Spring security

Spring Security est un framework permettant d'ajouter aux applications Spring authentification et contrôle d'accès.

Spring security ajoute au Dispatcher servlet de Spring MVC un ensemble de filtres (servlet filters) qualifié de **filter chain**.

Intégration

Ajouter la dépendance suivante à **pom.xml** :

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

Authentification par défaut

L'intégration de Spring security met en place une authentification basique par défaut, avec un utilisateur de nom **user**, ayant le rôle **USER**,
dont le password s'affiche dans la console de démarrage de Spring, utilisable en mode développement :

```
Using generated security password: 90990698-6bb0-4953-8151-fe8010f2547a

This generated password is for development use only. Your security configuration must be updated before running your application in production.
```

Il est possible de modifier l'utilisateur par défaut dans **application.properties** :

```
spring.security.user.name=boris
spring.security.user.password=boris-password
spring.security.user.roles=manager
```

Configuration

La configuration se fait au travers d'une classe de configuration activant la sécurité :

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http.authorizeHttpRequests(req -> req
                .requestMatchers("/", "/css/**", "/js/**", "/img/**",
"/register/**").permitAll()
                .anyRequest().authenticated()
        );
        return http.build();
    }
}
```

Les chaînes passées à **requestMatchers** sont interprétées comme des patterns de chemin. Il n'est plus nécessaire d'utiliser **AntPathRequestMatcher** (déprécié dans les versions récentes).

La protection CSRF est **activée par défaut**. Ne la désactivez pas (**csrf.disable()**) sans raison valable, voir la section Mise en place CSRF.

Authentification depuis une BDD

Entity

Créer ou reprendre votre entity **User** pour gérer les utilisateurs et leur rôle.

Elle devra implémenter l'interface **UserDetails** :

```
@Entity
public class User implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(length = 50, unique = true)
    private String login;

    @Column(length = 255)
    private String password;

    @Column(length = 255)
```

```
    private String email;

    @ManyToOne
    private Role role;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        List<GrantedAuthority> authorities = new ArrayList<>();
        if (role != null) {
            authorities.add(new SimpleGrantedAuthority(role.getName()));
        }
        return authorities;
    }

    @Override
    public String getUsername() {
        return login;
    }

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

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

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

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

    // getters / setters ...
}
```

Créer le repository associé et ajouter une méthode permettant de rechercher un User par son login ou son email :

```
public interface UserRepository extends CrudRepository<User, Long> {
    Optional<User> findByLoginOrEmail(String login, String email);
}
```

UserDetailsService

Créer un Service implémentant **UserDetailsService**.

Ce service retourne depuis la base de données un utilisateur recherché par son login ou son email, sous la forme d'une instance de **UserDetails** :

```
@Service
public class UserService implements UserDetailsService {

    private final UserRepository uRepo;
    private final RoleRepository rRepo;
    private final PasswordEncoder pEncoder;

    public UserService(UserRepository uRepo, RoleRepository rRepo, PasswordEncoder
pEncoder) {
        this.uRepo = uRepo;
        this.rRepo = rRepo;
        this.pEncoder = pEncoder;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        return uRepo.findByLoginOrEmail(username, username)
                .orElseThrow(() -> new UsernameNotFoundException("Utilisateur
inconnu"));
    }

    public void encodePassword(User user) {
        user.setPassword(pEncoder.encode(user.getPassword()));
    }
}
```

Préférez l'**injection par constructeur** à **@Autowired** sur les champs : elle facilite les tests et évite les dépendances cachées.

Password encoder

Déclarer dans la configuration un **PasswordEncoder** (hashage Bcrypt) :

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        ...
        return http.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
```

```
        return new BCryptPasswordEncoder();
    }
}
```

Spring Boot détecte automatiquement un bean **UserDetailsService** (ici **UserService**, déclaré via **@Service**) et un bean **PasswordEncoder**, et configure l'authentification en conséquence.
Il n'est donc **pas nécessaire** de déclarer manuellement un **DaoAuthenticationProvider**, ni de redéclarer **UserService** avec **@Bean** ou **@Primary**.

Attention aux **dépendances circulaires** : si **UserService** injecte **PasswordEncoder** et que le bean **PasswordEncoder** est défini dans une classe qui injecte **UserService**, Spring échouera au démarrage. Ici, **SecurityConfig** ne dépend pas de **UserService**, donc tout va bien.

Création d'utilisateur

Ne pas oublier de hasher le password du User avant création :

```
@Service
public class UserService implements UserDetailsService {

    ...

    public User createUser(String login, String password) {
        User user = new User();
        user.setLogin(login);
        user.setPassword(password);
        encodePassword(user);
        user.setRole(getRole("ROLE_USER"));
        return uRepo.save(user);
    }
}
```

Récupération de l'utilisateur connecté

Récupération de l'utilisateur connecté en tant que **ModelAttribute** pour utilisation dans les vues :

```
@ControllerAdvice
public class MainAdvice {
    @ModelAttribute("activeUser")
    public User activeUser(Authentication auth) {
        return (auth == null) ? null : (User) auth.getPrincipal();
    }
}
```

Login form personnalisée

Créer un template pour le formulaire de connexion :

```
<form method="post" action="./login">
    <input type="text" name="username" placeholder="E-mail address or username">
    <input type="password" name="password" placeholder="Password">
    <!-- Token CSRF (voir section CSRF) -->
    <input type="hidden" name="{{_csrf.parameterName}}" value="{{_csrf.token}}"/>
    <button type="submit">Login</button>
</form>
```

Ajouter une route pour le **login** :

```
@Configuration
public class WebConfiguration implements WebMvcConfigurer {

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

Modifier la configuration dans **SecurityConfig** :

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
                ...
                .formLogin(form -> form
                        .loginPage("/login")
                        .defaultSuccessUrl("/", true)
                        .permitAll()
                );
        return http.build();
    }
}
```

Connexion à la console H2

Pour autoriser l'accès à la console H2 (**uniquement en développement**) :

1. Ignorer la protection CSRF pour la console
2. Autoriser la route vers la console **/h2-console**
3. Autoriser l'utilisation des iframes pour l'origine locale

```
@Configuration
```

```
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
                .csrf(csrf ->
csrf.ignoringRequestMatchers(PathRequest.toH2Console()))   // (1)
                .authorizeHttpRequests(req -> req
                        .requestMatchers(PathRequest.toH2Console()).permitAll()
// (2)
                        .requestMatchers("/", "/css/**", "/js/**").permitAll()
                        .anyRequest().authenticated()
                )
                .headers(headers -> headers
.frameOptions(HeadersConfigurer.FrameOptionsConfig::sameOrigin)  // (3)
                );
        return http.build();
    }
}
```

La console H2 ne doit **jamais** être exposée en production. Idéalement, placez cette configuration dans un profil Spring dédié (**@Profile("dev")**).

Rôles et authorities

Spring Security distingue les authorities, accessibles par une simple chaîne (encapsulée ensuite dans un objet de type **GrantedAuthority**) : **USER, ADMIN…**
et le rôle équivalent, authority préfixée par **ROLE_** : **ROLE_USER, ROLE_ADMIN…**.

Il est possible de faire référence à l'un ou à l'autre indifféremment, à condition de respecter le préfixe :

- **hasRole("ADMIN")** vérifie l'authority **ROLE_ADMIN** (le préfixe est ajouté automatiquement)
- **hasAuthority("ROLE_ADMIN")** vérifie la même chose, mais le préfixe doit être écrit explicitement

Le rôle des utilisateurs pourra donc être stocké en base de données, sous la forme d'une chaîne, ou d'une liste de chaînes.

Hiérarchie des authorities

Il est possible de définir les authorities de manière hiérarchique :

```
    @Bean
    static RoleHierarchy roleHierarchy() {
        return RoleHierarchyImpl.fromHierarchy("""
                ROLE_ADMIN > ROLE_REDACTOR
                ROLE_REDACTOR > ROLE_USER
                """);
    }
```

```
    @Bean
    static MethodSecurityExpressionHandler
methodSecurityExpressionHandler(RoleHierarchy roleHierarchy) {
        DefaultMethodSecurityExpressionHandler expressionHandler = new
DefaultMethodSecurityExpressionHandler();
        expressionHandler.setRoleHierarchy(roleHierarchy);
        return expressionHandler;
    }
```

Les utilisateurs peuvent dans ce cas n'avoir qu'une seule authority, qui leur en confère plusieurs grâce à la hiérarchie des rôles.

Les méthodes **static** sont recommandées pour ces beans, afin qu'ils soient instanciés tôt dans le cycle de vie du contexte Spring.
La méthode **RoleHierarchyImpl.fromHierarchy(...)** est disponible à partir de Spring Security 6.3 ; pour les versions antérieures, utiliser **RoleHierarchyImpl.withDefaultRolePrefix()** ou **setHierarchy(...)**.

Configuration des rôles

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http.authorizeHttpRequests(req -> req
                .requestMatchers("/css/**", "/assets/**", "/login/**").permitAll()
// (1)
                .requestMatchers("/admin/**").hasRole("ADMIN")
// (2)
                .requestMatchers("/user/**").hasAuthority("ROLE_USER")
// (3)
                .requestMatchers("/staff/**").hasAnyAuthority("ROLE_USER",
"ROLE_ADMIN", "ROLE_MANAGER") // (4)
                .anyRequest().authenticated()
// (5)
        );
        return http.build();
    }
}
```

Méthode	Description
1 - **permitAll**	Pas de sécurisation sur les urls commençant par **/css**, **/assets**, **/login**.
2 - **hasRole**	Les utilisateurs ayant le rôle **ADMIN** (authority **ROLE_ADMIN**) peuvent accéder aux urls commençant par **/admin**.
3 - **hasAuthority**	Les utilisateurs ayant l'authority **ROLE_USER** peuvent accéder aux urls commençant par **/user**.
4 - **hasAnyAuthority**	Les utilisateurs ayant l'une des authorities **ROLE_USER**, **ROLE_ADMIN** ou **ROLE_MANAGER** peuvent accéder aux urls commençant par **/staff**.
5 - **authenticated**	Pour toutes les autres URLs, il faut être authentifié.

Ordre des règles : elles sont évaluées dans l'ordre de déclaration, la première qui correspond s'applique. Placez toujours les règles les plus spécifiques en premier, et **anyRequest()** en dernier.

Defense in depth

Spring security permet de définir des autorisations sur les méthodes sollicitées, au-delà des autorisations qu'il est possible de définir sur les URLs.

Activation

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(
    prePostEnabled = true,
    securedEnabled = true,
    jsr250Enabled = true
)
public class SecurityConfig {
    ...
}
```

Propriété	Défaut	Description
prePostEnabled	true	permet d'utiliser les annotations **@PreAuthorize** et **@PostAuthorize**.
securedEnabled	false	permet d'utiliser l'annotation **@Secured**.
jsr250Enabled	false	permet d'utiliser l'annotation **@RolesAllowed**.

Dans les versions récentes de Spring Boot, **jakarta.annotation.security.RolesAllowed** remplace l'ancien package **javax**.

@Secured

```
    @Secured("ROLE_ADMIN")
    public String adminOnly() {
        return "admin";
    }
```

@RolesAllowed

```
    @RolesAllowed({"ROLE_USER", "ROLE_MANAGER"})
    public String userOrManager() {
        return "userOrManager";
    }
```

@PreAuthorize

Permet d'agir avant l'exécution de la méthode :

- Avec usage des SpEL
- En testant éventuellement les paramètres passés.

```
@PreAuthorize("hasRole('VIEWER')")
public String getUsernameInUpperCase() {
    return getUserName().toUpperCase();
}
```

Comparaison du paramètre **username** avec le nom de l'utilisateur connecté :

```
@PreAuthorize("#username == authentication.principal.username")
public List<String> getRoles(String username) {
    ...
}
```

@PostAuthorize

Permet d'agir après l'exécution de la méthode :

- Avec usage des SpEL
- En testant éventuellement le retour.

```
@PostAuthorize("returnObject.username == authentication.principal.username")
public CustomUser loadUserDetail(String username) {
    return userRoleRepository.loadUserByUserName(username);
}
```

Voir la documentation officielle sur la sécurité des méthodes.

Mise en place CSRF

Objectif : éviter les attaques CSRF

Activation

La protection CSRF est **activée par défaut** dans Spring Security : il suffit de ne pas la désactiver.
Si votre configuration contient la ligne suivante, supprimez-la :

```
http.csrf(AbstractHttpConfigurer::disable)
```

Exception éventuelle pour la console H2 (uniquement en développement) :

```
http.csrf(csrf -> csrf.ignoringRequestMatchers(PathRequest.toH2Console()))
```

Intégration dans les vues

Dans **application.properties**, exposer les attributs de requête dans les vues (exemple pour Mustache) :

```
spring.mustache.servlet.expose-request-attributes=true
```

Utilisation dans un formulaire :

```
<form method="post" action="...">
    ...
    <input type="hidden"
           name="{{_csrf.parameterName}}"
           value="{{_csrf.token}}"/>
    ...
</form>
```

Avec **Thymeleaf**, le token CSRF est inséré automatiquement dans les formulaires utilisant **th:action**.

Attention à la route **/logout** : avec CSRF activé, elle doit être accédée uniquement via un **POST** incluant le token CSRF.

Voir Spring Cross Site Request Forgery

From:
http://slamwiki2.kobject.net/ - **SlamWiki 2.1**

Permanent link:
http://slamwiki2.kobject.net/web/framework/spring/security?rev=1793008381

Last update: **2026/10/26 10:53**