Sécuriser son application avec Spring Security

16. Introduction à la sécurité avec Spring Security

La sécurité est un aspect critique de toute application moderne, en particulier lorsque vous gérez des données d'investigation ou des dashboards administratifs. **Spring Security** est le framework de référence de l'écosystème Spring pour gérer l'**authentification** (vérifier qui vous êtes) et l'**autorisation** (vérifier ce que vous avez le droit de faire).

Par défaut, dès que vous ajoutez la dépendance `spring-boot-starter-security` à votre projet, Spring Security sécurise **tous** vos endpoints HTTP. Il génère automatiquement un mot de passe aléatoire dans la console au démarrage et bloque l'accès à vos API avec un mécanisme appelé **Basic Authentication**.

> **Basic Auth vs JWT (JSON Web Token) :**
> • **Basic Auth :** Envoie l'identifiant et le mot de passe à chaque requête dans l'en-tête HTTP. C'est simple, mais peu adapté aux architectures découplées modernes (comme un client mobile Flutter ou un dashboard Angular web).
> • **JWT :** Le client s'authentifie une seule fois via un endpoint de login. Le serveur lui renvoie un jeton (token) chiffré et signé. Le client stocke ce jeton et l'envoie dans l'en-tête de ses prochaines requêtes (`Authorization: Bearer <token>`). C'est la méthode standard pour les API REST sans état (stateless).

17. Configuration de base de Spring Security

Dans les versions modernes de Spring Security (Spring Boot 3+), la configuration s'effectue de manière programmatique à l'aide d'un composant (Bean) de type `SecurityFilterChain`.

Voici comment configurer votre application pour définir des règles d'accès précises (par exemple, autoriser tout le monde à lire les indices publics, mais exiger un rôle d'administrateur pour modifier ou supprimer des suspects) :

```
package com.studiogames.mysterybackend.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
```

apcpedagogie.com

```
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;

import static org.springframework.security.config.Customizer.withDefaults;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            // Désactiver le CSRF (généralement inutile pour les API REST stateless)
            .csrf(csrf -> csrf.disable())

            // Configuration des autorisations sur les requêtes HTTP
            .authorizeHttpRequests(auth -> auth
                // Permettre l'accès public à l'endpoint de statut
                .requestMatchers("/api/cases/vanishing-diplomat/status").permitAll()
                // Restreindre les requêtes d'écriture/modification aux Administrateurs
                .requestMatchers("/api/suspects/**").hasRole("ADMIN")
                // Toutes les autres requêtes nécessitent une simple authentification
                .anyRequest().authenticated()
            )
            // Activer l'authentification Basic (par défaut pour les tests rapides)
            .httpBasic(withDefaults());

        return http.build();
    }

    // Configuration de utilisateurs de test en mémoire (In-Memory Authentication)
    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails admin = User.withDefaultPasswordEncoder()
                .username("detective_chief")
                .password("secure123")
                .roles("ADMIN")
                .build();

        return new InMemoryUserDetailsManager(admin);
    }
}
```

18. Architecture de l'authentification par jeton (JWT)

Pour implémenter une architecture JWT robuste, l'application doit suivre le cycle de vie suivant :

Étape	Acteur	Description de l'action
1	Client (Flutter/Angular)	Envoie les identifiants (username/password) via une requête POST `/api/auth/login`.

Étape	Acteur	Description de l'action
2	Serveur (Spring Boot)	Vérifie les identifiants. Si corrects, il génère un JWT signé avec une clé secrète et le renvoie au client.
3	Client (Flutter/Angular)	Stocke le token localement et l'ajoute dans l'en-tête `Authorization` pour chaque appel d'API ultérieur.
4	Serveur (Spring Boot)	Un filtre de sécurité (JWT Filter) intercepte la requête, valide la signature du token et extrait les rôles sans interroger la base de données.

19. Implémentation pratique d'un utilitaire JWT

Pour utiliser JWT, vous devez ajouter une bibliothèque de manipulation de tokens comme `io.jsonwebtoken:jjwt` à votre fichier de build. Voici la structure de la classe de service chargée de la gestion technique des jetons :

```
package com.studiogames.mysterybackend.security;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.HashMap;
import java.util.Map;

@Component
public class JwtTokenUtil {

    private final String SECRET_KEY =
"ma_cle_secrete_tres_longue_pour_le_jeu_de_mystere_noir";
    private final long TOKEN_VALIDITY = 3600000; // 1 heure en millisecondes

    // Générer un token pour un utilisateur
    public String generateToken(String username) {
        Map<String, Object> claims = new HashMap<>();
        return Jwts.builder()
                .setClaims(claims)
                .setSubject(username)
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + TOKEN_VALIDITY))
                .signWith(SignatureAlgorithm.HS256, SECRET_KEY)
                .compact();
    }

    // Extraire le nom d'utilisateur à partir du token
```

```
    public String extractUsername(String token) {
        return extractAllClaims(token).getSubject();
    }

    // Vérifier si le token a expiré
    public boolean isTokenExpired(String token) {
        return extractAllClaims(token).getExpiration().before(new Date());
    }

    // Valider le token
    public boolean validateToken(String token, String username) {
        final String extractedUsername = extractUsername(token);
        return (extractedUsername.equals(username) && !isTokenExpired(token));
    }

    private Claims extractAllClaims(String token) {
        return Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token).getBody();
    }
}
```

Étape suivante pour un projet réel : Pour finaliser l'intégration JWT, vous devez créer un `OncePerRequestFilter` personnalisé. Ce filtre va lire l'en-tête HTTP de chaque requête entrante, valider le token à l'aide de la classe `JwtTokenUtil` ci-dessus, puis injecter l'identité de l'utilisateur directement dans le contexte de sécurité de Spring (`SecurityContextHolder`).

*Rédigé et structuré par : **Med Khalil Kribi***

Source : https://apcpedagogie.com/securiser-application-spring-security · téléchargé le 07/11/2026 · © APC Pédagogie · apcpedagogie.com