Spring Security

Introduction

Qu'est-ce que Spring Security ?

- Framework de sécurité puissant et hautement personnalisable pour les applications Java
- S'intègre facilement avec les projets basés sur Spring, tels que Spring Boot, Spring MVC et Spring Data, pour fournir une sécurité transparente dans tout l'écosystème Spring

Architecture de Spring Security

Organisation des dossiers

```
src/
├── main/
│   └── java/
│       └── com/
│           └── example/
│               ├── config/
│               │   └── SecurityConfig.java
│               ├── controller/
│               │   └── UserController.java
│               ├── model/
│               │   └── User.java
│               ├── repository/
│               │   └── UserRepository.java
│               ├── service/
│               │   └── UserService.java
│   └── resources/
│       └── static/
│       └── templates/
│       └── application.properties
└── test/
    └── java/
        └── com/
            └── example/
                ├── controller/
                │   └── UserControllerTest.java
                ├── repository/
                │   └── UserRepositoryTest.java
                ├── service/
                    └── UserServiceTest.java
```

Référentiel utilisateur

Qu'est-ce que c'est ?

- Aussi appelé "gestionnaire d'utilisateurs"
- Stocke et gère les informations relatives aux utilisateurs autorisés à accéder à l'application

Que contient-il ?

- Données d'identification et d'authentification des utilisateurs (nom d'utilisateur, mot de passe cryptés, rôles et autorisations)
- Ces informations sont utilisées par le système de sécurité pour vérifier l'identité des utilisateurs lors de la phase d'authentification et pour déterminer les autorisations appropriées lors de la phase d'autorisation

Comment Spring Security gère le référentiel utilisateur ?

- Le référentiel utilisateur est représenté par l'interface "UserDetailsService" qui est fournie par Spring Security
- Vous devez créer votre propre classe d'implémentation de UserDetailsService pour spécifier comment les détails de l'utilisateur (nom d'utilisateur, mot de passe crypté, rôles et autorisations associés) sont récupérés à partir de votre source de données
- Ces détails sont ensuite utilisés pour construire un objet "UserDetails", qui représente l'utilisateur dans le cadre de Spring Security.

L'implémentation "UserDetailsServiceImpl"

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    // Charge les détails de l'utilisateur en fonction du nom d'utilisateur
spécifié lors du processus d'authentification
    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByUsername(username); // Recherche un
utilisateur dans la base de données en utilisant le nom d'utilisateur spécifié
        if (user == null) {
            throw new UsernameNotFoundException("User not found");
        }
        return new CustomUserDetails(user);
    }
}
```

Dans le Repository

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    User findByUsername(String username); // Prend un nom d'utilisateur en
paramètre et renvoie un objet User
}
```

Les modes d'authentifications

A quoi servent-ils ?

Garantir que seules les personnes autorisées peuvent accéder aux ressources pour la :

- Sécurité de l'application
- Protection des données sensibles
- Gestion des autorisations
- Expérience utilisateur personnalisée
- Conformité réglementaire

Quels sont les modes ?

- Session
- JWT

Session

A quoi servent les sessions ?

- Maintenir un état entre les différentes requêtes HTTP d'un utilisateur
- Essentiel pour stocker des informations d'authentification et d'autorisation

Gestion automatique de la session

- Spring Security gère automatiquement la création et la gestion des sessions pour les utilisateurs authentifiés.
- Lorsqu'un utilisateur se connecte avec succès, une session lui est attribuée et un identifiant de session unique est généré.
- Cet identifiant de session est stocké, soit dans un cookie, soit dans l'URL, et est utilisé pour associer les requêtes ultérieures à la session correspondante.

Stockage des informations d'authentification

- Lorsqu'un utilisateur est authentifié, les informations d'authentification, telles que l'objet "Authentication" représentant l'utilisateur connecté, sont stockées dans la session.
- Cela permet de conserver l'état de l'authentification entre les différentes requêtes, ce qui facilite la vérification de l'identité de l'utilisateur lors des accès ultérieurs.

Configuration de la gestion de session

- Spring Security offre des mécanismes de configuration flexibles pour la gestion des sessions.
- Vous pouvez spécifier des paramètres tels que la durée de vie de la session, le mécanisme de stockage (par exemple, en mémoire, dans une base de données), la stratégie d'invalidation de la session, etc.
- Cela vous permet d'adapter la gestion des sessions en fonction des besoins de votre application.

Protection contre les attaques liées aux sessions

- Spring Security intègre des fonctionnalités de protection contre les attaques liées aux sessions (protection contre les attaques de fixation de session (session fixation attacks), invalidation de la session après la déconnexion de l'utilisateur, etc)
- Cela renforce la sécurité de votre application en garantissant que les sessions sont gérées de manière sûre et fiable

---

Utilisation de sessions pour l'autorisation

- Outre l'authentification, les sessions peuvent également être utilisées pour stocker des informations d'autorisation spécifiques à l'utilisateur connecté.
- Par exemple, vous pouvez stocker les rôles ou les autorisations de l'utilisateur dans la session et les utiliser pour prendre des décisions d'autorisation lors des accès ultérieurs.

Configurer la gestion des sessions

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
            .antMatchers("/admin/**").hasRole("ADMIN") // Les URL commençant par
"/admin" nécessitent le rôle "ADMIN"
            .antMatchers("/user/**").hasRole("USER") // Les URL commençant par
"/user" nécessitent le rôle "USER"
            .anyRequest().authenticated() // Toutes les autres URL nécessitent une
authentification
            .and()
            .formLogin() // Active la configuration de la connexion basée sur un
formulaire
            .and()
            .logout().logoutSuccessUrl("/login?logout") // Active la configuration
```

```
de la déconnexion
            .and()
            .exceptionHandling().accessDeniedPage("/403") // Active la gestion des
exceptions liées à la sécurité
            .and()
            .sessionManagement()
                .maximumSessions(1)
                    .sessionRegistry(sessionRegistry()) // Spécifie qu'un
utilisateur ne peut avoir qu'une seule session active à la fois
                    .invalidSessionUrl("/login") // Redirige l'utilisateur vers
l'URL "/login" si une session invalide est détectée
                    .expiredUrl("/login?expired"); // Redirige vers "/login?
expired" si la session a expiré
}

    // Publier les événements liés à la session
    @Bean
    public HttpSessionEventPublisher httpSessionEventPublisher() {
        return new HttpSessionEventPublisher();
    }
```

JWT

Qu'est-ce que JWT ?

- JWT (JSON Web Token) est un standard ouvert (RFC 7519) utilisé pour l'échange sécurisé de données entre parties, généralement utilisé pour l'authentification et l'autorisation dans les applications web.
- Avec Spring Security, JWT est pris en charge grâce à diverses bibliothèques et fonctionnalités intégrées.

Structure du JWT

Chaîne de caractères encodée au format JSON qui est composée de trois parties :

- l'en-tête (header) : métadonnées sur le token
- la charge utile (payload) : informations spécifiques à l'application (par exemple, le nom d'utilisateur, les rôles)
- la signature (signature) : permet de vérifier l'intégrité du JWT

Génération et validation du JWT

- Avec Spring Security, vous pouvez générer un JWT lorsqu'un utilisateur s'authentifie avec succès.
- Les informations d'identification de l'utilisateur, telles que le nom d'utilisateur et les rôles, sont ajoutées à la charge utile du JWT.
- Lorsque le JWT est envoyé avec les requêtes ultérieures, Spring Security peut le valider en vérifiant la signature et extraire les informations d'authentification et d'autorisation nécessaires.

Filtre JWT dans Spring Security

- Pour intégrer JWT avec Spring Security, vous pouvez utiliser le filtre JwtAuthenticationFilter fourni par Spring Security.
- Ce filtre intercepte les requêtes, extrait le JWT du header ou du corps de la requête, et le valide en utilisant la clé secrète configurée.
- Il vérifie également si l'utilisateur a les autorisations nécessaires pour accéder à la ressource demandée.

Configuration de Spring Security pour JWT

Pour utiliser JWT avec Spring Security, vous devez configurer certains éléments :

- définition d'un TokenProvider qui gère la création
- validation et la signature du JWT
- configuration des règles d'autorisation basées sur le JWT
- gestion des exceptions liées à l'authentification et l'autorisation, etc

---

Implémentation de JWT

Ajout de dépendance

1. Ajouter les dépendances nécessaires dans votre fichier pom.xml pour utiliser les fonctionnalités JWT de Spring Security

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
```

Création de la classe "JwtTokenProvider"

2. Créez une classe JwtTokenProvider qui est responsable de la génération, de la validation et de la récupération des informations d'un JWT

```
import io.jsonwebtoken.*;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.core.Authentication;
import org.springframework.stereotype.Component;

import java.util.Date;

@Component
public class JwtTokenProvider {
```

```
    @Value("${jwt.secret}")
    private String jwtSecret;

    @Value("${jwt.expiration}")
    private int jwtExpiration;

    public String generateToken(Authentication authentication) {
        UserDetailsImpl userDetails = (UserDetailsImpl)
authentication.getPrincipal();

        Date now = new Date();
        Date expiryDate = new Date(now.getTime() + jwtExpiration);

        return Jwts.builder()
                .setSubject(Long.toString(userDetails.getId()))
                .setIssuedAt(new Date())
                .setExpiration(expiryDate)
                .signWith(SignatureAlgorithm.HS512, jwtSecret)
                .compact();
    }

    public Long getUserIdFromToken(String token) {
        Claims claims = Jwts.parser()
                .setSigningKey(jwtSecret)
                .parseClaimsJws(token)
                .getBody();

        return Long.parseLong(claims.getSubject());
    }

    public boolean validateToken(String token) {
        try {
            Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(token);
            return true;
        } catch (SignatureException ex) {
            System.out.println("Invalid JWT signature");
        } catch (MalformedJwtException ex) {
            System.out.println("Invalid JWT token");
        } catch (ExpiredJwtException ex) {
            System.out.println("Expired JWT token");
        } catch (UnsupportedJwtException ex) {
            System.out.println("Unsupported JWT token");
        } catch (IllegalArgumentException ex) {
            System.out.println("JWT claims string is empty");
        }
        return false;
    }
}
```

JwtTokenProvider et SecurityConfig

3. Utiliser le JwtTokenProvider dans votre classe SecurityConfig pour configurer Spring Security pour utiliser les JWT

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint;

    @Autowired
    private UserDetailsService userDetailsService;

    @Autowired
    private JwtRequestFilter jwtRequestFilter;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
                .authorizeRequests()
                .antMatchers("/authenticate").permitAll()
                .anyRequest().authenticated()
                .and()

.exceptionHandling().authenticationEntryPoint(jwtAuthenticationEntryPoint)
                .and()

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

        http.addFilterBefore(jwtRequestFilter,
UsernamePasswordAuthenticationFilter.class);
    }

    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {

auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());
    }

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

    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }
}
```

Sécurisation des routes

Qu'est-ce que c'est ?

- Fait référence à la protection des différentes URL (ou routes) de votre application contre les accès non autorisés.
- L'objectif est de restreindre l'accès aux ressources sensibles de votre application aux utilisateurs authentifiés et autorisés.

Plus en détails

- Lorsque vous configurez Spring Security, vous pouvez définir des règles de sécurité pour spécifier qui est autorisé à accéder à certaines URL ou à effectuer certaines actions.
- Vous pouvez contrôler les autorisations en fonction des rôles des utilisateurs, des privilèges spécifiques, des conditions personnalisées, etc.

Exemple

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/public").permitAll() // Route publique accessible à
tous
                .antMatchers("/admin").hasRole("ADMIN") // Route accessible
uniquement aux utilisateurs ayant le rôle "ADMIN"
                .anyRequest().authenticated() // Toutes les autres routes
nécessitent une authentification
                .and()
            .formLogin() // Configuration de l'authentification basée sur les
formulaires
                .loginPage("/login") // Page de connexion personnalisée
                .permitAll()
                .and()
            .logout()
                .permitAll();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .inMemoryAuthentication()
                .withUser("user").password("{noop}password").roles("USER") //
```

```
Utilisateur avec le rôle "USER"
                .and()
                .withUser("admin").password("{noop}password").roles("ADMIN"); //
Utilisateur avec le rôle "ADMIN"
    }

    @Override
    public void configure(WebSecurity web) throws Exception {
        web
            .ignoring()
                .antMatchers("/css/**", "/js/**", "/images/**"); // Ignorer les
ressources statiques lors de l'application de la sécurité
    }
}
```

Tests d'une application sécurisée (tests unitaires)

A quoi ça sert ?

Vérifier si les mécanismes de sécurité fonctionnent correctement et si les ressources sont correctement protégées

---

Exemple total de code

User.java

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

    private String username;
    private String password;
    private String role;

    // Constructeurs

    public User() {
    }

    public User(String username, String password, String role) {
        this.username = username;
        this.password = password;
        this.role = role;
    }
```

```
    // Getters and Setters

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

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

    public String getPassword() {
        return password;
    }

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

    public String getRole() {
        return role;
    }

    public void setRole(String role) {
        this.role = role;
    }
}
```

SecurityConfig.java

```
@Configuration // Classe de configuration
@EnableWebSecurity // Active la configuration de la sécurité Web pour votre
application
@EnableGlobalMethodSecurity(prePostEnabled = true) // Active la sécurisation des
méthodes dans Spring Security
public class SecurityConfig {

    @Autowired
    private UserDetailsService userDetailsService;

    // Configure l'authentification
    // Charge les détails de l'utilisateur et encode le mot de passe
    @Override
```

```
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {

auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());
    }

    // Configure les règles d'autorisation et de sécurité HTTP pour les
différentes URL de l'appli
    // Configure la gestion de session
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .antMatchers("/admin/**").hasRole("ADMIN") // Les URL commençant
par "/admin" nécessitent le rôle "ADMIN"
                .antMatchers("/user/**").hasRole("USER") // Les URL commençant par
"/user" nécessitent le rôle "USER"
                .anyRequest().authenticated() // Toutes les autres URL nécessitent
une authentification
                .and()
                .formLogin()
                .and()
                .logout().logoutSuccessUrl("/login?logout")
                .and()
                .exceptionHandling().accessDeniedPage("/403")
                .and()
                .sessionManagement()
                    .maximumSessions(1)
                        .sessionRegistry(sessionRegistry()) // Spécifie qu'un
utilisateur ne peut avoir qu'une seule session active à la fois
                        .invalidSessionUrl("/login") // Redirige l'utilisateur
vers l'URL "/login" si une session invalide est détectée
                        .expiredUrl("/login?expired"); // Redirige vers "/login?
expired" si la session a expiré
    }

    // Utilisé pour encoder les mots de passe des utilisateurs
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
     }

    @Bean
    public SpringSessionBackedSessionRegistry sessionRegistry() {
        return new SpringSessionBackedSessionRegistry(sessionRepository());
    }

    // Publier les événements liés à la session
    @Bean
    public HttpSessionEventPublisher httpSessionEventPublisher() {
        return new HttpSessionEventPublisher();
    }
}
```

UserController

```
@RestController
@RequestMapping("/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping
    public ResponseEntity<List<User>> getAllUsers() {
        List<User> users = userService.getAllUsers();
        return new ResponseEntity<>(users, HttpStatus.OK);
    }

    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable Long id) {
        User user = userService.getUserById(id);
        if (user != null) {
            return new ResponseEntity<>(user, HttpStatus.OK);
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }

    @PreAuthorize("hasRole('ADMIN')")
    @PostMapping
    public ResponseEntity<User> createUser(@RequestBody User user) {
        User createdUser = userService.createUser(user);
        return new ResponseEntity<>(createdUser, HttpStatus.CREATED);
    }

    @PreAuthorize("hasRole('ADMIN')")
    @PutMapping("/{id}")
    public ResponseEntity<User> updateUser(@PathVariable Long id, @RequestBody
User user) {
        User updatedUser = userService.updateUser(id, user);
        if (updatedUser != null) {
            return new ResponseEntity<>(updatedUser, HttpStatus.OK);
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }

    @PreAuthorize("hasRole('ADMIN')")
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
        userService.deleteUser(id);
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }
}
```

UserDetailsServiceImpl

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByUsername(username);
        if (user == null) {
            throw new UsernameNotFoundException("Utilisateur non trouvé : " +
username);
        }

        // Créer l'objet UserDetails à partir des informations de l'utilisateur
        List<GrantedAuthority> authorities = buildUserAuthority(user.getRoles());
        return buildUserDetails(user, authorities);
    }

    // Créer l'objet User à partir des informations de l'utilisateur
    private org.springframework.security.core.userdetails.User
buildUserDetails(User user, List<GrantedAuthority> authorities) {
        return new org.springframework.security.core.userdetails.User(
                user.getUsername(),
                user.getPassword(),
                user.isEnabled(),
                true,
                true,
                true,
                authorities
        );
    }

    // Créer une liste d'objets GrantedAuthority à partir des rôles de
l'utilisateur
    private List<GrantedAuthority> buildUserAuthority(Set<Role> userRoles) {
        List<GrantedAuthority> authorities = new ArrayList<>();
        for (Role role : userRoles) {
            authorities.add(new SimpleGrantedAuthority(role.getName()));
        }
        return authorities;
    }
}
```

suite