EPI-Polytec
Ecole Privée Internationale d'Ingénieurs
المدرسة الدولية العليا الخاصة للمهندسين بسوسة

Leçon 2 : Sécurité basique des API Rest

Spring Security

Spring Security est un Framework de sécurité léger qui fournit une authentification et un support d'autorisation afin de sécuriser les applications Spring Boot.

Spring Security permet de préconfigurer et de personnaliser des fonctions de sécurité au sein d'une application Spring Boot.

Les principales fonctionnalités de Spring Security incluent :

1. **Authentification** : Spring Security gère les processus d'authentification en prenant en charge différents mécanismes, tels que l'authentification basée sur les formulaires, l'authentification basée sur les jetons, etc. Il facilite également l'intégration avec des fournisseurs d'authentification externes à travers le protocole Oauth2 (ex : serveur keyclock, …)
2. **Autorisation** : Spring Security permet de définir des règles d'autorisation pour contrôler l'accès aux ressources de l'application. Il prend en charge des mécanismes flexibles tels que les expressions de contrôle d'accès basées sur les rôles, les annotations de sécurité et les stratégies de sécurité personnalisées.
3. **Gestion des sessions** : Spring Security gère les sessions utilisateur de manière sécurisée et offre des fonctionnalités pour contrôler et gérer les sessions, telles que la configuration de la durée de validité des sessions, la gestion des sessions concurrentes, etc.

Authentification vs Autorisation

1. Authentification (Authentication)

    But : déterminer qui êtes-vous ? (who are you ?)

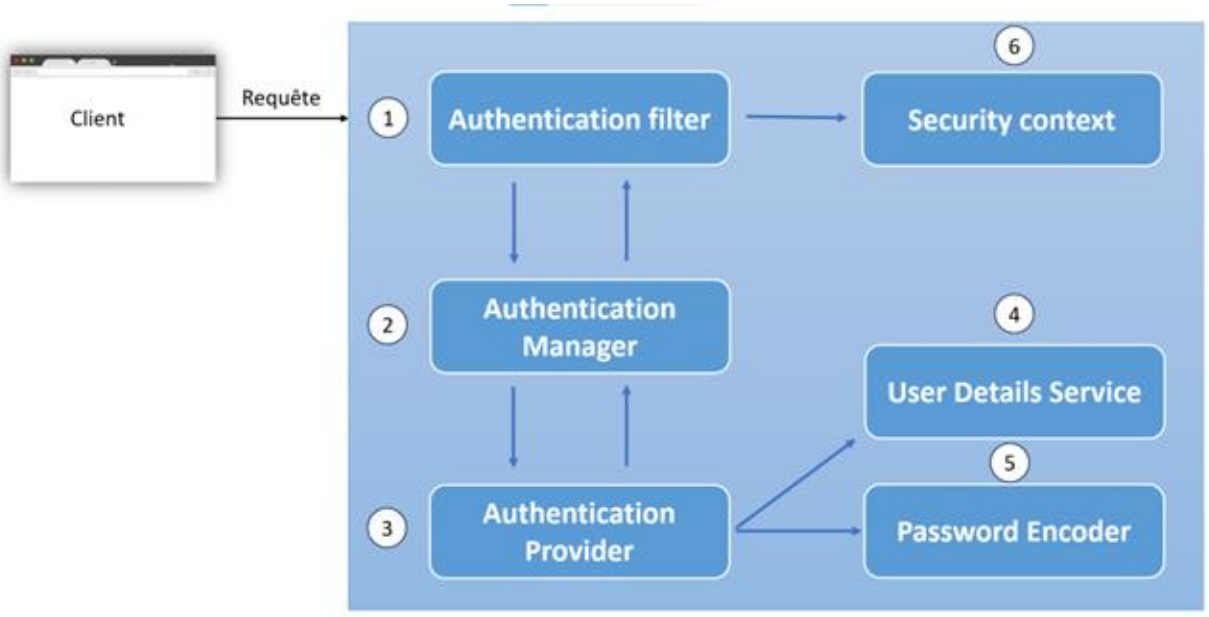
    Moyen : login (username) et password

2. Autorisation (Authorization)

    But : déterminer que pouvez-vous faire dans l'application (what are you allowed to do)

    Moyen : le(s) rôle(s) attribué(s) à chaque utilisateur,

Architecture de Spring Security



Etape 1 : la requête envoyée par le client est interceptée par `Authentication filter`

Etape 2 : Ensuite `l'Authentication manager` prend la responsabilité de cette requête.

Etape 3 : l'`Authentication manager` utilise l'`Authentication provider` qui implémente la logique de l'Authentification (Basic, Token, Oauth2, ...)

Etape 4&5 : `l'Authentication provider` recherche et valide l'utilisateur en utilisant `Password Encoder` et `User Details Service`

Etape 6 : la requête retourne ensuite à `Authentication filter` et si l'authentification réussit, le détail de l'authentification est sauvegardé (selon la politique de sécurité) dans `Security Context`.

Expression Lambda

Une expression lambda en Java permet d'écrire du code de manière concise et expressive. C'est une fonction anonyme qui prend des paramètres et renvoie une valeur et peut être implémentée directement dans le corps d'une méthode. Sa syntaxe la plus simple est :

`paramètre(s) -> expression`

1. Liste des paramètres : C'est la liste des paramètres que la fonction lambda prend en entrée. Elle peut être vide si la fonction ne nécessite pas de paramètres, ou contenir un ou plusieurs paramètres séparés par des virgules.

2. Opérateur "->" : Cet opérateur est utilisé pour séparer les paramètres du corps de la fonction lambda.

3. Corps de la fonction : C'est le bloc de code qui est exécuté lorsque la fonction lambda est invoquée. Il peut s'agir d'une seule instruction ou d'un bloc de code entre accolades.

Les Beans

Dans Spring Boot, un bean est un objet géré par le conteneur d'inversion de contrôle (IoC) de Spring. Il s'agit essentiellement d'une instance d'une classe qui est créée, configurée et gérée par Spring context pour faciliter son injection lors de son utilisation.

Spring Boot utilise une configuration basée sur les annotations, ce qui signifie qu'on peut utiliser des annotations pour indiquer à Spring comment créer et configurer les beans.

Pour qu'une classe soit considérée comme un bean Spring, on peut lui appliquer l'annotation `@Component` ou l'une de ses annotations dérivées telles que `@Service`, `@Repository` ou `@Controller`. Ces annotations permettent à Spring de détecter automatiquement les classes et de les traiter comme des beans.

L'annotation `@Bean` est utilisée dans une classe de configuration Spring pour déclarer une méthode qui va retourner un bean géré par Spring context. Elle indique à Spring de traiter le résultat de cette méthode comme un bean et de le rendre disponible pour l'injection de dépendances dans d'autres parties de l'application. Cette méthode doit être définie dans une classe annotée avec `@Configuration`.

TP2 : Sécurité basique des API Rest du TP1

Ajout de la dépendance de Spring Security

Dans ce TP on va aborder la sécurité basique des APIs Rest c-a-d une authentification par username et password.

Le principe de l'authentification basique est que pour consommer un API Rest par une requête HTTP, l'utilisateur doit envoyer son username et password. Si l'authentification réussi et l'utilisateur a le rôle (autorisation) requis, la réponse est généré.

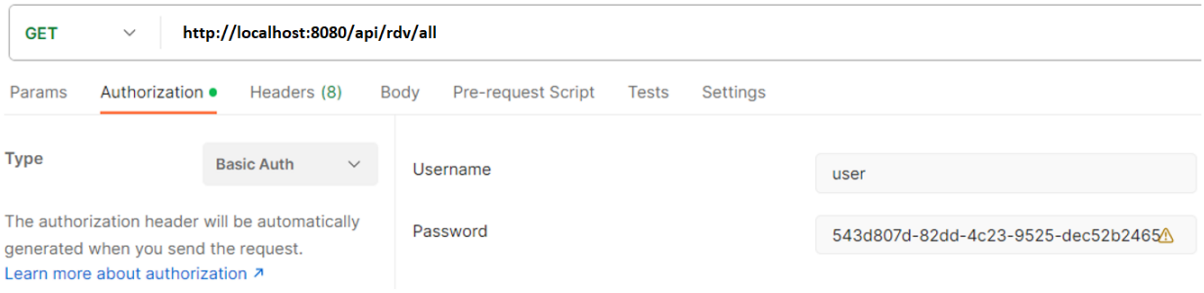
Pour commencer, on doit ajouter au fichier `pom.xml` du projet la dépendance correspondante à Spring Security. Rechercher cette dépendance (dans le site Maven repository) avec les mots clés `spring security starter`.. Mettre à jour le contenu du fichier `pom.xml`.

Au redémarrage de l'application Spring Security sécurise tous les accès aux API Rest de l'application (Endpoints) et à la documentation Swagger par authentification basique avec login user et mot de passe généré qu'on récupère de la console :

```
Using generated security password: 543d807d-82dd-4c23-9525-dec52b246507

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

Tester dans Postman l'API qui, avec la méthode Get, retourne la liste de tous les rdvs avec l'url : `http://localhost:8080/api/rdv/all` et introduisez les paramètres d'authentification basique :



Configurer la sécurité

Comme il est indiqué dans la capture de la console précédente, on doit configurer Spring Security avec une classe de configuration. Créer un package `security` contenant la classe de configuration `SecurityConfig`. La classe doit être annoté par `@Configuration` et `@EnableWebSecurity` pour préciser son rôle.

```
package com.example.medicalapp.security;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

}
```

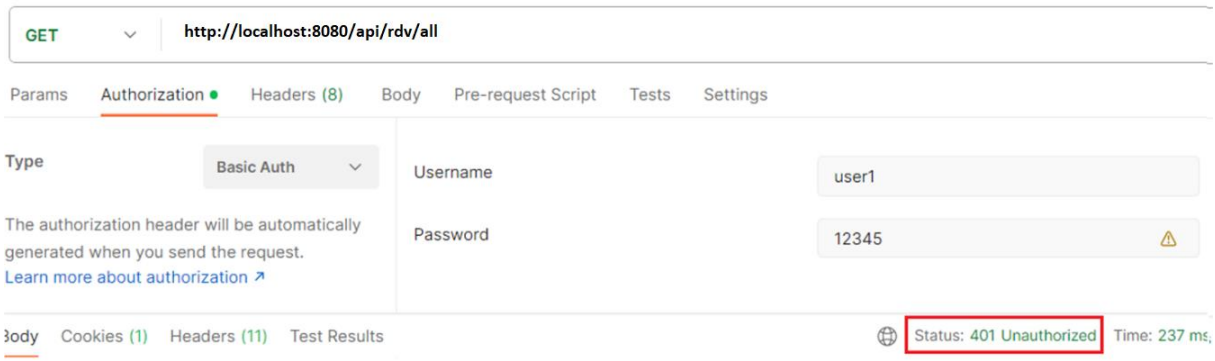
La première méthode va indiquer que les détails de chaque utilisateur seront stockés de manière non persistante dans la mémoire. C'est une méthode qui retourne des objets **Bean** de type la classe `InMemoryUserDetailsManager` correspondants aux utilisateurs.

```
@Bean
public InMemoryUserDetailsManager inMemoryUserDetailsManager(){

    UserDetails user1 = User.withUsername("user1").password("12345").authorities("USER").build();
    UserDetails admin = User.withUsername("admin").password("admin").authorities("ADMIN","USER").build();
    return new InMemoryUserDetailsManager(user1,admin);
}
```

On définit 2 utilisateurs (de type Interface `UserDetails`), chacun ayant son username, passoword et autorités (rôles). La méthode retourne l'instanciation de 2 objets Bean de Type la classe `InMemoryUserDetailsManager` .

On redémarre l'application, Spring Boot ne propose plus le mot de passe du user par défaut dans la console. En tentant de s'authentifier avec l'un de ses utilisateurs sur un Endpoint, l'accès est refusé :



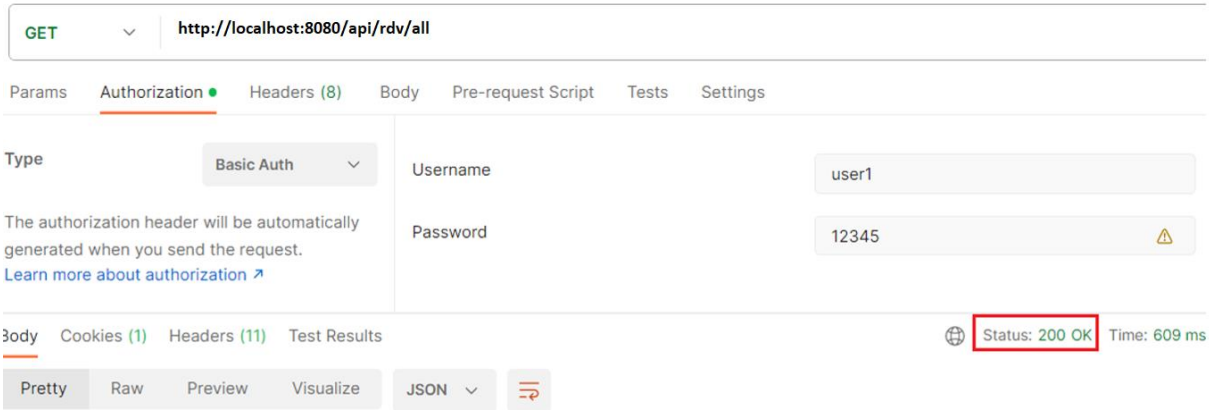
Pourtant les paramètres sont corrects, Pourquoi alors l'accès est refusé ?

La réponse est simple, Spring Security ne tolère pas des mots de passe en plein texte et il faut obligatoirement les **crypter**. L'algorithme de cryptage utilisé est `Bcrypt`.

Définissez une méthode annotée par `@Bean` nommée `passwordEncoder` de type `PasswordEncoder` qui retourne une instance de `BCryptPasswordEncoder`.

Utiliser ce Bean dans la méthode `inMemoryUserDetailsManager` pour crypter les mots de passe avec la méthode `encode(password)`.

Redémarrer l'application et tester de nouveau, cette fois ça fonctionne :



Cette configuration fonctionne lorsqu'on veut sécurisez toutes les requêtes entrantes à l'application. Si on veut personnaliser certains accès il faut définir une 3ème méthode qui met en place un Authentication Filter qui intercepte la requête.

Cette méthode est de type l'interface `SecurityFilterChain` et prend un paramètre de type `HttpSecurity` (qui représente la requête).

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {
    return httpSecurity
            .sessionManagement(sm->sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .csrf(csrf->csrf.disable())
            .authorizeHttpRequests(ar->ar.anyRequest().authenticated())
            .httpBasic(Customizer.withDefaults())
            .build();
}
```

➢ La première méthode appliquée à `httpSecurity` est `sessionManagement` qui indique avec STATELESS que Spring Security ne créera jamais de Session et ne l'utilisera jamais pour obtenir SecurityContext.

➢ Lorsque les services (API REST) de l'application sont exposés à des clients (autres que les navigateurs) il faut désactiver csrf.

➢ L'appel de `authorizeHttpRequests` indique à Spring Security que tout Endpoint de l'application nécessite une authentification afin d'autoriser l'accès.

➢ On précise que la méthode d'authentification est basique.

Traitement 1 :

- Ajouter une autorisation qui permet à l'admin seulement d'ajouter un patient, un médecin et un Rdv.
- Ajouter une autorisation qui permet à tout utilisateur de consulter la liste des médecins (sans authentification)

Traitement 2 : changer la première autorisation et la définir directement dans les actions du RestController, pour cela :

- Commenter cette instruction ou le code équivalent.

- Annoter le classe `SecurityConfig` par `@EnableMethodSecurity(prePostEnabled=true)` Ce qui permet d'affecter l'autorisation d'accès aux méthodes (actions) dans le contrôleur Rest.

- Ajouter sur chaque action de `RdvRestController` l'annotation `@PreAuthorize` qui précise l'autorisation à donner pour chaque autorité ADMIN ou USER. Exemple :

```
@PostMapping("add")
@PreAuthorize("hasAuthority('ADMIN')")
public Rdv add(@RequestBody Rdv rdv){
    return iServiceRdv.addRdv(rdv);
}
```