TP7 : Sécuriser l'application Web avec Spring Security

Spring Security est un service permettant d'authentifier des utilisateurs par login et mot de passe. c-à-d permettre l'accès aux interfaces de l'application web pour des utilisateurs autorisés et selon leurs rôles. Il existe plusieurs manières d'utiliser Spring Security (avec une version de Spring Boot < 3), on va l'aborder dans ce TP de 2 manières :
- les paramètres d'authentification et les rôles seront sauvegardés dans la mémoire (sans BD)
- les paramètres d'authentification et les rôles seront sauvegardés dans la BD.

I- Paramètres d'accès sauvegardés dans la mémoire

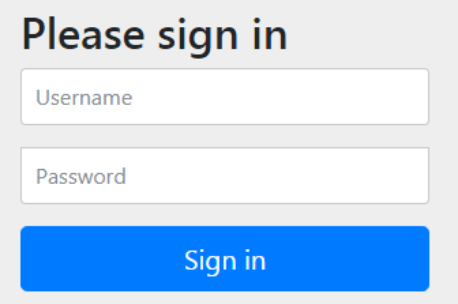
1. Ajouter la dépendance security au projet

Commençons par ajouter la dépendance de Spring Security dans pom.xml.

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

2. Configuration Basique par défaut

Une fois le projet redémarré, Spring security installe un filtre par défaut sur toutes les requêtes HTTP envoyés vers l'application. c-à-d que pour accéder à n'importe quel route, il faut préciser un nom d'utilisateur et un mot de passe dans un formulaire web générer par Spring Boot



Par défaut l'utilisateur est **user** avec un mot de passe qui est généré à chaque redémarrage du serveur et qui est donnée dans la console de l'IDE (par exemple) :

```
Using generated security password: 0a7e1f07-b448-4f37-9308-732f17aebe23
```

Démarrer l'application et tester l'accès avec le login et mot de passe généré avec l'URL `http://localhost:8080`

3. Personnaliser la configuration par défaut

Afin de personnaliser la configuration de Spring Security par défaut, on doit créer une classe `SecurityConfig.java` dans le package `com.gestion.filmotheque.security` qui hérite de la classe `WebSecurityConfigurerAdapter` et annoté par `@Configuration`

```
import org.springframework.context.annotation.Configuration;

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

}
```

Cette classe va redéfinir la méthode `configure(HttpSecurity http)` qui indique par quel moyen l'authentification sera faite (par un formulaire) et sur quelles requêtes HTTP on va exiger l'authentification (toutes les requêtes pour l'instant) :

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.formLogin();
        http.authorizeRequests().anyRequest().authenticated();
    }
}
```

L'autre méthode à redéfinir dans cette classe est `configure(AuthenticationManagerBuilder auth)` qui indique :

- quels sont les login et password des users qui ont le droit de s'authentifier
- ou sont mémorisé ces logins et password
- quel est le ou les rôles de chaque users

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication().withUser("user1").password("{noop}1234").roles("USER");
    auth.inMemoryAuthentication().withUser("admin").password("{noop}admin").roles("ADMIN");

}
```

Par défaut Spring Boot encode le mot de passe saisie dans le formulaire d'authentification. L'expression `{noop}` indique qu'il n'y aura pas d'encodage de mot de passe saisie avant de le comparer avec le mot de passe définit.

Tester l'accès à l'application avec ces 2 utilisateurs.

4. Encodage du mot de passe

Si on veut appliquer l'encodage, il faut définir dans la classe de configuration une méthode qui définit l'algorithme d'encodage pour les mots de passe :

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

`@Bean` indique qu'au démarrage de l'application, Spring Boot va appeler cette méthode pour retourner objet de type `PasswordEncoder` qui contient un objet qui va encoder le mot de passe selon l'algorithme BCrypt. Cet objet sera placé dans le contexte de l'application.

Pour encoder le mot de passe saisie il suffit de lui appliquer la méthode `enccode` de l'objet `passwordEncoder` :

```java
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    PasswordEncoder passwordEncoder = passwordEncoder();
    auth.inMemoryAuthentication().withUser("user1").password(passwordEncoder.encode("1234")).roles(
    auth.inMemoryAuthentication().withUser("admin").password(passwordEncoder.encode("admin")).roles
```

Pour connaitre la valeur cryptée du mot de passe, il suffit de l'afficher dans la console avec `System.out.println(passwordEncoder.encode("..."));`

5. Restriction d'accès

L'application web possède des interfaces publiques et d'autres qu'on veut restreindre l'accès. Les routes définies dans le `HomeController` et les ressources statiques seront permises d'accès. Les routes concernant `FilmController`, `CategorieController` et `ActeurController` auront un accès restreint.

La vue index contiendra un lien hypertexte vers la route `/login` pour se connecter et qu'après l'authentification l'utilisateur sera redirigé vers la route `/film/all`.

Ces restrictions seront définies dans la première méthode `configure` :

```java
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.formLogin().defaultSuccessUrl("/film/all", true);
    http.logout().logoutSuccessUrl("/");
    http.authorizeRequests().antMatchers("/","/cat/**","/detailfilm/**", "/images/**", "/photos/**").permitAll();
    http.authorizeRequests().antMatchers("/film/new/**", "/film/delete/**", "/film/edit/**").hasRole("ADMIN");
    http.authorizeRequests().anyRequest().authenticated();
```

4. Personnaliser l'affichage de la vue Thymeleaf selon le rôle de l'utilisateur

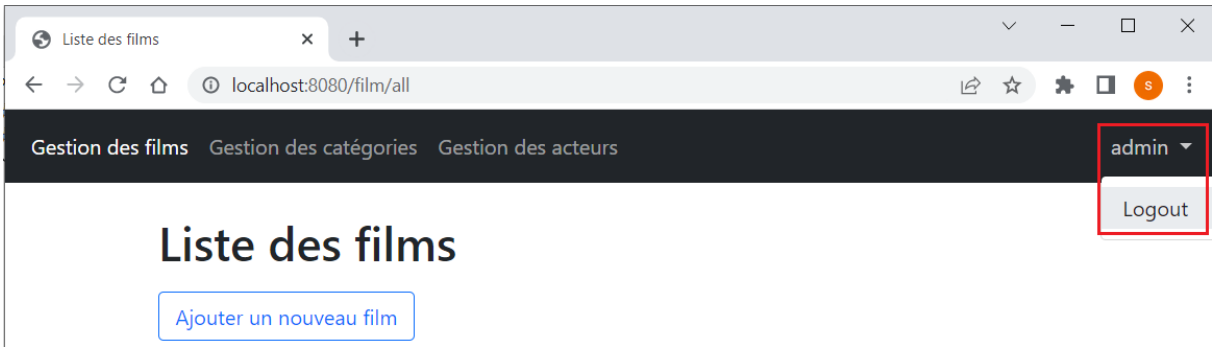
Pour pouvoir personnaliser l'affichage dans une vue Thymeleaf, il faut d'abord ajouter la dépendance suivante à pom.xml :

```xml
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity5</artifactId>
</dependency>
```

Ensuite il faut ajouter le namesapace suivant dans la vue `affiche.html` pour les films :

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

Une fois authentifié (avec admin par exemple), on va afficher dans la barre de navigation le nom de l'utilisateur connecté avec un lien vers la route `/logout` :



Voici le code HTML à ajouter dans la barre de navigation :

```
    </ul>
    <ul class="navbar-nav">
    <li class="nav-item dropdown">
      <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown">
      <span sec:authentication="name"></span>
      </a>
      <ul class="dropdown-menu">
        <li><a class="dropdown-item" href="/logout">Logout</a></li>
      </ul>
    </li>
    </ul>
```

Après la confirmation de la déconnexion, c'est la vue `index.html` qui est affichée associée à la route `/`.

Normalement l'utilisateur ayant le rôle USER n'a pas le droit de réaliser toutes les opérations CRUD (il n'a le droit que d'afficher, rechercher et filtrer en mode GET), le reste des opérations sont autorisées uniquement pour le rôle ADMIN.

Donc on va restreindre l'affichage des liens pour ajouter, modifier et supprimer un film, respectivement une catégorie et un acteur uniquement pour le rôle ADMIN :

```
<div class="my-3" sec:authorize="hasRole('ADMIN')">
    <a href="/film/new" class="btn btn-outline-primary">Ajouter un nouveau film</a>
</div>
```

```
<td>
<a th:href="@{/film/show/{id} (id=${f.id})}" class="btn btn-outline-info" >Détails</a>
<span sec:authorize="hasRole('ADMIN')">
<a th:href="@{/film/delete/{id} (id=${f.id})}" class="btn btn-outline-danger"
onclick="return confirm('Voulez vous vraiment supprimer ce film?')">Supprimer</a>
<a th:href="@{/film/edit/{id} (id=${f.id})}" class="btn btn-outline-warning" >Modifier</a>
</span>
</td>
```

Tester l'ensemble des fonctionnalités pour chaque rôle d'utilisateur.

II- Paramètres d'accès sauvegardés dans la BD

Maintenant on va appliquer le principe de l'authentification dans l'application web tout en sauvegardant les utilisateurs et leurs rôles dans la BD. Ceci est permis à travers l'utilisation de l'interface `UserDetailsService`.

L'interface `UserDetailsService` est utilisée pour récupérer les données relatives à l'utilisateur. Il a une seule méthode nommée `loadUserByUsername()` qui peut être redéfinie pour personnaliser le processus de recherche de l'utilisateur dans la BD.

Cette interface est utilisée par la couche service pour charger les détails de l'utilisateur lors de l'authentification.

1. Définition de la couche model : les entités AppUser et AppRole et leurs Repository

On commence par définir deux entités `AppUser` et `AppRole` (pour éviter la confusion avec les classes `User` et `Role` de Spring Security).

```java
package com.gestion.filmotheque.entities;

import java.util.List;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class AppUser {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(nullable = false, unique = true)
    private String username;
    private String password;
    @ManyToMany(fetch = FetchType.EAGER)
    private List<AppRole> appRoles;
}
```

```java
package com.gestion.filmotheque.entities;

import java.util.List;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class AppRole {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(nullable = false, unique = true)
    private String rolename;
    private String description;
    @ManyToMany(mappedBy = "appRoles")
    private List<AppUser> appUsers;
}
```

```java
package com.gestion.filmotheque.repository;

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

public interface AppUserRepository extends JpaRepository<AppUser, Integer> {

    AppUser findByUsername(String username);
}
```

```java
package com.gestion.filmotheque.repository;

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

public interface AppRoleRepository extends JpaRepository<AppRole, Integer>{

    AppRole findByRolename(String rolename);
}
```

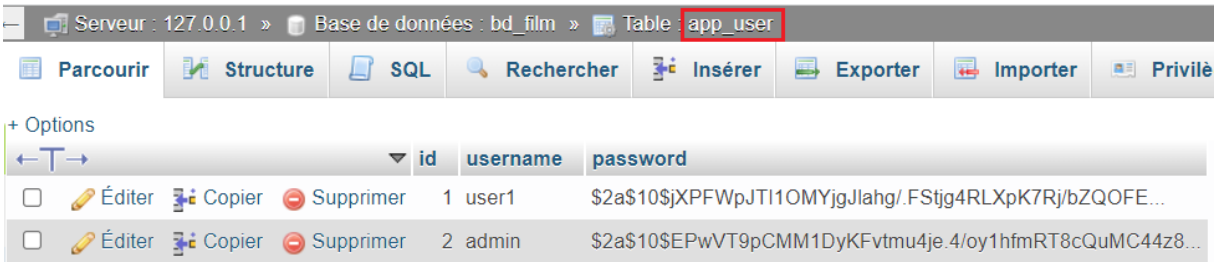
2. Définition de la coucher métier : la classe Service et son interface

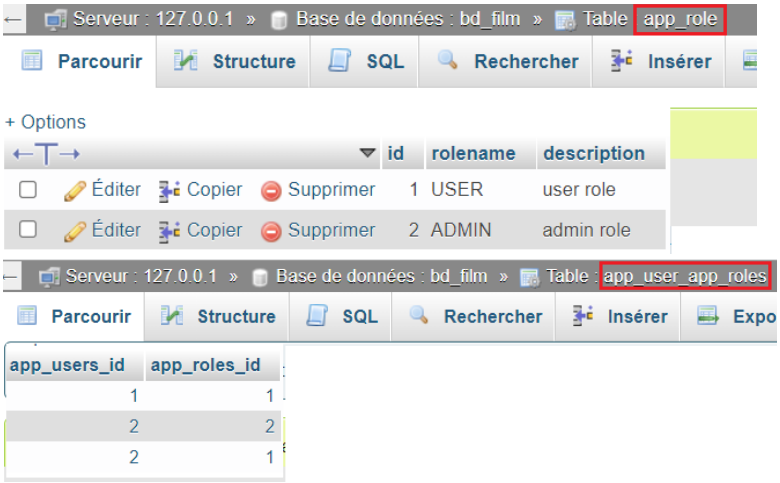
```java
package com.gestion.filmotheque.service;

import com.gestion.filmotheque.entities.AppUser;

public interface IServiceSecurity {

    public AppUser loadUserByUserName(String username);
    // les autres méthodes CRUD sur les users et roles
}
```

```java
package com.gestion.filmotheque.service;

import org.springframework.beans.factory.annotation.Autowired;

@Service
public class ServiceSecurity implements IServiceSecurity{

    @Autowired
    AppUserRepository appUserRepository;

    @Override
    public AppUser loadUserByUserName(String username) {
        return appUserRepository.findByUsername(username);
    }

}
```

3. Insertion des données dans la BD

Démarrer l'application pour générer les nouvelles tables dans la BD puis réaliser les insertions des données suivantes dans la BD (les mots de passe sont cryptés par Bcrypt) :





4. l'implémentation de UserDetailsService

On va définir une classe Service nommée `UserDetailsServiceImpl` qui va implémenter l'interface `UserDetailsservice`. Cette classe va redéfinir la méthode `loadUserByUserName()` en cherchant l'utilisateur par son `username` puis générer un objet de type `UserDetails` (interface implémentée par la classe User de Spring Security). Cet objet contient le `username`, le `password` et la collection des autorités (les rôles) qu'il possède :

```
package com.gestion.filmotheque.service;

import java.util.ArrayList;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    ServiceSecurity serviceSecurity;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        AppUser appUser = serviceSecurity.loadUserByUserName(username);
        Collection<GrantedAuthority> authorities = new ArrayList<GrantedAuthority>();
        appUser.getAppRoles().forEach(role->{
            SimpleGrantedAuthority simpleGrantedAuthority = new SimpleGrantedAuthority(role.getRolename());
            authorities.add(simpleGrantedAuthority);
        });
        User user = new User(appUser.getUsername(), appUser.getPassword(), authorities);
        return user;
    }

}
```

Puis dans la classe `SecurityConfig` on va injecter une dépendance de cette classe pour l'utiliser dans la première méthode configure :

```
import com.gestion.filmotheque.service.UserDetailsServiceImpl;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    UserDetailsServiceImpl userDetailsServiceImpl;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        /* PasswordEncoder passwordEncoder = passwordEncoder();
        auth.inMemoryAuthentication().withUser("user1").password(passwordEncoder.enc
        auth.inMemoryAuthentication().withUser("admin").password(passwordEncoder.enc
         */
        auth.userDetailsService(userDetailsServiceImpl);
    }
```

Maintenant l'utilisateur authentifié possède des autorités (authorities) et non des rôles. C'est pour cette raison qu'il faut changer la méthode `hasRole()` par `hasAuthority()` dans la deuxième méthode `configure` :

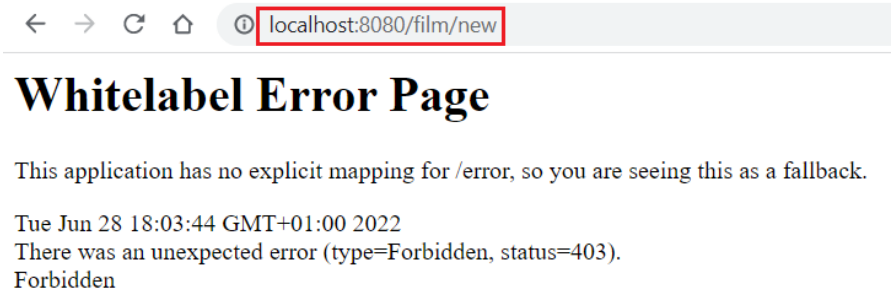
```java
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.formLogin().defaultSuccessUrl("/film/all", true);
    http.logout().logoutSuccessUrl("/");
    http.authorizeRequests().antMatchers("/","/cat/**","/detailfilm/**", "/images/**", "/photos/**").permitAll();
    http.authorizeRequests().antMatchers("/film/new/**", "/film/delete/**", "/film/edit/**").hasAuthority("ADMIN");
    http.authorizeRequests().anyRequest().authenticated();

}
```

De même dans la vue `affiche.html` des films il faut aussi effectuer le changement :

```html
<div class="my-3" sec:authorize="hasAuthority('ADMIN')">
    <a href="/film/new" class="btn btn-outline-primary">Ajouter un nouveau film</a>
</div>
```

5. Gestion personnalisée des accès non autorisé (denied)

C'est le cas par exemple de l'utilisateur `user1` qui tente d'accéder à une route dont il n'est pas autorisé (par exemple `http://localhost:8080/film/new` pour afficher le formulaire d'ajout d'un film). Naturellement Spring Security va interdire cet accès et va afficher une page avec le code 403 Access Denied.



Pour personnaliser cet affichage on devra traiter ces accès non autorisés en les redirigeant vers une route personnalisée. Dans la deuxième méthode configure ajouter la ligne suivante :

```java
http.authorizeRequests().anyRequest().authenticated();
http.exceptionHandling().accessDeniedPage("/forbidden");
```

Maintenant, il suffit de créer un controleur qui renvoi une vue contenant un message approprié pour la route `/forbidden`.