7. 스프링 보안(Spring Security)

인증(Authentication) vs 인가(Authorization)

인증(Authentication)과 **인가(Authorization)**는
웹 보안 및 애플리케이션 설계에서 **가장 기본적이고 중요한 개념**입니다.
두 용어는 혼동되기 쉽지만, **완전히 다른 목적**을 가지고 있으며,
Spring Security와 같은 보안 프레임워크에서 **별도로 처리됩니다.**

✅ 1. 정의 비교

개념	정의
인증 (Authentication)	**"너는 누구냐?"** 사용자의 신원을 확인하는 절차 (로그인 등)
인가 (Authorization)	**"너는 이걸 할 수 있냐?"** 인증된 사용자가 **무엇을 할 수 있는지** 결정하는 절차 (권한 검사 등)

✅ 2. 비유적 설명

- **인증**: 당신이 **누구인지 확인** (예: ID/PW로 문 열기)
- **인가**: 당신이 **무슨 권한이 있는지 결정** (예: VIP 라운지 입장 가능 여부)

✅ 3. 인증(Authentication) 상세

항목	설명
목적	사용자 확인 (로그인 성공 여부)
기준	자격 증명 (ID, 비밀번호, 토큰 등)
결과	SecurityContext에 `Authentication` 객체 저장
실패 시	`401 Unauthorized`

Spring Security 예:

```
UsernamePasswordAuthenticationToken auth =
    new UsernamePasswordAuthenticationToken(username, password);
authenticationManager.authenticate(auth);
```

✅ 4. 인가(Authorization) 상세

항목	설명
목적	접근 권한 부여/차단
기준	사용자에 부여된 권한(Role, Authority)
방식	URL, 메서드, 도메인 별 권한 검사
실패 시	`403 Forbidden`

Spring Security 예:

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

✅ 5. Spring Security에서의 구분

단계	처리 내용
인증(Authentication)	로그인 → Authentication 객체 생성 → SecurityContext에 저장
인가(Authorization)	요청 시 `AccessDecisionManager`가 권한 체크 수행

✅ 6. HTTP 상태 코드 차이

상태 코드	상황
`401 Unauthorized`	인증이 되지 않은 사용자 (로그인 실패, 토큰 없음 등)
`403 Forbidden`	인증은 되었지만 권한이 없는 사용자 (예: ROLE_USER로 관리자 API 요청)

✅ 7. JWT 기반 인증/인가 흐름 예

```
[클라이언트] → 로그인 요청
           → JWT 토큰 발급 받음

[클라이언트] → API 요청 시 Authorization: Bearer <token> 헤더 포함

[서버]
    1. 토큰을 디코드하여 사용자 확인 → 인증(Authentication)
    2. 사용자 권한 확인 → 인가(Authorization)
    3. 권한이 없으면 403 Forbidden
```

✅ 8. 예시 코드 비교

인증 처리 예 (Spring Security)

```
@PostMapping("/login")
public ResponseEntity<?> login(@RequestBody LoginDto dto) {
    Authentication authentication = authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(dto.getUsername(), dto.getPassword())
    );
    SecurityContextHolder.getContext().setAuthentication(authentication);
    return ResponseEntity.ok(tokenProvider.generateToken(authentication));
}
```

인가 처리 예

```
@PreAuthorize("hasRole('ADMIN')")
@GetMapping("/admin/users")
public List<User> getAllUsers() {
    return userService.findAll();
}
```

✅ 9. 결론 요약

항목	인증(Authentication)	인가(Authorization)
질문	"너 누구야?"	"너 이거 해도 돼?"
목적	신원 확인	권한 확인
입력	ID, 비밀번호, 토큰 등	인증된 사용자에 부여된 권한
실패 시 상태 코드	401 Unauthorized	403 Forbidden
Spring 위치	`AuthenticationManager`	`AccessDecisionManager`, `@PreAuthorize`

👉 정리

- 로그인은 "인증"
- 권한 확인은 "인가"
- 둘은 **순차적**, **구분되며**, **협력함**

Spring Security 기본 구조

Spring Security는 Spring 기반 애플리케이션의 **인증(Authentication)**과 **인가(Authorization)**를 제공하는 **보안 프레임워크**입니다.
웹 보안, 메서드 보안, JWT 인증, 세션 기반 인증 등 모든 보안 요구 사항을 통합적으로 처리할 수 있도록 설계되어 있습니다.

아래는 Spring Security의 **기본 구조, 핵심 흐름, 필터 체인, 주요 컴포넌트**에 대한 정리입니다.

✅ 1. Spring Security 전체 구조 흐름

```
[HTTP 요청]
   ↓
Spring Security Filter Chain (보안 필터들)
   ↓
인증 처리(Authentication)
   ↓
인가 처리(Authorization)
   ↓
Controller, Service 등 실제 로직 실행
```

✅ 2. 핵심 컴포넌트

컴포넌트	역할
`SecurityFilterChain`	모든 보안 관련 필터를 포함한 체인
`AuthenticationManager`	로그인 인증을 처리하는 중심 객체
`AuthenticationProvider`	실제 인증을 수행 (DB 조회 등)
`UserDetailsService`	사용자 정보를 로딩하는 서비스
`PasswordEncoder`	비밀번호 암호화 및 검증
`SecurityContextHolder`	인증된 사용자 정보 저장소
`AccessDecisionManager`	인가 결정 처리

✅ 3. 인증(Authentication) 흐름 요약

```
1. 사용자가 로그인 요청 (POST /login)
2. UsernamePasswordAuthenticationFilter가 요청 가로챔
3. AuthenticationManager → AuthenticationProvider 호출
4. UserDetailsService로 사용자 정보 조회
5. 비밀번호 검증 후 Authentication 객체 생성
6. SecurityContextHolder에 Authentication 저장
```

✅ 4. 인가(Authorization) 흐름 요약

```
1. 인증된 사용자의 Authentication에서 권한(Role)을 꺼냄
2. 요청한 URL 또는 메서드가 허용된 권한인지 판단
3. 허용: 컨트롤러 진입 / 거부: 403 Forbidden 반환
```

✅ 5. Spring Security Filter Chain

Spring Security는 다음과 같은 **수십 개의 필터를 순차적으로 실행**합니다:

필터	설명
`SecurityContextPersistenceFilter`	인증 정보의 세션 저장/복원
`UsernamePasswordAuthenticationFilter`	로그인 처리 (기본 폼 로그인)
`BasicAuthenticationFilter`	HTTP Basic 인증 처리
`BearerTokenAuthenticationFilter`	JWT 토큰 인증
`ExceptionTranslationFilter`	예외 처리 및 403/401 반환
`FilterSecurityInterceptor`	인가 처리 수행 (권한 검사)

→ 가장 중요한 필터는 `UsernamePasswordAuthenticationFilter` (인증)
→ 가장 마지막은 `FilterSecurityInterceptor` (인가)

✅ 6. 기본 설정 예시 (Spring Boot 3.x 이상)

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

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

✅ 7. 사용자 인증 설정

```
@Bean
public UserDetailsService userDetailsService() {
    return username -> userRepository.findByUsername(username)
        .map(CustomUserDetails::new)
        .orElseThrow(() -> new UsernameNotFoundException("사용자 없음"));
}
```

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

✅ 8. 로그인 성공 후 Authentication 정보 저장

```
Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
String username = authentication.getName();
List<GrantedAuthority> authorities = (List<GrantedAuthority>)
authentication.getAuthorities();
```

✅ 9. 인가 어노테이션

어노테이션	설명
`@Secured("ROLE_ADMIN")`	단순 권한 체크
`@PreAuthorize("hasRole('ADMIN')")`	메서드 실행 전 검사
`@PostAuthorize`	메서드 실행 후 결과 기반 검사
`@RolesAllowed`	JSR-250 호환 어노테이션

✅ 10. 정리 구조도

```
[클라이언트 요청]
       ↓
[SecurityFilterChain]
       ↓
[UsernamePasswordAuthenticationFilter] ← 인증 수행
       ↓
[AuthenticationManager] + [UserDetailsService]
       ↓
[SecurityContextHolder] ← 인증 정보 저장
       ↓
[FilterSecurityInterceptor] ← 인가(권한 확인)
       ↓
[컨트롤러 진입]
```

✅ 결론

- **Spring Security는 필터 기반 보안 체계**
- 인증은 `AuthenticationManager`와 `UserDetailsService` 중심
- 인가는 `AccessDecisionManager`, `FilterSecurityInterceptor`가 담당
- Spring Boot에서는 보안 설정을 `SecurityFilterChain`을 통해 선언적으로 구성
- `SecurityContextHolder`에 인증 정보가 저장되며, 인가 처리 시 참조됨

사용자 정의 인증 필터

사용자 정의 인증 필터(Custom Authentication Filter)는
Spring Security에서 기본 제공하는 `UsernamePasswordAuthenticationFilter`를 대체하거나 확장하여,
특정 인증 방식(JWT, API Key, 소셜 로그인, OTP 등)에 맞춘 인증 로직을 구현할 때 사용된다.

이제 **사용자 정의 인증 필터의 개념, 구성 요소, 구현 절차, 전체 흐름**을 정리하겠다.

✅ 1. 왜 사용자 정의 인증 필터가 필요한가?

기본적으로 Spring Security는 `formLogin()` 기반의 **ID/PW 로그인**만을 지원한다.
하지만 실무에서는 다음과 같은 **맞춤 인증 방식**이 필요하다:

- JWT 인증 헤더 처리
- API Key 인증
- OAuth 토큰 인증
- QR/OTP 기반 인증
- 모바일 로그인/사설 인증 시스템

→ 이때 **사용자 정의 인증 필터**를 만들어 `SecurityFilterChain`에 등록하여 처리한다.

✅ 2. 전체 구조 요약

```
[요청 (Authorization: Bearer xxx)]
   ↓
[CustomAuthenticationFilter]  ← 인증 시도
   ↓
[AuthenticationManager]       ← 인증 위임
   ↓
[CustomAuthenticationProvider] ← 인증 수행
   ↓
[Authentication 객체 생성]
   ↓
[SecurityContextHolder에 저장]
```

✅ 3. 사용자 정의 필터 구현 절차

✔ 1) Authentication 객체 정의 (선택적)

```
public class JwtAuthenticationToken extends AbstractAuthenticationToken {
    private final String token;

    public JwtAuthenticationToken(String token) {
        super(null);
        this.token = token;
        setAuthenticated(false); // 인증 전
    }

    @Override
    public Object getPrincipal() {
        return token;
    }

    @Override
    public Object getCredentials() {
        return token;
    }
}
```

✔ 2) CustomAuthenticationFilter 구현

```
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final AuthenticationManager authenticationManager;

    public JwtAuthenticationFilter(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
    }

    @Override
```

```
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain chain)
            throws ServletException, IOException {

        String token = request.getHeader("Authorization");

        if (token != null && token.startsWith("Bearer ")) {
            token = token.substring(7);

            Authentication authRequest = new JwtAuthenticationToken(token);
            Authentication authResult =
authenticationManager.authenticate(authRequest);

            SecurityContextHolder.getContext().setAuthentication(authResult);
        }

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

✔ 3) CustomAuthenticationProvider 구현

```
@Component
public class JwtAuthenticationProvider implements AuthenticationProvider {

    @Override
    public Authentication authenticate(Authentication authentication) throws
AuthenticationException {
        String token = (String) authentication.getCredentials();

        // 토큰 유효성 검증
        if (validateToken(token)) {
            String username = extractUsername(token);
            List<GrantedAuthority> authorities = List.of(new
SimpleGrantedAuthority("ROLE_USER"));

            return new UsernamePasswordAuthenticationToken(username, token,
authorities);
        }

        throw new BadCredentialsException("Invalid JWT Token");
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return JwtAuthenticationToken.class.isAssignableFrom(authentication);
    }
}
```

✔ 4) SecurityConfig에 필터 등록

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    private final JwtAuthenticationProvider jwtAuthenticationProvider;

    public SecurityConfig(JwtAuthenticationProvider provider) {
        this.jwtAuthenticationProvider = provider;
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        AuthenticationManager authManager = authenticationManager(http);

        return http
            .csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/public/**").permitAll()
                .anyRequest().authenticated()
            )
            .addFilterBefore(new JwtAuthenticationFilter(authManager),
UsernamePasswordAuthenticationFilter.class)
            .build();
    }

    @Bean
    public AuthenticationManager authenticationManager(HttpSecurity http) throws
Exception {
        return http
            .getSharedObject(AuthenticationManagerBuilder.class)
            .authenticationProvider(jwtAuthenticationProvider)
            .build();
    }
}
```

✅ 4. 인증 후 처리

Spring Security는 필터 통과 후 `SecurityContextHolder`에 `Authentication`을 저장한다.
→ 이후 `@AuthenticationPrincipal`, `SecurityContextHolder.getContext()` 등을 통해 사용자 정보 접근 가능

✅ 5. 실패 처리 (선택적)

```
@Override
protected void doFilterInternal(...) {
    try {
        // 인증 로직
    } catch (AuthenticationException ex) {
        response.setStatus(HttpStatus.UNAUTHORIZED.value());
        response.getWriter().write("Authentication Failed: " + ex.getMessage());
        return;
    }

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

✅ 6. 전체 흐름 요약

```
[JWT 인증 요청]
   ↓
[CustomAuthenticationFilter]  → 토큰 추출
   ↓
[JwtAuthenticationToken 생성]  → 인증 전
   ↓
[AuthenticationManager] 위임
   ↓
[JwtAuthenticationProvider] → 토큰 검증 + 사용자 정보 로딩
   ↓
[UsernamePasswordAuthenticationToken 반환] → 인증 완료
   ↓
[SecurityContextHolder에 저장]
   ↓
[컨트롤러 진입 → 인증 정보 사용 가능]
```

✅ 7. 결론

항목	설명
목적	기본 로그인 방식 외의 **맞춤 인증 처리** 구현
필수 구성	`CustomFilter`, `AuthenticationToken`, `AuthenticationProvider`
등록 위치	`SecurityFilterChain.addFilterBefore(...)`
실무 예	JWT, OAuth2, API Key 인증, 사설 로그인

JWT 기반 인증 처리

JWT 기반 인증 처리는 Spring Security에서 가장 많이 사용하는 **토큰 기반 인증 방식**입니다.
세션 없이도 클라이언트가 로그인 상태를 유지할 수 있고,
REST API 환경에서 **무상태(Stateless)**로 인증 처리를 할 수 있는 강력한 방법입니다.

아래는 **JWT 기반 인증의 전체 구조, 흐름, 구현 전략**을 순서대로 정리한 내용입니다.

✅ 1. JWT 인증이란?

> JWT (JSON Web Token)는 **사용자 정보를 서명된 토큰**으로 인코딩하여
> 클라이언트에게 전달하고, 이후 요청마다 인증을 토큰 기반으로 처리하는 방식이다.

✅ 2. 전체 인증 흐름 요약

```
[1] 로그인 요청 (ID/PW)
   ↓
[2] 서버에서 JWT 발급 (accessToken)
   ↓
[3] 클라이언트가 이후 요청 시 헤더에 토큰 포함
   ↓
[4] 필터에서 토큰 검증
   ↓
[5] 인증 성공 시 SecurityContext에 저장
   ↓
[6] 인가 처리 → 컨트롤러 실행
```

✅ 3. JWT 구조

```
eyJhbGciOiJIUzI1NiJ9.          // Header
eyJzdWIiOiJ1c2VySWQiLCJyb2xlIjpbIlVTRVIiXX0.     // Payload (claim)
7dbsafsd97dsaf...              // Signature
```

구성	설명
Header	alg, typ
Payload	사용자 정보 (Claims)
Signature	비밀 키로 서명

✅ 4. JWT 발급 (로그인 시)

요청

```
POST /login
Content-Type: application/json

{
  "username": "kim",
  "password": "1234"
}
```

응답

```
{
  "accessToken": "eyJhbGci...xxx",
  "refreshToken": "eyJhbGci...yyy"
}
```

✅ 5. JWT 발급 코드 (TokenProvider)

```
@Component
public class JwtTokenProvider {

    private final String secretKey = "very-secret-key";
    private final long expiration = 1000 * 60 * 60; // 1시간

    public String generateToken(String username, List<String> roles) {
        return Jwts.builder()
            .setSubject(username)
            .claim("roles", roles)
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + expiration))
            .signWith(SignatureAlgorithm.HS256, secretKey.getBytes())
            .compact();
    }

    public Claims parseToken(String token) {
        return Jwts.parser()
            .setSigningKey(secretKey.getBytes())
            .parseClaimsJws(token)
            .getBody();
    }

    public boolean validate(String token) {
        try {
            parseToken(token);
            return true;
        } catch (JwtException e) {
            return false;
```

```
        }
    }
}
```

✅ 6. 인증 필터 (`JwtAuthenticationFilter`)

```
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final JwtTokenProvider tokenProvider;

    public JwtAuthenticationFilter(JwtTokenProvider provider) {
        this.tokenProvider = provider;
    }

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

        String token = resolveToken(request);
        if (token != null && tokenProvider.validate(token)) {
            Claims claims = tokenProvider.parseToken(token);
            String username = claims.getSubject();
            List<GrantedAuthority> authorities = ((List<?>) claims.get("roles")).stream()
                .map(role -> new SimpleGrantedAuthority("ROLE_" + role))
                .toList();

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

        chain.doFilter(request, response);
    }

    private String resolveToken(HttpServletRequest request) {
        String header = request.getHeader("Authorization");
        return (header != null && header.startsWith("Bearer ")) ? header.substring(7) : null;
    }
}
```

✅ 7. Security 설정에 필터 등록

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

```

```
    private final JwtTokenProvider tokenProvider;

    public SecurityConfig(JwtTokenProvider tokenProvider) {
        this.tokenProvider = tokenProvider;
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .csrf(AbstractHttpConfigurer::disable)
            .sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/login", "/signup").permitAll()
                .anyRequest().authenticated()
            )
            .addFilterBefore(new JwtAuthenticationFilter(tokenProvider),
UsernamePasswordAuthenticationFilter.class);

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

✅ 8. 인증 후 사용자 정보 접근

```
@GetMapping("/me")
public String getUserInfo(Authentication authentication) {
    return authentication.getName(); // username
}
```

또는

```
@GetMapping("/me")
public String getUserInfo(@AuthenticationPrincipal UserDetails user) {
    return user.getUsername();
}
```

✅ 9. 상태 코드 예외 처리 (선택 사항)

```
@ExceptionHandler(AccessDeniedException.class)
public ResponseEntity<ErrorResponse> handleForbidden() {
    return ResponseEntity.status(403).body(new ErrorResponse(403, "접근 거부"));
}

@ExceptionHandler(AuthenticationException.class)
public ResponseEntity<ErrorResponse> handleUnauthorized() {
    return ResponseEntity.status(401).body(new ErrorResponse(401, "인증 실패"));
}
```

✅ 10. 결론 요약

항목	설명
인증 방식	로그인 시 JWT 발급 후, 이후 모든 요청에 포함
Stateless	세션 없이 인증 처리 (REST에 적합)
인증 흐름	JWT → 필터 → `Authentication` 저장
인가 처리	JWT 내 Role 기반 → `hasRole`, `@PreAuthorize` 활용
장점	서버 확장성, 무세션, 모바일/SPA 친화적
단점	토큰 탈취 시 대응 어려움 → 만료 시간 설정, RefreshToken 도입 필요

OAuth2 로그인 처리

OAuth2 로그인 처리는 Spring Security에서 소셜 로그인(Google, Kakao, Naver, GitHub 등)을 구현하기 위한 **표준 인증 프로토콜**입니다.
Spring Boot에서는 이를 간편하게 구성할 수 있도록 `spring-boot-starter-oauth2-client` 모듈을 제공합니다.

✅ 1. OAuth2 로그인 흐름 개요

```
[1] 사용자가 "소셜 로그인" 버튼 클릭
   ↓
[2] 인증 서버(Google 등)로 리디렉션
   ↓
[3] 사용자 로그인 → 권한 부여
   ↓
[4] 콜백 URL로 Access Token 반환
   ↓
[5] 리소스 서버에서 사용자 정보 조회
   ↓
[6] Spring Security가 사용자 인증 완료
```

✅ 2. 의존성 추가 (Gradle)

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

✅ 3. application.yml 설정 예 (Google)

```yaml
spring:
  security:
    oauth2:
      client:
        registration:
          google:
            client-id: YOUR_CLIENT_ID
            client-secret: YOUR_CLIENT_SECRET
            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
```

✅ 4. 기본 로그인 URL

```
GET /oauth2/authorization/google
```

→ 이 URL에 접속하면 Spring Security가 자동으로 Google 로그인으로 리디렉션

✅ 5. OAuth2 기본 구조

컴포넌트	역할
`OAuth2LoginAuthenticationFilter`	인증 시작 필터
`OAuth2UserService`	사용자 정보 조회
`OAuth2User`	OAuth 제공자에서 받은 사용자 정보
`PrincipalOauth2UserService`	사용자 커스텀 매핑

✅ 6. 사용자 정보 커스터마이징

1) 커스텀 OAuth2UserService

```java
@Service
public class CustomOAuth2UserService extends DefaultOAuth2UserService {

    @Override
    public OAuth2User loadUser(OAuth2UserRequest request) throws
OAuth2AuthenticationException {
        OAuth2User oAuth2User = super.loadUser(request);

```

```
        String registrationId = request.getClientRegistration().getRegistrationId(); // "google", "naver"
        String userNameAttrName = request.getClientRegistration()
            .getProviderDetails().getUserInfoEndpoint().getUserNameAttributeName();

        Map<String, Object> attributes = oAuth2User.getAttributes();

        // 사용자 정보 추출
        String email = (String) attributes.get("email");
        String name = (String) attributes.get("name");

        return new DefaultOAuth2User(
            Collections.singleton(new SimpleGrantedAuthority("ROLE_USER")),
            attributes,
            userNameAttrName
        );
    }
}
```

2) SecurityConfig에 등록

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    private final CustomOAuth2UserService customOAuth2UserService;

    public SecurityConfig(CustomOAuth2UserService service) {
        this.customOAuth2UserService = service;
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/", "/login/**").permitAll()
                .anyRequest().authenticated()
            )
            .oauth2Login(oauth -> oauth
                .loginPage("/login")
                .userInfoEndpoint(userInfo -> userInfo
                    .userService(customOAuth2UserService)
                )
            );
        return http.build();
    }
}
```

✅ 7. 로그인 후 사용자 정보 접근

```
@GetMapping("/profile")
public String profile(@AuthenticationPrincipal OAuth2User user) {
    return "Hello, " + user.getAttribute("name");
}
```

✅ 8. 실무 확장 전략

항목	설명
커스텀 UserDetails	OAuth2User → 로컬 UserDetails 변환
회원가입 연동	처음 로그인 시 DB 저장 및 역할 부여
JWT 연동	OAuth2 로그인 성공 시 JWT 발급 후 전달
상태값 저장	`OAuth2AuthorizationRequestRepository` 사용
소셜 연동	Google, Kakao, Naver, GitHub 동시 등록 가능

✅ 9. 예시 응답 (Google)

```
{
  "sub": "123456789",
  "name": "Kim",
  "email": "kim@example.com",
  "picture": "https://profile.jpg"
}
```

→ `OAuth2User.getAttributes()` 를 통해 전체 접근 가능

✅ 10. 결론

항목	설명
인증 방식	OAuth2 표준 기반 (Authorization Code Grant)
URL	`/oauth2/authorization/{provider}`
사용자 정보	`OAuth2UserService` 로 커스터마이징
확장성	JWT 발급, DB 연동, 추가 권한 부여 가능
권장 전략	① OAuth2 로그인 → ② 사용자 정보 저장 → ③ JWT 발급 후 응답

CSRF, CORS, 세션 보안

웹 보안에서 반드시 이해하고 대응해야 할 3가지 핵심 주제가 바로 다음입니다:

- **CSRF (Cross-Site Request Forgery)**
- **CORS (Cross-Origin Resource Sharing)**
- **세션 보안(Session Security)**

이들은 서로 다른 보안 영역이지만, Spring Security에서는 **명확한 제어 도구와 설정 방법**이 존재합니다.
아래에 이 세 가지를 구조적으로 설명하고, 실무 대응 방법까지 정리합니다.

✅ 1. CSRF (Cross-Site Request Forgery)

📌 개념

> **사용자가 의도하지 않은 요청을 악의적인 사이트가 자동으로 수행**하게 만드는 공격

- 사용자가 로그인한 상태에서, 공격자가 **정상 사이트의 권한을 도용**하여 요청을 전송
- 예: 사용자 인증 쿠키가 자동 전송되는 특성을 이용

📌 예시 공격

```
<img src="https://bank.com/transfer?amount=10000&to=attacker" />
```

→ 브라우저가 `bank.com`에 로그인된 상태라면 자동으로 전송됨

✅ Spring Security에서 기본 방어

Spring Security는 `csrf()` **보호를 기본적으로 활성화**한다.
→ `POST`, `PUT`, `DELETE` 요청은 **CSRF 토큰이 없으면 403 Forbidden**

✅ 설정 예시

```
http
    .csrf(csrf -> csrf
        .ignoringRequestMatchers("/api/**") // REST API는 무상태 → CSRF 비활성화
    );
```

✅ 비활성화 (무상태 API)

```
http.csrf(AbstractHttpConfigurer::disable); // JWT 기반 REST API에서 사용
```

✅ 2. CORS (Cross-Origin Resource Sharing)

📌 개념

> **브라우저가 다른 Origin(출처)의 리소스 요청을 차단**하는 기본 보안 정책 (Same-Origin Policy)

- 예: `frontend.com`에서 `api.backend.com`에 요청 → 차단됨
- 브라우저는 `OPTIONS` 프리플라이트 요청으로 허용 여부를 먼저 검증

✅ Spring 설정 방법

```
@Bean
public WebMvcConfigurer corsConfigurer() {
    return new WebMvcConfigurer() {
        @Override
        public void addCorsMappings(CorsRegistry registry) {
            registry.addMapping("/**")
                .allowedOrigins("http://localhost:3000")
                .allowedMethods("GET", "POST", "PUT", "DELETE")
                .allowCredentials(true);
        }
    };
}
```

> `allowCredentials(true)` 설정 시, `allowedOrigins("\*")` **사용 불가**

✅ SecurityConfig에서 CORS 허용

```
http.cors(Customizer.withDefaults());
```

→ `WebMvcConfigurer`와 함께 작동하도록 함

✅ 3. 세션 보안

📌 개념

> 서버가 브라우저의 상태를 기억하기 위해 유지하는 저장 공간 → 공격 대상이 될 수 있음

✅ 세션 고정 공격 방지 (Session Fixation)

Spring Security는 로그인 시 자동으로 **세션 ID를 변경**하여 공격 방지

```
http.sessionManagement(session ->
    session.sessionFixation(SessionFixationConfigurer::migrateSession)
);
```

- `migrateSession` (기본): 기존 세션 속성 유지하면서 새로운 ID 부여

✅ 세션 하이재킹 대응

전략	설명
HTTPS 사용	쿠키 탈취 방지
`HttpOnly` 쿠키	JavaScript 접근 차단
세션 타임아웃	유휴 세션 자동 만료 (`server.servlet.session.timeout`)
최대 동시 세션 제한	한 사용자당 1회 로그인 제한

```
http.sessionManagement(session -> session
    .maximumSessions(1)
    .maxSessionsPreventsLogin(true)
);
```

✅ Stateless 방식 (JWT 인증)

JWT 기반 로그인에서는 세션을 완전히 사용하지 않는다.

```
http.sessionManagement(session ->
    session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
);
```

✅ 정리 비교표

항목	목적	위협	Spring 처리 방식
CSRF	악의적 사이트의 요청 위조 방지	인증된 사용자의 쿠키 악용	`csrf().enable()` (기본 활성화)
CORS	브라우저 출처 간 리소스 공유 제한	다른 Origin 요청 차단	`http.cors()` + `CorsRegistry`
세션 보안	인증된 세션 탈취, 고정 방지	Session Fixation, Hijacking	`sessionManagement()` 설정

✅ 결론

보안 요소	REST API 환경에서는?
CSRF	❌ 비활성화 권장 (`csrf().disable()`)
CORS	✅ 반드시 명시적 허용 필요 (`http.cors()`)

보안 요소	REST API 환경에서는?
세션	✅ `SessionCreationPolicy.STATELESS` 설정 필수

비밀번호 암호화 (BCrypt)

Spring Security에서 **비밀번호 암호화의 표준 방식**은 `BCryptPasswordEncoder`를 사용하는 것이다.
BCrypt는 해시 기반의 단방향 암호화 알고리즘으로,
역산이 불가능하고, 무차별 대입 공격(Brute Force)에 매우 강한 구조를 가지고 있다.

✅ 1. 왜 비밀번호를 암호화해야 하는가?

이유	설명
보안성	평문 비밀번호 저장은 치명적인 보안 위협
해킹 시 피해 최소화	DB 유출되더라도 복호화가 불가능해야 함
단방향성	사용자의 입력과 저장된 해시값을 비교 (복호화 아님)
난수성	동일한 입력값 → 매번 다른 해시값 생성 (Salt 자동 포함)

✅ 2. BCrypt 특징 요약

항목	내용
알고리즘	단방향 해시 함수
Salt	내부적으로 자동 생성 및 포함
반복 횟수(cost)	기본 10회, 높일수록 보안 증가
복호화	불가능 (단, 입력값을 다시 암호화해 비교는 가능)
안전성	Rainbow Table, Brute Force에 강함

✅ 3. Spring에서 `BCryptPasswordEncoder` 사용 방법

1) 의존성 포함 (Spring Boot Security 사용 시 기본 포함)

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

---

2) PasswordEncoder Bean 등록

```
@Configuration
public class SecurityConfig {

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

---

3) 비밀번호 암호화 예

```
@Autowired
private PasswordEncoder passwordEncoder;

public void registerUser(String rawPassword) {
    String encodedPassword = passwordEncoder.encode(rawPassword);
    userRepository.save(new User(..., encodedPassword));
}
```

→ `encode()` 는 내부적으로 salt 포함 + 해시 실행

---

4) 로그인 시 비밀번호 비교

```
public void login(String rawPassword, String encodedPasswordFromDB) {
    if (!passwordEncoder.matches(rawPassword, encodedPasswordFromDB)) {
        throw new IllegalArgumentException("비밀번호가 틀립니다.");
    }
}
```

→ `matches()` 는 내부적으로 salt를 추출해 재해시하여 비교

✅ 4. BCrypt 해시 예시

```
$2a$10$uOZk8sVdrxoKqfnSEVUEu.HKn8Rv7TCvMSxtdMQmLt93ZKk/AF12e
```

구성	설명
`$2a$`	알고리즘 버전
`10`	cost factor (2^10 반복)
`uOZk8...`	salt + hashed password

✅ 5. PasswordEncoder 확장 전략

Spring Security의 `DelegatingPasswordEncoder`를 사용하면 **다양한 인코딩 알고리즘을 혼합 지원**할 수 있다:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return PasswordEncoderFactories.createDelegatingPasswordEncoder();
}
```

→ 기본은 BCrypt, 기존 SHA-1, PBKDF2 등도 사용 가능
→ `{bcrypt}hashed_string` 형태로 접두어 저장

✅ 6. 비밀번호 변경 시 전략

- 새로운 비밀번호는 반드시 다시 `encode()` 후 저장
- 같은 값이라도 해시값이 매번 다르므로 재저장 필요
- 기존 `matches()`는 정상 작동함

✅ 7. 결론 요약

항목	설명
알고리즘	`BCryptPasswordEncoder` 사용
encode()	비밀번호 해시 + salt 포함 저장
matches()	입력값 → 해시 비교
복호화	❌ 불가능 (단방향)
저장 위치	DB에 해시값만 저장