Spring Boot et **Spring Security** (InMemory)

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Spring Security

But de la sécurité

Interdire, à un utilisateur, l'accès à une ressource à laquelle il n'a pas droit

Spring Security

But de la sécurité

Interdire, à un utilisateur, l'accès à une ressource à laquelle il n'a pas droit

Deux questions

- Qui veut accéder à la ressource ? (Authentification)
- A t-il le droit d'y accéder ? (Autorisation)

Spring Security

Ce qu'il nous faut

Spring Security

Spring Security

Ce qu'il nous faut

Spring Security

Deux systèmes d'authentification

- **Stateful** (avec état) : stocke les données de l'utilisateur dans les sessions et les cookies
- **Stateless** (sans état) : stocke les données dans un jeton (token) qui sera délivré au client

Spring Security

Authentification **Stateful**

- Le client envoie une requête **HTTP** de type `POST` contenant le nom d'utilisateur et le mot de passe.
- Le serveur reçoit la requête et récupère les rôles de l'utilisateur.
- Dans un **Map** (`SESSION_ID` ⇒ clé, données utilisateur ⇒ valeur) nommé `sessions`, il ajoute une `entry` associée à l'utilisateur connecté.
- Le serveur retourne une réponse avec le code 200 et stocke dans les cookies du navigateur le `SESSION_ID` attribué à l'utilisateur.
- Chaque fois que le client envoie une requête au serveur, le `SESSION_ID` sera automatiquement attaché à l'entête de la requête.
- Le serveur vérifiera chaque fois que le `SESSION_ID` est bien présent dans le dictionnaire `sessions`.

Authentification **Stateless**

- Le client envoie une requête **HTTP** de type `POST` contenant le nom d'utilisateur et le mot de passe.
- Le serveur reçoit la requête et récupère les rôles de l'utilisateur.
- Le serveur génère un jeton (token) contenant les données utilisateur + signature
- Le serveur retourne une réponse avec le code 200 et un jeton. Le client doit stocker le jeton pour le réutiliser (dans le **Local Storage** par exemple).
- Chaque fois que le client envoie une requête au serveur, il doit ajouter le jeton à l'entête de la requête.
- Le serveur vérifiera chaque fois que la signature du jeton.

Spring Security

Remarque

En utilisant un système de connexion de type **Stateful** et les cookies, on est exposé à une faille de sécurité appelé **CSRF**

Spring Security

Remarque

En utilisant un système de connexion de type **Stateful** et les cookies, on est exposé à une faille de sécurité appelé **CSRF**

CSRF ou **C**ross-**S**ite **R**equest **F**orgery (via un exemple)

- Un utilisateur non authentifié `X` veut exécuter une action sur une application alors qu'il n'est pas authentifié
- Cette application utilise un système d'authentification **Stateful** (avec les cookies)
- `X` transmet alors à un utilisateur `U` authentifié à cette même application une image (par exemple) derrière laquelle se cache un script
- `U` clique sur l'image et envoie donc une requête **HTTP** avec son `SESSION_ID` sans qu'il le sache

Spring Security

Quelles solutions pour les attaques **CSRF** ?

- Éviter d'utiliser la méthode **HTTP** `GET` pour l'exécution d'une action.
- Demander une confirmation pour chaque action critique
- Utiliser un jeton de validité (un jeton intégré dans tous les formulaires comme champ caché)
- ...

Spring Security

Deux possibilité pour le stockage des utilisateurs

- En mémoire (InMemory)
- En base de données

Spring Security

Deux possibilité pour le stockage des utilisateurs

- En mémoire (InMemory)
- En base de données

Dans ce chapitre

- On utilisera un système d'authentification **Stateful**
- Les utilisateurs seront stockées en mémoire

Spring Security

Création de projet Spring Boot

- Aller dans `File > New > Other`
- Chercher `Spring`, dans `Spring Boot` sélectionner `Spring Starter Project` et cliquer sur `Next >`
- Saisir
  - `spring-boot-security-in-memory` dans `Name`,
  - `com.example` dans `Group`,
  - `spring-boot-security-in-memory` dans `Artifact`
  - `com.example.demo` dans `Package`
- Cliquer sur `Next`
- Chercher et cocher les cases correspondantes aux `Spring Data JPA`, `MySQL Driver`, `Lombok`, `Spring Web`, `Spring Boot DevTools` et `Spring Security`
- Cliquer sur `Next` puis sur `Finish`

Spring Security

Explication

- Le package contenant le point d'entrée de notre application (la classe contenant le `puclic static void main`) est `com.example.demo`
- Tous les autres packages `dao`, `model`... doivent être dans le package `demo`.

Spring Security

Explication

- Le package contenant le point d'entrée de notre application (la classe contenant le `puclic static void main`) est `com.example.demo`
- Tous les autres packages `dao`, `model`... doivent être dans le package `demo`.

Pour la suite, nous considérons

- une entité `Personne` à définir dans `com.example.demo.model`
- une interface DAO `PersonneRepository` à définir dans `com.example.demo.dao`
- un contrôleur REST `PersonneController` à définir dans `com.example.demo.controller`

Spring Boot & REST

Créons une entité `Personne` **dans** `com.example.demo.model`

```
package com.example.demo.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;

@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long num;
    @NonNull
    private String nom;
    @NonNull
    private String prenom;
}
```

Spring Boot & REST

Préparons notre interface DAO `PersonneRepository`

```
package com.example.demo.dao;

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

import com.example.demo.model.Personne;

public interface PersonneRepository extends JpaRepository<Personne,
   Long> {

}
```

Créons le contrôleur REST suivant

```
@CrossOrigin
@RestController
public class PersonneRestController {

  @Autowired
  private PersonneRepository personneRepository;

  @GetMapping("/personnes")
  public List<Personne> getPersonnes() {
    return personneRepository.findAll();
  }

  @GetMapping("/personnes/{id}")
  public Personne getPersonne(@PathVariable("id") long id) {
    return personneRepository.findById(id).orElse(null);
  }

  @PostMapping("/personnes")
  public Personne addPersonne(@RequestBody Personne personne) {
    return personneRepository.save(personne);
  }
}
```

Spring Security

Dans `application.properties`, ajoutons les données permettant la connexion à la base de données et la configuration de `Hibernate`

```
spring.datasource.url = jdbc:mysql://localhost:3306/security_memory?createDatabaseIfNotExist=
  true
spring.datasource.username = root
spring.datasource.password = root
spring.jpa.hibernate.ddl-auto = update
spring.jpa.show-sql = true
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.naming.physical-strategy = org.hibernate.boot.model.naming.
  PhysicalNamingStrategyStandardImpl
```


L'ajout de la propriété `spring.jpa.hibernate.naming.physical-strategy` permet de forcer **Hibernate** à utiliser les mêmes noms pour les tables et les colonnes que les entités et les attributs.

Spring Security

Pour alimenter la base de données avec quelques données au démarrage de l'application

```
@SpringBootApplication
public class SpringBootSecurityInMemoryApplication implements ApplicationRunner {

    @Autowired
    private PersonneRepository personneRepository;

    public static void main(String[] args) {
        SpringApplication.run(SpringBootSecurityInMemoryApplication.class, args);
    }

    @Override
    public void run(ApplicationArguments args) throws Exception {
        Personne personne1 = new Personne("wick", "john");
        Personne personne2 = new Personne("dalton", "jack");
        Personne personne3 = new Personne("maggio", "carol");
        Personne personne4 = new Personne("cohen", "sophie");
        personneRepository.saveAll(Arrays.asList(personne1, personne2, personne3, personne4));
    }
}
```

Spring Security

Lancez l'application et vérifiez la génération d'un mot de passe dans la console

Using generated security password : 895a8a45-d296-4ee6-a246-421d6dd6e64e

Spring Security

Lancez l'application et vérifiez la génération d'un mot de passe dans la console

Using generated security password : 895a8a45-d296-4ee6-a246-421d6dd6e64e

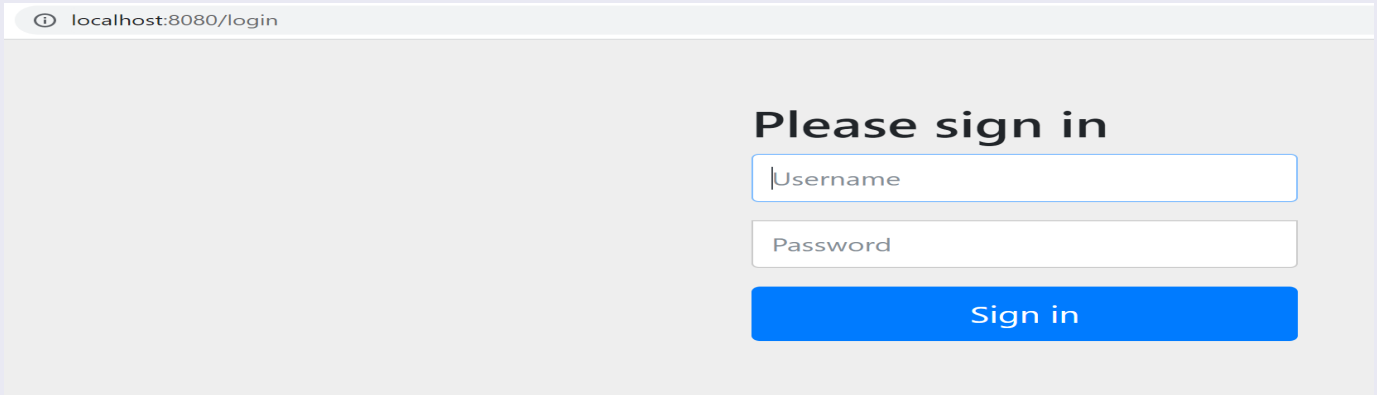
Remarque

Ce mot de passe est à utiliser pour accéder aux ressources.

Spring Security

Pour tester

- Allez à `http://localhost:8080/` et vérifiez l'affichage de l'interface d'authentification



- Utilisez `user` comme nom d'utilisateur et le mot de passé généré et affiché dans la console pour la connexion

Spring Security

Étapes

- Créer une classe `SecurityConfig` pour la configuration de la sécurité dans un package `com.example.demo.configuration`
- Définir les noms d'utilisateurs et les mots de passe autorisés

Spring Security

Dans `com.example.demo.configuration`**, définissons la classe** `SecurityConfig`

```
package com.example.demo.configuration;

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

@Configuration
@EnableWebSecurity
public class SecurityConfig {

}
```

Spring Security

Ajoutons un premier bean pour lister les utilisateurs autorisés

```
package com.example.demo.configuration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public InMemoryUserDetailsManager inMemoryUserDetailsManager() {
        return new InMemoryUserDetailsManager(
            User.builder().username("user").password("user").roles("USER").build(),
            User.builder().username("admin").password("admin").roles("USER", "ADMIN").build()
        );
    }
}
```

Et un deuxième pour configurer le filtre

```
package com.example.demo.configuration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public InMemoryUserDetailsManager inMemoryUserDetailsManager() {
        return new InMemoryUserDetailsManager(
            User.builder().username("user").password("user").roles("USER").build(),
            User.builder().username("admin").password("admin").roles("USER", "ADMIN").build()
        );
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
                .csrf(csrf -> csrf.disable())
                .authorizeHttpRequests((authz) -> authz.anyRequest().fullyAuthenticated())
                .httpBasic(Customizer.withDefaults())
                .build();
    }
}
```

Spring Security

Explication

- `csrf().disable()` : pour désactiver les jetons **CSRF** car les données ne proviennent pas d'un formulaire.
- `authorizeHttpRequests` : pour spécifier les routes autorisées ou celles qui nécessitent une authentification.
- `httpBasic(Customizer.withDefaults())` : pour afficher un formulaire de connexion basique avec deux champs : un premier pour le nom d'utilisateur et un deuxième pour l'utilisateur.

Spring Security

Remarque

- Allez à `http://localhost:8080/personnes` et vérifiez l'affichage d'un message d'erreur relatif au `PasswordEncoder`.
- Par défaut, **Spring Security** encode les mots de passe.

Spring Security

Désactivons l'utilisation de mots de passe en ajoutant l'option `{noop}` **(pour** `NoOpPasswordEncoder`**)**

```
@Bean
public InMemoryUserDetailsManager inMemoryUserDetailsManager() {
    return new InMemoryUserDetailsManager(
        User.builder().username("user").password("{noop}user").roles("USER").build(),
        User.builder().username("admin").password("{noop}admin").roles("USER", "ADMIN").build()
    );
}
```

Spring Security

Désactivons l'utilisation de mots de passe en ajoutant l'option `{noop}` **(pour** `NoOpPasswordEncoder`**)**

```
@Bean
public InMemoryUserDetailsManager inMemoryUserDetailsManager() {
    return new InMemoryUserDetailsManager(
        User.builder().username("user").password("{noop}user").roles("USER").build(),
        User.builder().username("admin").password("{noop}admin").roles("USER", "ADMIN").build()
    );
}
```


Autres options

- `{bcrypt}` : pour **BCryptPasswordEncoder**
- `{sha256}` : pour **StandardPasswordEncoder**
- `{scrypt}` : pour **SCryptPasswordEncoder**
- ...

Spring Security

Pour tester

- Allez à `http://localhost:8080/personnes` et vérifiez l'affichage de l'interface d'authentification
- Utilisez `user` comme nom d'utilisateur et mot de passé et vérifiez que la ressource est accessible

Spring Security

Pour tester

- Allez à `http://localhost:8080/personnes` et vérifiez l'affichage de l'interface d'authentification
- Utilisez `user` comme nom d'utilisateur et mot de passé et vérifiez que la ressource est accessible

Pour accéder à la ressource depuis **Postman**

- Dans la liste déroulante, choisir `GET` puis saisir l'URL vers notre web service `http://localhost:8080/personnes`
- Ensuite cliquer sur `Authorization` et choisissez `Basic Auth`
- Utilisez `user` comme nom d'utilisateur et mot de passé et vérifiez que la ressource est accessible

Spring Security

Deux solutions pour autoriser (ou pas) les utilisateurs authentifiés à accéder à une ressource

- Dans la classe de configuration `SecurityConfig`
- Avec les annotations sur les ressources (contrôleurs et services)

Spring Security

Autorisons l'accès à certaines ressources uniquement aux utilisateurs ayant le droit

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http
            .csrf(csrf -> csrf.disable())
            .authorizeHttpRequests((authz) -> authz
                    .requestMatchers(HttpMethod.GET, "/personnes").hasRole("USER")
                    .requestMatchers(HttpMethod.POST, "/personnes").hasRole("ADMIN")
                    .anyRequest().authenticated())
            .httpBasic(Customizer.withDefaults())
            .build();
}
```

Spring Security

Autorisons l'accès à certaines ressources uniquement aux utilisateurs ayant le droit

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http
            .csrf(csrf -> csrf.disable())
            .authorizeHttpRequests((authz) -> authz
                    .requestMatchers(HttpMethod.GET, "/personnes").hasRole("USER")
                    .requestMatchers(HttpMethod.POST, "/personnes").hasRole("ADMIN")
                    .anyRequest().authenticated())
            .httpBasic(Customizer.withDefaults())
            .build();
}
```


Explication

- Les utilisateurs ayant le rôle `USER` seront autorisés à consulter la liste de personnes
- Les utilisateurs ayant le rôle `ADMIN` seront autorisés à ajouter une nouvelle personne

Spring Security

Commençons par commenter les deux lignes suivantes

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http
            .csrf(csrf -> csrf.disable())
            .authorizeHttpRequests((authz) -> authz
//                  .requestMatchers(HttpMethod.GET, "/personnes").hasRole("USER")
//                  .requestMatchers(HttpMethod.POST, "/personnes").hasRole("ADMIN")
                .anyRequest().authenticated())
            .httpBasic(Customizer.withDefaults())
            .build();
}
```

Spring Security

Étapes

- Activer les annotations de sécurité (comme `@Secured("ROLE")`) dans `SecurityConfig`
- Annoter les ressources avec `@Secured("ROLE")` (ou autres)

Spring Security

Le contrôleur `PersonneRestController`

```
@RestController
@Secured("ROLE_ADMIN")
public class PersonneRestController {

    @Autowired
    private PersonneRepository personneRepository;
```

Spring Security

Le contrôleur `PersonneRestController`

```
@RestController
@Secured("ROLE_ADMIN")
public class PersonneRestController {

    @Autowired
    private PersonneRepository personneRepository;
```

- `@Secured("ROLE_ADMIN")` : rend l'accès à la route `/personnes` réservé aux utilisateurs ayant le rôle `ROLE_ADMIN`
- Cette annotation peut être appliquée sur une classe ou une méthode
- On peut autoriser plusieurs rôles ainsi `@Secured({...,...})`

Spring Security

Pour activer les annotations de sécurité, il faut annoter `SecurityConfig` **par** `EnableMethodSecurity`

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

    // contenu précédent

}
```

Spring Security

Pour activer les annotations de sécurité, il faut annoter `SecurityConfig` par `EnableMethodSecurity`

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

    // contenu précédent

}
```


Explication

`securedEnabled = true` : pour activer l'annotation Spring `@Secured`

Spring Security

Pour tester

- Connectez-vous avec `(admin, admin)` et vérifiez que vous avez accès aux `GET` et `POST`
- Connectez-vous avec `(user, user)` et vérifiez qu'un message d'erreur `403 Forbidden` lorsque vous essayez d'accéder aux `GET` et `POST`

Spring Security

`@Secured` **peut être remplacée par** `@RolesAllowed`

```
@RestController
@RolesAllowed("ADMIN")
public class PersonneRestController {

    @Autowired
    private PersonneRepository personneRepository;
```

Spring Security

`@Secured` **peut être remplacée par** `@RolesAllowed`

```
@RestController
@RolesAllowed("ADMIN")
public class PersonneRestController {

    @Autowired
    private PersonneRepository personneRepository;
```

- `@Secured("ROLE_ADMIN")` ≡ `@RolesAllowed("ADMIN")`
- `@Secured("ROLE_ADMIN")` est une annotation **Spring**
- `@RolesAllowed("ADMIN")` est une annotation **Java**

Spring Security

Dans `SecurityConfig`, il faudrait également activer l'annotation `@RolesAllowed`

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(securedEnabled = true, jsr250Enabled = true)
public class SecurityConfig {

    // contenu précédent

}
```

Spring Security

Dans `SecurityConfig`, il faudrait également activer l'annotation `@RolesAllowed`

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(securedEnabled = true, jsr250Enabled = true)
public class SecurityConfig {

    // contenu précédent

}
```


Explication

- `securedEnabled = true` : pour activer l'annotation **Spring** `@Secured`
- `jsr250Enabled = true` : pour activer les annotations **Java** de la standard **JSR250** telles que `@RolesAllowed`

Spring Security

Pour tester

- Connectez-vous avec `(admin, admin)` et vérifiez que vous avez accès aux `GET` et `POST`
- Connectez-vous avec `(user, user)` et vérifiez qu'un message d'erreur `403 Forbidden` lorsque vous essayez d'accéder aux `GET` et `POST`

Le contrôleur `PersonneRestController`

```
@RestController
@RolesAllowed("USER")
public class PersonneRestController {

        @Autowired
        private PersonneRepository personneRepository;

        @GetMapping("/personnes")
        public List<Personne> getPersonnes() { ... }

        @GetMapping("/personnes/{id}")
        public Personne getPersonne(@PathVariable("id") long id)  { ... }

        @RolesAllowed("ADMIN")
        @PostMapping("/personnes")
        public Personne addPersonne(@RequestBody Personne personne)  { ... }

}
```

Le contrôleur `PersonneRestController`

```
@RestController
@RolesAllowed("USER")
public class PersonneRestController {

        @Autowired
        private PersonneRepository personneRepository;

        @GetMapping("/personnes")
        public List<Personne> getPersonnes() { ... }

        @GetMapping("/personnes/{id}")
        public Personne getPersonne(@PathVariable("id") long id)  { ... }

        @RolesAllowed("ADMIN")
        @PostMapping("/personnes")
        public Personne addPersonne(@RequestBody Personne personne)  { ... }

}
```


- Les deux `GET` sont uniquement accessibles aux utilisateurs ayant le rôle `ROLE_USER` (elles prennent la sécurité du contrôleur)
- Le `POST` est accessible seulement aux utilisateurs ayant le rôle `ROLE_ADMIN` (la sécurité de la méthode `addPersonne()` annule la sécurité du contrôleur `ROLE_USER`)

Modifions les rôles associés à la méthode `findAll` dans `PersonneService`

```
@RestController
public class PersonneRestController {

        @Autowired
        private PersonneRepository personneRepository;

        @Secured({ "ROLE_ADMIN", "ROLE_USER" })
        @GetMapping("/personnes")
        public List<Personne> getPersonnes() { ... }

        @GetMapping("/personnes/{id}")
        public Personne getPersonne(@PathVariable("id") long id) { ... }

        @PostMapping("/personnes")
        public Personne addPersonne(@RequestBody Personne personne) { ... }
}
```

Modifions les rôles associés à la méthode `findAll` dans `PersonneService`

```
@RestController
public class PersonneRestController {

        @Autowired
        private PersonneRepository personneRepository;

        @Secured({ "ROLE_ADMIN", "ROLE_USER" })
        @GetMapping("/personnes")
        public List<Personne> getPersonnes() { ... }

        @GetMapping("/personnes/{id}")
        public Personne getPersonne(@PathVariable("id") long id) { ... }

        @PostMapping("/personnes")
        public Personne addPersonne(@RequestBody Personne personne) { ... }
}
```


- La méthode `getPersonnes` est uniquement accessible aux utilisateurs ayant le rôle `ROLE_ADMIN` **OU** le `ROLE_USER`
- Ce n'est pas un **ET**

Modifions les rôles associés à la méthode `findAll` dans `PersonneService`

```
@RestController
public class PersonneRestController {

        @Autowired
        private PersonneRepository personneRepository;

        @Secured({ "ROLE_ADMIN", "ROLE_USER" })
        @GetMapping("/personnes")
        public List<Personne> getPersonnes() { ... }

        @GetMapping("/personnes/{id}")
        public Personne getPersonne(@PathVariable("id") long id) { ... }

        @PostMapping("/personnes")
        public Personne addPersonne(@RequestBody Personne personne) { ... }
}
```


- La méthode `getPersonnes` est uniquement accessible aux utilisateurs ayant le rôle `ROLE_ADMIN` **OU** le `ROLE_USER`
- Ce n'est pas un **ET**

Impossible d'exiger deux rôles avec `@Secured` ni `@RolesAllowed`

Pour exiger deux rôles (ou plus), on utilise `@PreAuthorize`

```
@RestController
public class PersonneRestController {

        @Autowired
        private PersonneRepository personneRepository;

        @PreAuthorize("hasRole('ROLE_USER') and hasRole('ROLE_ADMIN')")
        @GetMapping("/personnes")
        public List<Personne> getPersonnes() { ... }

        @GetMapping("/personnes/{id}")
        public Personne getPersonne(@PathVariable("id") long id) { ... }

        @PostMapping("/personnes")
        public Personne addPersonne(@RequestBody Personne personne) { ... }
}
```

Pour exiger deux rôles (ou plus), on utilise `@PreAuthorize`

```
@RestController
public class PersonneRestController {

        @Autowired
        private PersonneRepository personneRepository;

        @PreAuthorize("hasRole('ROLE_USER') and hasRole('ROLE_ADMIN')")
        @GetMapping("/personnes")
        public List<Personne> getPersonnes() { ... }

        @GetMapping("/personnes/{id}")
        public Personne getPersonne(@PathVariable("id") long id) { ... }

        @PostMapping("/personnes")
        public Personne addPersonne(@RequestBody Personne personne) { ... }
}
```


Explication

La méthode `getPersonnes` est uniquement accessible aux utilisateurs ayant les deux rôles `ROLE_USER` et `ROLE_ADMIN` Les annotations `@PreAuthorize` et `@PostAuthorize` sont, par défaut, activées par `@EnableMethodSecurity`

Spring Security

Pour tester

- Connectez-vous avec `(admin, admin)` et vérifiez que vous avez accès à la méthode `getPersonnes()`
- Connectez-vous avec `(user, user)` et vérifiez qu'un message d'erreur `403 Forbidden` lorsque vous essayez d'accéder à `getPersonnes()`

Spring Security

Exercice 1

Tester, avec **Postman**, les trois méthodes **HTTP** `put` et `delete` qui permettront de modifier ou supprimer une personne.

Spring Security

Exercice 2

Créer une application Frontend qui permet à un utilisateur, via des interfaces graphiques) la gestion de personnes (ajout, modification, suppression, consultation et recherche) en utilisant les web services définis par **Spring**.

Dans `personne.service.ts`, on ajoute l'utilisateur à l'entête de la requête HTTP

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Personne } from '../interfaces/personne';

@Injectable({ providedIn: 'root' })
export class PersonneService {
  url = 'http://localhost:8080/personnes';
  headers: HttpHeaders;
  constructor(private http: HttpClient) {
    const username = 'wick';
    const password = 'wick';
    const user = username + ':' + password;
    this.headers = new HttpHeaders().set('Authorization', 'Basic ' +
      btoa(user));
  }
  getAll() {
    return this.http.get<Array<Personne>>(this.url, { headers: this.
      headers });
  }
  add(personne: Personne) {
    return this.http.post(this.url, personne, { headers: this.headers
      });
  }
}
```

Spring Security

Exercice

Créer une application Frontend qui permet à un utilisateur, via des interfaces graphiques) la gestion de personnes (ajout, modification, suppression, consultation et recherche) en utilisant les web services définis par **Spring**.

Spring Security

Dans `personne.service.ts` (Angular), on ajoute l'utilisateur à l'entête de la requête HTTP

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Personne } from '../interfaces/personne';

@Injectable({ providedIn: 'root' })
export class PersonneService {
  url = 'http://localhost:8080/personnes';
  headers: HttpHeaders;
  constructor(private http: HttpClient) {
    const username = 'wick';
    const password = 'wick';
    const user = username + ':' + password;
    this.headers = new HttpHeaders().set('Authorization', 'Basic ' + btoa(user));
  }
  getAll() {
    return this.http.get<Array<Personne>>(this.url, { headers: this.headers });
  }
  add(personne: Personne) {
    return this.http.post(this.url, personne, { headers: this.headers });
  }
}
```