# **Spring Boot** et **Spring Security** (DB)

**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


1. Introduction
2. Exemple
   - `SecurityConfig`
   - `User` et `Role`
   - `UserDetailsService`
   - `UserDetails`
3. Utilisation de BCrypt

# Spring Security

## But de la sécurité

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

# Spring Security

## But de la sécurité

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

## Deux questions

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

# Spring Security

## Ce qu'il nous faut

Spring Security

# Spring Security

## Ce qu'il nous faut

Spring Security

## Deux systèmes d'authentification

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

# Spring Security

## Authentification **Stateful**

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

## Authentification **Stateless**

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

# Spring Security

**Remarque**

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

# Spring Security

## Remarque

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



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

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

# Spring Security

## Quelles solutions pour les attaques **CSRF** ?

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

# Spring Security

## Deux possibilité pour le stockage des utilisateurs

- En mémoire (InMemory)
- En base de données

# Spring Security

## Deux possibilité pour le stockage des utilisateurs

- En mémoire (InMemory)
- En base de données

## Dans ce chapitre

- On utilisera un système d'authentification **Stateful**
- Les utilisateurs seront stockées en base de données

# Spring Security

## Création de projet Spring Boot

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

# Spring Security

## Explication

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

# Spring Security

## Explication

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

## Pour la suite, nous considérons

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

# Spring Boot & REST

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

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

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

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

# Spring Boot & REST

**Préparons notre interface DAO** `PersonneRepository`

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

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

import com.example.demo.model.Personne;

public interface PersonneRepository extends JpaRepository<Personne,
   Long> {

}
```

## Créons le contrôleur REST suivant

```
@CrossOrigin
@RestController
public class PersonneRestController {

  @Autowired
  private PersonneRepository personneRepository;

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

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

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

# Spring Security

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

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



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

# Spring Security

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

```
@SpringBootApplication
public class SpringBootSecurityInDbApplication implements ApplicationRunner {

    @Autowired
    private PersonneRepository personneRepository;

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

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

# Spring Security

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

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

# Spring Security

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

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

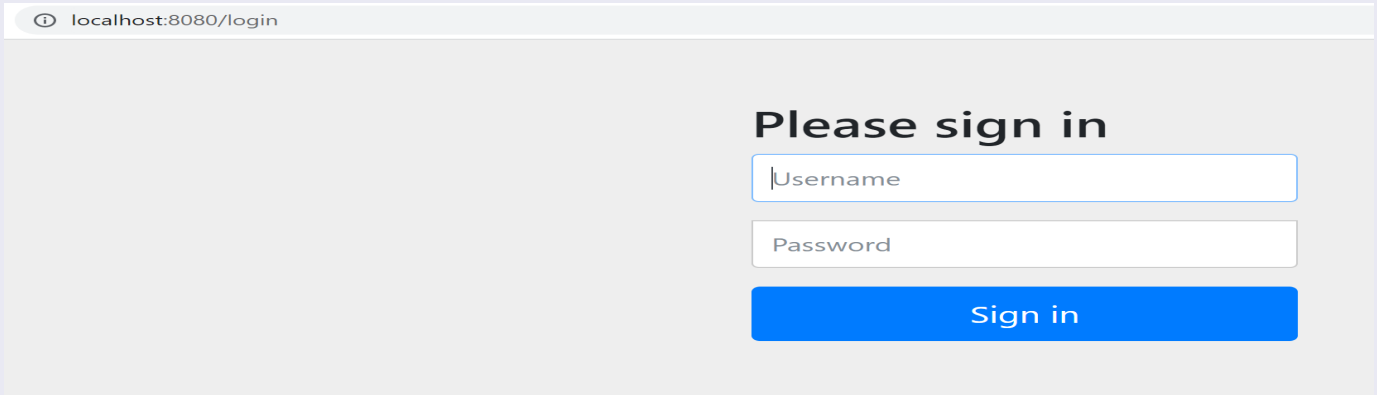
**Remarque**

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

# Spring Security

## Pour tester

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




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

# Spring Security

## Étapes

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

# Spring Security

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

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

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

@Configuration
@EnableWebSecurity
public class SecurityConfig {

}
```

# Spring Security

**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

**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
    public static NoOpPasswordEncoder passwordEncoder() {
        return (NoOpPasswordEncoder) NoOpPasswordEncoder.getInstance();
    }
}
```

# Spring Security

**Et encore deux beans un pour les utilisateurs et leurs mots de passe et un pour configurer le filtre**

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Autowired
    private UserDetailsService userDetailsService;

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

    @Bean
    AuthenticationManager authenticationManager() {
        DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
        authProvider.setUserDetailsService(userDetailsService);
        authProvider.setPasswordEncoder(passwordEncoder());
        return new ProviderManager(authProvider);
    }

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

# 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 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();
  }
}
```

# Spring Security

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

```
@SpringBootApplication
public class CoursSpringBootSecurityInDbApplication {

  public static void main(String[] args) {
    SpringApplication.run(CoursSpringBootSecurityInDbApplication.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

## Pour tester

Il faut aller à `http://localhost:8080/personnes`, s'authentifier avec les identifiants d'un utilisateur de la table `users`

# Spring Security

## Pour ajouter une personne

- utiliser **Postman** en précisant la méthode et l'**URL** `http://localhost:8080/personnes`
  - dans `Headers`, préciser la clé `Content-Type` et la valeur `application/json`
  - dans `Body`, cocher `raw` et sélectionner `JSON(application/json)`

# Spring Security

## Pour ajouter une personne

- utiliser **Postman** en précisant la méthode et l'**URL** `http://localhost:8080/personnes`
  - dans `Headers`, préciser la clé `Content-Type` et la valeur `application/json`
  - dans `Body`, cocher `raw` et sélectionner `JSON(application/json)`



**Exemple de valeurs à persister**

```
{
    "nom": "dalton",
    "prenom": "jack"
}
```

# Spring Security

**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

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

```
@SpringBootApplication
public class CoursSpringBootSecurityInDbApplication {

  public static void main(String[] args) {
    SpringApplication.run(CoursSpringBootSecurityInDbApplication.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

## Remarque

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