Spring Security

1. 개념

Spring Security는 **스프링 기반 애플리케이션의 보안(인증과 인가)** 을 담당하는 프레임워크임.

요약하면, "누가 접근할 수 있고, 무엇을 할 수 있는가"를 관리함.

애플리케이션에 로그인, 권한 제어, 세션 관리, 토큰 검증 등을 통합적으로 제공함.

스프링 시큐리티는 **필터 기반 구조**로 작동함.

모든 요청이 컨트롤러로 들어가기 전에 필터 체인을 거쳐 인증과 인가를 처리함.

2. 핵심 개념

(1) 인증 (Authentication)

사용자가 누구인지 확인하는 과정임.

로그인 시 아이디, 비밀번호를 입력받아 DB의 정보와 비교함.

성공하면 `Authentication` 객체가 생성되어 `SecurityContext`에 저장됨.

(2) 인가 (Authorization)

인증된 사용자가 특정 리소스에 접근할 권한이 있는지를 확인하는 과정임.

예: 관리자만 `/admin/**` 접근 가능, 일반 사용자는 `/user/**`만 접근 가능.

3. 동작 구조

스프링 시큐리티는 필터 체인을 통해 요청을 가로채고 인증/인가 절차를 거침.

요청 흐름은 다음과 같음.

1. 사용자가 `/login` 요청을 보냄
2. `UsernamePasswordAuthenticationFilter`가 요청을 가로챔
3. `AuthenticationManager`가 인증을 시도함
4. `AuthenticationProvider`가 실제로 사용자 정보를 확인함 (DB 조회 등)

5. 인증 성공 시 `SecurityContextHolder`에 인증 정보를 저장함
6. 이후 모든 요청에서 이 인증 정보를 기반으로 권한을 검사함

---

4. 기본 설정 예시 (Spring Boot 3.x 기준)

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception
{
        http
            .csrf(csrf -> csrf.disable()) // 개발 시에는 비활성화
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/login", "/signup", "/css/**").permitAll() // 누구
나 접근 가능
                .requestMatchers("/admin/**").hasRole("ADMIN") // 관리자 권한 필
요
                .anyRequest().authenticated() // 나머지는 로그인 필요
            )
            .formLogin(form -> form
                .loginPage("/login") // 커스텀 로그인 페이지
                .defaultSuccessUrl("/") // 로그인 성공 시 이동할 경로
                .permitAll()
            )
            .logout(logout -> logout
                .logoutUrl("/logout")
                .logoutSuccessUrl("/login?logout")
            );

        return http.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder(); // 비밀번호 암호화용
```

```
    }
}
```

이 설정은 로그인, 접근 권한, 로그아웃 등을 제어함.

`BCryptPasswordEncoder`로 비밀번호를 안전하게 암호화함.

---

5. 사용자 정보 관리 (UserDetailsService)

Spring Security는 사용자 정보를 `UserDetails` 형태로 관리함.

DB에서 사용자 정보를 불러와 인증 과정에서 사용함.

```
@Service
public class CustomUserDetailsService implements UserDetailsService {

    private final UserRepository userRepository;

    public CustomUserDetailsService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws Usern
ameNotFoundException {
        UserEntity user = userRepository.findByUsername(username)
                .orElseThrow(() -> new UsernameNotFoundException("User not f
ound"));

        return org.springframework.security.core.userdetails.User.builder()
                .username(user.getUsername())
                .password(user.getPassword())
                .roles(user.getRole())
                .build();
    }
}
```

---

6. 비밀번호 암호화

평문 비밀번호를 저장하면 안 됨.

`BCryptPasswordEncoder`를 사용해 암호화함.

```
PasswordEncoder encoder = new BCryptPasswordEncoder();
String encoded = encoder.encode("mypassword");
```

인증 시 `encoder.matches(rawPassword, encodedPassword)` 로 비교함.

---

7. 확장 가능한 구조

Spring Security는 다양한 인증 방식을 지원함.

- 세션 기반 인증
- JWT 기반 인증 (토큰 사용)
- OAuth2 소셜 로그인 (Google, Kakao, Github 등)
- 커스텀 필터 기반 인증 (IP 제한, 추가 보안 로직 등)

기본 폼 로그인 외에도 REST API, 모바일 앱 등 환경에 맞게 확장 가능함.

---

8. 심화 학습 포인트

기초 설정을 익혔다면 아래 주제들을 추가로 학습해야 실무에 적용 가능함.

(1) 세션 기반 인증

- 로그인 성공 시 세션 생성, 서버 메모리에 저장
- 간단하지만 확장성 낮음 (서버 여러 대면 세션 공유 필요)

(2) JWT 기반 인증

- 세션 없이 토큰으로 인증 상태 유지
- 프론트엔드-백엔드 분리형 구조에 적합
- `OncePerRequestFilter` 를 상속해 토큰 검증 필터 구현함
- Refresh Token 개념도 함께 학습 필요

(3) OAuth2 / OIDC (소셜 로그인)

- 외부 인증 서비스(구글, 카카오 등)를 통해 로그인
- Access Token, Refresh Token, Authorization Code Flow 개념 숙지 필요
- OIDC는 OAuth2에 사용자 ID 토큰이 추가된 형태임

(4) 커스텀 필터

- 특정 요청 전후에 동작할 보안 로직 추가 가능
- 예: JWT 인증 필터, IP 필터
- `addFilterBefore`, `addFilterAfter` 로 순서 지정 가능

(5) 예외 처리

- 인증 실패 시 → `AuthenticationEntryPoint`
- 인가 실패 시 → `AccessDeniedHandler`
- REST API에서는 JSON 형태의 에러 응답을 직접 구성해야 함

(6) SecurityContext와 ThreadLocal

- 인증 정보는 `SecurityContextHolder`에 저장됨
- 내부적으로 `ThreadLocal`로 관리되므로 비동기/멀티스레드 환경에서는 주의 필요

(7) CSRF, CORS 보안

- **CSRF**: 폼 로그인 시 중요. REST API에서는 비활성화하는 경우가 많음
- **CORS**: 프론트엔드(React, Vue 등)와 통신할 때 필수 설정

(8) 권한 관리 설계

- ROLE 기반뿐만 아니라 Permission 기반 설계도 가능
- 메서드 단위 권한 제어: `@PreAuthorize`, `@PostAuthorize`

```
@PreAuthorize("hasRole('ADMIN')")
public void deleteUser(Long id) { ... }
```

(9) 테스트

- `@WithMockUser`, `SecurityMockMvcRequestPostProcessors.user()` 활용

- JWT 인증 시에는 토큰을 직접 주입해 테스트 수행

(10) 구조화된 보안 패키지 구성

보안 관련 클래스를 명확히 분리하면 유지보수성이 높아짐

```
/config/SecurityConfig.java
/security/filter/JwtFilter.java
/security/service/CustomUserDetailsService.java
/security/handler/CustomAccessDeniedHandler.java
```

---

9. 최신 트렌드

- Spring Boot 3.x 이상에서는 `WebSecurityConfigurerAdapter` 완전 제거됨 → `SecurityFilterChain` 방식 사용
- 세션 기반보다는 JWT 기반 인증이 일반적임
- OAuth2 클라이언트 통합 (`spring-boot-starter-oauth2-client`) 이 기본 제공됨
- “Stateless + OAuth2 + JWT” 조합이 실무 표준으로 자리 잡음

---

10. 정리

구분	핵심 내용
인증	사용자 신원 확인 (로그인)
인가	접근 권한 제어
구조	필터 체인 기반
확장성	JWT, OAuth2, 커스텀 필터 등 지원
실무 포인트	예외 처리, 권한 설계, 테스트, CSRF/CORS 대응

결론

Spring Security는 단순히 로그인 기능을 제공하는 라이브러리가 아님.

애플리케이션 전반의 **보안 아키텍처를 구성하는 핵심 프레임워크**임.

기초 수준에서는 로그인/로그아웃과 권한 제어를 이해하면 되고,

심화 단계에서는 JWT, OAuth2, 커스텀 필터, 예외 처리, 권한 설계 등을 학습해야 함.