AuthorizationManager<RequestAuthorizationContext>` pour un contexte de requête

```java
public class FormationAuthorizationManager implements AuthorizationManager<RequestAuthorizationContext> {
  @Override
  @Nullable
  public AuthorizationDecision check(Supplier<Authentication> authSup, RequestAuthorizationContext authCtx) {
      // TODO ...
      return AuthorityAuthorizationManager.hasRole("ADMIN").check(authSup, authCtx);
  }
}
```

  - La méthode `check` retourne
    - `AuthorizationDecision(true)` si autorisé
    - `AuthorizationDecision(false)` si non autorisé
    - `null` si n'arrive pas à déterminer une décision

ACCESS

- `AccessDecisionManager` et `AccessDecisionVoter`, remplacés par
  - `AuthorizationManager`
- `FilterSecurityInterceptor`, remplacé par
  - `AuthorizationFilter`

NOTE : Depuis la version 5.7, sont dépréciés :
`AccessDecisionManager`
`AccessDecisionVoter`
`FilterSecurityInterceptor`

ACCESS

- Pour ajouter une hiérarchie des rôles (rôle ou autorisation au sens large)
  - ROLE_ADMIN inclura ROLE_USER et ROLE_GUEST
  - ROLE_USER inclura ROLE_GUEST

```
@Bean
public void RoleHierarchy eroleHierarchy() {
  RoleHierarchyImpl roleHierarchy = new RoleHierarchyImpl();
  String hierarchy = """
    ROLE_ADMIN > ROLE_USER
    ROLE_USER > ROLE_GUEST
  """;

  roleHierarchy.setHierarchy(hierarchy);

  return roleHierarchy;
}
```

ACCESS

- Pour récupérer la liste des autorisations complétée de la hiérarchie

```
roleHierarchy.getReachableGrantedAuthorities(auth.getAuthorities());
```

  - Sinon, Spring le fait seul avec les gestionnaires d'autorisation disponibles

USER DETAILS

LES UTILISATEURS EN BDD

USER DETAILS

- Mise en place d'un `UserDetailsService` spécifique
  - Il en existe par défaut dans Spring Security
    - `InMemoryUserDetailsManager`
    - `JdbcUserDetailsManager`

USER DETAILS

- Nous avons jusqu'ici défini nos utilisateurs dans la configuration
- Pour utiliser une liste d'utilisateur en base de données
  - Définition d'un **@Service** qui implémente `UserDetailsService`
  - Son rôle est de rechercher un utilisateur par son nom d'utilisateur
    - La cohérence du mot de passe se fera après avoir récupéré l'utilisateur

```
@Service
public class JpaUserDetailsService implements UserDetailsService {
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        return null;
    }
}
```

USER DETAILS

- La classe étant annotée de **@Service**
  - Spring Boot la reconnait comme classe `UserDetailsService` à utiliser
    - Attention cependant, s'il en existe plusieurs dans le contexte de Spring, il ne saura pas laquelle utiliser
      - Spring répondra alors par « No AuthenticationProvider found »
  - Son rôle sera de retourner un `UserDetails` (en allant chercher un utilisateur en base par exemple)

USER DETAILS

- `UserDetailsService` manipule la classe `UserDetails`
  - C'est cette classe qui joue le rôle de l'utilisateur à authentifier
  - Il faut créer une classe qui encapsule le modèle Utilisateur, et qui implémente `UserDetails`

```
public class SecurityUser implements UserDetails {
}
```

USER DETAILS

- Il faut ensuite modifier le comportement des méthodes de l'interface

Méthode	Fonction	Valeur de retour
getAuthorities	Liste des autorisations, des rôles	Une liste d'autorisations
getPassword	Récupérer le mot de passe	Le mot de passe encodé
getUsername	Récupérer le nom d'utilisateur	Le nom d'utilisateur
isAccountNonExpired	Vérifie que le compte n'a pas expiré	Vrai
isAccountNonLocked	Vérifie que le compte n'est pas verrouillé	Vrai
isCredentialsNonExpired	Vérifie que les identifiants n'ont pas expirés	Vrai
isEnabled	Vérifie que le compte est actif	Vrai

USER DETAILS

```
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
  return List.of(
    new SimpleGrantedAuthority("ROLE_ADMIN")
  );
}
```

```
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
  return List.of(
    () -> "ROLE_ADMIN"
  );
}
```

USER DETAILS

- Il faut ensuite indiquer comment sont chiffrés les mots de passe
  - En utilisant un **PasswordEncoder**
    - Pas de chiffrement

```
@Bean
public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
}
```

    - Chiffrement Blowfish (bcrypt)

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

EXERCICE

- Modifier la sécurité
  - Les utilisateurs sont stockés en base de données (utiliser une base de donnée embarquée H2)
  - Leur mot de passe est chiffré !

AUTH PROVIDER

AUTHENTICATION PROVIDER

AUTHENTICATION PROVIDER

- Il est possible de fabriquer son propre fournisseur d'authentification
  - Une classe qui implémente `AuthenticationProvider`
  - Managée par Spring, elle sera automatiquement reconnue par Spring Boot Security
    - Et remplacera le fournisseur existant par défaut … et donc le `UserDetailsService` éventuel

AUTHENTICATION PROVIDER

```java
@Override
public Authentication authenticate(Authentication auth) throws AuthenticationException {
  String username = auth.getName();
  String password = auth.getCredentials().toString();

  if ("admin".equals(username) && "123456$".equals(password)) {
    return new UsernamePasswordAuthenticationToken(username, password, List.of(() -> "ROLE_ADMIN"));
  }

  else {
    throw new BadCredentialsException("Echec de l'authentification");
  }
}

@Override
public boolean supports(Class<?> auth) {
  return auth.equals(UsernamePasswordAuthenticationToken.class);
}
```

AUTHENTICATION PROVIDER

- Pour utiliser plusieurs `AuthenticationProvider`
  - Déclaration d'une nouvelle instance `AuthenticationManager`

```
@Bean
public AuthenticationManager authManager(HttpSecurity http, JpaUserDetailsService authService,
FormationAuthenticationProvider formationAuthProvider) throws Exception {
  AuthenticationManagerBuilder authManagerBuilder = http.getSharedObject(AuthenticationManagerBuilder.class);

  authManagerBuilder.authenticationProvider(formationAuthProvider);
  authManagerBuilder.userDetailsService(authService);

  return authManagerBuilder.build();
}
```

AUTHENTICATION PROVIDER

- Puis l'utiliser dans la configuration du `FilterChain`
  - Pour demander à Spring de l'utiliser

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http, AuthenticationManager authManager) throws Exception {
  http
    .httpBasic()
    .and()
    .authorizeHttpRequests(authManagerRequest -> {
      authManagerRequest.requestMatchers("/api/hello").permitAll();
    })
    .authenticationManager(authManager);

  return http.build();
}
```

AUTHENTICATION PROVIDER

- Spring ne gère qu'un seul `AuthenticationManager` à la fois
- Un `AuthenticationManager` ne gère qu'un seul `AuthenticationProvider`
  - Mais on peut lui associe un (et un seul)
    - `InMemoryAuthentication`
    - `LdapAuthentication`
    - `UserDetailsService`
  - Si besoin d'en associer plus
    - un `AuthenticationProvider` ou un `UserDetailsService` peut en appeler un autre

EXERCICE

- Remplacer le service d'authentification par un nouveau provider
  - Il vérifie l'utilisateur en base et recherche ses autorisations, comme le service

CSRF

CROSS-SITE REQUEST FORGERY

CSRF

- Par défaut, la protection CSRF est activée

```
http.csrf().disable();
```

```
http.csrf(c -> c.ignoringRequestMatchers("/api/hello", "..."));
```

CORS

CROSS-ORIGIN RESOURCE SHARING

CORS

- Lorsqu'une requête HTTP est envoyée par le navigateur, via Javascript
  - Le navigateur s'attend à recevoir les en-têtes suivantes
    - `Access-Control-Allow-Headers`
    - `Access-Control-Allow-Methods`
    - `Access-Control-Allow-Origins`
  - Cette politique permet de n'autoriser que les requêtes
    - D'un domaine concerné (*origins*)
    - Pour une ou des méthodes HTTP (*methods* – par défaut, **GET** et **HEAD** autorisés)
    - Avec éventuellement des en-têtes HTTP (*headers*)
  - Et si les headers n'existent pas dans la réponse HTTP, le navigateur empêche l'action de se réaliser

CORS

- Lorsque le navigateur envoie une requête HTTP avec des en-têtes particulières
  - Un *Content-Type* dans une requête **POST** par exemple
  - Le navigateur effectue une première requête **OPTIONS**, qu'on appelle *pre-flight*
    - Son objectif est de déterminer si la politique CORS configurée autorise
      - Les en-têtes **HTTP** utilisés
      - La méthode **HTTP** utilisée
      - Pour l'origine du site

CORS

- Par défaut, aucune en-tête HTTP CORS n'est ajoutée

```
http.cors(Customizer.withDefaults());
```

  - Ici, Spring cherchera un **bean** *corsConfigurationSource* de type `CorsConfigurationSource`

CORS

- Qu'on peut spécifier en tant que **bean**

```
@Bean
public CorsConfigurationSource corsConfigurationSource() {
  UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
  CorsConfiguration configuration = new CorsConfiguration();

  configuration.setAllowedHeaders(List.of("Content-Type"));
  configuration.setAllowedMethods(List.of("GET", "POST"));
  configuration.setAllowedOrigins(List.of("*"));

  source.registerCorsConfiguration("/**", configuration);

  return source;
}
```

CORS

- Ou qu'on peut créer et utiliser ici

```
http.cors(c -> {
  CorsConfigurationSource source = request -> {
    CorsConfiguration config = new CorsConfiguration();

    config.setAllowedHeaders(List.of("*"));
    config.setAllowedMethods(List.of("*"));
    config.setAllowedOrigins(List.of("*"));

    return config;
  };

  c.configurationSource(source);
});
```

  - En utilisant « * » sur la liste des méthodes et des en-têtes
    - Spring Security répondra que la seule méthode autorisée est celle interrogée, idem pour les en-têtes

CORS

- Il est également possible d'appliquer une en-tête CORS avec l'annotation **@CrossOrigin**
  - Sur une classe contrôleur ou sur une méthode par exemple

JWT

LES JETONS JSON

JWT

- Le protocole HTTP(S) est un protocole dit « déconnecté »
  - Chaque requête est indépendante
  - Il n'y a, par défaut, aucune stratégie de cohérence d'utilisateur

- Pour reconnaitre un utilisateur
  - Utilisation des sessions (côté serveur)
  - Utilisation des cookies (côté client – navigateur Web par exemple)
  - Les cookies sont envoyés à chaque requête, par le client, dans les en-têtes HTTP
    - Cookie
    - Sensible aux attaques CSRF

JWT

- JSON **W**eb **T**oken
  - Jeton généré par un serveur d'authentification, transmis au client
  - Remplace les sessions et cookies
  - Envoyés à chaque requête, dans les en-têtes HTTP
    - Authorization, avec le type d'autorisation « Bearer »
    - Il est aussi possible de le stocker et l'envoyer en tant que Cookie, mais attention aux attaques CSRF !

- Contrairement au « Cookie de session », le jeton peut contenir des informations
  - Elles sont signées à l'aide d'une clé privée détenue par le serveur
  - Lorsque le serveur reçoit un jeton, il compare la signature pour vérifier la validité

JWT

- Un jeton JWT est composé
  - Header
    - Contenant l'algorithme utilisé pour la signature et le type de jeton (JWT)
    - Au format JSON, encodé en Base64
  - Payload
    - Contenant les informations du jeton (nom d'utilisateur, scope, date d'émission, date d'expiration, etc.)
    - Au format JSON, encodé en Base64
  - Signature
    - Concaténation de Header et Payload, chiffrée avec la clé privée
    - Empêche donc l'altération des données

https://jwt.ms/

JWT

- Exemple Header JWT

```
{
  "alg":"HS512",
  "typ":"JWT"
}
```

- Exemple Payload JWT

```
{
  "iat": 1669658768,
  "exp": 1669662368,
  "sub": "jeremy",
  "scope": "produit.read"
}
```

JWT

- Exemple JWT

```
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJqZXJlbXkiLCJzY29wZSI6InByb2R1aXQucmVhZCIsImlhdCI6MTY2OTY1ODc2OCwiZXhwIjoxNjY5NjYyMzY4fQ.r46SL8XB43Wzxql6ZvcpsAhvuxwLnEhMqeKZRvq86MFxffsibxBHq9m9jwv47iDCPel5xvhC33h0-6OcmNITLA
```

- Header
- Payload
- Signature

https://jwt.io/

JWT

- Pour utiliser le jeton, on ajoute « Authorization » à l'en-tête HTTP
- `Authorization` `Bearer Token`

- « Bearer » fait référence au type d'authorization spécifique
  - Souvent utilisé pour les jetons d'accès de ce type
- Il en existe d'autres
  - Basic — Authentification HTTP-Basic en Base64
  - Digest — Authentification HTTP-Basic en MD5
  - Negociate — Kerberos pour systèmes MS Windows
  - AWS4-HMAC-SHA256 — Pour AWS

JWT – TESTER

- Manipuler un Jeton
  - Consulter le jeton Azure ou Google
    - jwt.io ou jwt.ms
  - Fabriquer ou modifier un jeton
    - jwt.io

- Et en effet, le header et le payload sont en clair

JWT – EXERCICE

- Inclure les dépendances
  - jjwt-jackson & jjwt-impl (de io.jsonwebtoken)
  - Générer une clé SHA512
  - Utiliser un temps d'expiration de 3600000 ms

```
public String generateJwtToken(Authentication authentication) {
  SecretKey key = Keys.hmacShaKeyFor(this.jwtSecret.getBytes(StandardCharsets.UTF_8));

  return Jwts.builder()
    .setSubject(authentication.getName())
    .setIssuedAt(new Date())
    .setExpiration(new Date((new Date()).getTime() + this.jwtExpirationMs))
    .signWith(key)
    .compact();
}
```

JWT – EXERCICE

- Ajouter un filtre pour intercepter l'en-tête HTTP « Authorization »
  - Le jeton sera placé après « Bearer »
  - Retrouver l'utilisateur, y associer le ou les autorisations
  - Désactiver la protection CSRF
    - Qui n'est plus nécessaire car un jeton est demandé à chaque requête
  - Désactiver le stockage de la session utilisateur
    - L'utilisateur sera retrouvé grâce au jeton

HTTPS

FORCER LE HTTPS

HTTPS

- Dans Spring Boot, il est possible d'activer le protocole HTTPS
  - Dans le fichier application.properties

```
server.ssl.enabled = true
server.port = 8443
```

- Générer un certificat auto-signé
  - PKCS12 : Public Key Cryptographic Standards, protection par mot de passe

```
keytool -genkeypair -alias formation -keyalg RSA -keysize 2048 -storetype PKCS12 -keystore formation.p12 -validity 3650
```

- Puis inclure dans la configuration

```
server.ssl.key-store-type = PKCS12
server.ssl.key-store = classpath:keystore/formation.p12
server.ssl.key-store-password = ******
server.ssl.key-alias = formation
```

EXERCICE

- Mettre en place la configuration HTTPS

PAGE LOGIN

PAGE DE CONNEXION PERSONNALISÉE

PAGE LOGIN

- Par défaut, Spring propose une page d'authentification
  - Possible de la remplacer par une page personnalisée

PAGE LOGIN

- Configuration de la page de login

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .formLogin()
            .loginPage("/login")
            .loginProcessingUrl("/perform_login")
            .defaultSuccessUrl("/home", true)
            .failureUrl("/login?erreur");

    return http.build();
}
```

PAGE LOGIN

- Si la protection CSRF est activée
  - Cette protection demande un jeton à chaque envoie de formulaire
  - Il faut donc préciser ce jeton dans les paramètres envoyés via le formulaire
    - On peut utiliser un champ caché pour que ce ne soit pas visible

```
<input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}" />
```

```
<input type="hidden" th:name="${_csrf.parameterName}" th:value="${_csrf.token}" />
```

EXERCICE

- Page de connexion personnalisée
  - Afficher un message d'erreur sur la page de connexion si les identifiants saisis sont faux

ANNOTATIONS

ACCÈS PAR LES ANNOTATIONS

ACCÈS PAR ANNOTATIONS

- **@Secured**
- **@RolesAllowed**
- **@PreAuthorize / @PostAuthorize**
- **@PreFilter / @PostFilter**
- Annotations à utiliser
  - Sur une classe
  - Sur une méthode

ACCÈS PAR ANNOTATIONS

- **@PreAuthorize / @PostAuthorize**
  - Se base sur Spring Expression Language (SpEL)
  - **@PreAuthorize**("hasRole('1') and hasRole('2')")
  - ROLE_1 ET ROLE_2
- **@Secured**
  - Se base sur une liste
  - **@Secured**({ "ROLE_ADMIN", "ROLE_USER" })
  - ROLE_ADMIN OU ROLE_USER
- **@RolesAllowed**
  - Se base sur une liste
  - **@RolesAllowed**({ " ROLE_ADMIN", "ROLE_USER " })
  - ROLE_ADMIN OU ROLE_USER
  - Standard JAVA (JSR250)

ACCÈS PAR ANNOTATIONS

- Il faut activer les annotations dans la configuration

```
@EnableMethodSecurity(prePostEnabled = true, securedEnabled = true)
```

ACCÈS PAR ANNOTATIONS

- ***@PreAuthorize*** / ***@PostAuthorize***
  - hasRole("role")
  - hasAuthority("authorite")
  - hasAnyRole("role1", "role2")
  - hasAnyAuthority("authorite1", "authorite2")
  - isAnonymous()
  - isAuthenticated()
  - hasPermission()
    - #arg, returnObject

- A placer sur une méthode
  - D'un contrôleur
  - D'un service
  - D'une **Repository**

ACCÈS PAR ANNOTATIONS

- ***@PreFilter*** / ***@PostFilter***
  - hasRole("role")
  - hasAuthority("authorite")
  - hasAnyRole("role1", "role2")
  - hasAnyAuthority("authorite1", "authorite2")
  - isAnonymous()
  - isAuthenticated()
  - hasPermission()
    - #arg, filterObject

- A placer sur une méthode
  - D'un contrôleur
  - D'un service
  - D'une **Repository**

ACCÈS PAR ANNOTATIONS

- Il est possible d'utiliser des méta-annotations
  - Pour regrouper plusieurs annotations
  - Pour éviter d'écrire plusieurs fois les mêmes autorisations

```
@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("hasRole('ADMIN')")
public @interface IsAdmin { }
```

  - Dans cet exemple, l'utilisation de **@IsAdmin** remplacera l'annotation **@PreAuthorize**(…)

EXERCICE

- Sécuriser le contrôleur par des annotations

VUES

ACCÈS DANS LES VUES

ACCÈS DANS LES VUES

- Si utilisation des JSP
  - Nécessite la dépendance **spring-security-taglibs**
  - Nécessite la taglib dans chaque JSP

```
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="sec" %>
```

```
<sec:authorize access="hasRole('USER')">
  <!-- -->
</sec:authorize>
```

ACCÈS DANS LES VUES

- Si utilisation de Thymeleaf
  - Nécessite la dépendance **thymeleaf-extras-springsecurity4**
  - Nécessite l'ajout du dialect **SpringSecurityDialect**
  - Nécessite l'ajout de l'espace de nom XML

ACCÈS DANS LES VUES

- Dans chaque vue

```
<html xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
```

```
<div sec:authorize="hasRole('USER')">
  <!-- -->
</div>
```

EXERCICE

- Dans la vue de **HomeController**, afficher
  - « Bonjour administrateur » si c'est un administrateur
  - « Bonjour utilisateur » si c'est un utilisateur

EXERCICE

- Modifier les autorisations pour **ProduitController**
  - Les utilisateurs peuvent voir la liste des produits
    - Sans les boutons d'action
    - Sans le lien « ajouter un produit »

PERMISSIONS

LES PERMISSIONS

ACCÈS – HASPERMISSION

- Pour aller plus loin dans la configuration de la sécurité
  - Configuration de permissions avec *hasPermission*
  - Permet de définir des permissions sur un objet

ACCÈS – HASPERMISSION

- Plutôt que d'utiliser

```
@PreAuthorize("hasAuthority('PRODUIT_READ_PERMISSION')")
public String findAll() {
  //...
}
```

- On pourra utiliser
  - Et être plus précis

```
@PreAuthorize("hasPermission(null, 'produit', 'read')")
public String findAll() {
  //...
}
```

```
@PreAuthorize("hasPermission(#id, 'produit', 'read')")
public Produit findById(int id) {
  //...
}
```

```
@PostAuthorize("hasPermission(returnObject, 'read')")
public Produit findById(int id) {
  //...
}
```

ACCÈS – HASPERMISSION

- Création d'une classe qui implémente `PermissionEvaluator`
  - Implémentation des méthodes *hasPermission*
    - Permettent de déterminer si l'utilisateur a les permissions nécessaires pour accéder à l'objet

```
public boolean hasPermission(Authentication auth, Object targetDomainObject, Object permission) {
  if (targetDomainObject == null) {
    return false;
  }

  String targetType = targetDomainObject.getClass().getSimpleName().toUpperCase();
  return this.hasPrivilege(auth, targetType, permission.toString());
}
```

```
public boolean hasPermission(Authentication auth, Serializable targetId, String targetType, Object permission) {
  return this.hasPrivilege(auth, targetType, permission.toString());
}
```

ACCÈS – HASPERMISSION

```
private boolean hasPrivilege(Authentication auth, String targetType, String permission) {
  if ((auth == null) || (targetType == null) || !(permission instanceof String)) {
    return false;
  }

  return auth.getAuthorities().stream()
    .anyMatch(gauth ->
      gauth.getAuthority().startsWith(targetType.toUpperCase() + "_" + permission.toUpperCase())
    );
}
```

ACCÈS – HASPERMISSION

- Exemple d'autorisations pour un utilisateur `UserDetails`
  - On lui donne le rôle USER
  - On lui donne les privilèges de lecture et d'écriture sur les produits

```
public Collection<? extends GrantedAuthority> getAuthorities() {
  List<GrantedAuthority> myAuthorities = new ArrayList<GrantedAuthority>();

  myAuthorities.add(new SimpleGrantedAuthority("ROLE_USER"));
  myAuthorities.add(new SimpleGrantedAuthority("PRODUIT_READ_PRIVILEGE"));
  myAuthorities.add(new SimpleGrantedAuthority("PRODUIT_WRITE_PRIVILEGE"));

  return myAuthorities;
}
```

ACCÈS – HASPERMISSION

- Il faut ensuite activer ce `PermissionEvalutor` dans les expressions

```
@Bean
public static MethodSecurityExpressionHandler methodSecurityExpressionHandler() {
  DefaultMethodSecurityExpressionHandler handler = new DefaultMethodSecurityExpressionHandler();

  handler.setPermissionEvaluator(new FormationPermissionEvaluator());

  return handler;
}
```

EXERCICE

- Mettre en place la sécurité via des permissions
  - Les utilisateurs peuvent voir la liste des utilisateurs
  - Les administrateurs peuvent voir et modifier les utilisateurs

EXPRESSIONS

EVALUER SES PROPRES EXPRESSIONS

ACCÈS – EXPRESSIONS

- Il est possible, parce qu'on est limité dans la syntaxe des permissions
  - De créer de nouvelles expressions à utiliser en complément de *hasPermission*
    - On parlait de `SecurityExpressionRoot`, mais Spring Security tend vers une non-utilisation

ACCÈS – EXPRESSIONS

- Plutôt que d'utiliser

```
@PreAuthorize("hasRole('ADMIN') or hasPermission(#id, 'Produit', 'read')")
public Produit findById(int id) {
  //...
}
```

- On pourra utiliser (par exemple)

```
@PreAuthorize("@expr.isAdmin() or @expr.isReadable(#id, 'Produit')")
public Produit findById(int id) {
  //...
}
```

```
@PostAuthorize("@expr.isAdmin() or @expr.isReadable(returnObject)")
public Produit findById(int id) {
  //...
}
```

ACCÈS – EXPRESSIONS

- Créer une nouvelle classe Component

```
@Component("expr")
public class FormationExpression {
  public Boolean isAdmin() {
    Authentication auth = SecurityContextHolder.getContext().getAuthentication();

    return AuthorityAuthorizationManager.hasRole("ADMIN").check(() -> auth, null).isGranted();
  }
}
```

OAUTH2

FOURNISSEURS OIDC

OAUTH2

- Problématique : application de composition d'album photo
  - L'utilisateur peut charger une image depuis son poste (pas de problème ici)
  - L'utilisateur peut vouloir partager des photos enregistrées sur un Cloud
    - L'application peut demander à l'utilisateur propriétaire ses identifiants (login / password)
    - Et **jurer** qu'elle ne gardera rien en mémoire et ne fera que ce qu'elle est sensée faire : charger une image

- Avant l'arrivée de OAuth
  - Les identifiants étaient demandées (et souvent stockés en clair !)
  - L'application avait accès à tout ce qu'avait accès l'utilisateur propriétaire

OAUTH2

- OAuth2 pour **O**pen **Auth 2**.0
  - Comment une application peut-elle accéder à une ressource protégée, au nom de son propriétaire, sans connaitre ses identifiants (login / mot de passe) ?

- OAuth2, contrairement à la première version, **doit** s'exécuter en HTTP**S**
  - L'aspect de sécurité pour la confidentialité des données est délégué au protocole TLS

OAUTH2

- On distingue l'authentification et l'autorisation
- Authentification
  - Reconnaissance d'un utilisateur via ses identifiants de connexion
- Autorisation
  - Accès (ou non) à une ressource ou un ensemble de ressources
  - Exemple : un administrateur a plus d'autorisations qu'un utilisateur lambda

OAUTH2

- De façon classique, un processus d'authentification et d'autorisation
  - Un serveur, en mesure d'authentifier et d'autoriser l'accès à une ressource
  - Un utilisateur, en mesure de fournir ses identifiants
- Pour OAuth, le processus est un peu plus complexe
  - Propriétaire de la ressource **Resource owner**
    - Une entité (utilisateur par exemple) en mesure de donner l'accès à une ressource protégée
  - Client **Client**
    - Une entité (application ou site web) qui demande l'accès à une ressource protégée
  - Serveur de la ressource **Resource server**
    - Le serveur qui héberge la ressource protégée
  - Serveur d'autorisation **Authorization server**
    - Le serveur qui délivre le droit d'accès à la ressource protégée au client après avoir authentifié le propriétaire

OAUTH2

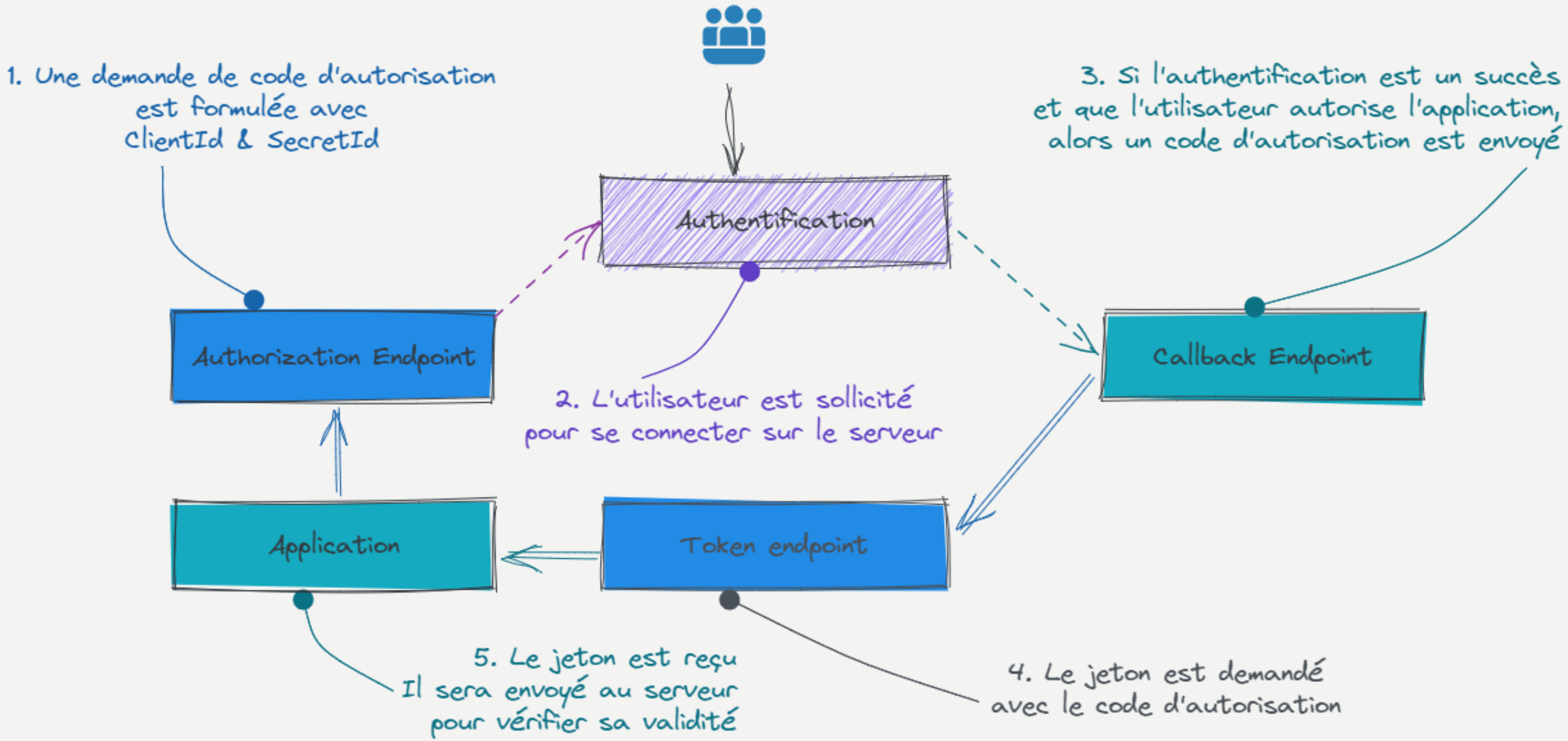
- La demande d'autorisation est toujours initiée par le client
  - Qu'il faut enregistrer auprès du serveur d'autorisation
- Son enregistrement nécessite
  - Un identifiant, l'identifiant du client
  - Un mot de passe, le secret du client
  - Une ou plusieurs URL de redirection, pour indiquer au client l'état de l'autorisation

OAUTH2

- La demande du client se traduit par la délivrance d'un jeton (token)
- Il en existe 2 types
  - Jeton d'accès **Access Token**
    - Jeton qui permet d'accès à une ressource protégée
    - Durée de validité
    - Peut avoir une portée (scope) limitée
      - Par exemple : lecture seule sur les photos uniquement
  - Jeton de rafraîchissement **Refresh Token**
    - Jeton qui permet d'obtenir un nouveau jeton d'accès sans l'intervention du propriétaire
    - Durée de validité, plus longue que le jeton d'accès

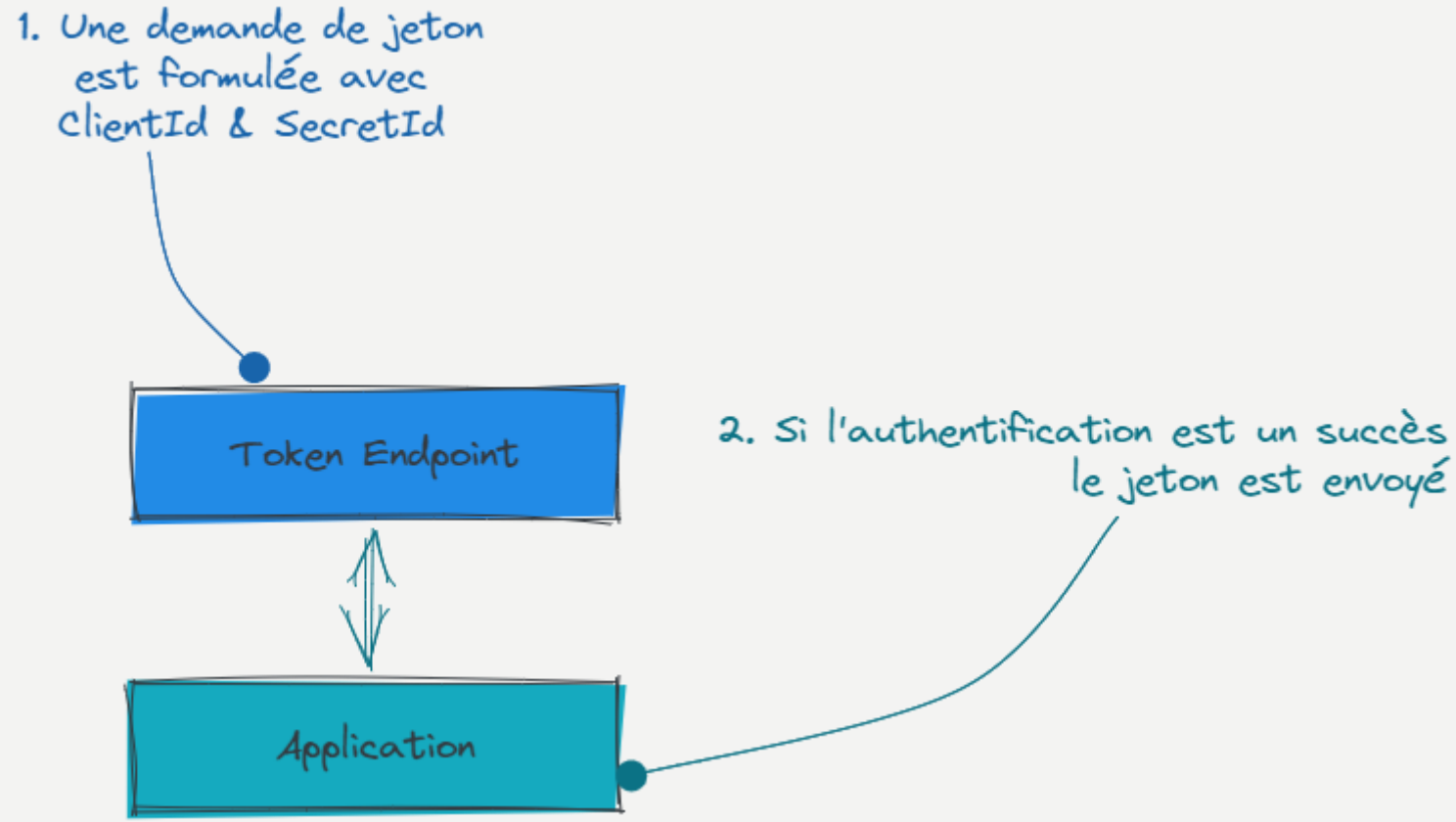
OAUTH2

- Type de demande : « Authorization Code »



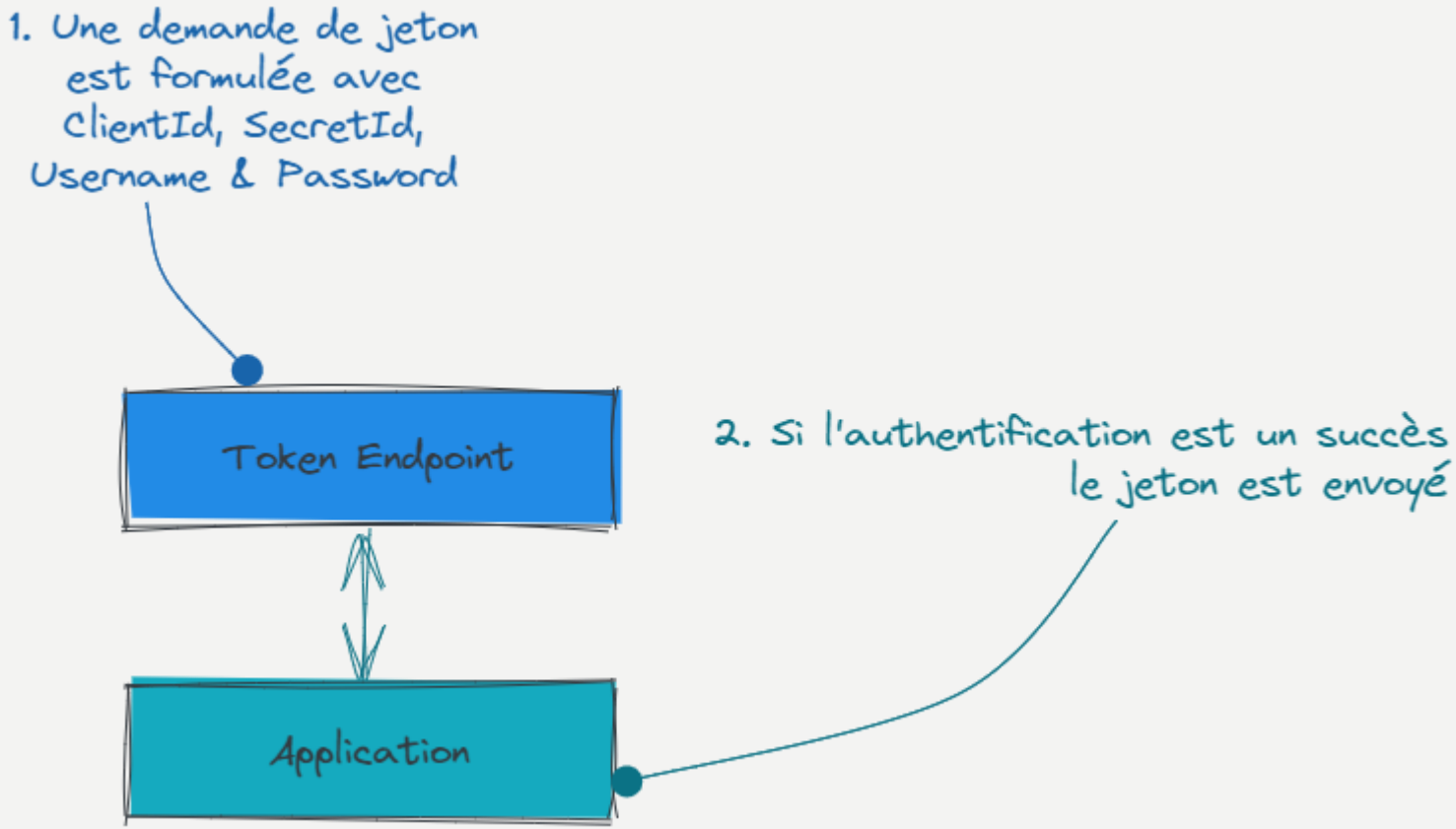
OAUTH2

- Type de demande : « Client Credentials »



OAUTH2

- Type de demande : « Password Credentials »



OAUTH2 – TESTER

- Tester avec Postman une connexion OAuth2 Azure ou Google
  - portal.azure.com
  - console.cloud.google.com

OAUTH2

- **O**pen**ID** **C**onnect
  - Surcouche à OAuth2
  - Plus ouvert
  - Utilise les jetons d'accès au format JWT
    - Permet d'obtenir des informations sur l'utilisateur, sans en faire une demande spéciale (ces informations peuvent être stockées dans le jeton – ces informations sont en clair, et la clé permet de garantir que les données n'ont pas été altérées)

OAUTH2

- Utiliser le starter **spring-boot-starter-oauth2-client**
  - Activer la page de login oauth2

```
http.oauth2Login();
```

  - Configurer l'id et le secret de l'application (et éventuellement d'autres paramètres)

```
spring.security.oauth2.client.registration.google.client-id = XXX
spring.security.oauth2.client.registration.google.client-secret = XXX
```

Spring Security utilise la session pour stocker les jetons de l'utilisateur connecté

OAUTH2

- Pour relier les autorisations d'un utilisateur OIDC
  - Utiliser une classe qui implémente `GrantedAuthoritiesMapper`
    - Gérée par Spring, elle se déclenche à la connexion (OAuth)

```
@Component
public class OAuthAuthoritiesMapper implements GrantedAuthoritiesMapper {

}
```

OAUTH2

```java
@Override
public Collection<? extends GrantedAuthority> mapAuthorities(Collection<? extends GrantedAuthority> authorities) {
  List<GrantedAuthority> mappedAuthorities = new ArrayList<>();

  authorities.forEach(authority -> {
    if (authority instanceof OAuth2UserAuthority oauthUserAuthority) {
      // Récupérer les infos de l'utilisateur
      Map<String, Object> userAttributes = oauthUserAuthority.getAttributes();

      mappedAuthorities.add(new SimpleGrantedAuthority("ROLE_ADMIN"));
    }

    else if (authority instanceof OidcUserAuthority oidcUserAuthority) {
      // Récupérer les infos de l'utilisateur
      Map<String, Object> userAttributes = oidcUserAuthority.getAttributes();
    }
  });

  return mappedAuthorities;
}
```

OAUTH2

- Exemple de configuration plus complète pour une authentification plus spécifique

```
#OAuth2 Spécifique
spring.security.oauth2.client.registration.formation.client-name = Formation
spring.security.oauth2.client.registration.formation.client-id = XXX
spring.security.oauth2.client.registration.formation.client-secret = XXX
spring.security.oauth2.client.registration.formation.authorization-grant-type = authorization_code
spring.security.oauth2.client.registration.formation.redirect-uri = http://localhost:8080/login/oauth2/code/formation

spring.security.oauth2.client.provider.formation.token-uri = endpoint pour récupérer un jeton (avec un code d'autorisation par ex.)
spring.security.oauth2.client.provider.formation.authorization-uri = endpoint pour récupérer un code d'autorisation
spring.security.oauth2.client.provider.formation.user-info-uri = endpoint pour récupérer les infos user
spring.security.oauth2.client.provider.formation.user-name-attribute = nom attribut pour le nom de l'user
```

EXERCICE

- Implémenter une authentification par OAuth2 Google

OAUTH2 – RESOURCESERVER

- Pour faire de Spring un serveur de ressource (et un relais de vérification d'un jeton)
  - Utiliser le starter **spring-boot-starter-oauth2-resource-server**
  - Activer le login via le serveur de ressource oauth2

```
http.oauth2ResourceServer().jwt();
```

  - Configurer les « relais » JWT

```
spring.security.oauth2.resourceserver.jwt.issuer-uri = https://accounts.google.com
spring.security.oauth2.resourceserver.jwt.jwk-set-uri = https://www.googleapis.com/oauth2/v3/certs
```

OAUTH2 – RESOURCESERVER

- Pour relier les autorisations d'un utilisateur OIDC
  - Utiliser un `Converter`

```
private Converter<Jwt, Collection<GrantedAuthority>> jwtConverter() {
  return jwt -> {
    List<GrantedAuthority> mappedAuthorities = new ArrayList<>();
    mappedAuthorities.add(new SimpleGrantedAuthority("ROLE_ADMIN"));

    return mappedAuthorities;
  };
}
```

OAUTH2 – RESOURCESERVER

- Utilisé par un `JwtAuthenticationConverter`

```
private JwtAuthenticationConverter jwtAuthenticationConverter() {
  JwtAuthenticationConverter jwtAuthenticationConverter = new JwtAuthenticationConverter();
  jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(this.jwtConverter());

  return jwtAuthenticationConverter;
}
```

- Utilisé dans la configuration du serveur de ressource

```
http.oauth2ResourceServer()
  .jwt()
  .jwtAuthenticationConverter(this.jwtAuthenticationConverter());
```

EXERCICE

- Récupérer un jeton via Postman et l'utiliser pour se connecter à l'application Spring