

Cours Approfondi Spring Security 6

Sécuriser vos applications Spring Boot : des formulaires aux tokens JWT

Les Piliers de Spring Security



Authentification (AuthN)

"Qui êtes-vous ?"

Le processus de vérification de l'identité (login, mot de passe, token).



Autorisation (AuthZ)

"Qu'avez-vous le droit de faire ?"

Le processus de vérification des permissions (rôles, scopes).



Principal

"L'utilisateur connecté."

L'objet représentant l'entité authentifiée ([UserDetails](#)).

La Révolution de Spring Security 6

🚫 Fin de `WebSecurityConfigurerAdapter`

On n'hérite plus de cette classe. La configuration devient basée sur les composants.

⚙️ Configuration par `@Bean`

On définit un ou plusieurs Beans de type `SecurityFilterChain` pour configurer la sécurité.

⟨/⟩ DSL Lambda Systématisée

La configuration est plus concise et lisible (ex: `http.authorizeHttpRequests(auth → ...)`).

↗️ `requestMatchers()` remplace `antMatchers()`

Une API plus claire pour définir les permissions sur les URLs (plus de confusion avec `mvcMatchers`).

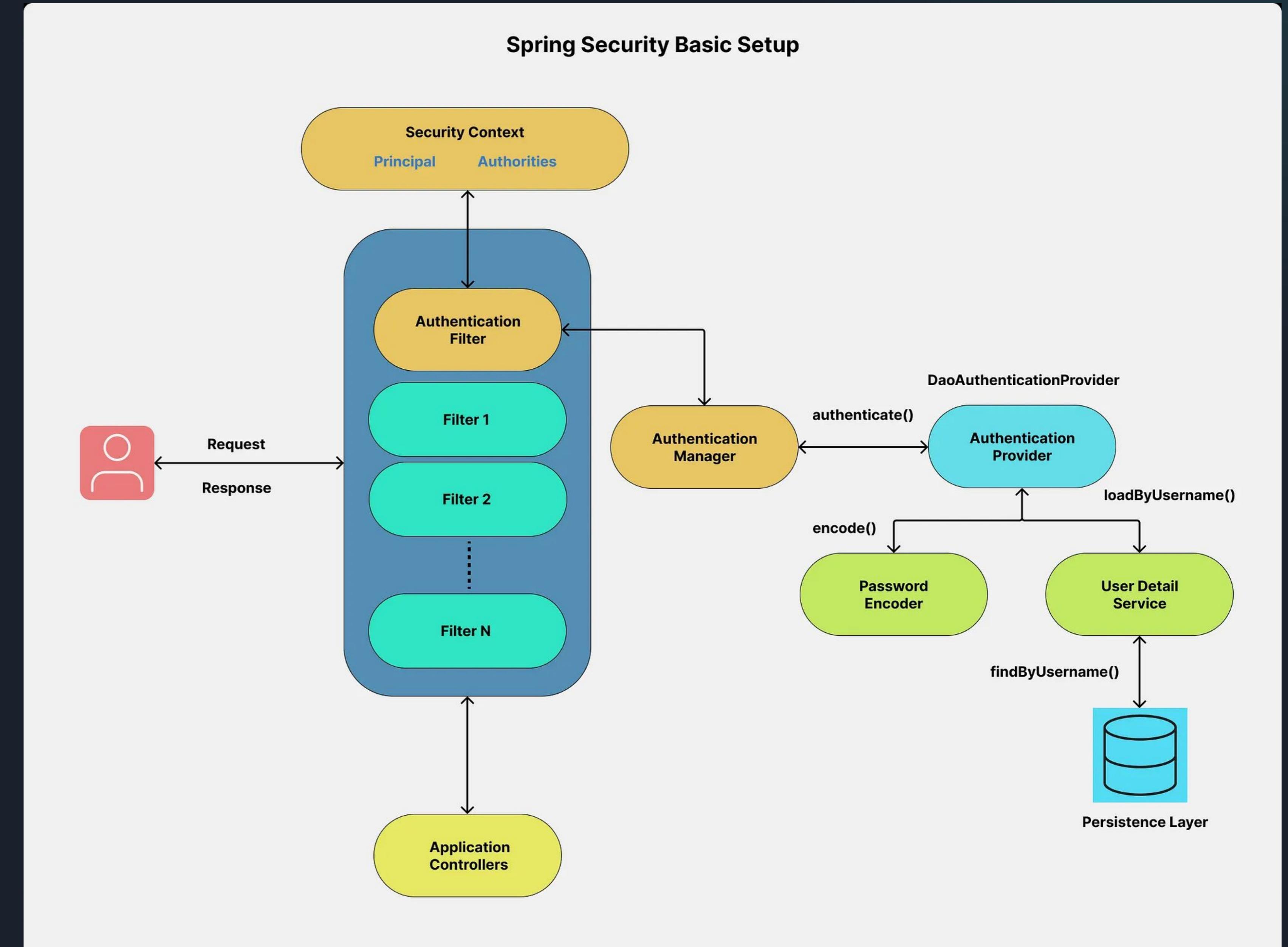
Architecture : La Chaîne de Filtres

Le SecurityFilterChain

Chaque requête HTTP passe à travers une chaîne de filtres. Chaque filtre a une responsabilité unique :

- ✓ **CsrfFilter** (Protection CSRF)
- ✓ **UsernamePasswordAuthenticationFilter** (Login)
- ✓ **BearerTokenAuthenticationFilter** (JWT)
- ✓ **AuthorizationFilter** (Vérification des rôles)

Avec Spring Security 6, on définit cette chaîne via un **@Bean**.



Mise en Place : Le Starter

Le "Starter" Spring Security

Pour activer Spring Security, il suffit d'ajouter la dépendance `spring-boot-starter-security` à votre `pom.xml`.

Dès cet instant, **toute votre application est sécurisée par défaut** (login par formulaire et mot de passe généré en console).

```
org.springframework.boot  
spring-boot-starter-security
```

```
org.springframework.boot  
spring-boot-starter-web
```

Section 1

Sécurité Web "Stateful" (Form Login)

Exemple : La Sécurité par Défaut

"Out-of-the-Box"

Sans aucune configuration, si vous ajoutez le starter, Spring Security :

- ✓ Protège tous les points d'accès.
- ✓ Génère un utilisateur `user` avec un mot de passe aléatoire (affiché au démarrage).
- ✓ Met en place une page de login par formulaire (auto-générée).
- ✓ Active la protection CSRF.

C'est un excellent point de départ "sécurisé par défaut".

Please sign in

Username

Password

Sign in

Exemple : Le `SecurityFilterChain`

La Configuration Minimale

C'est la nouvelle façon de configurer. On crée une classe `@Configuration` et on y déclare un `@Bean`.

Cet exemple configure la sécurité web (`@EnableWebSecurity`) et définit une chaîne de filtres qui active le login par formulaire (`formLogin`) pour toutes les requêtes (`anyRequest`) authentifiées.

```
// SecurityConfig.java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(
        HttpSecurity http)

        throws Exception {
            http
                .authorizeHttpRequests(authz → authz
                    // Toutes les requêtes nécessitent une
                    authentication
                    .anyRequest().authenticated()
                )
                // Active le formulaire de login par défaut
                .formLogin(Customizer.withDefaults());

            return http.build();
    }
}
```

Exemple : Autoriser des URLs Publiques

Autorisations (AuthZ)

L'ordre est crucial : les règles les plus spécifiques doivent être déclarées en premier.

Ici, nous permettons l'accès public (`permitAll`) aux ressources statiques et à la page d'accueil, tout en exigeant le rôle `ADMIN` pour le chemin `/admin`.

```
// ... dans la méthode securityFilterChain ...
http
    .authorizeHttpRequests(authz → authz
        // Autoriser les ressources statiques et la home
        .requestMatchers("/", "/home", "/css/**", "/js/**")
        .permitAll()

        // Exiger le rôle ADMIN pour /admin
        .requestMatchers("/admin/**")
        .hasRole("ADMIN") // "ROLE_ADMIN" en base

        // Le reste doit être authentifié
        .anyRequest()
        .authenticated()
    )
    .formLogin(Customizer.withDefaults());
```

Exemple : Personnaliser Form Login

Page de Login Personnalisée

Si la page de login par défaut ne vous convient pas, vous pouvez spécifier la vôtre.

Vous devez créer un `@Controller` qui expose `/login` (en `GET`) et autoriser l'accès public à cette page.

Spring s'occupe de gérer le `POST` vers `/login` (ou `loginProcessingUrl`).

```
// ... dans la méthode securityFilterChain ...
http
    // ... authorizeHttpRequests ...
    .authorizeHttpRequests(authz → authz
        .requestMatchers("/login").permitAll() // IMPORTANT
        .anyRequest().authenticated()
    )
    .formLogin(form → form
        // URL de notre page de login personnalisée
        .loginPage("/login")

        // URL de traitement (le
        // doit pointer ici)
        .loginProcessingUrl("/perform_login")

        // Redirection après succès
        .defaultSuccessUrl("/dashboard", true)

        // Redirection après échec
        .failureUrl("/login?error=true")
    );
}
```

Exemple : Personnaliser Logout

Configuration de la Déconnexion

Par défaut, le logout est activé sur `/logout` (en `POST` pour des raisons de sécurité CSRF).

On peut personnaliser l'URL de redirection et les actions à effectuer, comme invalider la session ou supprimer les cookies.

```
// ... dans la méthode securityFilterChain ...
http
    // ... formLogin ...
    .logout(logout → logout
        // L'URL qui déclenche la déconnexion
        .logoutUrl("/perform_logout")
        // Redirection après succès
        .logoutSuccessUrl("/login?logout=true")

        // Invalider la session HTTP
        .invalidateHttpSession(true)

        // Supprimer le cookie de session
        .deleteCookies("JSESSIONID")
    );
}
```

Protection CSRF (Cross-Site Request Forgery)

Qu'est-ce que c'est ?

Une attaque qui force un utilisateur authentifié à exécuter des actions non désirées (ex: un lien malveillant sur un autre site qui déclenche un **POST** vers [/transfer-money](#)).

Spring Security active la protection CSRF par défaut pour les applications "stateful". Il génère un token `_csrf` qui doit être inclus dans tous les formulaires **POST**, **PUT**, **DELETE**.

Important : On le désactive pour les API "stateless".

```
// Exemple de formulaire Thymeleaf avec CSRF  
th:action="@{/perform_login}" method="post">
```

```
    type="hidden"  
    th:name="${_csrf.parameterName}"  
    th:value="${_csrf.token}" />
```

```
// Pour désactiver CSRF (NON RECOMMANDÉ pour le web stateful)  
http.csrf(csrf → csrf.disable());
```

Section 2

Gestion des Utilisateurs

Exemple : Utilisateurs en Mémoire (Test)

Pour le Développement

Pour des tests rapides ou des tutoriels, vous pouvez définir des utilisateurs directement en mémoire.

Cela se fait en définissant un `@Bean` de type `InMemoryUserDetailsManager`.

Note : N'utilisez jamais cela en production.

```
// Dans une classe @Configuration
@Bean
public InMemoryUserDetailsManager userDetailsService(
    PasswordEncoder
    passwordEncoder) {

    UserDetails user = User.withUsername("user")
        .password(passwordEncoder.encode("password"))
        .roles("USER")
        .build();

    UserDetails admin = User.withUsername("admin")
        .password(passwordEncoder.encode("admin123"))
        .roles("USER", "ADMIN")
        .build();

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

Exemple : UserDetailsService (JPA)

L'Approche Standard (BDD)

Pour charger des utilisateurs depuis votre base de données (JPA, JDBC...), vous devez implémenter l'interface `UserDetailsService`.

Sa seule méthode, `loadUserByUsername`, doit trouver l'utilisateur dans votre BDD et le retourner sous forme d'un objet `UserDetails` (que Spring Security comprendra).

```
// JpaUserDetailsService.java
@Service
public class JpaUserDetailsService implements
UserDetailsService {

    @Autowired
    private UtilisateurRepository utilisateurRepo;

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {
        // 1. Trouver l'entité JPA (Utilisateur)
        Utilisateur user =
            utilisateurRepo.findByUsername(username)
                .orElseThrow(() →
                    new UsernameNotFoundException("Utilisateur non
trouvé"));

        // 2. La convertir en UserDetails Spring
        return new SecurityUser(user); // Classe wrapper (voir
slide suivante)
    }
}
```

Exemple : Implémenter UserDetails

Le "Wrapper" de Sécurité

UserDetails est l'interface que Spring Security utilise.

Votre entité JPA (ex: Utilisateur) ne doit pas implémenter UserDetails directement.

La bonne pratique est de créer une classe "Wrapper" (ex: SecurityUser) qui implémente UserDetails et qui encapsule (wrap) votre entité JPA.

Cette classe fait le pont entre votre domaine et Spring Security.

```
// SecurityUser.java
public class SecurityUser implements UserDetails {
    private final Utilisateur user;
    public SecurityUser(Utilisateur user) { this.user = user; }

    @Override
    public String getUsername() { return user.getUsername(); }

    @Override
    public String getPassword() { return user.getPassword(); }

    @Override
    public Collection<GrantedAuthority> getAuthorities()
    {
        // Convertir les Rôles (String) en GrantedAuthority
        return user.getRoles().stream()
            .map(role → new SimpleGrantedAuthority("ROLE_" +
role))
            .collect(Collectors.toList());
    }

    // ... isAccountNonExpired, isEnabled ... (généralement
```

Exemple : Le `PasswordEncoder`

Stockage Sécurisé

Spring Security 6 **impose** l'utilisation d'un encodeur de mot de passe. Les mots de passe en clair sont refusés.

Vous devez déclarer un `@Bean` de type `PasswordEncoder`. L'implémentation recommandée est `BCryptPasswordEncoder`, qui gère le hachage et le "salage" (salt) automatiquement.

Spring l'injectera partout où il en a besoin (ex: pour `UserDetailsService`).

```
// Dans SecurityConfig.java ou une autre classe @Configuration  
@Bean  
public PasswordEncoder passwordEncoder() {  
    // Algorithme de hachage fort et adaptatif  
    return new BCryptPasswordEncoder();  
}  
  
// --- Utilisation lors de l'enregistrement ---  
@Service  
public class UserService {  
    @Autowired private PasswordEncoder passwordEncoder;  
  
    public void registerUser(String username, String rawPassword) {  
        Utilisateur user = new Utilisateur();  
        user.setUsername(username);  
        // On encode le mot de passe avant de le sauvegarder  
        user.setPassword(passwordEncoder.encode(rawPassword));  
        // ... sauvegarde ...  
    }  
}
```

Section 3

Autorisation Avancée

Autorisation : URL vs. Méthode

1. Sécurité par URL (dans SecurityFilterChain)

Contrôle d'accès "grossier". Définit quelles URLs sont accessibles.

Ex:

```
.requestMatchers("/admin/**").hasRole("ADMIN")
```

Avantage : Centralisé et facile à lire.

Inconvénient : Ne gère pas la logique métier (ex: "un utilisateur peut-il modifier *ce* post ?").

2. Sécurité par Méthode (@PreAuthorize)

Contrôle d'accès "fin". S'applique directement sur vos méthodes de @Service ou @Controller .

Ex: @PreAuthorize("hasRole('ADMIN') or #username == authentication.name")

Avantage : Extrêmement puissant, utilise Spring Expression Language (SpEL).

Inconvénient : Dispersé dans le code.

Exemple : Sécurité par Méthode

Activer `@PreAuthorize`

Pour utiliser les annotations de sécurité au niveau des méthodes, vous devez d'abord l'activer.

1. Ajoutez `@EnableMethodSecurity` à votre `SecurityConfig`.

2. Annotez vos méthodes (généralement dans la couche `@Service`) avec `@PreAuthorize`.

SpEL (Spring Expression Language) vous donne accès à des variables comme `authentication` (l'utilisateur connecté) et aux arguments de la méthode.

```
// 1. Activer dans SecurityConfig.java
@Configuration
@EnableWebSecurity
@EnableMethodSecurity // Active @PreAuthorize
public class SecurityConfig { ... }

// 2. Utiliser dans le Service
@Service
public class PostService {

    // Tout le monde peut lister les posts
    public List<Post> findAll() { ... }

    // Seul un ADMIN peut supprimer n'importe quel post
    @PreAuthorize("hasRole('ADMIN')")
    public void deletePost(Long id) { ... }

    // L'auteur du post OU un admin peut le modifier
    @PreAuthorize(" @postRepository.findById(#id).get().getAuthor()
        = " +
                    "authentication.name or hasRole('ADMIN')")
20   public Post updatePost(Long id, String content) { ... }
}
```

Section 4

Sécurité "Stateless" (API & JWT)

Configuration : API Stateless

Stateful vs. Stateless

Jusqu'à présent, nous étions "Stateful" (avec sessions `JSESSIONID` et protection `CSRF`).

Les API REST (pour mobile, SPA) doivent être "Stateless". Le client (ex: React) envoie un **token** (ex: JWT) à chaque requête.

Nous devons donc :

1. ✓ Désactiver la création de sessions.
2. ✓ Désactiver la protection CSRF (inutile sans session).

```
// ... dans la méthode securityFilterChain ...
http
    // 1. Désactiver CSRF (non nécessaire pour stateless)
    .csrf(csrf → csrf.disable())

    // 2. Définir la politique de session à STATELESS
    .sessionManagement(session → session
        .sessionCreationPolicy(
            SessionCreationPolicy.STATELESS
        )
    )

    .authorizeHttpRequests(authz → authz
        // L'endpoint de login doit être public
        .requestMatchers("/api/auth/login").permitAll()
        .anyRequest().authenticated()
    );
// ... mais comment valider le token ?
```

Options de Sécurité Stateless



Serveur de Ressources (OAuth2)

Le standard. Vous utilisez un serveur d'autorisation externe (Keycloak, Okta, Auth0). Spring Security valide les tokens JWT signés par cet émetteur (issuer).

Recommandé pour les microservices.



Authentification Basique (HTTP Basic)

Simple (`http.httpBasic()`), mais envoie le login/mdp à chaque requête (encodé en Base64). **Non recommandé en production (sauf HTTPS).**



Filtre JWT Personnalisé

Approche "manuelle". Vous créez votre propre endpoint de login qui génère un JWT, et un filtre personnalisé qui valide ce token à chaque requête.

Courant pour les monolithes.

Exemple : Serveur de Ressources (OAuth2)

Valider les JWT (Standard)

C'est l'approche la plus simple et la plus robuste si vous avez un serveur d'autorisation.

1. Ajoutez le starter `spring-boot-starter-oauth2-resource-server`.

2. Ajoutez l'URL de votre émetteur (issuer) dans `application.properties`.

3. Configurez le `SecurityFilterChain`.

```
// application.properties
spring.security.oauth2.resourceserver.jwt.issuer-
uri=https://votre-serveur-auth.com/realm/mon-realm

// SecurityConfig.java
@Bean
public SecurityFilterChain apiSecurityFilterChain(
    HttpSecurity http)
throws Exception {
    http
        .csrf(csrf → csrf.disable())
        .sessionManagement(s →
            s.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(authz → authz
            .anyRequest().authenticated()
        )
    // Active la validation JWT via l'issuer-uri
    .oauth2ResourceServer(oauth2 → oauth2
        .jwt(Customizer.withDefaults())
    );
}

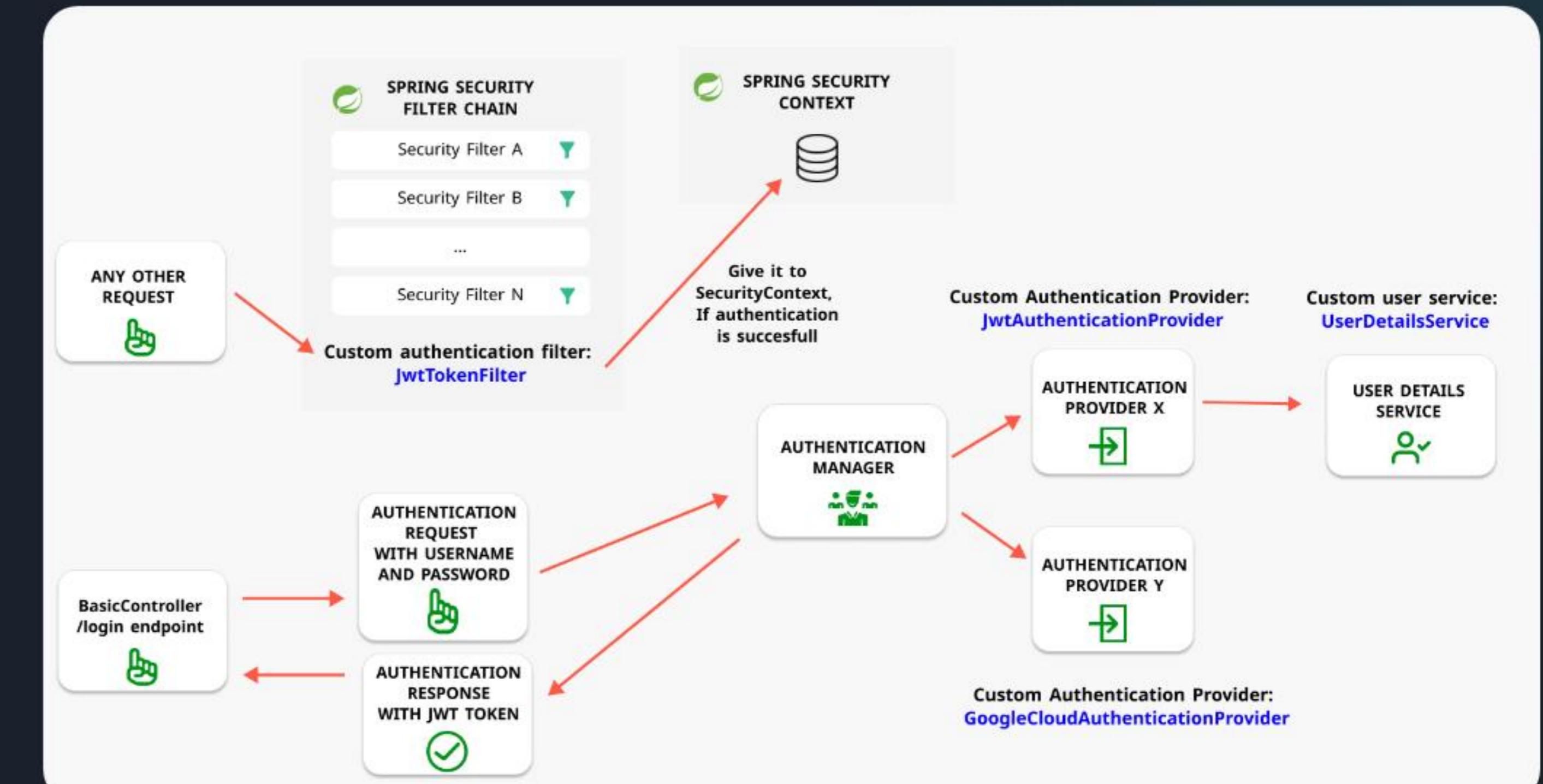
return http.build();
```

Exemple : Filtre JWT Personnalisé

L'Approche Manuelle

Si vous n'avez pas de serveur OAuth2, vous devez gérer le cycle de vie du token vous-même :

1. Un `/login` qui génère un JWT (ex: avec la lib `jjwt`).
2. Un filtre personnalisé (`JwtAuthFilter`) qui hérite de `OncePerRequestFilter`.
3. Ce filtre doit :
 - ✓ Lire l'en-tête `Authorization: Bearer ...`
 - ✓ Valider le token (signature, expiration).
 - ✓ Extraire le username du token.
 - ✓ Charger le `UserDetails` (via `UserDetailsService`).
 - ✓ Créer un `UsernamePasswordAuthenticationToken` et le placer dans le `SecurityContextHolder`.



Exemple : Configuration CORS (API)

Cross-Origin Resource Sharing

Si votre API (ex: `api.mondomaine.com`) est appelée par un Frontend (ex: `app.mondomaine.com`), le navigateur bloquera les requêtes à cause de la politique "Same-Origin".

CORS est nécessaire. La méthode recommandée est de fournir un `@Bean CorsConfigurationSource`.

N'oubliez pas d'activer `.cors()` dans votre `SecurityFilterChain` !

```
// Dans SecurityConfig.java
@Bean
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();
    // Autoriser les requêtes depuis votre frontend

    configuration.setAllowedOrigins(List.of("http://localhost:3000"));
    configuration.setAllowedMethods(List.of("GET", "POST", "PUT",
 ""));
    configuration.setAllowedHeaders(List.of("Authorization",
 "Content-Type"));
    configuration.setAllowCredentials(true);

    UrlBasedCorsConfigurationSource source = new
 UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", configuration);
    return source;
}

// N'oubliez pas de l'activer dans la chaîne de filtres :
http.cors(Customizer.withDefaults()); // Active le Bean ci-dessus
```

Résumé des Bonnes Pratiques (Spring 6)

Configuration par `@Bean`

Utilisez un `@Bean SecurityFilterChain`. N'utilisez plus `WebSecurityConfigurerAdapter`.

DSL Lambda

Adoptez la syntaxe lambda (`.authorizeHttpRequests(auth → ...)`) pour la clarté.

Encodez les Mots de Passe

Toujours fournir un `@Bean PasswordEncoder` (`BCrypt` est le standard).

UserDetailsService

Implémentez cette interface pour connecter Spring Security à votre base de données utilisateurs (JPA/JDBC).

Stateless pour les API

Désactivez `csrf()` et configurez les sessions en `STATELESS`. Utilisez `oauth2ResourceServer` pour valider les