12. 보안 (Spring Security)

Spring Security 구조와 필터 체인

Spring Security는 Spring 기반 애플리케이션에서 **인증(Authentication)**과 **인가(Authorization)**를 중심으로 **보안 기능을 제공하는 핵심 프레임워크**이다.
그 중심에는 **Filter Chain 기반의 아키텍처**가 있으며, **모듈화된 보안 필터들이 HTTP 요청을 통제**한다.

📘 1. Spring Security 전체 구조 개요

```
Client → Security Filter Chain → Authentication → Authorization → Controller
```

구성 요소	역할
FilterChainProxy	모든 보안 필터를 묶는 진입점
SecurityFilterChain	URL 별 필터 목록 정의
Security Filters	요청을 가로채 인증/인가 처리
AuthenticationManager	사용자 인증 로직 (예: 로그인 처리)
AuthorizationManager	접근 허용 여부 판단 (예: ROLE 검사)
UserDetailsService	사용자 정보 조회
PasswordEncoder	비밀번호 암호화 및 비교
SecurityContextHolder	현재 인증된 사용자 정보 저장소

🔐 2. 필터 체인 (Security Filter Chain)

Spring Security는 **Servlet Filter 기반**으로 동작하며,
하나의 요청이 다음과 같은 필터 체인을 거치며 처리된다.

✅ 필터 체인 구성 예시

```
❶ SecurityContextPersistenceFilter
❷ UsernamePasswordAuthenticationFilter
❸ BasicAuthenticationFilter
❹ ExceptionTranslationFilter
❺ FilterSecurityInterceptor
...
```

🧩 3. 주요 필터 설명

필터 이름	역할
SecurityContextPersistenceFilter	`SecurityContext`를 요청 전후로 저장/복원
UsernamePasswordAuthenticationFilter	로그인 폼 제출 요청을 처리 (기본 `/login`)
BasicAuthenticationFilter	`Authorization: Basic` 헤더 처리
BearerTokenAuthenticationFilter	JWT 등 Bearer Token 처리 (스프링 시큐리티 6)
ExceptionTranslationFilter	인증/인가 실패 → 예외 처리 (401, 403)
FilterSecurityInterceptor	권한 검사 (인가 처리) 수행

> ⚠️ 필터 순서 중요: 인증 → 예외 처리 → 인가 순서대로 구성됨

📜 4. 인증(Authentication) 흐름

```
① 로그인 요청 (POST /login)
  ↓
② UsernamePasswordAuthenticationFilter → AuthenticationManager
  ↓
③ UserDetailsService로 사용자 조회
  ↓
④ PasswordEncoder로 비밀번호 확인
  ↓
⑤ 성공 시 Authentication 객체 생성
  ↓
⑥ SecurityContextHolder에 저장
  ↓
⑦ 컨트롤러 실행
```

📜 5. 인가(Authorization) 흐름

```
① 요청 수신
  ↓
② FilterSecurityInterceptor가 인증 객체 확인
  ↓
③ @PreAuthorize 또는 URL 권한 검사
  ↓
④ 허용 → 컨트롤러 실행 / 차단 → 403 Forbidden
```

⚙️ 6. Filter Chain 설정 방식 (Spring Security 6 기준)

✅ SecurityFilterChain 설정 (Java Config)

```
@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .csrf().disable()
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/public/**").permitAll()
            .anyRequest().authenticated()
        )
        .formLogin(Customizer.withDefaults());
    return http.build();
}
```

🔧 7. 커스터마이징 예시

✅ 사용자 인증 처리 커스터마이징

```
@Bean
UserDetailsService userDetailsService() {
    return username -> userRepository.findByUsername(username)
        .map(user -> new User(user.getUsername(), user.getPassword(), authorities))
        .orElseThrow(() -> new UsernameNotFoundException("User not found"));
}
```

✅ 비밀번호 암호화 설정

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

🧠 8. 실무 전략

항목	전략
JWT 기반 보안 구조	`BearerTokenAuthenticationFilter` 사용
REST API 보안	`stateless` + Token + 커스텀 필터
커스텀 로그인 처리	`UsernamePasswordAuthenticationFilter` 교체
컨트롤러 권한 제어	`@PreAuthorize("hasRole('ADMIN')")`
예외 응답 커스터마이징	`AuthenticationEntryPoint`, `AccessDeniedHandler` 구현

📌 9. 요약

구성 요소	설명
SecurityFilterChain	보안 필터 목록 구성
주요 필터	인증 → 예외 → 인가 처리 필터로 구성
인증(Authentication)	로그인 처리: ID/PW, 토큰 등
인가(Authorization)	권한 확인: URL, 메서드, 롤(Role) 등
실무 사용 전략	필터 체인 분리, JWT 인증 확장, 예외 핸들러 분리

Form 로그인 처리

Spring Security에서의 Form 로그인 처리는 가장 기본적이고 널리 사용되는 **인증(Authentication)** 방식이다.
사용자가 **ID와 비밀번호를 입력하여 로그인하면**,
Spring Security의 **필터 체인과 인증 매니저가 이를 검증하고 세션에 인증 정보를 저장**한다.

📘 1. Form 로그인 처리 개요

항목	설명
기본 로그인 URL	`POST /login`
기본 로그인 페이지	`/login` (자동 생성)
인증 처리 필터	`UsernamePasswordAuthenticationFilter`
인증 처리 클래스	`AuthenticationManager`, `UserDetailsService`, `PasswordEncoder`
결과 저장 위치	`SecurityContextHolder`, `HttpSession` (기본 설정)

🔄 2. 인증 처리 흐름

```
① 사용자가 /login에 ID/PW 제출
   ↓
② UsernamePasswordAuthenticationFilter가 요청 처리
   ↓
③ AuthenticationManager에게 인증 위임
   ↓
④ UserDetailsService가 사용자 정보 조회
   ↓
⑤ PasswordEncoder로 비밀번호 일치 여부 검증
   ↓
⑥ 성공 시 Authentication 객체 생성 후 SecurityContextHolder에 저장
   ↓
⑦ 인증된 상태로 요청 처리 가능
```

🧩 3. 기본 설정 (Spring Security 6 이상)

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .csrf().disable()
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/login", "/css/**", "/js/**").permitAll()
            .anyRequest().authenticated()
        )
        .formLogin(Customizer.withDefaults()); // 기본 로그인 폼 사용
    return http.build();
}
```

→ `/login` 페이지 자동 생성
→ `POST /login`으로 로그인 요청하면 처리됨

💉 4. 사용자 정의 로그인 페이지

```
.formLogin(form -> form
    .loginPage("/login")            // 커스텀 로그인 페이지
    .loginProcessingUrl("/login")   // 로그인 폼 action
    .usernameParameter("username")  // 기본: username
    .passwordParameter("password")  // 기본: password
    .defaultSuccessUrl("/home", true) // 로그인 성공 시 이동
    .failureUrl("/login?error=true")  // 로그인 실패 시 이동
)
```

📃 5. 로그인 HTML 폼 예시

```
<form method="post" action="/login">
    <input type="text" name="username" placeholder="아이디" />
    <input type="password" name="password" placeholder="비밀번호" />
    <button type="submit">로그인</button>
</form>
```

🔐 6. 사용자 인증 구성 요소

✅ 1. UserDetailsService

```
@Bean
UserDetailsService userDetailsService() {
    return username -> {
        User user = userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException(username));
```

```
        return new org.springframework.security.core.userdetails.User(
            user.getUsername(),
            user.getPassword(),
            List.of(new SimpleGrantedAuthority("ROLE_USER"))
        );
    };
}
```

✅ 2. PasswordEncoder

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

→ 사용자 등록 시 비밀번호를 반드시 암호화하여 저장해야 인증 성공

🔧 7. 로그인 성공/실패 핸들러 커스터마이징

```
.formLogin(form -> form
    .successHandler((request, response, authentication) -> {
        response.sendRedirect("/home");
    })
    .failureHandler((request, response, exception) -> {
        response.sendRedirect("/login?error=true");
    })
)
```

🧠 8. 실무 전략

항목	전략
비밀번호 평문 저장 금지	반드시 `BCryptPasswordEncoder` 사용
로그인 시도 제한	사용자 정의 로그인 핸들러 + DB 기록
실패 응답 개선	`failureHandler`에서 이유 분석 가능
커스텀 로그인 페이지	`/login`에 Thymeleaf or HTML 페이지 제작
세션 고정 보호, 동시 로그인 제한	`sessionManagement()` 설정 고려

📌 9. 요약

항목	설명
기본 URL	`/login`, `POST /login`
인증 필터	`UsernamePasswordAuthenticationFilter`
사용자 정의 설정	`.formLogin().loginPage(...)` 등
필수 구성 요소	`UserDetailsService`, `PasswordEncoder`
인증 결과 저장 위치	`SecurityContextHolder` → 세션 or 토큰 기반 전환 가능

사용자 인증

Spring Security에서 **사용자 인증(Authentication)**은 클라이언트가 시스템에 접근하려고 할 때,
제공한 자격 증명(아이디, 비밀번호, 토큰 등)이 유효한지를 검증하는 과정이다.
이는 Spring Security 구조의 핵심이며, 인증 성공 시 **SecurityContext에 인증 정보가 저장되고**
이후 모든 요청은 이 정보를 바탕으로 인가(Authorization) 처리된다.

📘 1. 사용자 인증이란?

> 사용자의 ID와 비밀번호(또는 토큰, 인증서 등)를 검증하여
> **"누구인지 증명하는 과정"**

🔄 2. 인증 흐름 요약

```
① 클라이언트가 ID/PW 또는 토큰과 함께 요청
   ↓
② 인증 필터(예: UsernamePasswordAuthenticationFilter) → AuthenticationManager 호출
   ↓
③ AuthenticationManager → UserDetailsService → 사용자 조회
   ↓
④ 비밀번호 비교 → 성공 시 Authentication 객체 생성
   ↓
⑤ SecurityContextHolder에 저장
   ↓
⑥ 인증 성공 상태로 컨트롤러 동작
```

🧩 3. 인증에 사용되는 핵심 구성 요소

구성 요소	설명
`Authentication`	인증 객체 (ID, PW, 권한 등 포함)
`AuthenticationManager`	인증을 처리하는 인터페이스

구성 요소	설명
`UsernamePasswordAuthenticationToken`	인증 전/후 상태를 표현하는 객체
`UserDetailsService`	사용자 정보를 로드하는 서비스
`PasswordEncoder`	비밀번호 암호화/검증 도구
`SecurityContextHolder`	인증 정보를 저장하는 ThreadLocal

🔐 4. 인증 처리 클래스 구조

✅ 1. AuthenticationManager

```
public interface AuthenticationManager {
    Authentication authenticate(Authentication authentication) throws
AuthenticationException;
}
```

→ 기본 구현체: `ProviderManager`

✅ 2. UserDetailsService 구현

```
@Bean
public UserDetailsService userDetailsService() {
    return username -> {
        User user = userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException("사용자 없음"));

        return new org.springframework.security.core.userdetails.User(
            user.getUsername(),
            user.getPassword(),
            List.of(new SimpleGrantedAuthority("ROLE_USER"))
        );
    };
}
```

✅ 3. PasswordEncoder 설정

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

- 사용자 등록 시에도 `passwordEncoder.encode()` 로 저장
- 로그인 시 자동으로 매칭 (`matches(raw, encoded)`)

🧾 5. 인증 객체: `Authentication`

인증 전 (로그인 요청 시)

```
new UsernamePasswordAuthenticationToken(username, password);
```

인증 후 (성공 후 저장될 때)

```
new UsernamePasswordAuthenticationToken(userDetails, null, authorities);
```

→ 이후 `SecurityContextHolder.getContext().getAuthentication()`으로 현재 인증된 사용자 접근 가능

🧪 6. 커스터마이징 가능한 인증 전략

시점	확장 가능 요소
사용자 조회	`UserDetailsService` 구현
비밀번호 비교	`PasswordEncoder` 교체
인증 처리	`AuthenticationManager`, `AuthenticationProvider` 구현
인증 필터 교체	`UsernamePasswordAuthenticationFilter` 커스터마이징
인증 성공/실패 핸들러	`AuthenticationSuccessHandler`, `AuthenticationFailureHandler`

✅ 7. 실무 전략

항목	전략
사용자 인증	`UserDetailsService`에서 DB 조회
비밀번호 보안	`BCryptPasswordEncoder`로 암호화
사용자 정보 저장소	`SecurityContextHolder` + 세션 or JWT
커스텀 인증 처리	자체 `AuthenticationProvider` 구현 가능
인증 실패시 상세 응답	`AuthenticationEntryPoint` 커스터마이징

📌 8. 요약

항목	설명
인증이란	사용자 ID/PW 또는 토큰 검증
기본 처리 구조	AuthenticationManager → UserDetailsService → PasswordEncoder

항목	설명
인증 성공 시	Authentication 객체 생성 → SecurityContext에 저장
필수 구성 요소	Authentication, UserDetails, PasswordEncoder
확장 전략	DB 조회/비밀번호/토큰 처리/응답 방식까지 모두 커스터마이징 가능

사용자 권한 처리

- ## `@Secured`, `@PreAuthorize`, `@PostAuthorize`

Spring Security에서는 메서드 수준에서 접근 제어를 할 수 있도록 여러 애노테이션을 제공한다.
그중에서도 `@Secured`, `@PreAuthorize`, `@PostAuthorize`는
메서드 호출 전/후에 권한이나 조건을 검사하기 위해 사용된다.

이 세 가지는 목적은 비슷하지만 **지원 방식, 표현력, 사용 시기**가 다르며,
실무에서 보안 로직을 선언적으로 작성할 수 있게 해주는 중요한 기능이다.

📘 1. 개요 비교

애노테이션	사용 시점	SpEL 지원	특징
`@Secured`	호출 **전**	❌	간단한 **역할(Role)** 기반 제어
`@PreAuthorize`	호출 **전**	✅	SpEL 기반, 다양한 조건 지원
`@PostAuthorize`	호출 **후**	✅	반환 값 기반 인가 가능

> 📌 SpEL (Spring Expression Language) 사용 여부가 큰 차이점

🔐 2. 공통 전제: 설정 필요

Spring Security 6+에서는 반드시 아래 어노테이션 활성화 필요:

```
@Configuration
@EnableMethodSecurity(securedEnabled = true, prePostEnabled = true)
public class SecurityConfig { }
```

- `securedEnabled = true` → `@Secured` 사용 가능
- `prePostEnabled = true` → `@PreAuthorize`, `@PostAuthorize` 사용 가능

✅ 3. `@Secured`

개요

- 단순한 Role 기반 인가에 적합
- 문자열 배열로 `"ROLE_USER"` 등 지정

예제

```
@Secured("ROLE_ADMIN")
public void deleteUser(Long id) { ... }

@Secured({"ROLE_USER", "ROLE_MANAGER"})
public void viewDashboard() { ... }
```

→ 메서드 호출 전에 현재 사용자의 권한 목록 중 하나라도 매칭되면 통과

✅ 4. `@PreAuthorize`

개요

- 메서드 **호출 전에 인가 조건 판단**
- **SpEL(Spring Expression Language)** 사용 가능
- 인자, 인증 정보, 권한 등 복합 조건 작성 가능

예제

```
@PreAuthorize("hasRole('ADMIN')")
public void createUser(UserDto dto) { ... }

@PreAuthorize("#id == authentication.principal.id")
public void updateUser(Long id, UserDto dto) { ... }

@PreAuthorize("hasAuthority('SCOPE_read')")
public void readScope() { ... }
```

✅ 5. `@PostAuthorize`

개요

- 메서드 **호출 후에 반환 값을 기준으로 인가 판단**
- 주로 **결과 기반 보안 판단** 시 사용 (예: 자신의 데이터만 보기)

예제

```
@PostAuthorize("returnObject.owner == authentication.name")
public Document getDocument(Long id) {
    return documentService.findById(id);
}
```

→ 문서 반환 후에 **반환 객체의 소유자가 현재 사용자와 같아야 허용**

🔧 6. SpEL에서 사용 가능한 키워드

표현	설명
`authentication`	현재 인증 정보 (Authentication 객체)
`authentication.name`	로그인한 사용자 ID
`authentication.authorities`	사용자의 권한 목록
`#paramName`	메서드 파라미터 접근
`returnObject`	반환된 객체 (`@PostAuthorize`에서 사용)

🧠 7. 실무 전략 비교

상황	추천 애노테이션
단순한 Role 기반 제어	`@Secured`
인자 기반 조건 (ID 비교 등)	`@PreAuthorize`
리턴 객체 조건 비교	`@PostAuthorize`
OAuth2 Scope 기반 제어	`@PreAuthorize("hasAuthority('SCOPE_xxx')")`
복합 조건 판단	`@PreAuthorize` + SpEL 사용

📌 8. 요약 비교표

항목	`@Secured`	`@PreAuthorize`	`@PostAuthorize`
사용 시점	호출 전	호출 전	호출 후
SpEL 지원	❌	✅	✅
권한 표현	문자열 (`ROLE_`)	자유로운 SpEL 사용 가능	SpEL로 리턴 값 분석 가능
실무 사용도	낮음	높음	중간 (특수 목적)

커스텀 인증 및 인가

Spring Security에서 **커스텀 인증(Authentication)** 및 **커스텀 인가(Authorization)**는
기본 제공되는 ID/PW 기반 인증과 `hasRole`, `hasAuthority` 같은 단순 권한 체크로는 부족할 때 사용된다.
복잡한 로그인 로직, 외부 시스템 연동, 세분화된 접근 제어를 요구하는 실무에서는
직접 인증 로직과 인가 로직을 작성하여 Spring Security의 내부 구조에 맞게 통합해야 한다.

📘 1. 커스텀 인증(Authentication) 구성 요소

구성 요소	설명
`AuthenticationFilter`	요청 가로채서 인증 객체 생성 (예: ID/PW, 토큰 등)
`AuthenticationManager`	인증 객체를 받아서 인증 시도
`AuthenticationProvider`	실제 인증 로직 수행
`UserDetailsService`	사용자 조회
`PasswordEncoder`	비밀번호 검증 또는 무력화 가능
`SecurityContextHolder`	인증 성공 후 결과 저장소

🧩 2. 커스텀 인증 흐름 구조

```
클라이언트 요청 (ex: /login, /api/login)
  ↓
[1] 커스텀 필터(AuthenticationFilter) 등록
  ↓
[2] AuthenticationManager에 인증 요청
  ↓
[3] AuthenticationProvider에서 사용자 조회 및 인증
  ↓
[4] 성공 시 SecurityContext에 저장
```

🔧 3. 커스텀 인증 필터 구현

```
public class CustomAuthenticationFilter extends UsernamePasswordAuthenticationFilter {

    public CustomAuthenticationFilter(AuthenticationManager authenticationManager) {
        super.setAuthenticationManager(authenticationManager);
        setFilterProcessesUrl("/custom-login"); // 로그인 URL 변경 가능
    }

    @Override
    public Authentication attemptAuthentication(HttpServletRequest request,
HttpServletResponse response)
        throws AuthenticationException {
```

```
11
12          String username = request.getParameter("username");
13          String password = request.getParameter("password");
14
15          Authentication token = new UsernamePasswordAuthenticationToken(username,
    password);
16          return this.getAuthenticationManager().authenticate(token);
17      }
18  }
```

→ 이 필터는 `/custom-login`으로 들어온 로그인 요청을 가로채어 `UsernamePasswordAuthenticationToken`을 만들고 `AuthenticationManager`에 위임함

🔐 4. 커스텀 `AuthenticationProvider` 구현

```
1   @Component
2   public class CustomAuthenticationProvider implements AuthenticationProvider {
3
4       @Autowired
5       private UserDetailsService userDetailsService;
6
7       @Autowired
8       private PasswordEncoder passwordEncoder;
9
10      @Override
11      public Authentication authenticate(Authentication authentication) {
12          String username = authentication.getName();
13          String password = authentication.getCredentials().toString();
14
15          UserDetails user = userDetailsService.loadUserByUsername(username);
16
17          if (!passwordEncoder.matches(password, user.getPassword())) {
18              throw new BadCredentialsException("비밀번호가 일치하지 않습니다.");
19          }
20
21          return new UsernamePasswordAuthenticationToken(user, null,
    user.getAuthorities());
22      }
23
24      @Override
25      public boolean supports(Class<?> authentication) {
26          return
    UsernamePasswordAuthenticationToken.class.isAssignableFrom(authentication);
27      }
28  }
```

🛡️ 5. 커스텀 인가(Authorization)

✅ 1. 표현식 기반 인가 확장

```
@Component("customPermissionEvaluator")
public class CustomPermissionEvaluator implements PermissionEvaluator {

    @Override
    public boolean hasPermission(Authentication auth, Object target, Object permission)
{
        // 로직 예: 해당 사용자가 특정 게시물의 소유자인지 확인
        return true;
    }

    @Override
    public boolean hasPermission(Authentication auth, Serializable targetId, String
targetType, Object permission) {
        return false;
    }
}
```

→ SpEL에서 사용:

```
@PreAuthorize("@customPermissionEvaluator.hasPermission(authentication, #doc, 'write')")
```

✅ 2. `AuthorizationManager` 기반 커스터마이징 (Spring Security 6+)

```
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/admin/**").access(this::checkAdmin)
            .anyRequest().permitAll()
        );
    return http.build();
}

private AuthorizationDecision checkAdmin(Authentication auth, HttpServletRequest req) {
    boolean allowed = auth != null && auth.getAuthorities().stream()
        .anyMatch(granted -> granted.getAuthority().equals("ROLE_ADMIN"));
    return new AuthorizationDecision(allowed);
}
```

🧠 6. 실무 전략 요약

목적	커스터마이징 위치
로그인 파라미터 구조 변경	`AuthenticationFilter`
외부 인증 시스템 연동	`AuthenticationProvider`
ID/PW 아닌 토큰 기반 인증	필터 교체 + 커스텀 토큰 인증
응답 JSON 커스터마이징	`AuthenticationSuccessHandler`, `FailureHandler`
권한 체크 로직 복잡함	`PermissionEvaluator` 또는 `AuthorizationManager`

📌 7. 요약

항목	설명
커스텀 인증 필터	로그인 요청을 직접 처리하는 필터
AuthenticationProvider	인증 로직(비밀번호 검증 등) 담당
인가 표현 확장	SpEL + `@PreAuthorize` + 커스텀 Bean
인증 성공 저장 위치	`SecurityContextHolder`
실무 사용 예시	ID/PW 로그인, JWT 토큰, API 키 인증 등

암호화

- # PasswordEncoder, BCrypt

  `PasswordEncoder`는 Spring Security에서 **비밀번호를 안전하게 암호화하고 검증하는 인터페이스**이다.
  그중 `BCryptPasswordEncoder`**는 가장 널리 사용되는 구현체**로,
  보안성과 실전성을 고려한 **단방향 해시 방식**을 제공한다.

  ### 📘 1. `PasswordEncoder`란?

  Spring Security에서 비밀번호를 처리하는 **표준 인터페이스**로,

  ```
  public interface PasswordEncoder {
      String encode(CharSequence rawPassword);
      boolean matches(CharSequence rawPassword, String encodedPassword);
  }
  ```

  - `encode()` : 비밀번호를 **단방향 해시로 암호화**
  - `matches()` : 사용자가 입력한 비밀번호와 저장된 해시값을 비교

🔐 2. `BCryptPasswordEncoder`란?

항목	설명
알고리즘	**bcrypt** (Blowfish 기반 단방향 해시 함수)
특징	**Salt 자동 생성**, 반복 연산(cost factor) 설정 가능
장점	Rainbow Table 공격 방지, 느린 해시 → brute-force 저항
단점	해시 결과 길이 김 (~60자), 속도 느림 (의도된 특징)

⚙️ 3. Spring Boot 설정 예시

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder(); // 기본 strength = 10
}
```

→ Spring Security는 `UserDetailsService`와 함께 이 인코더를 자동으로 사용

🔄 4. 비밀번호 저장 및 검증 흐름

✅ 비밀번호 저장 시

```
String rawPassword = "1234";
String encoded = passwordEncoder.encode(rawPassword);
// 결과: $2a$10$GnUePkW...
```

✅ 로그인 검증 시

```
passwordEncoder.matches("1234", "$2a$10$GnUePkW...")
→ true 반환
```

> ✅ **암호화된 결과는 항상 다름**
> 같은 비밀번호라도 **매번 다른 salt**가 붙기 때문에 hash값이 매번 바뀜

🧪 5. BCrypt 해시 결과 구조

```
$2a$10$EGL3kShs7aZ6SvL3Kckzj.fEQIVz6KDHR1yzXnE9pPZPQqW06.Qmy
 ↑   ↑        ↑
 |   |        └── 해시된 결과
 |   └─────────── cost factor (2^10 = 1024회 반복)
 └─────────────── 알고리즘 버전 (2a, 2b 등)
```

🔐 6. `BCryptPasswordEncoder` 생성자

```
new BCryptPasswordEncoder(); // 기본 cost = 10
new BCryptPasswordEncoder(12); // 더 느리게, 더 안전하게
```

> ✅ cost factor는 연산 횟수를 의미
> 값이 클수록 보안 강하지만 인증 속도는 느려짐 (12~14 실무 권장)

🧠 7. 실무 전략

항목	전략
비밀번호는 반드시 해시 저장	절대 평문 저장 금지
BCrypt 사용 강력 추천	현재 기준 가장 안전하고 검증된 방식
Salt 수동 관리 ❌	`BCryptPasswordEncoder`는 자동 관리
사용자 등록 시 `encode()` 호출	비밀번호 저장 시 반드시 암호화
로그인 시 `matches()` 사용	평문과 해시 비교는 직접 하지 않음
연산 비용(cost) 높이기	서버 성능 허용 시 `12~14` 정도 권장

⚠️ 8. 잘못된 사용 사례 (피해야 할 것)

- `equals()`로 비밀번호 비교
  → `matches()`를 반드시 사용
- 해시를 두 번 이상 하는 방식
  → `encode(encode(password))`는 안됨
- Salt 없이 SHA-256 등 일반 해시 사용
  → 빠르고 예측 가능 → 공격 취약

📌 9. 요약

항목	설명
목적	비밀번호 암호화 및 검증
기본 구현체	`BCryptPasswordEncoder`
알고리즘 특징	Salt + 반복 연산 기반 해시
encode() 결과 특징	비밀번호 같아도 해시값 매번 다름
실무 권장 사항	`matches()`, cost factor 조정, 저장 전 반드시 encode()

세션 관리

Spring Security의 **세션 관리(session management)**는 인증된 사용자에 대해 **세션을 생성하고 유지하거나 제한하며, 보안 취약점(예: 세션 고정, 세션 탈취)을 방지**하는 데 중점을 둔다.
이는 특히 로그인 이후 인증 상태를 유지해야 하는 **Form 로그인, OAuth2 로그인 기반의 시스템에서 매우 중요**하다.

📘 1. 세션(Session)이란?

HTTP는 기본적으로 **무상태(stateless)**이므로,
사용자 인증 상태를 유지하려면 서버에 **세션을 생성하여 사용자 정보를 저장**해야 함.

구성 요소	설명
`HttpSession`	사용자 상태(인증 정보 등)를 저장하는 객체
`JSESSIONID` 쿠키	클라이언트가 세션을 식별하기 위한 키
`SecurityContext`	인증된 사용자 정보를 담은 객체 (SecurityContextHolder에 저장됨)

🔁 2. 기본 동작 (Form 로그인 기반)

```
1. 로그인 성공 → Authentication 객체 생성
2. SecurityContextHolder에 저장
3. HttpSession에 SecurityContext 저장
4. 이후 요청마다 JSESSIONID로 사용자 상태 확인
```

> ✅ 즉, Spring Security는 기본적으로 **세션 기반 인증을 제공**하며, 이 때 사용자의 인증 정보를 세션에 저장함.

⚙️ 3. 세션 관리 설정 예시

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .sessionManagement(session -> session
            .sessionFixation().migrateSession()     // 세션 고정 보호
            .maximumSessions(1)                     // 동시 로그인 1회 제한
            .maxSessionsPreventsLogin(false)        // 이전 세션 만료 처리
            .expiredUrl("/login?expired=true")      // 만료 시 이동할 URL
        );
    return http.build();
}
```

🔐 4. 주요 세션 보안 옵션 설명

✅ 세션 고정 공격 방지 (Session Fixation Protection)

설정	설명
`.sessionFixation().migrateSession()`	로그인 시 새로운 세션 ID 생성 (기본값)
`.sessionFixation().none()`	세션 ID 그대로 유지 (취약점 발생 가능)
`.sessionFixation().newSession()`	새 세션만 생성, 기존 속성도 삭제

> ✅ 기본은 `migrateSession()`으로 충분 (세션 ID만 변경, 속성 유지)

✅ 동시 로그인 제한 (Concurrent Session Control)

설정	설명
`.maximumSessions(n)`	한 명의 사용자가 n개까지 로그인 허용
`.maxSessionsPreventsLogin(true)`	n개 초과 시 새 로그인 거부 (기존 세션 유지)
`.maxSessionsPreventsLogin(false)`	n개 초과 시 기존 세션 무효화 (기본)

→ 보통 보안이 중요한 시스템은 `1`로 설정 + `true` 옵션 추가

✅ 세션 만료 시 처리

```
.expiredUrl("/login?expired=true")
```

→ 기존 세션이 무효화되었을 때 사용자를 리디렉션할 URL 지정

💣 5. 세션 만료 처리 시 주의사항

- 세션이 만료되면 `SecurityContext`도 사라짐
- 이때 인증되지 않은 요청은 `AuthenticationEntryPoint`로 전달됨
- 필요 시 커스텀 처리 가능 (예: JSON 응답)

📜 6. 커스텀 세션 이벤트 핸들링

✅ 세션 만료 감지 리스너

```
@Component
public class SessionExpiredListener implements SessionInformationExpiredStrategy {
    @Override
```

```
    public void onExpiredSessionDetected(SessionInformationExpiredEvent event) throws
IOException {
        HttpServletResponse response = event.getResponse();
        response.sendRedirect("/login?expired=true");
    }
}
```

→ `.expiredSessionStrategy(new SessionExpiredListener())`로 등록 가능

🧠 7. 실무 전략 요약

목적	전략
세션 탈취 방지	`migrateSession()` 사용 (기본값)
보안 강화용 동시 로그인 제한	`maximumSessions(1)` + 이전 세션 만료
세션 만료 시 UX 고려	`expiredUrl()` 또는 커스텀 핸들러
REST API 시스템 (JWT 등)	`.sessionCreationPolicy(STATELESS)` 로 세션 제거
세션 저장소 분산화 필요 시	Redis 기반 `Spring Session` 사용

📌 8. 요약

항목	설명
세션 사용 목적	인증 정보 유지
기본 저장 위치	`HttpSession` → `JSESSIONID` 쿠키
고정 공격 방지	`migrateSession()` (기본값)
동시 로그인 제한	`maximumSessions(n)` 사용
만료 시 이동 경로	`.expiredUrl(...)` 설정 가능

Remember-me 기능

Spring Security의 **Remember-me** 기능은 사용자가 로그인 후 **브라우저를 닫았다가 다시 열어도 인증 상태를 유지**할 수 있도록 쿠키 기반으로 인증 정보를 저장해주는 기능이다.
주로 **"로그인 유지" 체크박스**처럼 사용되며, 인증을 세션에만 의존하지 않게 해주는 중요한 확장이다.

📘 1. Remember-me란?

> 사용자가 세션 만료나 브라우저 재시작 후에도 **쿠키를 통해 인증 상태를 복원**할 수 있게 해주는 Spring Security 기능

특징	설명
인증 지속성	세션 없이도 로그인 유지 가능
쿠키 기반	인증 정보를 쿠키에 저장
단기 세션 + 장기 쿠키 조합	UX 향상 목적
자동 로그인 위험 존재	보안 설정 주의 필요

🔄 2. 인증 흐름 요약

```
① 로그인 성공 + remember-me 요청
  ↓
② 서버가 remember-me 쿠키 생성 (자동 로그인 토큰 포함)
  ↓
③ 브라우저 재접속 시 → remember-me 쿠키 존재 확인
  ↓
④ 유효성 검증 후 → 자동 인증 처리 → SecurityContext에 저장
```

🔐 3. Remember-me 동작 방식: 2가지 모드

방식	설명
✅ **Token 기반 (기본값)**	username + 만료시간 + secret key + 해시 → 쿠키 저장
✅ **Persistent 기반**	DB에 토큰 저장 + 쿠키에는 토큰 키만 저장 → 더 안전

⚙️ 4. 기본 설정 예시 (Token-based)

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .formLogin(Customizer.withDefaults())
        .rememberMe(remember -> remember
            .key("secure-and-random-key")
            .rememberMeParameter("remember-me")  // 체크박스 name
            .tokenValiditySeconds(60 * 60 * 24 * 7)  // 7일
        );
    return http.build();
}
```

✅ 로그인 HTML 예시

```
<form method="post" action="/login">
  <input type="text" name="username" />
  <input type="password" name="password" />
  <label><input type="checkbox" name="remember-me" /> 로그인 유지</label>
  <button type="submit">로그인</button>
</form>
```

🧩 5. Persistent Token 기반 설정 (DB 저장 방식)

✅ DB 테이블 생성

```
CREATE TABLE persistent_logins (
  username VARCHAR(64) NOT NULL,
  series VARCHAR(64) PRIMARY KEY,
  token VARCHAR(64) NOT NULL,
  last_used TIMESTAMP NOT NULL
);
```

✅ 설정 코드

```
@Bean
public PersistentTokenRepository tokenRepository(DataSource dataSource) {
    JdbcTokenRepositoryImpl repo = new JdbcTokenRepositoryImpl();
    repo.setDataSource(dataSource);
    return repo;
}

@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .rememberMe(remember -> remember
            .tokenRepository(tokenRepository(dataSource))
            .tokenValiditySeconds(60 * 60 * 24 * 14)  // 14일
            .userDetailsService(userDetailsService)
        );
    return http.build();
}
```

→ Persistent 방식은 **쿠키 탈취에 더 안전**, DB에서 토큰 관리 가능

🧠 6. 실무 전략

목적	전략
로그인 유지 UX 제공	기본 토큰 방식 사용 가능

목적	전략
보안 강화 필요	Persistent + DB 저장 방식 사용
장기 유지 제한	`tokenValiditySeconds` 짧게 조절
쿠키 탈취 대응	HTTPS + HttpOnly + Secure 설정
사용자 로그아웃 시 쿠키 삭제 필요	`logout().deleteCookies("remember-me")` 설정

📓 7. 커스텀 Remember-me 서비스 구현 (Optional)

```
public class CustomRememberMeService extends PersistentTokenBasedRememberMeServices {

    public CustomRememberMeService(String key, UserDetailsService userDetailsService,
                                   PersistentTokenRepository tokenRepository) {
        super(key, userDetailsService, tokenRepository);
    }

    @Override
    protected void onLoginSuccess(...) {
        // 커스텀 로직 가능 (예: 토큰 발행 알림 등)
    }
}
```

🔒 8. 보안 주의사항

위험 요소	대응 방안
쿠키 탈취 가능성	**HTTPS**, `Secure`, `HttpOnly` 설정
오래된 토큰 재사용	만료 시간 단축 + Logout 시 삭제
동일 토큰 재사용	Persistent 방식은 새 토큰 발급함 (추천)

📌 9. 요약

항목	설명
목적	로그인 유지 쿠키 제공
기본 동작	Token 기반 쿠키 생성
보안 강화 방식	Persistent 방식 + DB 토큰 저장
설정 핵심	`rememberMe().key()`, `tokenValiditySeconds()`
실무 권장	7~14일 유지 + 로그아웃 시 쿠키 삭제 + Secure 설정

CSRF 보호

Spring Security는 **CSRF(Cross-Site Request Forgery, 사이트 간 요청 위조)** 공격으로부터
웹 애플리케이션을 보호하기 위해 **기본적으로 CSRF 보호 기능을 활성화**한다.
이 기능은 **악의적인 제3자가 사용자의 인증 정보를 도용해 의도하지 않은 요청을 보내는 행위**를 막는 데 중점을 둔다.

📘 1. CSRF란?

> 인증된 사용자의 권한을 악용하여 **다른 사이트에서 사용자 몰래 요청을 보내는 공격**
> (예: 로그인된 상태에서 악성 스크립트가 `/transfer` 요청을 자동 제출)

✅ 예시 (공격 시나리오)

```
<!-- 악성 사이트에 심어진 폼 -->
<form method="post" action="https://bank.com/transfer">
  <input type="hidden" name="to" value="attacker" />
  <input type="hidden" name="amount" value="1000000" />
</form>
<script>document.forms[0].submit();</script>
```

→ 사용자가 로그인된 상태라면 이 요청은 **자신도 모르게 실행됨**

🔐 2. Spring Security의 CSRF 보호 원리

항목	설명
보호 대상 HTTP 메서드	`POST`, `PUT`, `DELETE`, `PATCH` 등 **상태를 변경하는 요청**
보호 방법	요청마다 **CSRF 토큰을 검증**하여 위조 여부 확인
토큰 저장 위치	서버: `HttpSession` 또는 `CsrfTokenRepository`
토큰 전달 방식	클라이언트: **폼 필드 또는 HTTP 헤더**로 전송

🧪 3. 기본 동작

Spring Security는 다음 방식으로 동작함:

```
① 서버가 요청한 HTML에 CSRF 토큰 삽입 (폼 or 메타태그)
② 클라이언트가 이 토큰을 함께 제출
③ 서버는 요청의 토큰과 세션 저장 토큰을 비교
④ 일치하면 요청 허용, 아니면 403 Forbidden
```

📄 4. CSRF 토큰 HTML 폼 삽입 예시 (Thymeleaf)

```
<form th:action="@{/update}" method="post">
    <input type="text" name="data" />
    <input type="hidden" th:name="${_csrf.parameterName}" th:value="${_csrf.token}" />
    <button type="submit">전송</button>
</form>
```

→ `${_csrf}` 객체는 Spring이 자동으로 View에 주입

⚙️ 5. 설정 예시

✅ 기본 설정 (보호 활성, 생략 가능)

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .csrf(Customizer.withDefaults()) // 또는 생략
        .authorizeHttpRequests(auth -> auth
            .anyRequest().authenticated()
        );
    return http.build();
}
```

✅ 비활성화 (주의!)

```
http.csrf(csrf -> csrf.disable());
```

→ REST API or 외부 토큰 기반 인증(JWT 등)에서 사용 가능
→ 반드시 **정당한 이유**가 있을 때만 비활성화할 것

🧩 6. REST API에서의 CSRF 처리

✅ 1. 완전히 비활성화 (JWT 사용 시 권장)

```
http.csrf(csrf -> csrf.disable());
http.sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
```

✅ 2. CSRF 토큰을 헤더로 전송

서버:

```
http
    .csrf(csrf -> csrf
        .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
    );
```

클라이언트 (JS):

```
fetch('/update', {
  method: 'POST',
  headers: {
    'X-CSRF-TOKEN': csrfToken
  },
  body: JSON.stringify(data)
});
```

🧠 7. 실무 전략

상황	전략
HTML Form 기반 웹 앱	CSRF 기본값 그대로 사용
JS 기반 SPA + 서버 세션 사용	`CookieCsrfTokenRepository` 로 쿠키 전달
JWT 기반 REST API	`csrf().disable()` + `STATELESS` 모드
테스트 시 403 오류 발생	CSRF 토큰 미포함 여부 확인 필요
커스텀 처리 필요	`CsrfTokenRepository` 직접 구현 가능

📌 8. 요약

항목	설명
기본 보호 대상	POST, PUT, DELETE, PATCH 요청
클라이언트 전달 방식	hidden input or `X-CSRF-TOKEN` 헤더
토큰 저장 위치	세션 or 쿠키
비활성화 필요 시	JWT 사용, API 서버, 외부 인증 구조
실무 권장	HTML Form은 활성화 유지, REST API는 필요시 비활성화

CORS 설정

📘 1. CORS란?

> Cross-Origin Resource Sharing
> **다른 Origin의 리소스를 요청하는 것을 허용할 것인지 결정하는 정책**

✅ Origin이란?

```
Origin = [프로토콜 + 도메인 + 포트]

예:
http://localhost:3000 ≠ http://localhost:8080
```

→ 백엔드가 `localhost:8080`인데, 프론트가 `localhost:3000`에서 요청하면 **CORS 오류 발생**

🔁 2. CORS 요청 흐름

```
1. 프론트에서 다른 Origin으로 요청
2. 브라우저가 먼저 Preflight 요청 (OPTIONS)
3. 서버가 응답 헤더로 허용 여부 명시 (Access-Control-Allow-Origin 등)
4. 허용되면 실제 요청 진행, 아니면 CORS 에러 발생
```

🧩 3. Spring에서의 CORS 설정 위치

설정 위치	설명
Spring MVC (`@CrossOrigin`)	Controller 수준 or 전역 CORS
Spring Security (`CorsConfigurationSource`)	인증 필터보다 먼저 CORS 허용 필요

> 📌 Spring Security를 사용하는 경우 반드시 **SecurityFilterChain에 CORS 허용 코드 추가**해야 실제 동작함.

✅ 4. Spring Security + MVC 통합 CORS 설정 예시

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .cors(Customizer.withDefaults()) // Security가 cors 필터를 등록하게 함
        .csrf(csrf -> csrf.disable())
        .authorizeHttpRequests(auth -> auth
            .anyRequest().permitAll()
        );
    return http.build();
}

@Bean
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration config = new CorsConfiguration();
    config.setAllowedOrigins(List.of("http://localhost:3000")); // 허용할 origin
    config.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE"));
    config.setAllowedHeaders(List.of("*"));
    config.setAllowCredentials(true); // 쿠키 허용 시 필요

```

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

📜 5. 컨트롤러 단위 설정: `@CrossOrigin`

```
@CrossOrigin(origins = "http://localhost:3000")
@RestController
public class ApiController {
    @GetMapping("/data")
    public String getData() {
        return "ok";
    }
}
```

→ 이 방식은 보안 필터 이전에 작동하므로 **Spring Security와 함께 쓸 때는 효과 없음**
→ **단순한 테스트용**으로만 권장

🔒 6. CORS 헤더 요약

헤더	설명
`Access-Control-Allow-Origin`	허용할 origin 지정 (`*`, 특정 도메인)
`Access-Control-Allow-Methods`	허용할 HTTP 메서드
`Access-Control-Allow-Headers`	허용할 요청 헤더 (`Content-Type`, `Authorization` 등)
`Access-Control-Allow-Credentials`	쿠키/인증정보 전달 허용 여부 (`true`)
`Access-Control-Max-Age`	preflight 결과를 캐시할 시간(초)

🧠 7. 실무 전략

상황	권장 설정
로컬 개발: React ↔ Spring Boot	`http://localhost:3000` 만 허용
운영 환경: 도메인 분리	실제 도메인만 명시 (`https://myapp.com`)
REST API에서 쿠키 인증 사용	`Allow-Credentials: true` + `Allowed-Origin`에 `*` 사용 금지
모든 요청 허용 (테스트용만)	`config.setAllowedOrigins(List.of("*"));` 단, `credentials=false` 필수

⚠️ 8. 주의사항

문제 상황	원인
CORS 설정했는데 여전히 403	**Spring Security 필터 체인에서 cors() 생략**됨
쿠키가 전달되지 않음	`credentials=true` + 클라이언트의 `withCredentials=true` 필요
`Access-Control-Allow-Origin` 헤더 없음	설정이 누락되었거나 origin이 다름

📌 9. 요약

항목	설명
설정 위치	Spring Security 필터 체인 + `CorsConfigurationSource`
브라우저 요청 처리	Preflight → 응답 헤더로 허용 여부 판단
보안 설정 핵심	`allowCredentials`, `allowedOrigins`, `allowedMethods` 조합
실무 권장 전략	도메인 명시 + 인증 필요 시 Credentials true 설정

JWT 기반 인증

JWT(Json Web Token) 기반 인증은 Spring Security에서 **세션을 사용하지 않고 클라이언트에 토큰을 발급해 인증 상태를 유지**하는 방식이다.
RESTful API 서버에서는 세션을 상태 없이 유지하기 위해, **Spring Security의 기본 세션 인증 대신 JWT 토큰 방식**을 선호한다.

📘 1. JWT란?

> 클라이언트와 서버 간 **서버 인증 정보를 안전하게 전달하는 방식**으로,
> 자체 서명된 **JSON 기반 토큰**

✅ 구조

JWT는 `.`으로 구분된 3부분으로 구성됨:

```
[header].[payload].[signature]
```

예:

```
eyJhbGciOiJIUzI1NiJ9.       // Header: 알고리즘, 타입
eyJ1c2VySWQiOjEsInJvbGUiOiJVU0VSIn0.  // Payload: 사용자 정보
aBc123...                   // Signature: 서명
```

🔐 2. Spring Security에서의 JWT 인증 흐름

```
① 사용자가 로그인 요청 (ID, PW)
② 인증 성공 → JWT 토큰 발급
③ 이후 요청 시 HTTP Header에 JWT 포함
④ JWTAuthenticationFilter가 요청을 가로채고 토큰 검증
⑤ 성공 시 SecurityContext에 인증 정보 저장
⑥ 이후 컨트롤러는 인증 정보에 접근 가능
```

⚙️ 3. JWT 구성 필수 요소

항목	설명
비밀 키 (Secret)	토큰 서명을 위한 키 (`HS256`, `RS256` 등)
토큰 유효 시간	`exp` (만료 시간) 필드 포함
클레임(Claim)	사용자 ID, 권한 등 포함 가능
서명(Signature)	위변조 여부 확인

🧩 4. JWT 발급 예제 (로그인 시)

```
String token = Jwts.builder()
    .setSubject(username)
    .claim("role", "USER")
    .setIssuedAt(new Date())
    .setExpiration(new Date(System.currentTimeMillis() + 3600_000)) // 1시간
    .signWith(SignatureAlgorithm.HS256, secretKey)
    .compact();
```

→ 클라이언트는 이 토큰을 **Authorization 헤더에 추가**해서 사용

```
Authorization: Bearer <token>
```

📜 5. JWTAuthenticationFilter 구현 (요청 필터링)

```
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final String secretKey = "my-secure-secret";

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws ServletException,
IOException {

```

```
        String header = request.getHeader("Authorization");
        if (header == null || !header.startsWith("Bearer ")) {
            filterChain.doFilter(request, response);
            return;
        }

        String token = header.replace("Bearer ", "");
        try {
            Claims claims = Jwts.parser()
                .setSigningKey(secretKey)
                .parseClaimsJws(token)
                .getBody();

            String username = claims.getSubject();
            List<GrantedAuthority> authorities = List.of(new
SimpleGrantedAuthority("ROLE_USER"));

            Authentication auth = new UsernamePasswordAuthenticationToken(username,
null, authorities);
            SecurityContextHolder.getContext().setAuthentication(auth);

        } catch (JwtException e) {
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
            return;
        }

        filterChain.doFilter(request, response);
    }
}
```

🔧 6. Spring Security 설정에 필터 등록

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .csrf(csrf -> csrf.disable())
        .sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/login", "/public/**").permitAll()
            .anyRequest().authenticated()
        )
        .addFilterBefore(new JwtAuthenticationFilter(),
UsernamePasswordAuthenticationFilter.class);
    return http.build();
}
```

→ 세션을 완전히 비활성화하고, 인증은 JWT로만 처리

🧠 7. 실무 전략

항목	전략
토큰 유효기간 설정	너무 길게 설정하면 위험, 너무 짧으면 UX 저하
Refresh Token 분리	Access + Refresh 구조로 관리 (재발급 처리)
HTTPS 적용 필수	토큰 탈취 방지
로그아웃 처리	토큰은 무상태이므로, 블랙리스트 방식 고려
인증 실패 시 응답 처리	`AuthenticationEntryPoint`로 401 JSON 응답 구현
비밀 키 관리	`.properties`, Vault, 환경변수로 외부화

📌 8. 요약

항목	설명
인증 방식	세션 X, JWT O
요청 처리 필터	`JwtAuthenticationFilter`
SecurityContext 저장	인증 성공 시 직접 설정
헤더 사용 방식	`Authorization: Bearer <token>`
Stateless 설정	`SessionCreationPolicy.STATELESS` 필수

OAuth2 클라이언트 및 서버

Spring Security에서의 **OAuth 2.0 클라이언트와 리소스 서버 구성**은
외부 인증 서버(Google, Kakao, Naver 등) 또는 자체 인증 서버를 통해
인증(Authorization) 및 **리소스 접근 제어(Resource Access)**를 처리하기 위한 핵심 아키텍처이다.

OAuth2는 복잡하지만, Spring Security는 **클라이언트 및 서버 측 구현을 쉽게 할 수 있는 구성요소들을 제공**한다.

📘 1. OAuth2 아키텍처 핵심 구성요소

역할	설명
Authorization Server	토큰을 발급하는 인증 서버 (Google, 자체 서버 등)
Resource Server	보호된 API를 제공, 토큰 검증을 통해 접근 제어
Client	사용자를 대신해 리소스를 요청하는 애플리케이션
Resource Owner	사용자의 실제 계정 정보 및 권한 보유자
Access Token	권한을 부여받은 요청임을 증명하는 수단 (JWT 등)

🔄 2. 인증 흐름 (Authorization Code 방식 기준)

```
① Client → Authorization Server: 인증 요청 (/oauth2/authorize)
② 사용자 로그인 후 동의
③ Authorization Server → Client: Authorization Code 전달
④ Client → Authorization Server: Authorization Code로 Access Token 요청
⑤ Client → Resource Server: Access Token을 헤더에 담아 API 호출
⑥ Resource Server → Token 유효성 검증 후 응답
```

🧩 3. Spring Security - OAuth2 Client 구성

Spring Boot는 `spring-boot-starter-oauth2-client`를 통해 OAuth2 로그인 기능을 자동화함.

✅ 1. 의존성 추가

```
implementation 'org.springframework.boot:spring-boot-starter-oauth2-client'
```

✅ 2. application.yml 설정 (Google 예시)

```yaml
spring:
  security:
    oauth2:
      client:
        registration:
          google:
            client-id: xxx
            client-secret: yyy
            scope: profile, email
        provider:
          google:
            authorization-uri: https://accounts.google.com/o/oauth2/v2/auth
            token-uri: https://oauth2.googleapis.com/token
            user-info-uri: https://www.googleapis.com/oauth2/v3/userinfo
```

✅ 3. Security 설정

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/", "/login**").permitAll()
            .anyRequest().authenticated()
        )
        .oauth2Login(Customizer.withDefaults()); // 자동 로그인
    return http.build();
}
```

🛡️ 4. OAuth2 Resource Server 구성 (토큰 검증 API 서버)

✅ 1. 의존성 추가

```
implementation 'org.springframework.boot:spring-boot-starter-oauth2-resource-server'
implementation 'org.springframework.boot:spring-boot-starter-security'
```

✅ 2. application.yml 설정 (JWT 방식)

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          jwk-set-uri: https://authorization-server.com/oauth2/jwks
```

또는 직접 `public key` 등록 방식:

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          public-key-location: classpath:public.pem
```

✅ 3. Security 설정

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/public").permitAll()
            .anyRequest().authenticated()
        )
        .oauth2ResourceServer(oauth2 -> oauth2
            .jwt(Customizer.withDefaults())
        );
    return http.build();
}
```

🔐 5. JWT 기반 토큰 검증

토큰을 클라이언트가 헤더에 포함:

```
Authorization: Bearer <access_token>
```

→ Spring Security가 자동으로 `JwtDecoder`를 통해 서명 및 만료 여부 검증 후
`SecurityContext`에 `JwtAuthenticationToken` 저장

🧠 6. 실무 전략 요약

항목	전략
사용자 OAuth 로그인 처리	`spring-boot-starter-oauth2-client` 사용
자체 API 서버 보호	`oauth2-resource-server` + JWT 인증 사용
토큰 서명 방식	`HS256` (대칭) 또는 `RS256` (비대칭 키)
사용자 정보 매핑 커스터마이징	`OAuth2UserService` 재정의
클라이언트-서버 분리 대응	AccessToken → Authorization 헤더 사용

📌 7. 요약

역할	Spring 구성 요소
OAuth 클라이언트	`oauth2Login()` + `spring-boot-starter-oauth2-client`
리소스 서버	`oauth2ResourceServer()` + JWT 설정
사용자 인증 처리	Authorization Code Flow
토큰 전달 방식	`Authorization: Bearer <token>`
보안 검증	JWK, 서명, 만료 시간 확인 등 자동 수행