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 Boot

But de la sécurité

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

Spring Security & Spring Boot

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 Boot

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 Boot

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 Boot

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-mvc-security` dans `Name`,
  - `com.example` dans `Group`,
  - `spring-boot-mvc-security` 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 & Spring Boot

Pour la suite, nous récupérerons les classes/interfaces suivantes du projet `spring-boot-intro`

- l'entité `Personne` de `com.example.demo.entities`
- l'interface DAO `PersonneRepository` de `com.example.demo.repositories`
- le contrôleur `PersonneController` à définir dans `com.example.demo.controllers`

Spring Boot & REST

L'entité `Personne`

```
package com.example.demo.entities;

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 Integer id;
    @NonNull
    private String nom;
    @NonNull
    private String prenom;
}
```

Spring Boot & REST

L'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, Integer> {

}
```

Spring Security & Spring Boot

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

```
# spring mvc properties
spring.mvc.view.prefix=/views/
spring.mvc.view.suffix=.jsp

# server properties
server.servlet.context-path=/security

# data
spring.datasource.url=jdbc:mysql://localhost:3306/spring?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
```

Spring Security & Spring Boot

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

```
@SpringBootApplication
public class SpringBootMvcSecurityApplication implements ApplicationRunner {

    @Autowired
    private PersonneRepository personneRepository;

    public static void main(String[] args) {
        SpringApplication.run(SpringBootMvcSecurityApplication.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 & Spring Boot

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 & Spring Boot

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 & Spring Boot

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 & Spring Boot

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 & Spring Boot

Créer une classe `SecurityConfig` pour configurer la sécurité de l'application

```
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 & Spring Boot

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()
        );
    }
}
```

Spring Security & Spring Boot

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 & Spring Boot

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 & Spring Boot

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 & Spring Boot

Pour tester

- Allez à `http://localhost:8080/` 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 & Spring Boot

Questions

Comment

1. utiliser une page d'authentification personnalisée ?
2. rediriger vers la page d'accueil après authentification réussie ?
3. autoriser/interdire certaines routes ?

Spring Security & Spring Boot

Questions

Comment

1. utiliser une page d'authentification personnalisée ?
2. rediriger vers la page d'accueil après authentification réussie ?
3. autoriser/interdire certaines routes ?

Réponse

ajouter un `bean` pour configurer le filtre.

Spring Security & Spring Boot

Ajoutons un deuxième `bean` pour configurer le filtre

```
@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
                .authorizeHttpRequests((auth) -> auth
                        .dispatcherTypeMatchers(DispatcherType.FORWARD).permitAll()
                        .requestMatchers("/login").permitAll()
                        .requestMatchers("/showPersonnes").hasRole("ADMIN")
                        .anyRequest().authenticated()
                )
                .formLogin((form) -> form
                        .loginPage("/login")
                        .defaultSuccessUrl("/home", true)
                        .permitAll()
                )
                .logout((logout) -> logout.permitAll())
                .csrf(c -> c.disable());
        return http.build();
}
```

Spring Security & Spring Boot

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.
- `dispatcherTypeMatchers(DispatcherType.FORWARD))` : pour autoriser les forwards internes (vers `/webapp/views/login.jsp`)

Spring Security & Spring Boot

Créons la classe `MvcConfig` suivante

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

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class MvcConfig implements WebMvcConfigurer {

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

}
```

Spring Security & Spring Boot

Créons la classe `MvcConfig` suivante

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

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class MvcConfig implements WebMvcConfigurer {

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

}
```


Explication

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

Spring Security & Spring Boot

Contenu de `login.jsp`

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="jakarta.tags.core"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Authentification</title>
</head>
<body>
        <h1>Page d'authentification</h1>
        <form action="${ pageContext.request.contextPath }/login" method="post">
                <div>
                        Nom d'utilisateur <input type="text" name="username">
                </div>
                <div>
                        Mot de passe <input type="password" name="password">
                </div>
                <div>
                        <button>Se connecter</button>
                </div>
        </form>
        <p>
                <c:if test="${ param.error ne null }"> Identifiants incorrects</c:if>
        </p>
</body>
</html>
```

Spring Security & Spring Boot

Tester

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

Spring Security & Spring Boot

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

Spring Security & Spring Boot

Commençons par supprimer la méthode `InMemoryUserDetailsManager`

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

Spring Security & Spring Boot

Injectons `UserDetailsService`

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private UserDetailsService userDetailsService;

}
```

Spring Security & Spring Boot

Injectons `UserDetailsService`

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private 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 Boot

Définissons un bean pour l'encodage de mots de passe (on les cryptera dans une prochaine section)

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

import org.springframework.beans.factory.annotation.Autowired;
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.UserDetailsService;
import org.springframework.security.crypto.password.NoOpPasswordEncoder;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private UserDetailsService userDetailsService;

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

Spring Boot

Contenu de l'entité `Role`

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Role {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        Long id;
        @NonNull
        String titre;
}
```

Contenu de l'entité User

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class User {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        Long num;
        @NonNull
        String username;
        @NonNull
        String password;
        @NonNull
        @ManyToMany(cascade = { CascadeType.PERSIST, CascadeType.REMOVE
            }, fetch = FetchType.EAGER)
        List<Role> roles;
}
```

`fetch = FetchType.EAGER` : les `Role` seront chargés au même temps que les `User`.

Spring Boot

Contenu de `UserRepository`

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

Spring Boot

Contenu de `UserRepository`

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


La méthode `findByUsername` sera utilisée plus tard.

Spring Boot

Contenu de `UserRepository`

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

La méthode `findByUsername` sera utilisée plus tard.

Contenu de `RoleRepository`

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

}
```

Spring Boot

Créons une classe qui implémente l'interface `UserDetailsService`

```
package com.example.demo.security;

import org.springframework.stereotype.Service;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

}
```


- Cette classe implémente l'interface `UserDetailsService` et doit donc implémenter la méthode `loadUserByUserName()` qui retourne un objet de type `UserDetails` (interface).
- L'annotation Service nous permettra d'utiliser cette classe en faisant une injection de dépendance.

Spring Boot

Contenu de la classe qui implémente `UserDetailsService`

```
package com.example.demo.security;

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 com.example.demo.dao.UserRepository;
import com.example.demo.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);
    }
  }
}
```

Spring Boot

Créons une classe qui implémente l'interface `UserDetails`

```
package com.example.demo.security;

public class UserDetailsImpl  implements UserDetails {

}
```


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

Spring Boot

Créer une classe qui implémente l'interface `UserDetails`

```
package com.example.demo.security;

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 com.example.demo.model.Role;
import com.example.demo.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 Boot

`UserDetailsImpl` **(la suite)**

```
  @Override
  public String getUsername() {
    return user.getUsername();
  }
  @Override
  public String getPassword() {
    return user.getPassword();
  }
}
```

Spring Security & Spring Boot

Modifions la classe de démarrage pour ajouter des nouveaux utilisateurs avec un mot de passe

```
@SpringBootApplication
public class SpringBootMvcSecurityApplication {

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

  @Bean
  CommandLineRunner start(
      PersonneRepository personneRep, UserRepository userRep, PasswordEncoder encoder) {
    return args -> {
      personneRep.save(new Personne("Wick", "John"));
      personneRep.save(new Personne("Dalton", "Jack"));
      userRep.save(new User("user", "user", List.of(new Role("USER"))));
      userRep.save(new User("admin", "admin", List.of(new Role("ADMIN"))));
    };
  }
}
```

Spring Security & Spring Boot

Pour crypter les mots de passe avec BCrypt, on commence par remplacer le bean `NoOpPasswordEncoder` **par** `PasswordEncoder`

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

Spring Security & Spring Boot

Modifions la classe de démarrage pour ajouter des nouveaux utilisateurs avec un mot de passe crypté

```
@SpringBootApplication
public class SpringBootMvcSecurityApplication {

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

  @Bean
  CommandLineRunner start(
      PersonneRepository personneRep, UserRepository userRep, PasswordEncoder encoder) {
    return args -> {
      personneRep.save(new Personne("Wick", "John"));
      personneRep.save(new Personne("Dalton", "Jack"));
      userRep.save(new User("user", encoder.encode("user"), List.of(new Role("USER"))));
      userRep.save(new User("admin", encoder.encode("admin"), List.of(new Role("ADMIN"))));
    };
  }
}
```

Spring Security & Spring Boot

Remarque

Lancez le projet et allez vérifier dans la base de données que les deux mots de passe ajoutés sont cryptés.

Spring Security & Spring Boot

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 Boot

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 Boot

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 Boot

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

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

Spring Security & Spring Boot

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 Boot

Le contrôleur `PersonneController`

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

    @Autowired
    private PersonneRepository personneRepository;
```

Spring Security & Spring Boot

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 Boot

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 Boot

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 Boot

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 Boot

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 Boot

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 Boot

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 Boot

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 Boot

Pour exiger deux rôles (ou plus)

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

Spring Security & Spring Boot

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 Boot

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 Boot

`@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 Boot

`@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 Boot

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 Boot

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 Boot

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 Boot

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 Boot

Pour la déconnexion

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

Spring Security & Spring Boot

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>
```