Spring MVC : la gestion d'utilisateurs (Spring Security)

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

La sécurité

But de la sécurité

Empêcher un utilisateur d'accéder à une ressource à laquelle il n'a pas droit

La sécurité

But de la sécurité

Empêcher un utilisateur d'accéder à une ressource à laquelle il n'a pas droit

Deux étapes

- Qui veut accéder ?
- Est-ce qu'il a le droit d'accéder à cette ressource ?

La sécurité

Configuration de la sécurité

- En utilisant des données statiques (en mémoire, dans un fichier)
- En utilisant des données dynamiques provenant d'une base de données

La sécurité

Configuration de la sécurité

- En utilisant des données statiques (en mémoire, dans un fichier)
- En utilisant des données dynamiques provenant d'une base de données

Pour cela

On va utiliser un module de **Spring** à savoir `org.springframework.security`.

La connexion statique

Étapes

- Ajouter les deux dépendances `spring-security-web` et `spring-security-config` dans le `pom.xml`
- Créer deux nouvelles classes de configuration
  - `SecurityConfig` : contenant au moins un `bean` précisant les détails sur les utilisateurs et une méthode définissant les chemins autorisés et les chemins nécessitant une authentification (les filtres)
  - `MessageSecurityWebApplicationInitializer` pour enregistrer les filtres définis dans `SecurityConfig`
- Déclarer `SecurityConfig` dans la classe dérivée de `AbstractAnnotationConfigDispatcherServletInitializer` (le contrôleur frontal)

La connexion statique

Ajouter les dépendances suivantes

```
<dependencies>
  <!-- ... les autres dependances ... -->
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>5.1.1.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>5.1.1.RELEASE</version>
  </dependency>
</dependencies>
```

La connexion statique

Créer une classe `SecurityConfig`

```
package org.eclipse.FirstSpringMvc.configuration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.
 builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.
 HttpSecurity;
import org.springframework.security.config.annotation.web.configuration
 .EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration
 .WebSecurityConfigurerAdapter;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetailsService
 ;
import org.springframework.security.provisioning.
 InMemoryUserDetailsManager;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter{
```

La classe `SecurityConfig` (suite)

```
  @Bean
  public UserDetailsService userDetailsService() {
    InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager
     ();
    String password = passwordEncoder().encode("user");
    System.out.print("password␣"  + password);
    manager.createUser(User.withUsername("user").password(password)
    .roles("USER").build());
    return manager;
  }
  @Bean
  public  NoOpPasswordEncoder passwordEncoder() {
    return (NoOpPasswordEncoder) NoOpPasswordEncoder.getInstance();
  }
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().antMatchers("/login").permitAll();
    http.authorizeRequests().anyRequest().authenticated()
      .and()
      .formLogin().loginPage("/login")
      .and()
      .logout().permitAll();
  }
}
```

La connexion statique

Explication

- Le `bean` appelé `userDetailsService` permet de préciser le type d'utilisateur (ici statique indiqué avec `InMemoryUserDetailsManager`), son nom, son mot de passe et ses rôles.
- Dans la méthode `configure`, on définit l'accès à notre application.
  - La première instruction autorise l'accès à la route `/login`
  - La deuxième instruction demande à toutes les requêtes de s'authentifier, et on indique la route d'authentification (ici `/login` que l'on a rendue accessible avec la première instruction)

La connexion statique

Le bean `NoOpPasswordEncoder` est déprécié

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

La connexion statique

Le bean `NoOpPasswordEncoder` est déprécié

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

Si on veut crypter les mots de passe, on peut utiliser l'algorithme suivant

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

La connexion statique

On peut aussi fusionner les deux premiers beans en un seul

```
  @Bean
  public UserDetailsService userDetailsService() {
    InMemoryUserDetailsManager manager = new
     InMemoryUserDetailsManager();
    manager.createUser(User.withDefaultPasswordEncoder().
     username("user").password("user").roles("USER").build());
    return manager;
  }
  @Override
  protected void configure(HttpSecurity http) throws Exception
   {
    http.authorizeRequests().antMatchers("/login").permitAll();
    http.authorizeRequests().anyRequest().authenticated()
       .and()
       .formLogin().loginPage("/login")
       .and()
       .logout().permitAll();
  }
}
```

La connexion statique

Le `bean` appelé `userDetailsService` peut être aussi remplacé par la méthode suivante

```
@Autowired
public void configureGlobal(
 AuthenticationManagerBuilder auth) throws Exception
 {
  auth.inMemoryAuthentication().withUser(User.
   withDefaultPasswordEncoder().username("user").
   password("user").roles("USER").build());
}
```

La connexion statique

Mettre à jour la classe dispatcher

```
package org.eclipse.FirstSpringMvc.configuration;

import org.springframework.web.servlet.support.
 AbstractAnnotationConfigDispatcherServletInitializer;

public class MyWebInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

  @Override
  protected Class<?>[] getRootConfigClasses() {
    // TODO Auto-generated method stub
    return null;
  }

  @Override
  protected Class<?>[] getServletConfigClasses() {
    return new Class [] { ApplicationConfig.class, MvcConfig.class, SecurityConfig.class };
  }

  @Override
  protected String[] getServletMappings() {
    // TODO Auto-generated method stub
    return new String [] {"/"};
  }
}
```

La connexion statique

Enregistrer les filtres définies

```
package org.eclipse.FirstSpringMvc.configuration;

import org.springframework.security.web.context.
 AbstractSecurityWebApplicationInitializer;

public class
 MessageSecurityWebApplicationInitializer extends
 AbstractSecurityWebApplicationInitializer {

}
```

`MessageSecurityWebApplicationInitializer` permet d'enregistrer automatiquement les filtres pour chaque URL de notre application.

La connexion statique

Mettre à jour la classe `MvcConfig`

```
@Override
public void addViewControllers(ViewControllerRegistry registry)
  {
 registry.addViewController("/").setViewName("jsp/home");
 registry.addViewController("/login").setViewName("jsp/login");
}
```

La connexion statique

Mettre à jour la classe `MvcConfig`

```
@Override
public void addViewControllers(ViewControllerRegistry registry)
  {
 registry.addViewController("/").setViewName("jsp/home");
 registry.addViewController("/login").setViewName("jsp/login");
}
```

La route `/login` n'a pas de contrôleur qui l'intercepte, donc lorsqu'elle est saisie, c'est la vue `login.jsp` qui sera exécutée.

La connexion statique

Créer la vue `login.jsp`

```
<body>
  <form action="login" method="post">
    <div>
      <input type="text" name="username" placeholder="Login" />
    </div>
    <div>
      <input type="password" name="password" placeholder="Password" />
    </div>
    <div>
      <input type="submit" value="Connection" />
    </div>
    <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.
     token}" />
  </form>
</body>
```

Le dernier `input` permet à **Spring** d'assurer la sécurité de transmission de données, de se protéger de `Cross Site Request Forgery` et de rediriger vers la page demandée après connexion.

La connexion statique

Tester cela

- Tester la route `/personne` (vérifier la redirection vers `login`)
- Tester avec des faux identifiants
- Tester avec des identifiants corrects `user` et `user`

La connexion dynamique

Étapes

- Modifier le fichier `SecurityConfig`
- Créer deux entités `User` et `Role` et les `Repository` respectifs pour la gestion des utilisateurs (enregistrés dans la base de données)
- Préparer la couche métier qui va permettre de gérer l'accès à l'application
- Adapter la vue pour les messages d'erreurs

La connexion dynamique

Mettre à jour la méthode `configureGlobal` de la classe `SecurityConfig`

```
@Autowired
private UserDetailsService userDetailsService;

@Autowired
public void configureGlobal(
 AuthenticationManagerBuilder auth) throws Exception
 {
  auth.userDetailsService(userDetailsService);
}
```

On précise donc que les informations sur notre utilisateur se trouveront dans une classe qui implémente l'interface `UserDetailsService` et qui utilisera les deux entités `User` et `Role` que nous créerons plus tard.

La connexion dynamique

On définit un `bean` pour ne pas crypter les mots de passe

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

La connexion dynamique

Si on veut crypter les mots de passe ?

```
@Autowired
private UserDetailsService userDetailsService;
@Autowired
public void configureGlobal(
 AuthenticationManagerBuilder auth) throws Exception
 {
  auth.userDetailsService(userDetailsService).
   passwordEncoder(passwordEncoder());
}
```

Et on retourne une instance de l'algorithme de cryptage

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

La connexion dynamique

Créer l'entité `Role`

```
@Entity
@Table(name="roles")
public class Role {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY
   )
  private Long id;
  private String titre;

  // ajouter les getters / setters
```

La connexion dynamique

Créer l'entité `User`

```
@Entity
@Table(name = "users")
public class User implements Serializable {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long num;
  private String userName;
  private String password;
  @ManyToMany(cascade={CascadeType.PERSIST, CascadeType.
   REMOVE},fetch = FetchType.EAGER)
  private List <Role> roles = new ArrayList<Role>();
  // + getters & setters + addRole & removeRole
```

`fetch = FetchType.EAGER` pour indiquer que la relation doit être chargée en même temps que l'entité `User`.

La connexion dynamique

Créer le `repository` pour `User`

```
public interface UserRepository extends
 JpaRepository<User, Long> {
  public User findByUserName(String username);
}
```

La méthode `findByUserName` sera utilisée plus tard.

Et pour `Role`

```
public interface RoleRepository extends
 JpaRepository<Role, Long>{

}
```

La connexion dynamique

Ensuite

- Créer un package `org.eclipse.FirstSpringMvc.security`
- Créer deux classes dans ce package :
  - une classe `UserDetailsImpl` qui implémente l'interface `UserDetails` : elle implémente donc les méthodes permettant de retourner des informations sur l'utilisateur
  - une classe `UserDetailsServiceImpl` qui implémente l'interface `UserDetailsService` : elle implémente donc une méthode vérifiant l'existence d'un utilisateur selon la valeur de `userName` (en utilisant le repository)
- Scanner le package `org.eclipse.FirstSpringMvc.security` dans `ApplicationConfig`

La connexion dynamique

Créer une classe qui implémente l'interface `UserDetails`

```
package org.eclipse.FirstSpringMvc.security;

public class UserDetailsImpl  implements UserDetails
 {
}
```

Cette classe implémente l'interface `UserDetails`, elle doit donc implémenter toutes les méthodes que nous détaillerons dans le slide suivant

La connexion dynamique

Créer une classe qui implémente l'interface `UserDetails`

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import org.eclipse.FirstSpringMvc.model.Role;
import org.eclipse.FirstSpringMvc.model.User;

public class UserDetailsImpl  implements UserDetails{

  private User user;

  public  UserDetailsImpl(User user){
    this.user = user;
  }

  @Override
  public Collection<? extends GrantedAuthority> getAuthorities() {
    final List<GrantedAuthority> authorities = new ArrayList<GrantedAuthority>();
    for (final Role role : user.getRoles())
      authorities.add(new SimpleGrantedAuthority(role.getTitre()));
    return authorities;
  }
```

```
  @Override
  public boolean isAccountNonExpired() {
    return true;
  }
  @Override
  public boolean isAccountNonLocked() {
    return true;
  }
  @Override
  public boolean isCredentialsNonExpired() {
    return true;
  }
  @Override
  public boolean isEnabled() {
    return true;
  }
  @Override
  public String getUsername() {
    return user.getUserName();
  }
  @Override
  public String getPassword() {
    return user.getPassword();
  }
}
```

La connexion dynamique

Créer une classe qui implémente l'interface `UserDetailsService`

```
package org.eclipse.FirstSpringMvc.security;

import org.springframework.stereotype.Service;

@Service
public class UserDetailsServiceImpl implements
 UserDetailsService {
}
```

Cette classe implémente l'interface `UserDetailsService`, elle doit donc implémenter la méthode `loadUserByUserName()` qui retourne un objet de la classe qui implémente l'interface `UserDetails`.

On utilise l'annotation Service pour dire qu'il s'agit d'un `Component` qu'on peut injecter avec l'annotation `Autowired` et qui fait parti de la couche métier.

La connexion dynamique

Contenu de la classe `UserDetailsServiceImpl`

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetailsService
 ;
import org.springframework.security.core.userdetails.
 UsernameNotFoundException;
import org.springframework.stereotype.Service;
import org.eclipse.FirstSpringMvc.dao.UserRepository;
import org.eclipse.FirstSpringMvc.model.User;
@Service
public class UserDetailsServiceImpl implements UserDetailsService {
  @Autowired UserRepository userRepository;
  @Override
  public UserDetailsImpl loadUserByUsername(String username) throws
   UsernameNotFoundException {
    User user=userRepository.findByUserName(username);
    if(null == user){
      throw new UsernameNotFoundException("No user named "+username);
    }else{
      return new UserDetailsImpl(user);
    }
  }
}
```

La connexion dynamique

Scanner le package `org.eclipse.FirstSpringMvc.security` **dans** `ApplicationConfig`

```
// tous les imports et les annotations

@ComponentScan("org.eclipse.FirstSpringMvc.
 controller,␣org.eclipse.FirstSpringMvc.security")
public class ApplicationConfig {

  // le code precedent
}
```

La connexion dynamique

Modifier la vue `login.jsp`

```
<body >
  <form action="login" method="post">
    <div>
      <input type="text" name="username" placeholder="Login" />
    </div>
    <div>
      <input type="password" name="password" placeholder="Password" />
    </div>
    <div>
      <input type="submit" value="Connection" />
    </div>
    <c:if test="${param.error ne null}">
      <div >Invalid username and password.</div>
    </c:if>
    <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.
     token}" />
  </form>
</body>
```

La connexion dynamique

Avant de tester, créer trois utilisateurs avec des rôles différents

```
insert into roles values (1,"ROLE_ADMIN"),
(2,"ROLE_USER");

insert into users values (1,"wick","wick"),
(2,"john","john"),
(3,"alan","alan");

insert into users_roles values (1,1),
(2,2),
(1,2),
(3,1);
```

La connexion dynamique

Enfin

- Il faut ajouter le préfixe `c` pour la librairie `core` de la **JSTL** et activer le langage d'expression.

```
<%@ page isELIgnored="false"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/
 core" prefix="c" %>
```

- Il ne faut pas utiliser un algorithme de cryptage pour les mots de passe

L'utilisateur connecté

Pour récupérer les données relatives à l'utilisateur connecté, il faut écrire dans le contrôleur

```
UserDetailsImpl connectedUser = (UserDetailsImpl)
 SecurityContextHolder.getContext().
 getAuthentication().getPrincipal();

User user = userRepository.findByUserName(
 connectedUser.getUsername());
```

Les rôles

Étapes

- Activer les annotations de sécurité (comme `@Secured("ROLE")`) dans `SecurityConfig`
- Annoter des méthodes et/ou des classes avec `@Secured("ROLE")` (ou autres)

Les rôles

Étapes

- Activer les annotations de sécurité (comme `@Secured("ROLE")`) dans `SecurityConfig`
- Annoter des méthodes et/ou des classes avec `@Secured("ROLE")` (ou autres)

Pour notre exemple, supposons qu'on a deux rôles principaux

- `ROLE_ADMIN`
- `ROLE_USER`

Les rôles

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

```
@EnableGlobalMethodSecurity(
  securedEnabled = true,
  jsr250Enabled = true,
  prePostEnabled = true)
```

Les rôles

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

```
@EnableGlobalMethodSecurity(
  securedEnabled = true,
  jsr250Enabled = true,
  prePostEnabled = true)
```

Explication

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

Les rôles

Le contrôleur `PersonneController`

```
@Controller
@Secured("ROLE_ADMIN")
public class PersonneController {
  @Autowired
  private PersonneRepository personneRepository;

  @GetMapping("/personne")
```

Les rôles

Le contrôleur `PersonneController`

```
@Controller
@Secured("ROLE_ADMIN")
public class PersonneController {
  @Autowired
  private PersonneRepository personneRepository;

  @GetMapping("/personne")
```

- `@Secured("ROLE_ADMIN")` : rend l'accès à la route `/personne` unique aux utilisateurs ayant le rôle `ROLE_ADMIN`
- On peut remplacer `@Secured` par `@RolesAllowed`
- On peut aussi autoriser plusieurs rôles au même temps juste en ajoutant `@Secured({...,...})`

Les rôles

Le contrôleur `PersonneController`

```
@Controller
@Secured("ROLE_ADMIN")
public class PersonneController {
  @Autowired
  private PersonneRepository personneRepository;

  @GetMapping("/personne")
  public String firstMethod() { ... }

  @Secured("ROLE_USER")
  @GetMapping("/personnes")
  public String secondMethod() { ... }
```

Les rôles

Le contrôleur `PersonneController`

```
@Controller
@Secured("ROLE_ADMIN")
public class PersonneController {
  @Autowired
  private PersonneRepository personneRepository;

  @GetMapping("/personne")
  public String firstMethod() { ... }

  @Secured("ROLE_USER")
  @GetMapping("/personnes")
  public String secondMethod() { ... }
```

- Le chemin `/personne` est accessible seulement aux utilisateurs ayant le rôle `ROLE_ADMIN` (la méthode `firstMethod()` prend la sécurité du contrôleur)
- Le chemin `/personnes` est accessible seulement aux utilisateurs ayant le rôle `ROLE_USER` (la sécurité de la méthode `secondMethod()` annule la sécurité du contrôleur `ROLE_ADMIN`)

Les rôles

Le service `PersonneService`

```
package org.eclipse.FirstSpringMvc.service;

@Service
public class PersonneService {
  @Autowired
  private PersonneRepository personneRepository;
  @Secured("ROLE_ADMIN")
  public List <Personne> findAll(){
    return personneRepository.findAll();
  }
  @Secured("ROLE_USER")
  public Personne findById(Long id) {
    return personneRepository.findById(id).orElse(null);
  }
}
```

Les rôles

Le service `PersonneService`

```
package org.eclipse.FirstSpringMvc.service;

@Service
public class PersonneService {
  @Autowired
  private PersonneRepository personneRepository;
  @Secured("ROLE_ADMIN")
  public List <Personne> findAll(){
    return personneRepository.findAll();
  }
  @Secured("ROLE_USER")
  public Personne findById(Long id) {
    return personneRepository.findById(id).orElse(null);
  }
}
```

Il faut scanner les services dans `ApplicationConfig`

```
@ComponentScan("org.eclipse.FirstSpringMvc.controller, org.eclipse.
 FirstSpringMvc.security, org.eclipse.FirstSpringMvc.service")
public class ApplicationConfig {
```

Les rôles

Le contrôleur `ShowPersonneController`

```
@Controller
public class ShowPersonneController {

  @Autowired
  private PersonneService personneService;

  @GetMapping("/showPersonnes")
  public String showPersonnes(Model model) {
    ArrayList <Personne> personnes = (ArrayList<Personne>)
     personneService.findAll();
    model.addAttribute("personnes", personnes);
    return "jsp/showPersonnes";
  }

  @GetMapping("/showPersonne/{num}")
  public String showPersonne(Model model, @PathVariable long num) {
    Personne personne =  personneService.findById(num);
    model.addAttribute("personne", personne);
    return "jsp/showPersonne";
  }
}
```

Les rôles

La vue `showPersonnes.jsp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
 pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>List of Personnes</title>
  </head>
  <body>
    <h1>List of Persons (Private access)</h1>
    <table>
      <tr><td> <b>Nom</b></td><td> <b>Prenom</b></td></tr>
      <c:forEach items="${ personnes }" var = "elt">
        <tr><td> <c:out value="${ elt['nom'] }" />  </td>
        <td> <c:out value="${ elt['prenom'] }" /> </td></tr>
      </c:forEach>
    </table>
  </body>
</html>
```

Les rôles

La vue `showPersonne.jsp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
 pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html>
<html>
  <head>
    <title>List of Personnes</title>
  </head>
  <body>
    <h1>One person (Private access)</h1>
    <table>
      <tr>
          <td> <b>Nom</b></td><td> <i>Prenom</i></td>
      </tr>
      <tr>
          <td> <c:out value="${ personne['nom']}" />  </td>
          <td> <c:out value="${ personne['prenom'] }" /> </td>
      </tr>
    </table>
  </body>
</html>
```

Les rôles

Pour tester

- Se connecter avec deux comptes utilisateurs différents : un qui a le rôle `ROLE_ADMIN` et un deuxième qui a le rôle `ROLE_USER`
- Aller sur les routes `/showPersonne/1`, `/showPersonnes` avec les deux comptes utilisateurs

Les rôles

Le service `PersonneService`

```
package org.eclipse.FirstSpringMvc.service;

@Service
public class PersonneService {
  @Autowired
  private PersonneRepository personneRepository;
  @Secured({"ROLE_ADMIN","ROLE_USER"})
  public List <Personne> findAll(){
    return personneRepository.findAll();
  }
  // le reste ...
}
```

Remarques

- La méthode `findAll()` de `PersonneService` est accessible seulement aux utilisateurs ayant comme rôle `ROLE_ADMIN` **OU** `ROLE_USER`
- Ce n'est pas un **ET**

Les rôles

Le service `PersonneService`

```
package org.eclipse.FirstSpringMvc.service;

@Service
public class PersonneService {
  @Autowired
  private PersonneRepository personneRepository;
  @Secured({"ROLE_ADMIN","ROLE_USER"})
  public List <Personne> findAll(){
    return personneRepository.findAll();
  }
  // le reste ...
}
```

Remarques

- La méthode `findAll()` de `PersonneService` est accessible seulement aux utilisateurs ayant comme rôle `ROLE_ADMIN` **OU** `ROLE_USER`
- Ce n'est pas un **ET**

Impossible d'exiger deux rôles avec `@Secured`

Les rôles

Pour exiger deux rôles

```
package org.eclipse.FirstSpringMvc.service;

@Service
public class PersonneService {
  @Autowired
  private PersonneRepository personneRepository;
  @PreAuthorize("hasRole('ROLE_USER') and hasRole('
   ROLE_ADMIN')")
  public List <Personne> findAll(){
    return personneRepository.findAll();
  }
  // le reste ...
}
```

Les rôles

Autres annotations

- `@PreAuthorize("hasRole('ROLE_ADMIN')")` : fera la même chose que `@Secured("ROLE_ADMIN")`. Contrairement à cette dernière, `@PreAuthorize` autorise l'utilisation des SpEL (Spring Expression Language) (voir exemple suivant).

```
@PreAuthorize("#username == authentication.
 principal.username")
public String myMethod(String username) {
    //...
}
```

- La méthode `myMethod` sera exécutée si et seulement si la valeur de l'attribut `username` de la méthode `myMethod` est égal à celui de l'utilisateur principal.

Les rôles

Autres annotations

- `@PostAuthorize("hasRole('ROLE_ADMIN')")` : fera la même chose que `@PreAuthorize("hasRole('ROLE_ADMIN')")`. Cette dernière vérifiera le rôle avant d'exécuter la méthode et la première après.
- ...

On peut annoter un élément avec plusieurs annotations de sécurité : `@PostAuthorize(...)` et `@PreAuthorize(...)` par exemple.

Les rôles

Pour tester `@PostAuthorize(...)`, ajoutons la méthode suivante au service

```
package org.eclipse.FirstSpringMvc.service;

@Service
public class PersonneService {
  @Autowired
  private PersonneRepository personneRepository;
  @PostAuthorize("returnObject.nom == authentication.principal.
   username")
  public Personne getPersonne(String username) {
    Personne personne= personneRepository.findByNom(username);
    if (personne == null)
        personne = new Personne();
    return personne;
  }
  // le reste ...
}
```

N'oublions pas de définir `findByNom()` dans `PersonneRepository`

Les rôles

Appelons la nouvelle méthode de service dans le contrôleur

`ShowPersonneController`

```
@Controller
public class ShowPersonneController {
  @Autowired
  private PersonneService personneService;
  @GetMapping("/showPersonne/{nom}")
  public String showPersonne(@PathVariable String nom, Model
   model) {

    Personne personne =  personneService.getPersonne(nom);
    model.addAttribute("personne", personne);
    return "jsp/showPersonne";
  }
  //+ le code precedent
}
```

Les rôles

Pour tester

Pour accéder à la méthode du service précédent, il faut que l'utilisateur ait le même nom que la personne que l'on souhaite afficher ses détails dans la vue

La déconnexion

Pour la déconnexion

- Plus besoin de contrôleur
- Plus besoin de vider la session....

La déconnexion

Pour la déconnexion

- Plus besoin de contrôleur
- Plus besoin de vider la session....

Il faut juste avoir une URL `/logout` dans les vues

```
<c:url var="logoutUrl" value="/logout"/>
<form action="${logoutUrl}" method="post">
  <input type="submit" value="Deconnection" />
  <input type="hidden" name="${_csrf.parameterName}"
    value="${_csrf.token}"/>
</form>
```