
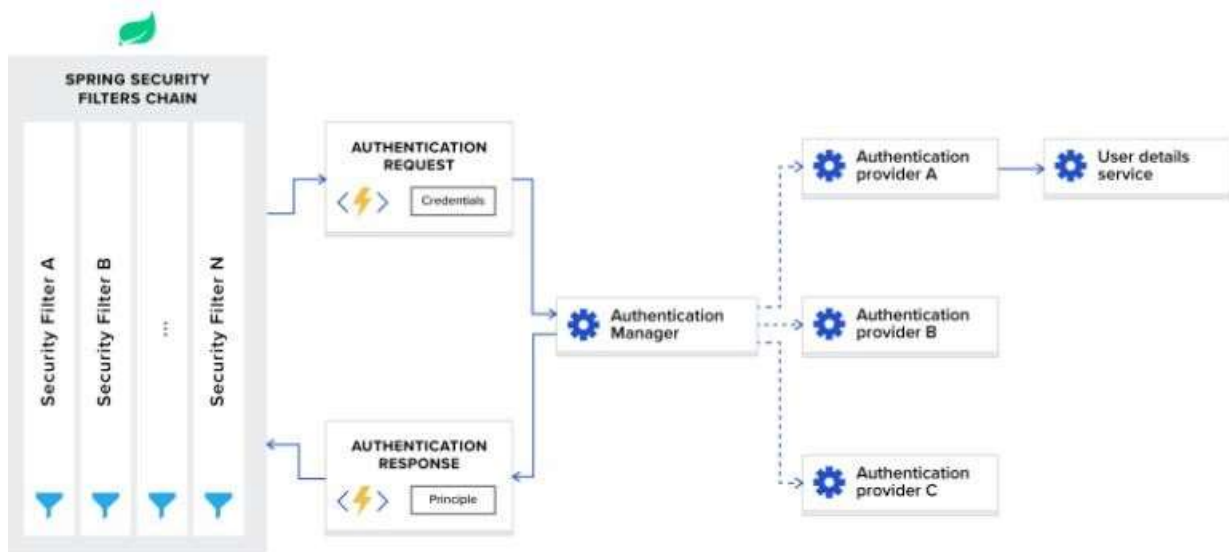
	<p align="center">Atelier N°7</p> <p align="center">Spring Security Framework (JPA & Authentication Provider)</p>	
---	--	---

Objectifs	Temps alloué	Outils
<ul style="list-style-type: none"> • Maitriser l'architecture du framework Spring Security • Savoir s'authentifier en utilisant JPA/MySQL • Savoir intégrer les filtres 	6 Heures	<ul style="list-style-type: none"> • STS 4 • MySQL Server

I. Architecture

Spring Security est un Framework de sécurité léger qui fournit une authentification et un support d'autorisation afin de sécuriser les applications Spring.



II. Travail demandé

Suivez les étapes suivantes pour mettre en place le framework Spring Security.

Structure du projet:

Partie 1 : Authentification des utilisateurs

1. Création de l'entité User

User.java, cette classe implémente l'interface **UserDetails**.

L'interface fournit des informations sur l'utilisateur principal. Les implémentations ne sont pas utilisées directement par Spring Security à des fins de sécurité. Ils stockent simplement les informations utilisateur qui sont ensuite encapsulées dans des objets d'authentification. Cela permet de stocker des informations non liées à la sécurité (telles que les adresses e-mail etc.) dans un emplacement approprié.

```
@Entity
@Table(name = "USER")
public class User implements Serializable, UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer userId;
    private String username;
    private String password;

    public Integer getUserId() {
        return userId;
    }

    public void setUserId(Integer userId) {
        this.userId = userId;
    }

    public String getUsername() {
        return username;
    }

    @Override
    public boolean isAccountNonExpired() {
        return false;
    }

    @Override
    public boolean isAccountNonLocked() {
        return false;
    }

    @Override
```

```

    public boolean isCredentialsNonExpired() {
        return false;
    }

    @Override
    public boolean isEnabled() {
        return false;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return null;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}

```

2. Implémentation du Repository UserRepository

La classe **UserRepository.java**, hérite de la classe **JpaRepository**.

La fonction **findUserWithName(String name)** permet de récupérer un User à partir de son nom d'utilisateur.

```

public interface UserRepository extends JpaRepository<User, String> {
    @Query(" select u from User u where u.username = ?1")
    Optional<User> findUserWithName(String username);
}

```

3. Implémentation du Service UserService

La classe **UserService.java**, implémente l'interface **UserDetailsService**.

L'interface **UserDetailsService** est utilisée pour récupérer les données liées à l'utilisateur. Il a une méthode nommée **loadUserByUsername** qui trouve une entité utilisateur basée sur le nom d'utilisateur et peut être substituée pour personnaliser le processus de recherche de l'utilisateur. Il est utilisé par **DaoAuthenticationProvider** pour charger des détails sur l'utilisateur lors de l'authentification.

```

@Service
@Slf4j
public class UserService implements UserDetailsService {

```

```

    private final UserRepository userRepository;

    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        Objects.requireNonNull(username);
        User user = userRepository.findUserWithName(username)
            .orElseThrow(() -> new UsernameNotFoundException("User not found"));
        return user;
    }
}

```

4. Implémentation de La classe AppAuthenticationProvider

Spring Security fournit une variété d'options pour effectuer l'authentification. Toutes ces options suivent un contrat simple. Une demande d'authentification est traitée par un AuthenticationProvider et un objet entièrement authentifié avec des informations d'identification complètes est renvoyé.

L'implémentation standard et la plus courante est le DaoAuthenticationProvider – qui récupère les détails de l'utilisateur à partir d'un simple DAO utilisateur en lecture seule – le UserDetailsService. Ce service de détails de l'utilisateur a uniquement accès au nom d'utilisateur afin de récupérer l'entité utilisateur.

```

public class AppAuthProvider extends DaoAuthenticationProvider {
    @Autowired
    UserService userDetailsService;

    @Override
    public Authentication authenticate(Authentication authentication) throws
AuthenticationException {
        UsernamePasswordAuthenticationToken auth = (UsernamePasswordAuthenticationToken)
authentication;
        String name = auth.getName();
        String password = auth.getCredentials().toString();
        UserDetails user = userDetailsService.loadUserByUsername(name);
        if (user == null) {
            throw new BadCredentialsException("Username/Password does not match for "
+ auth.getPrincipal());
        }
        return new UsernamePasswordAuthenticationToken(user, null,
user.getAuthorities());
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return true;
    }
}

```

```
}
```

5. Implémentation de la classe de configuration SecurityConfig

Pour personnaliser la configuration nous allons créer une classe qui hérite de **WebSecurityConfigurerAdapter**. Cette classe doit avoir les annotations **@EnableWebSecurity** et **@Configuration**. Les classes de configuration sont scannées au démarrage de l'application.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    public static final String AUTHORITIES_CLAIM_NAME = "roles";
    @Autowired
    UserService userDetailsService;
    @Autowired
    private AccessDeniedHandler accessDeniedHandler;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService);
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
        .authenticationProvider(getProvider())

        .formLogin()
        .loginProcessingUrl("/login")
        .successHandler(new AuthenticationLoginSuccessHandler())
        .failureHandler(new SimpleUrlAuthenticationFailureHandler())
        .and()
        .logout()
        .logoutUrl("/logout")
        .logoutSuccessHandler(new AuthenticationLogoutSuccessHandler())
        .invalidateHttpSession(true)
        .and()
        .authorizeRequests()
        .antMatchers("/login").permitAll()
        .antMatchers("/logout").permitAll()
        .anyRequest().authenticated();
    }

    private class AuthenticationLoginSuccessHandler extends
    SimpleUrlAuthenticationSuccessHandler {

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response,
    Authentication authentication) throws IOException, ServletException {
```

```

        response.setStatus(HttpServletResponse.SC_OK);
    }
}

private class AuthentificationLogoutSuccessHandler extends SimpleUrlLogoutSuccessHandler {
@Override
public void onLogoutSuccess(HttpServletRequest request, HttpServletResponse response,
Authentication) throws IOException, ServletException {
    response.setStatus(HttpServletResponse.SC_OK);
}

}

@Bean
public AuthenticationProvider getProvider() {
    AppAuthProvider provider = new AppAuthProvider();
    provider.setUserDetailsService(userDetailsService);
    return provider;
}

@Bean
public AccessDeniedHandler accessDeniedHandler(){
    return new AccessDeniedHandlerImpl();
}

}

```

6. Tester l'authentification : pour cela ajouter un compte admin à partir de l'application principale.

Partie2 : Personnaliser l'interface Login et Logout

1. Modifier la méthode configure comme suit :

```

http.formLogin().loginPage("/login");
...
http.authorizeRequests().antMatchers("/login").permitAll();
//pour faire fonctionner Bootstrap
http.authorizeRequests().antMatchers("/webjars/**").permitAll();
http.authorizeRequests().anyRequest().authenticated();

```

2. Ajouter la méthode login() au contrôleur SecurityController :

```

@GetMapping("/login")
public String login()
{
    return "login";
}

```

3. Créer la page login.html :

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org"
xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
layout:decorator="template">
<head>

```

```

<meta charset="utf-8">
<title>Connexion</title>
</head>
<body>
<div layout:fragment="Mycontent">
<div class="container mt-5">
<div class="col-md-6 offset-md-3">
<div class="card">
<div class="card-header">Connexion</div>
<div class="card-body">
<form th:action="@{login}" method="post">
<div class="form-group">
<label class="control-label" >Utilisateur :</label>
<input type="text" name="username">
</div>
<div class="form-group">
<label class="control-label" >Mot de passe :</label>
<input type="password" name="password" >
</div>
<button class="btn btn-primary">se connecter</button>
<button class="btn btn-primary">Ajouter un Compte</button>

</form>
</div>
</div>
</div>
</div>
</div>
</body>
</html>

```

4. Personnaliser le Logout

Ajouter la méthode logout() au contrôleur :

```

@GetMapping("/logout")
public String logout(HttpServletRequest request) throws ServletException
{
    request.logout();
    return "redirect:/login";
}

```

Partie3 : gestion des authorizations

1. Création de l'entité Role et de son repository

```

@Entity
public class Role {
    @Id
    @Column(name = "role_id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String name;
    public Integer getId() {
        return id;
    }
    public Role(String name) {
        super();
    }
}

```

```

        this.name = name;
    }
}

```

Dans le package DAO, définir `RoleRepository`.

2. Mettre à jour l'entité User

Afin d'ajouter la relation plusieurs à plusieurs entre l'entité User et l'entité Role, ajouter dans l'entité User :

```

@ManyToMany(cascade = CascadeType.MERGE, fetch = FetchType.EAGER)
@JoinTable(name = "users_roles", joinColumns = @JoinColumn(name = "user_id"),
inverseJoinColumns = @JoinColumn(name = "role_id"))

```

```
private Set<Role> roles = new HashSet<>();
```

Redéfinir la méthode `getAuthorities()` comme suit:

```

Set<Role> roles = this.getRoles();
List<SimpleGrantedAuthority> authorities = new ArrayList<>();

for (Role role : roles) {
    authorities.add(new SimpleGrantedAuthority(role.getName()));
}

return authorities;

```

3. Ajouter un utilisateur (SignUp):

Dans la classe UserService ajouter la méthode suivante :

```

public User saveUser(String username, String password, String confirmedPassword) {
    User appUser = new User();
    if (userRepository.findUserWithName(username).isPresent() == true)
        throw new RuntimeException("User already exists");
    if (!password.equals(confirmedPassword))
        throw new RuntimeException("Please confirm your password");
    appUser.setUsername(username);

    Set<Role> roles = new HashSet<Role>();
    Role r = new Role("ROLE_USER");
    roleRepository.save(r);
    roles.add(r);
    appUser.setRoles(roles);

    appUser.setPassword(bCryptPasswordEncoder.encode(password));
    userRepository.save(appUser);
    return appUser;
}

```


Dans la classe **UserController** ajouter une méthode qui utilise la méthode **saveUser** de la classe **UserService** pour ajouter un nouveau utilisateur en appuyant sur le bouton **Ajouter un Compte** de la page **login.html** utiliser la classe **UserForm** pour passer les paramètres dans le corps de la requête.

```
class UserForm{
    private String username;
    private String password;
    private String confirmedPassword;
}
```

Partie 2 : Spring Security et la Défense en profondeur

Jusqu'à présent, la sécurité que nous avons paramétrée ne concerne que les pages web de l'application et leur accès, ce qui est tout à fait suffisant dans la plupart des cas. Pour renforcer encore la sécurité, il existe la notion de "defense in depth", que je propose audacieusement de traduire par "défense en profondeur" : il s'agit de contrôler l'accès aux méthodes que l'on peut trouver dans les **@Controllers**, les **@Components**, les **@Services**, les **@Repositories**, et autres beans Spring. Cette approche est mise en place par le biais d'annotations affectant les méthodes publiques des beans. Pour rendre possible cette sécurité supplémentaire, il est nécessaire de l'autoriser explicitement dans la classe annotée **@Configuration** que nous avons survolée dans la partie précédente, en ajoutant l'annotation **@EnableGlobalMethodSecurity**.

Dans **pom.xml** ajouter cette dépendance

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

Pour activer la défense en profondeur ajouter dans **SecurityConfig** :

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

- **prePostEnabled** : permet d'utiliser les annotations **@PreAuthorize** et **@PostAuthorize**.

- **securedEnabled** : permet d'utiliser l'annotation *@Secured*.
- **jsr250Enabled** : permet d'utiliser l'annotation *@RolesAllowed*.

Pour plus d'informations :

<https://docs.spring.io/spring-security/site/docs/5.2.11.RELEASE/reference/html/authorization.html>