Spring Security avec Spring MVC

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 & Spring MVC

But de la sécurité

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

Spring Security & Spring MVC

But de la sécurité

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

Deux étapes

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

Spring Security & Spring MVC

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)

Spring Security & Spring MVC

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)

Solution

Utilisation de **Spring Security**

Spring Security & Spring MVC

Étapes

- Ajouter les deux dépendances `spring-security-web` et `spring-security-config` dans `pom.xml`
- Créer deux nouvelles classes :
  - `SecurityConfig` : classe de configuration contenant les détails sur les utilisateurs et précisant les chemins autorisés et les chemins nécessitant une authentification
  - `MessageSecurityWebApplicationInitializer` pour enregistrer les filtres définis dans `SecurityConfig`
- Déclarer `SecurityConfig` dans la classe dérivée de `AbstractAnnotationConfigDispatcherServletInitializer`

Spring Security & Spring MVC

Ajouter les dépendances suivantes

```
<dependencies>
  <!-- ... les autres dependances ... -->
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>5.2.1.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>5.2.1.RELEASE</version>
  </dependency>
</dependencies>
```

Spring Security & Spring MVC

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.crypto.password.NoOpPasswordEncoder
   ;
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 {
```

Contenu de 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();
  http.csrf().disable();
}
```

Spring Security & Spring MVC

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 indique le chemin vers la page d'authentification (ici `/login`) accessible grâce à la première instruction)

Spring Security & Spring MVC

Le bean `NoOpPasswordEncoder` est déprécié pour rappeler qu'il faut crypter les mots de passe

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

Spring Security & Spring MVC

Le bean `NoOpPasswordEncoder` est déprécié pour rappeler qu'il faut crypter les mots de passe

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

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

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

Spring Security & Spring MVC

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

Spring Security & Spring MVC

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());
}
```

Spring Security & Spring MVC

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 [] {"/"};
  }
}
```

Spring Security & Spring MVC

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.

Spring Security & Spring MVC

Mettre à jour la classe `MvcConfig`

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

Spring Security & Spring MVC

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, la vue `login.jsp` sera exécutée.

Contenu de `login.jsp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Login page</title>
</head>
<body>
  <h1>Login page</h1>
  <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>
</html>
```

Le dernier `input` permet à **Spring** d'assurer la sécurité de transmission de données, de se protéger de `Cross Site Request Forgery`

Spring Security & Spring MVC

Tester cela

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

Spring Security & Spring MVC

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 aux messages d'erreurs

Spring Security & Spring MVC

Commençons par modifier la méthode `configureGlobal` de `SecurityConfig`

```
@Autowired
private UserDetailsService userDetailsService;

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

Spring Security & Spring MVC

Commençons par modifier la méthode `configureGlobal` de `SecurityConfig`

```
@Autowired
private UserDetailsService userDetailsService;

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


Explication

- Les détails sur l'utilisateur seront chargés à partir de `userDetailsService`
- `UserDetailsService` est une interface définie par **Spring Security**
- Nous devons donc créer une classe qui l'implémente

Spring Security & Spring MVC

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

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

Spring Security & Spring MVC

Pour 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();
}
```

Spring Security & Spring MVC

Créons l'entité `Role`

```
@Entity
public class Role {

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

  // + les getters / setters
```

Spring Security & Spring MVC

Créons l'entité `User`

```
@Entity
public class User {

  @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
```

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

Créons `UserRepository`

```
package org.eclipse.firstspringmvc.dao;

import org.eclipse.firstspringmvc.model.User;
import org.springframework.data.jpa.repository.JpaRepository;

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

La méthode `findByUserName` sera utilisée plus tard.

Et `RoleRepository`

```
package org.eclipse.firstspringmvc.dao;

import org.eclipse.firstspringmvc.model.Role;
import org.springframework.data.jpa.repository.JpaRepository;

public interface RoleRepository extends JpaRepository<Role, Long> {

}
```

Ensuite, on va créer

- un package `org.eclipse.firstspringmvc.security`
- deux classes dans ce package :
  - une classe `UserDetailsServiceImpl` qui implémente l'interface `UserDetailsService` : elle implémente donc une méthode vérifiant l'existence d'un utilisateur selon la valeur de `userName` et retournant un objet de l'interface `UserDetails`
  - une classe `UserDetailsImpl` qui implémente l'interface `UserDetails` : elle implémente donc les méthodes retournant des informations sur l'utilisateur

Ensuite, on va créer

- un package `org.eclipse.firstspringmvc.security`
- deux classes dans ce package :
  - une classe `UserDetailsServiceImpl` qui implémente l'interface `UserDetailsService` : elle implémente donc une méthode vérifiant l'existence d'un utilisateur selon la valeur de `userName` et retournant un objet de l'interface `UserDetails`
  - une classe `UserDetailsImpl` qui implémente l'interface `UserDetails` : elle implémente donc les méthodes retournant des informations sur l'utilisateur

Remarque

N'oublions pas de scanner le package `org.eclipse.firstspringmvc.security` dans `ApplicationConfig`

Spring Security & Spring MVC

Créons une classe qui implémente l'interface `UserDetailsService`

```
package org.eclipse.firstspringmvc.security;

import org.springframework.security.core.userdetails.
   UserDetailsService;

public class UserDetailsServiceImpl implements
   UserDetailsService {

}
```

Spring Security & Spring MVC

Créons une classe qui implémente l'interface `UserDetailsService`

```
package org.eclipse.firstspringmvc.security;

import org.springframework.security.core.userdetails.
   UserDetailsService;

public class UserDetailsServiceImpl implements
   UserDetailsService {

}
```


Remarque

Cette classe implémente l'interface `UserDetailsService`, elle doit donc implémenter toutes ses méthodes abstraites.

Spring Security & Spring MVC

Ajoutons les méthodes à implémenter

```
public class UserDetailsServiceImpl implements UserDetailsService {

        @Override
        public UserDetails loadUserByUsername(String username) throws
           UsernameNotFoundException {
                return null;
        }

}
```

Spring Security & Spring MVC

Ajoutons les méthodes à implémenter

```
public class UserDetailsServiceImpl implements UserDetailsService {

        @Override
        public UserDetails loadUserByUsername(String username) throws
           UsernameNotFoundException {
                return null;
        }

}
```

Explication

- La classe `UserDetailsServiceImpl` doit implémenter la méthode `loadUserByUserName()` qui retourne un objet de la classe `UserDetailsImpl` qui implémente l'interface `UserDetails`.
- On utilise l'annotation `@Service` pour pouvoir injecter `UserDetailsServiceImpl` avec `@Autowired` dans `SecurityConfig`

Spring Security & Spring MVC

Ajoutons l'annotation `@Service` à `UserDetailsServiceImpl` et implémentons la méthode `loadUserByUserName()`

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService {

  @Autowired
  private UserRepository userRepository;

  @Override
  public UserDetailsImpl loadUserByUsername(String username) throws
     UsernameNotFoundException {
    User user = userRepository.findByUserName(username);
    if(user == null) {
      throw new UsernameNotFoundException("No user named " + username);
    } else {
      return new UserDetailsImpl(user);
    }
  }
}
```

Spring Security & Spring MVC

Créons la classe `UserDetailsImpl` qui implémente l'interface `UserDetails`

```
package org.eclipse.firstspringmvc.security;

public class UserDetailsImpl implements UserDetails {

}
```

Spring Security & Spring MVC

Créons la classe `UserDetailsImpl` qui implémente l'interface `UserDetails`

```
package org.eclipse.firstspringmvc.security;

public class UserDetailsImpl implements UserDetails {

}
```


Remarque

Cette classe implémente l'interface `UserDetails`, elle doit donc implémenter toutes ses méthodes abstraites

Spring Security & Spring MVC

Modifions le contenu de `UserDetailsImpl`

```
package org.eclipse.firstspringmvc.security;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

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

import org.springframework.security.core.authority.SimpleGrantedAuthority;
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;
  }
```

Spring Security & Spring MVC

```
  @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();
  }
}
```

Modifions la vue `login.jsp` en ajoutant l'affichage d'un message d'erreur

```
<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>
```

Modifions la vue `login.jsp` en ajoutant l'affichage d'un message d'erreur

```
<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>
```

N'oublions pas de définir le préfixe de la `JSTL`

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

Spring Security & Spring MVC

Avant de tester, lancez le projet pour que Spring crée les tables ensuite créez trois utilisateurs avec le script SQL suivant

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

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

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

Spring Security & Spring MVC

Pour tester

- `(alan, alan)` a le rôle `ROLE_ADMIN`
- `(john, john)` a le rôle `ROLE_USER`
- `(wick, wick)` a les deux rôles `ROLE_USER` et `ROLE_ADMIN`

Spring Security & Spring MVC

Avant de tester

N'utilisons pas d'algorithme de cryptage pour les mots de passe.

Spring Security & Spring MVC

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());
```

Par exemple, dans `HomeController` on ajoute le contenu précédent

```
@Controller
public class HomeController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping(path = {"/hello", "/"})
    public String sayHello(Model model) {
        UserDetailsImpl connectedUser = (UserDetailsImpl)
           SecurityContextHolder.getContext().getAuthentication().
           getPrincipal();

        User user = userRepository.findByUserName(connectedUser.
           getUsername());
        model.addAttribute("nom", user.getUserName());
        return "jsp/hello";
    }
}
```

Par exemple, dans `HomeController` on ajoute le contenu précédent

```
@Controller
public class HomeController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping(path = {"/hello", "/"})
    public String sayHello(Model model) {
        UserDetailsImpl connectedUser = (UserDetailsImpl)
           SecurityContextHolder.getContext().getAuthentication().
           getPrincipal();

        User user = userRepository.findByUserName(connectedUser.
           getUsername());
        model.addAttribute("nom", user.getUserName());
        return "jsp/hello";
    }
}
```

Pour tester, aller à `http://localhost:8080/firstspringmvc/hello`

Spring Security & Spring MVC

Étapes

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

Spring Security & Spring MVC

Étapes

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

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

- `ROLE_ADMIN`
- `ROLE_USER`

Spring Security & Spring MVC

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

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

Spring Security & Spring MVC

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`.

Spring Security & Spring MVC

Le contrôleur `PersonneController`

```
@Controller
@Secured("ROLE_ADMIN")
public class PersonneController {

    @Autowired
    private PersonneRepository personneRepository;
```

Spring Security & Spring MVC

Le contrôleur `PersonneController`

```
@Controller
@Secured("ROLE_ADMIN")
public class PersonneController {

    @Autowired
    private PersonneRepository personneRepository;
```

- `@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({...,...})`

Spring Security & Spring MVC

Pour tester

- Connectez-vous avec `(wick, wick)` et vérifiez que vous avez accès à la route `addPerson`
- Connectez-vous avec `(john, john)` et vérifiez qu'un message d'erreur `403 Forbidden` lorsque vous essayez d'accéder à la route `addPerson`

Le contrôleur `PersonneController`

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

        @GetMapping(value = "/addPerson")
        public String addPerson() { ... }

        @Secured("ROLE_USER")
        @GetMapping(value = "/showAll")
        public ModelAndView showAll() { ... }
```

Le contrôleur `PersonneController`

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

        @GetMapping(value = "/addPerson")
        public String addPerson() { ... }

        @Secured("ROLE_USER")
        @GetMapping(value = "/showAll")
        public ModelAndView showAll() { ... }
```


- La route `/addPerson` est accessible seulement aux utilisateurs ayant le rôle `ROLE_ADMIN` (la méthode `addPerson()` prend la sécurité du contrôleur)
- La route `/showAll` est accessible seulement aux utilisateurs ayant le rôle `ROLE_USER` (la sécurité de la méthode `showAll()` annule la sécurité du contrôleur `ROLE_ADMIN`)

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);
  }
}
```

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);
  }
}
```

N'oublions pas de scanner les services dans `ApplicationConfig`

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

Spring Security & Spring MVC

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";
  }
}
```

Spring Security & Spring MVC

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 Persons</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>
```

Spring Security & Spring MVC

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>One person</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>
```

Spring Security & Spring MVC

Pour tester

- Connectez-vous avec `(alan, alan)` et vérifiez que vous avez accès à `showPersonnes` mais pas à `showPersonne/1`
- Connectez-vous avec `(john, john)` et vérifiez que vous avez accès à `showPersonne/1` mais pas à `showPersonnes`
- Connectez-vous avec `(wick, wick)` et vérifiez que vous avez accès à la fois à `showPersonne/1` et à `showPersonnes`

Spring Security & Spring MVC

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

```
@Secured({ "ROLE_ADMIN", "ROLE_USER" })
public List<Personne> findAll() {
        return personneRepository.findAll();
}
```

Spring Security & Spring MVC

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

```
@Secured({ "ROLE_ADMIN", "ROLE_USER" })
public List<Personne> findAll() {
        return personneRepository.findAll();
}
```


Remarques

- La route `/showPersonnes` est accessible seulement aux utilisateurs ayant le rôle `ROLE_ADMIN` **OU** le `ROLE_USER`
- Ce n'est pas un **ET**

Spring Security & Spring MVC

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

```
@Secured({ "ROLE_ADMIN", "ROLE_USER" })
public List<Personne> findAll() {
        return personneRepository.findAll();
}
```


Remarques

- La route `/showPersonnes` est accessible seulement aux utilisateurs ayant le rôle `ROLE_ADMIN` **OU** le `ROLE_USER`
- Ce n'est pas un **ET**

Impossible d'exiger deux rôles avec `@Secured`

Spring Security & Spring MVC

Pour exiger deux rôles (ou plus)

```
@PreAuthorize("hasRole('ROLE_USER') and hasRole('
   ROLE_ADMIN')")
public List<Personne> findAll() {
        return personneRepository.findAll();
}
```

Spring Security & Spring MVC

Pour exiger deux rôles (ou plus)

```
@PreAuthorize("hasRole('ROLE_USER') and hasRole('
   ROLE_ADMIN')")
public List<Personne> findAll() {
        return personneRepository.findAll();
}
```

Pour tester

- Connectez-vous avec `(alan, alan)` ensuite `(john, john)` et vérifiez que vous n'avez pas accès à `showPersonnes`
- Connectez-vous avec `(wick, wick)` et vérifiez que vous avez accès à `showPersonnes`

Spring Security & Spring MVC

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.

Spring Security & Spring MVC

`@PostAuthorize`

`@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.

Spring Security & Spring MVC

`@PostAuthorize`

`@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.

Spring Security & Spring MVC

Pour tester `@PostAuthorize(...)`, ajoutons la méthode suivante dans `PersonneService`

```
@PostAuthorize("returnObject.nom == authentication.principal.
   username")
public Personne getPersonne(String username) {
    Personne personne= personneRepository.findByNom(username);
    if (personne == null)
        personne = new Personne();
    return personne;
}
```

Spring Security & Spring MVC

Pour tester `@PostAuthorize(...)`, ajoutons la méthode suivante dans `PersonneService`

```
@PostAuthorize("returnObject.nom == authentication.principal.
   username")
public Personne getPersonne(String username) {
    Personne personne= personneRepository.findByNom(username);
    if (personne == null)
        personne = new Personne();
    return personne;
}
```

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

Spring Security & Spring MVC

Appelons la nouvelle méthode de service dans le contrôleur
`ShowPersonneController`

```
@GetMapping("/showPersonne/{nom}")
public String showPersonne(@PathVariable String nom, Model
   model) {

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

Spring Security & Spring MVC

Appelons la nouvelle méthode de service dans le contrôleur
`ShowPersonneController`

```
@GetMapping("/showPersonne/{nom}")
public String showPersonne(@PathVariable String nom, Model
   model) {

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


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

Spring Security & Spring MVC

Pour la déconnexion

- Pas besoin de créer un contrôleur
- Pas besoin de vider la session....

Spring Security & Spring MVC

Pour la déconnexion

- Pas besoin de créer un contrôleur
- Pas 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>
```