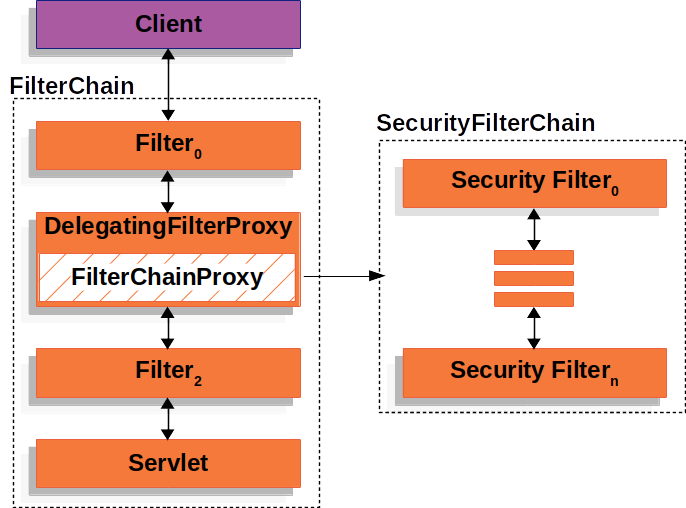
# Spring Security

## Giới thiệu về Spring Security

* Spring Security là một framework bảo mật mạnh mẽ và có thể tùy chỉnh cao. Nó là tiêu chuẩn “thực tế” cho việc bảo mật các ứng dụng dựa trên Spring. Nó cung cấp giải pháp toàn diện cho cả hai khía cạnh chính của bảo mật ứng dụng:
* **Xác thực** (**Authentication**): Chứng minh bạn là ai.
* **Phân quyền** (**Authorization**): Bạn được phép làm gì sau khi đã được xác thực.
* Spring Security được xây dựng dựa trên các Servlet Filter, cho phép nó chặn các request đến và xử lý các yêu cầu bảo mật trước khi chúng đến được controller hoặc các thành phần xử lý logic nghiệp vụ khác trong Spring MVC.



* Cài đặt (**Maven**):

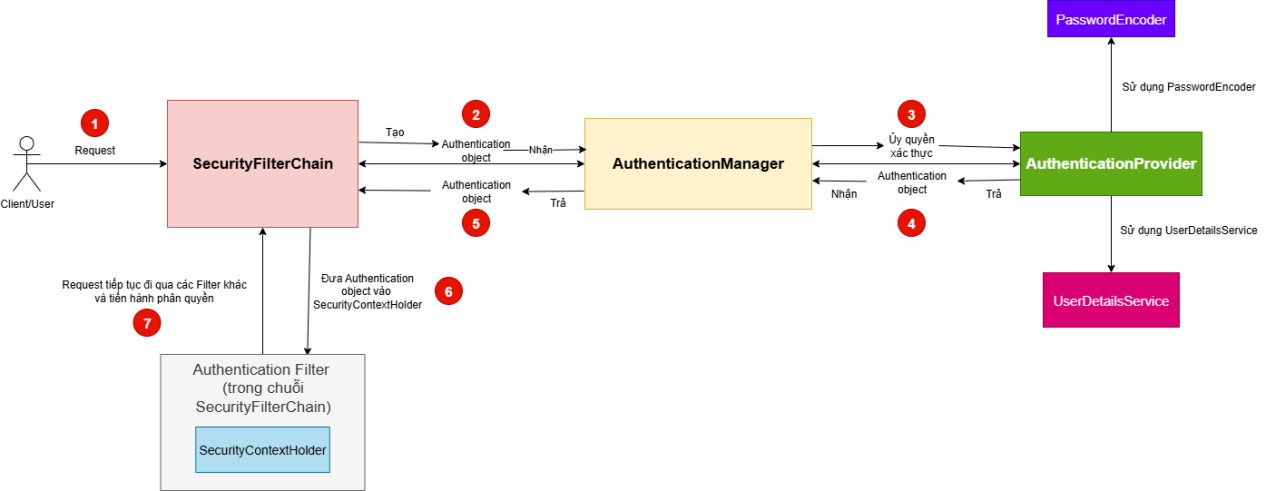
<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

## Luồng hoạt động cơ bản của Spring Security



1. **Request** đến **SecurityFilterChain**. Các Filter trong chuỗi bắt đầu xử lý. Một trong những Filter đầu tiên (ví dụ: UsernamePasswordAuthenticationFilter cho form login) sẽ cố gắng thực hiện **\*\*Xác thực\*\***. Nếu người dùng chưa được xác thực, Filter authentication sẽ cố gắng lấy thông tin đăng nhập (từ form, header, token…).
2. Thông tin đăng nhập được gói gọn trong một đối tượng **Authentication** và được gửi đến **AuthenticationManager**.
3. **AuthenticationManager** ủy quyền việc xác thực cho một hoặc nhiều **AuthenticationProvider** phù hợp. AuthenticationProvider (ví dụ: DaoAuthenticationProvider) sử dụng **UserDetailsService** để tải thông tin người dùng và **PasswordEncoder** để kiểm tra mật khẩu.

4, 5, 6. Nếu xác thực thành công, một đối tượng Authentication đầy đủ thông tin (bao gồm Principal và GrantedAuthoritys) được tạo ra và lưu vào SecurityContextHolder, đánh dấu người dùng đã được xác thực. Sau khi Xác thực hoàn tất (thành công hoặc thất bại), luồng xử lý tiếp tục. Nếu xác thực thất bại, một **AuthenticationEntryPoint** được kích hoạt (ví dụ: chuyển hướng đến trang login).

1. Nếu xác thực thành công, request tiếp tục đi qua các Filter khác. Khi request đến một tài nguyên được bảo vệ bằng cấu hình Phân quyền (ví dụ: .authorizeHttpRequests(...) hoặc @PreAuthorize), quá trình **\*\*Phân quyền\*\*** bắt đầu. Spring Security sẽ sử dụng thông tin **Authentication** hiện có trong **SecurityContextHolder** (bao gồm các GrantedAuthoritys) để kiểm tra quyền. **AccessDecisionManager**, với sự trợ giúp của các **AccessDecisionVoter**, sẽ đưa ra quyết định cuối cùng: cho phép hay từ chối truy cập. Nếu Phân quyền thành công, request được chuyển tiếp đến đích cuối cùng (Controller, Service…). Nếu thất bại, một ngoại lệ **AccessDeniedException** được ném ra, dẫn đến response 403 Forbidden.

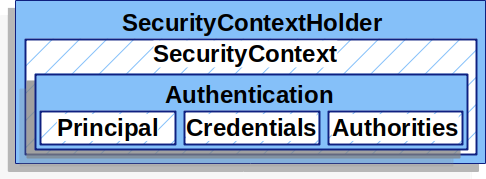
## Xác thực (Authentication) trong Spring Security

### Các thành phần để xử lý xác thực bao gồm:

* **Authentication** Object: Đây là trung tâm của quá trình xác thực. Nó chứa thông tin về yêu cầu xác thực (thường là username/password) và sau khi xác thực thành công, nó chứa thông tin chi tiết về người dùng đã xác thực (principal) và các quyền hạn (authorities).
* **AuthenticationManager**: Là giao diện chính để thực hiện xác thực. Nó nhận một đối tượng Authentication làm đầu vào và trả về một đối tượng Authentication đã được điền đầy đủ thông tin nếu thành công.
* **AuthenticationProvider**: Dependency được sử dụng bởi AuthenticationManager để thực hiện logic xác thực thực tế. Có nhiều loại provider khác nhau (ví dụ: để xác thực từ cơ sở dữ liệu, LDAP, OAuth2…).
* **UserDetailsService**: Một giao diện quan trọng được các AuthenticationProvider sử dụng để tải thông tin chi tiết về người dùng (username, password, enabled, authorities…). Kết quả trả về là một đối tượng UserDetails.
* **PasswordEncoder**: Giao diện được sử dụng để mã hóa (encode) và so sánh (match) mật khẩu. Việc lưu trữ mật khẩu dưới dạng văn bản gốc là cực kỳ nguy hiểm, do đó luôn cần mã hóa mật khẩu trước khi lưu vào DB. Spring Security cung cấp nhiều implement của giao diện này (ví dụ: BCryptPasswordEncoder là khuyến nghị phổ biến nhất).

### Quá trình xác thực cơ bản diễn ra như sau:

* Yêu cầu (ví dụ: POST đến /login) chứa thông tin đăng nhập (username, password) được gửi đến ứng dụng.
* Bộ lọc xác thực tương ứng bắt lấy yêu cầu này.
* Bộ lọc tạo ra một đối tượng **Authentication** (chưa được xác thực – unauthenticated) chứa thông tin đăng nhập.
* Đối tượng **AuthenticationManager** (trái tim của quy trình xác thực) nhận đối tượng **Authentication** này.
* **AuthenticationManager** ủy quyền việc xác thực cho một hoặc nhiều **AuthenticationProvider**.
* **AuthenticationProvider** (ví dụ: DaoAuthenticationProvider) sẽ lấy thông tin người dùng dựa trên username (thường thông qua **UserDetailsService**) và so sánh mật khẩu đã mã hóa (sử dụng **PasswordEncoder**).
* Nếu xác thực thành công, **AuthenticationProvider** trả về một đối tượng **Authentication** đã được xác thực (authenticated) chứa thông tin chi tiết của người dùng (UserDetails) và danh sách quyền hạn (Authorities).
* **AuthenticationManager** nhận đối tượng đã xác thực này và đưa vào **SecurityContextHolder**. SecurityContextHolder là nơi Spring Security lưu trữ thông tin bảo mật của người dùng hiện tại (theo Thread Local theo mặc định, hoặc theo request/session tùy cấu hình).
* Yêu cầu được tiếp tục xử lý hoặc bị từ chối nếu xác thực thất bại.



### Cấu hình xác thực với **SecurityFilterChain**:

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

@EnableWebSecurity

public class SecurityConfig {

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(authorize -> authorize

.requestMatchers("/public/\*\*", "/css/\*\*", "/js/\*\*").permitAll() // Cho phép truy cập công khai

.requestMatchers("/admin/\*\*").hasRole("ADMIN") // Yêu cầu vai trò ADMIN

.requestMatchers("/user/\*\*").hasAnyRole("USER", "ADMIN") // Yêu cầu vai trò USER hoặc ADMIN

.anyRequest().authenticated() // Tất cả các yêu cầu khác đều phải được xác thực

)

.formLogin(form -> form

.loginPage("/login") // Chỉ định URL trang đăng nhập tùy chỉnh

.defaultSuccessUrl("/") // URL chuyển hướng sau khi đăng nhập thành công

.permitAll() // Cho phép mọi người truy cập trang đăng nhập

)

.logout(logout -> logout

.logoutUrl("/logout") // Chỉ định URL đăng xuất

.logoutSuccessUrl("/login?logout") // URL chuyển hướng sau khi đăng xuất thành công

.permitAll() // Cho phép mọi người truy xuất URL đăng xuất

)

.csrf(csrf -> csrf.disable()); // Tắt CSRF

return http.build();

}

}

### Cấu hình nguồn dữ liệu người dùng: **UserDetailsService** và **UserDetails**

**UserDetailsService** là một interface đơn giản với một phương thức duy nhất: **loadUserByUsername(String username)**. Phương thức này chịu trách nhiệm lấy thông tin chi tiết của người dùng (dưới dạng đối tượng **UserDetails**) dựa trên tên đăng nhập được cung cấp.

Spring Security sẽ tự động tìm kiếm một **Bean kiểu UserDetailsService** và sử dụng nó trong quá trình xác thực.

**VD: Cấu hình In-Memory UserDetailsManager**

// ... imports ...

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

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.crypto.factory.PasswordEncoderFactories;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.security.provisioning.InMemoryUserDetailsManager;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

@EnableWebSecurity

public class SecurityConfig {

// ... securityFilterChain Bean ...

@Bean

public UserDetailsService userDetailsService(PasswordEncoder passwordEncoder) {

// Sử dụng builder pattern để tạo UserDetails

UserDetails user = User.withUsername("user")

.password(passwordEncoder.encode("password")) // Mật khẩu phải được mã hóa!

.roles("USER")

.build();

UserDetails admin = User.withUsername("admin")

.password(passwordEncoder.encode("adminpass")) // Mật khẩu phải được mã hóa!

.roles("ADMIN", "USER") // Một người dùng có thể có nhiều vai trò

.build();

// Trả về InMemoryUserDetailsManager với danh sách người dùng

return new InMemoryUserDetailsManager(user, admin);

}

@Bean

public PasswordEncoder passwordEncoder() {

// Sử dụng PasswordEncoderFactories.createDelegatingPasswordEncoder()

return PasswordEncoderFactories.createDelegatingPasswordEncoder();

}

}

**VD: Cấu hình Database Authentication với Custom UserDetailsService**

package com.bty.study.springsecuritypractice1.service;

import com.bty.study.springsecuritypractice1.entity.Account;

import com.bty.study.springsecuritypractice1.repository.AccountRepository;

import org.springframework.beans.factory.annotation.Autowired;

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.core.userdetails.UsernameNotFoundException;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.stereotype.Service;

@Service

public class CustomUserDetailsService implements UserDetailsService {

@Autowired

private AccountRepository accountRepository;

@Override

public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

Account account = accountRepository.findByUsername(username)

.orElseThrow(() -> new UsernameNotFoundException(username));

return User.builder()

.username(account.getUsername())

.password(account.getPassword())

.roles(account.getRole())

.build();

}

}

## Phân quyền (Authorization) trong Spring Security

### Các thành phần thực hiện phân quyền trong Spring Security

* **GrantedAuthority**: Giao diện đại diện cho một quyền hạn được cấp cho Principal (người dùng đã xác thực). Đây thường là các vai trò (Roles) như `ROLE\_ADMIN`, `ROLE\_USER`, hoặc các quyền cụ thể (Permissions) như `READ\_PRIVILEGE`, `WRITE\_PRIVILEGE`.
* **AccessDecisionManager**: Đây là điểm quyết định cuối cùng trong quá trình phân quyền. Nó nhận thông tin về người dùng đã xác thực, tài nguyên được bảo vệ và các thuộc tính bảo mật liên quan (ví dụ: URL pattern, annotation trên method) và đưa ra quyết định liệu truy cập có được cho phép hay không.
* **AccessDecisionVoter**: Dependency được sử dụng bởi AccessDecisionManager để “bỏ phiếu” về quyết định truy cập. Các Voter khác nhau có thể kiểm tra các loại thuộc tính bảo mật khác nhau (ví dụ: một voter kiểm tra vai trò, một voter khác kiểm tra quyền cụ thể).

### Cấu hình phân quyền trong Spring Security

#### Cấu Hình Bảo Mật Dựa Trên URL (URL-Based Security)

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

@EnableWebSecurity

public class SecurityConfig {

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.authorizeHttpRequests(authorizeRequests ->

authorizeRequests

// Cho phép truy cập các URL công khai mà không cần xác thực

.requestMatchers("/", "/public/\*\*", "/login", "/css/\*\*", "/js/\*\*").permitAll()

// Yêu cầu vai trò ADMIN cho các URL bắt đầu bằng /admin/

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

// Yêu cầu vai trò USER hoặc ADMIN cho các URL bắt đầu bằng /user/

.requestMatchers("/user/\*\*").hasAnyRole("USER", "ADMIN")

// Yêu cầu xác thực for tất cả các request còn lại

.anyRequest().authenticated()

)

.formLogin(formLogin ->

formLogin

.loginPage("/login") // Chỉ định trang đăng nhập tùy chỉnh

.permitAll() // Cho phép tất cả mọi người truy cập trang đăng nhập

)

.logout(logout ->

logout

.permitAll() // Cho phép tất cả mọi người logout

);

return http.build();

}

}

#### Cấu Hình Bảo Mật Dựa Trên Phương Thức (Method-Based Security)

Để bật bảo mật phương thức, ta sử dụng **@EnableMethodSecurity**

@Configuration

@EnableWebSecurity

@EnableMethodSecurity // Bật bảo mật phương thức

public class SecurityConfig {

//…

}

Sau khi bật **@EnableMethodSecurity**, ta có thể sử dụng các annotations sau trên các phương thức của Spring Beans (ví dụ: các phương thức trong lớp **@Service** hoặc **@Controller**):

**@PreAuthorize**: Kiểm tra quyền **trước** khi phương thức được thực thi. Đây là annotation mạnh mẽ nhất vì nó hỗ trợ Spring Expression Language (SpEL), cho phép ta viết các biểu thức kiểm tra phức tạp.

**@PostAuthorize**: Kiểm tra quyền **sau** khi phương thức đã thực thi (và giá trị trả về có sẵn). Hữu ích khi quyết định có cho phép trả về kết quả cho người dùng hay không, hoặc kiểm tra quyền dựa trên dữ liệu được trả về.

VD:

@Service

public class ProductService {

@PreAuthorize("hasRole('ADMIN')") // Chỉ ADMIN mới được gọi phương thức này TRƯỚC khi nó thực thi

public void addProduct(Product product) {

// logic thêm sản phẩm

}

@PreAuthorize("hasAnyRole('USER', 'ADMIN')") // USER hoặc ADMIN được phép

public Product getProductById(Long id) {

// logic lấy sản phẩm

return new Product();

}

@PostAuthorize("returnObject.owner == authentication.principal.username") // Chỉ trả về kết quả nếu người dùng là chủ sở hữu (SAU khi phương thức thực thi)

public Product getMyProduct(Long id) {

// logic lấy sản phẩm

Product p = new Product(); // Lấy product từ DB

p.setOwner("current\_user"); // Giả sử đây là owner từ DB

return p;

}

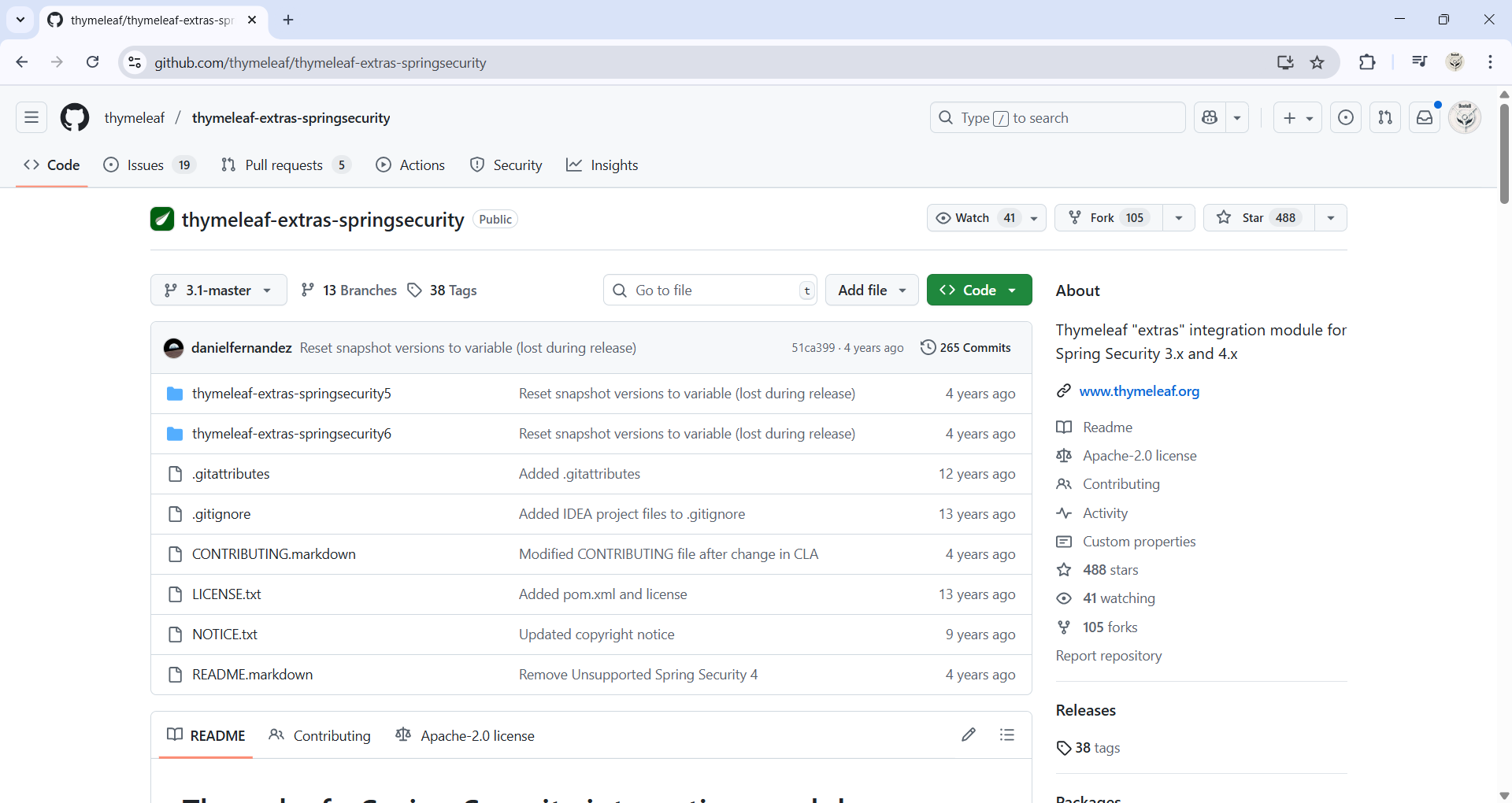
}

#### So sánh URL-Based Security và Method-Based Security

|  |  |  |
| --- | --- | --- |
|  | **URL-Based Security** | **Method-Based Security** |
| **Mức độ áp dụng** | Áp dụng cho toàn bộ URL/pattern URL | Áp dụng cho từng phương thức (method) cụ thể |
| **Cấu hình** | Trong **SecurityFilterChain** | Sử dụng annotations (**@PreAuthorize**, **@Secured**…) trên phương thức và bật **@EnableMethodSecurity** |
| **Tính linh hoạt** | Chủ yếu dựa vào path patterns và vai trò/quyền cơ bản | Sử dụng Spring EL (SpEL) rất mạnh mẽ, có thể kiểm tra dựa trên tham số, dữ liệu trả về, v.v. |
| **Ưu điểm** | Dễ dàng bảo vệ nhóm URL; Là lớp bảo vệ đầu tiên cho web request. | Kiểm soát chi tiết; Logic bảo mật gần với code nghiệp vụ; Dễ áp dụng cho các phương thức không phải endpoint web. |
| **Nhược điểm** | Khó áp dụng cho các trường hợp phân quyền phức tạp dựa trên dữ liệu; Không bảo vệ các phương thức được gọi nội bộ (không qua web request). | annotations có thể làm rối code; Yêu cầu cẩn thận khi sử dụng @PostAuthorize (có thể trả về dữ liệu nhạy cảm trước khi kiểm tra); Performance overhead nhỏ hơn so với URL filtering. |

## Spring Security trong Thymeleaf

<https://github.com/thymeleaf/thymeleaf-extras-springsecurity.git>



**Maven info**:

groupId: **org.thymeleaf.extras**

artifactId:

Spring Security 5 integration package: **thymeleaf-extras-springsecurity5**

Spring Security 6 integration package: **thymeleaf-extras-springsecurity6**

**Một số ví dụ**:

Lấy giá trị của thuộc tính của đối tượng authentication:

<div th:text="${#authentication.name}">

The value of the "name" property of the authentication object should appear here.

</div>

Thêm namespace:

<html xmlns:sec="http://www.thymeleaf.org/extras/spring-security">

Lấy giá trị của thuộc tính của đối tượng authentication thông qua sec:authentication:

<div sec:authentication="name">

The value of the "name" property of the authentication object should appear here.

</div>

Kiểm tra quyền để hiển thị:

<div sec:authorize="hasRole('ROLE\_ADMIN')">

This will only be displayed if authenticated user has role ROLE\_ADMIN.

</div>