Spring Security

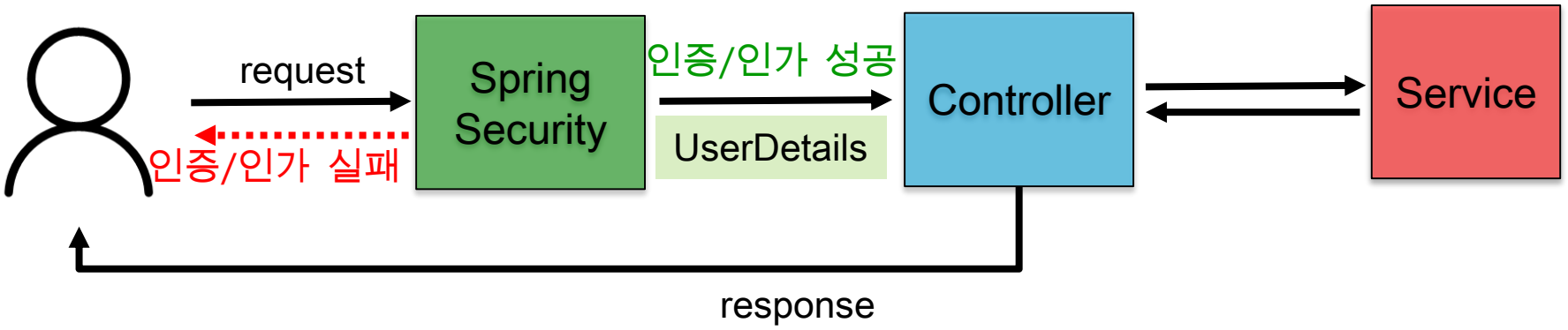
558280-1
2026년 봄학기
6/4/2026
박경신

Security

- 왜 보안이 중요한가?
  - 인증되지 않은 사용자의 접근 방지
  - 데이터 유출, 변조, 세션 탈취 방어
  - 법적 요구사항(GDPR 등) 및 사용자 신뢰 확보

Spring Security

- Spring Security
  - Spring 기반 보안 프레임워크
  - **인증 (Authentication) 및 인가 (Authorization) 처리, 세션/토큰 기반 인증 지원, 보안 필터 체계 제공**
  - 간단한 Annotation 기반으로 선언적 보안 설정이 가능
  - 다양한 인증 방식 지원 (Form Login, HTTP Basic, OAuth2, JWT, LDAP 등)



Spring Security 설정

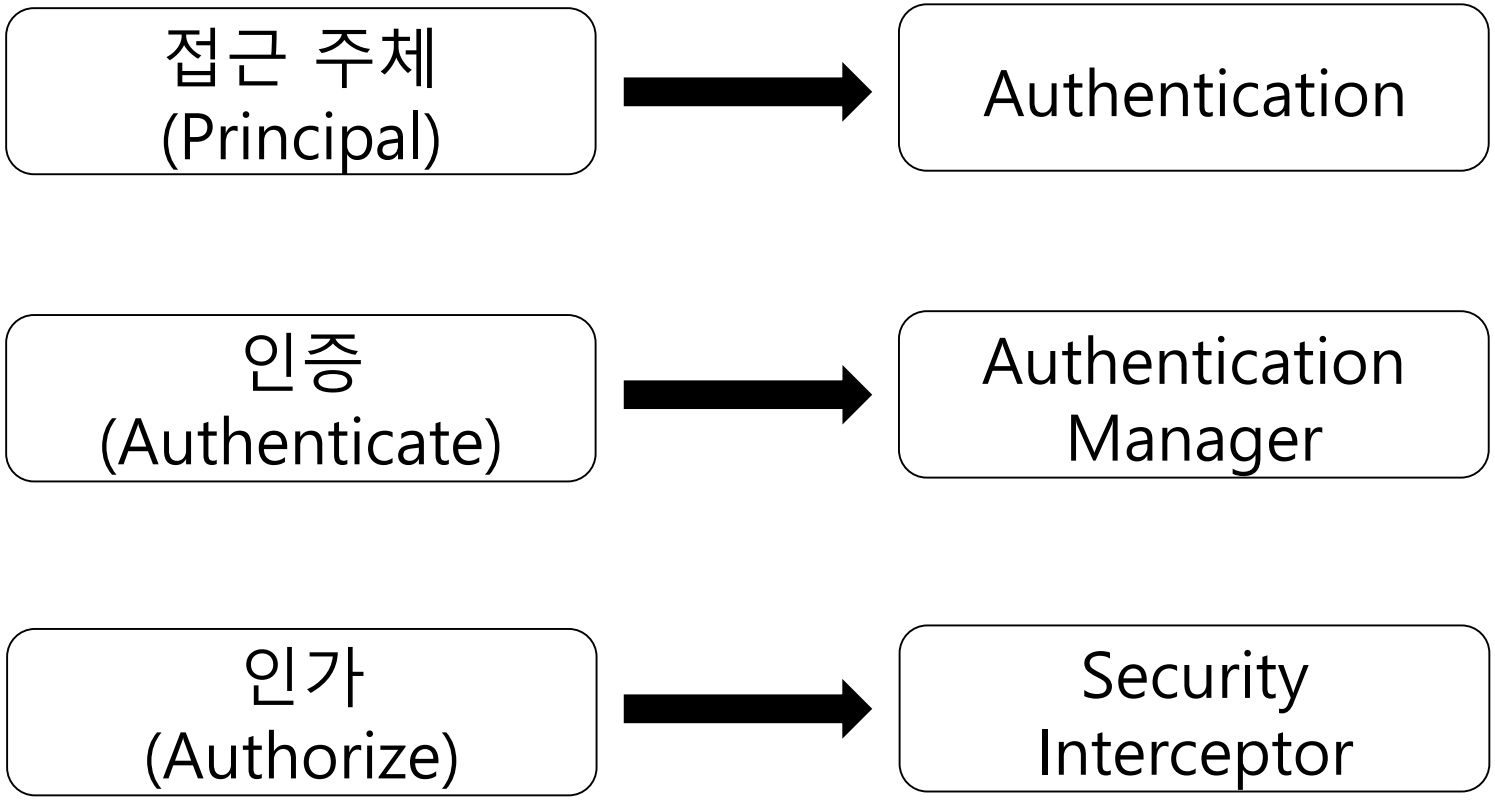
- Spring Security 설정 기본 구조 예시

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http.authorizeHttpRequests()
                .requestMatchers("/api/public/**").permitAll()
                .anyRequest().authenticated()
                .and()
                .formLogin()
                .build();
}
```

Security Fundamentals

- 접근 주체 (Principal)
  - 보호된 Resource에 접근하려는 사용자
- 인증 (Authentication)
  - 현재 사용자가 누구인지 검증하는 절차 (로그인)
- 인가 (Authorization)
  - 인증된 사용자가 특정 자원에 접근할 권한이 있는지 판단

Security Fundamentals



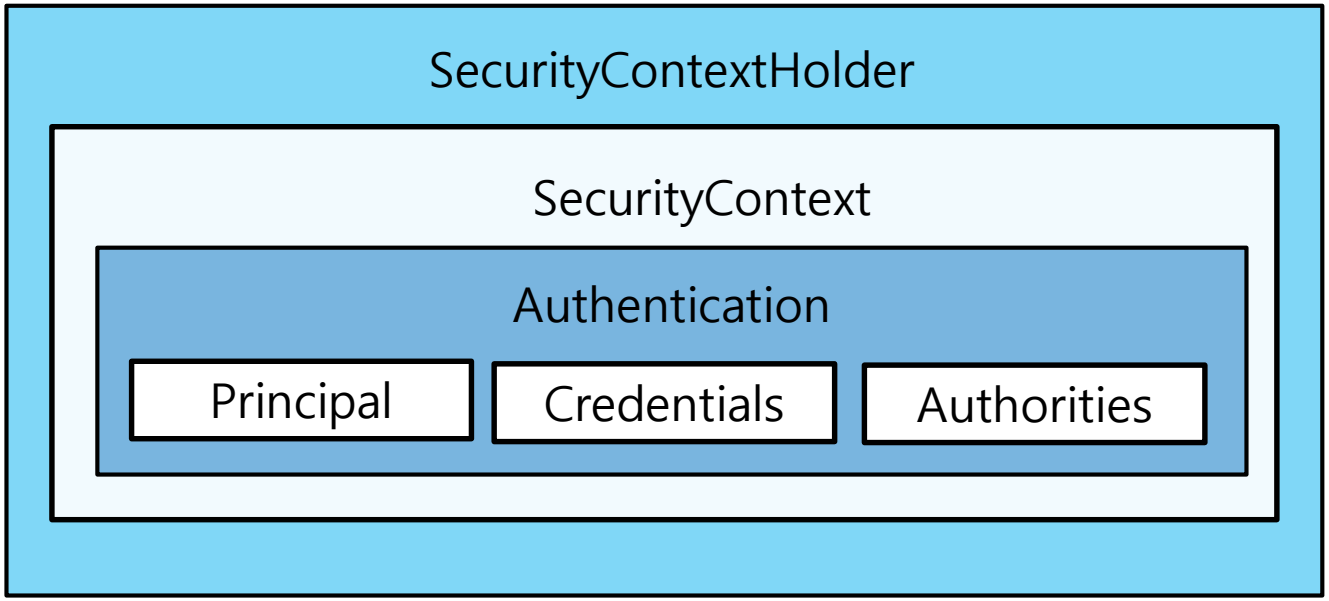
Authentication & SecurityContext

- Authentication
  - 현재 접근 주체(Principal) 정보를 담는 목적
  - 인증 요청할 때, 요청 정보를 담는 목적
- GrantedAuthority
  - 인증에서 접근 주체(Principal)에게 부여된 권한(Role, Permission 등)을 표현
- SecurityContext
  - 인증된 사용자의 인증 (Authentication) 및 기타 보안 정보를 보관
  - SecurityContext로부터 Authentication 객체를 구함
- SecurityContextHolder
  - SecurityContext에 대한 접근을 제공

https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html

SecurityContextHolder

- SecurityContextHolder
  - Spring Security가 인증된 사용자에 대한 세부 정보를 저장
  - Spring Security는 SecurityContextHolder가 어떻게 채워지는지 신경 쓰지 않으며, 값이 포함되어 있으면 현재 인증된 사용자로 사용됨



https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html

SecurityContextHolder

- SecurityContextHolder
  - ThreadLocal (default), InheritableThreadLocal, Global 에 SecurityContext를 보관

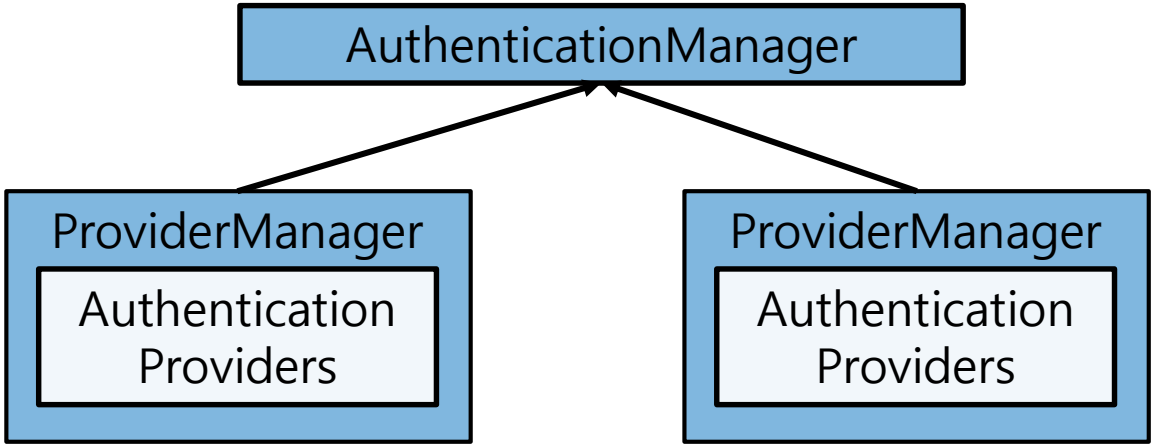
```
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();
String username = authentication.getName();
Object principal = authentication.getPrincipal();
If (principal instanceof UserDetails) {
        String username = ((UserDetails)principal).getUsername();
} else {
        String username = principal.toString();
}
Collection<? Extends GrantedAuthority> authorities =
authentication.getAuthorities();
```

Authentication Method

- Authentication 주요 메서드
  - String getName() – 사용자 이름
  - Object getCredential() – 증명 값 (비밀번호 등)
  - Object getPrincipal() – 인증 주체 정보
  - boolean isAuthenticated() – 인증 되었는지 여부
  - Collection<GrantedAuthority> getAuthorities() – 현재 사용자가 가진 권한 (GrantedAuthority)

AuthenticationManager

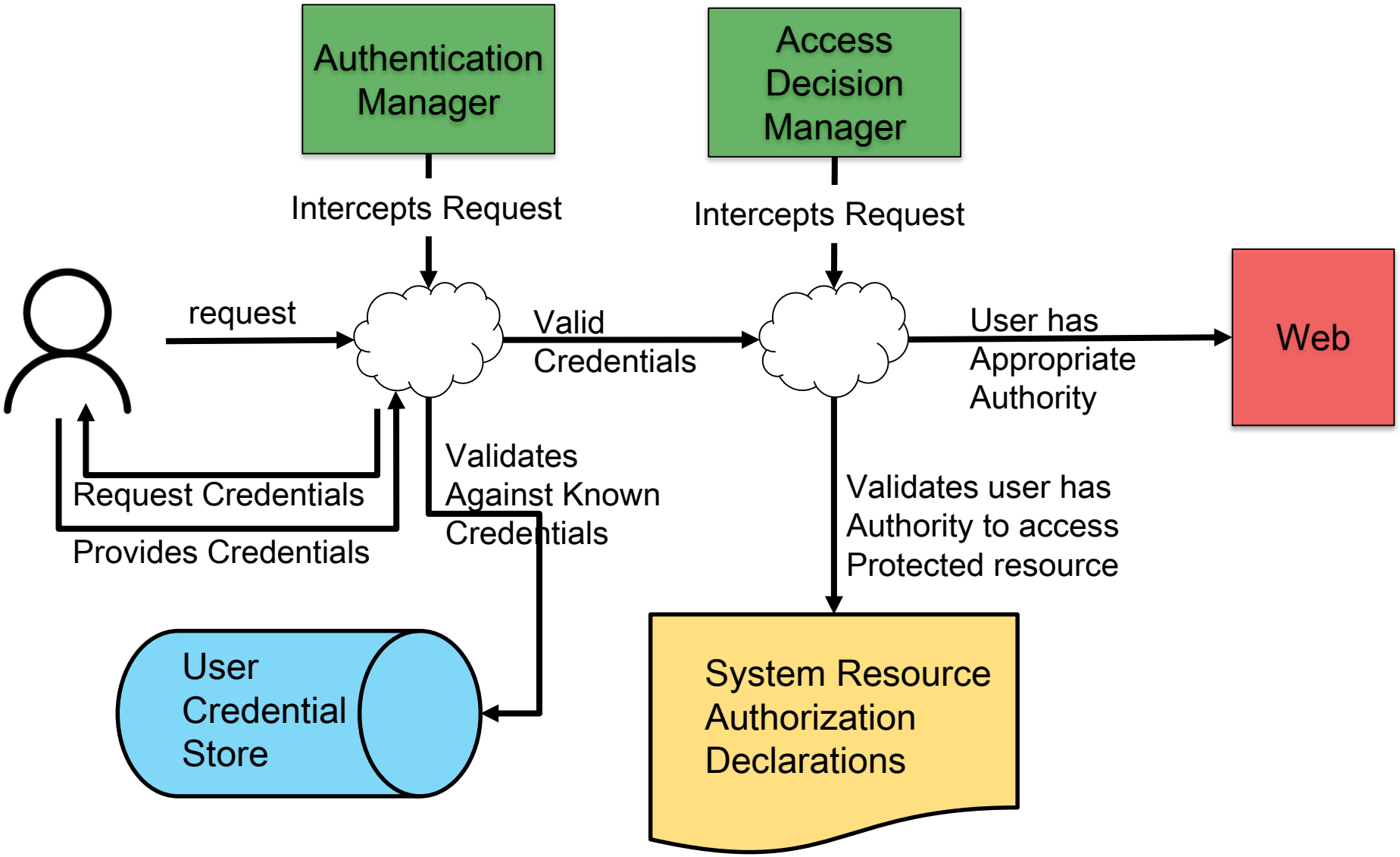
- AuthenticationManager는 인증(Authentication)을 처리
  - 인증 성공 시 인증 정보를 담고 있는 Authentication 객체를 반환
    - Spring Security는 반환한 Authentication 객체를 SecurityContext에 보관 및 인증 상태 유지를 위해 Session에 보관
  - 인증 실패 시 AuthenticationException 발생



(Abstract)SecurityInterceptor

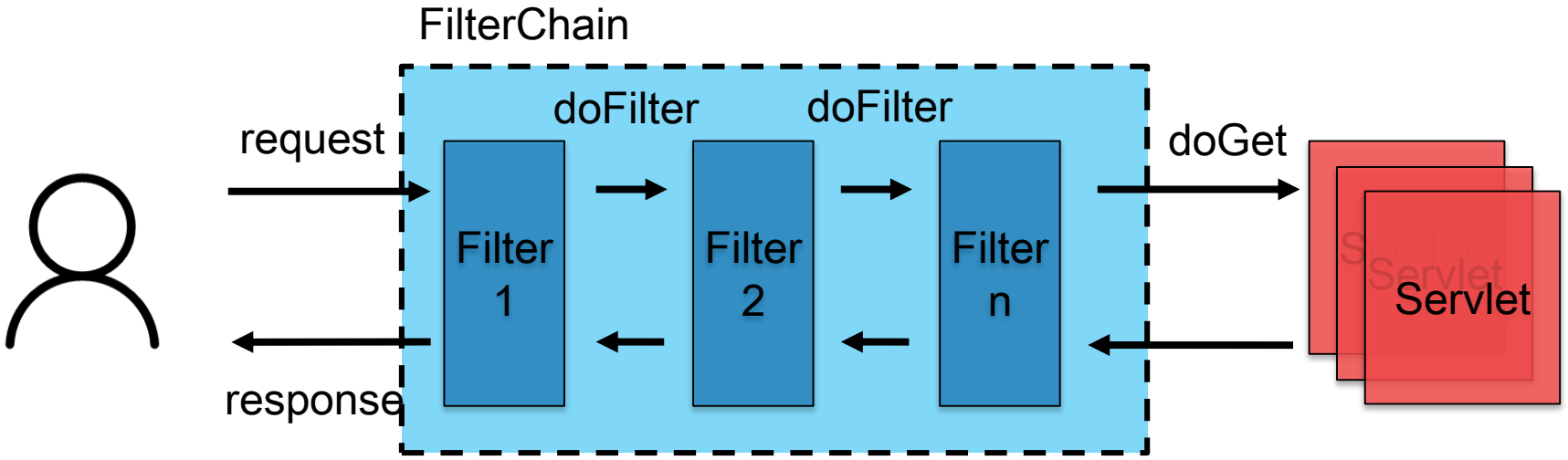
- SecurityInterceptor는 인가(Authorization)를 처리
  - 웹의 경우 FilterSecurityInterceptor 구현 사용
  - **AccessDecisionManager**에 권한 검사 위임
  - 사용자가 자원의 보안설정 기준으로, 접근 권한이 없을 경우 예외 발생

Spring Security Architecture



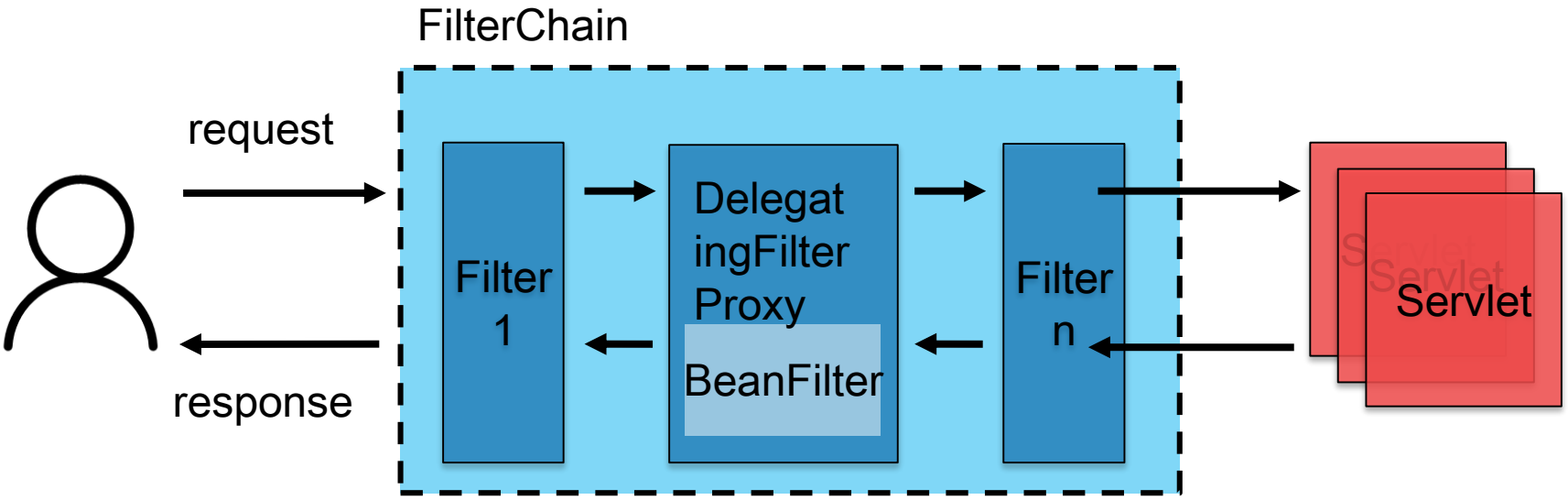
Security Filter Chain

- Spring Security는 보안 필터 체인을 이용한 보안 처리
- FilterChain 구조
  - DelegatingFilterProxy -> FilterChainProxy -> SecurityFilterChain



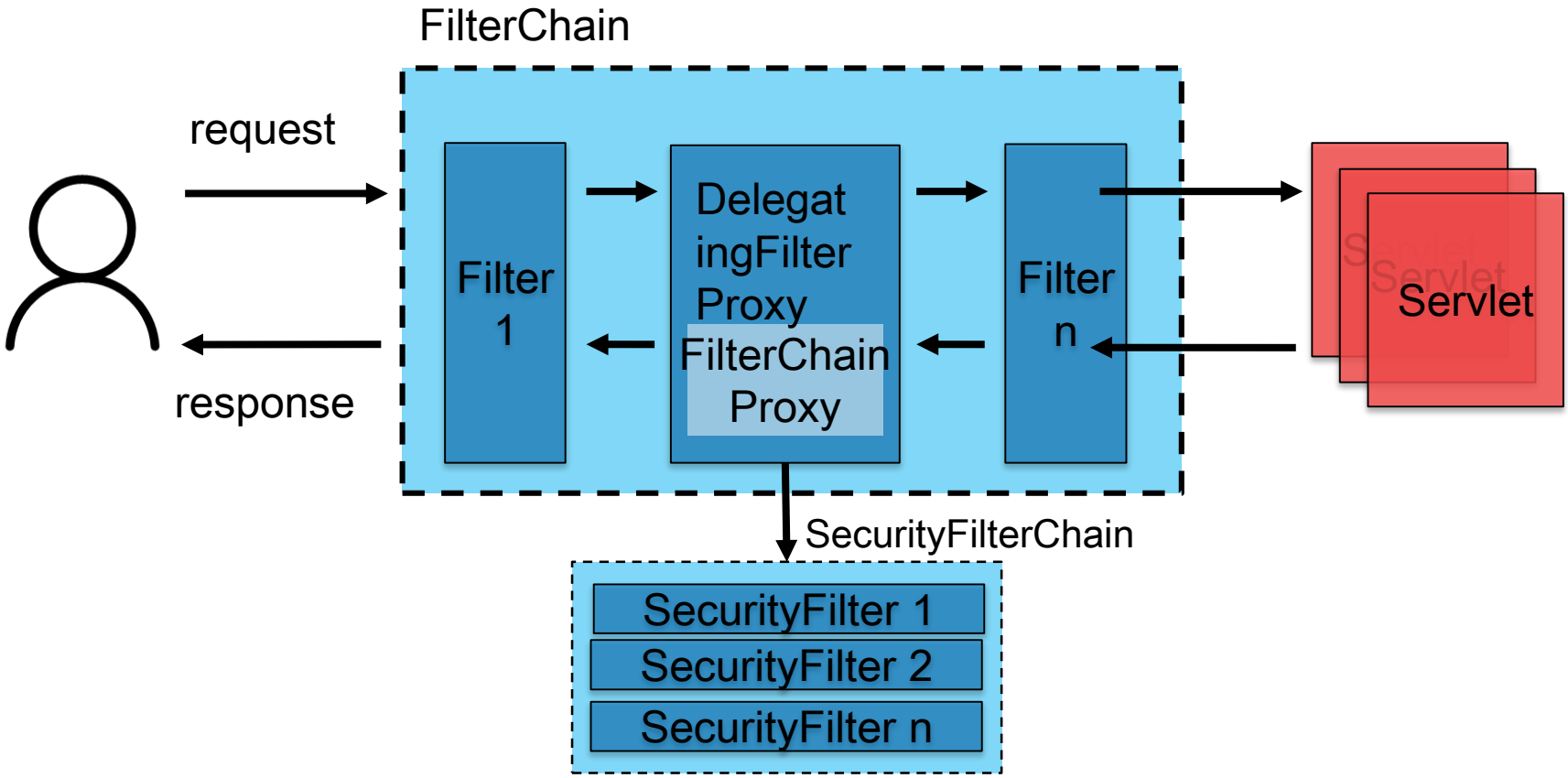
DelegatingFilterProxy

- Spring은 표준 Servlet 컨테이너 매커니즘을 통해 DelegatingFilterProxy를 등록하여 모든 작업을 Filter 인터페이스를 구현한 Spring Bean에 위임할 수 있음



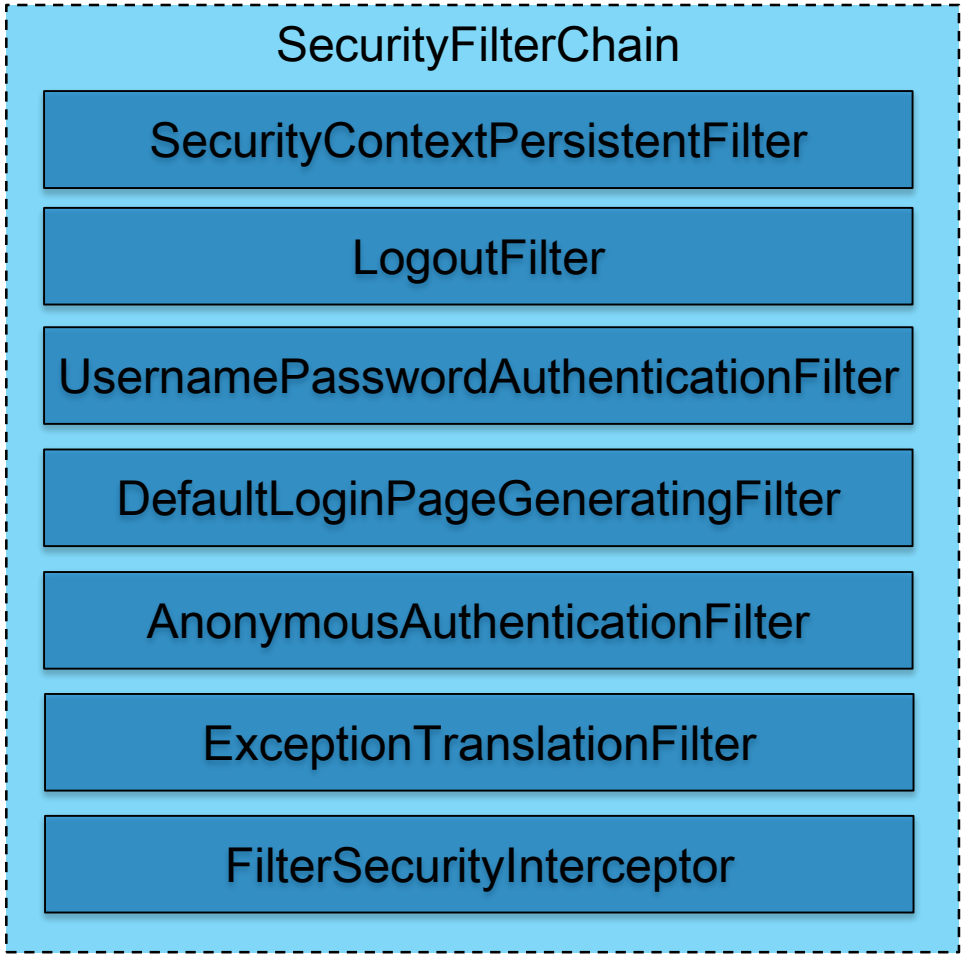
FilterChainProxy

- FilterChainProxy는 Spring Security에서 제공하는 특수 Filter로, SecurityFilterChain을 통해 여러 Security Filter (Spring Bean) 인스턴스에 위임할 수 있음



SecurityFilterChain

- 보안 필터 체인은 순서대로 필터에 따라 요청을 처리하고 체인 실행을 끝냄



Authentication 로딩

사용자 logout 요청 처리

사용자 authentication 요청 처리

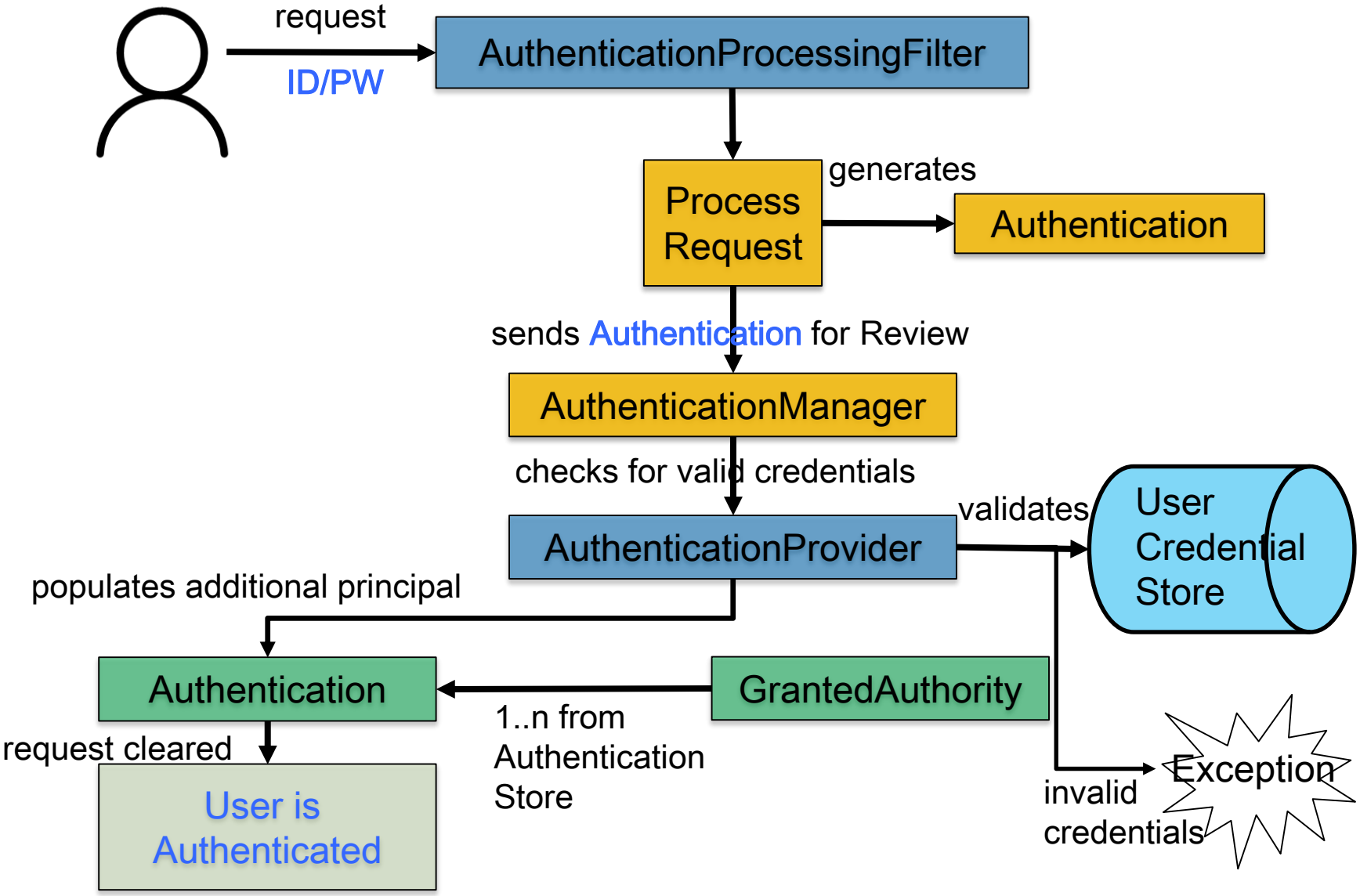
Login 폼 처리

Anonymous 사용자 처리

Exception 처리

접근 권한 검사

Authentication



Authentication

- 인증 (Authentication)
  - 웹 애플리케이션에 접근하려는 사용자의 신원을 확인하는 절차
  - 일반적으로 아이디/비밀번호를 사용
  - 인증이 완료되면 Authentication 객체가 생성되어 SecurityContextHolder에 저장됨
  - 예를 들어, 사용자가 loginForm에서 ID/PW를 입력하면 Authentication Filter가 검증하고 인증 성공 여부에 따라 흐름 결정
  - 기본적으로 Form 기반의 로그인, HTTP 기본 인증, Remember Me 인증, OAuth, LDAP 등 다양한 인증 방식을 지원

Authentication

- 인증 주요 요소
  - AuthenticationFilter -> AuthenticationManager -> AuthenticationProvider -> UserDetailsService -> UserDetails
  - PasswordEncoding (Bcryt 등)
  - RememberMe, Session 관리

Core Authentication Services

- AbstractAuthenticationProcessingFilter
  - 인증에 사용되는 Base Filter
- AuthenticationManager
  - Authentication(인증) 요청을 처리
- AuthenticationProvider
  - Authentication 수행 로직 진행
- UserDetailsService
  - 사용자 정보 조회 UserDetails 객체 반환을 담당
- UserDetails
  - Core User Information을 제공

AuthenticationManager

- AuthenticationManager 인터페이스
  - 인증 요청 처리의 핵심 진입점

```
public interface AuthenticationManager {
    // attempts to authenticate the passed Authentication object
    Authentication authenticate(Authentication authentication)
        throw AuthenticationException;
}
```

AuthenticationProvider

- AuthenticationProvider 인터페이스
  - 실제 인증 로직을 구현한 컴포넌트

```
public interface AuthenticationProvider {
    // perform authentication
    Authentication authenticate(Authentication authentication)
       throw AuthenticationException;

    // return true if this provider supports the indicated Authentication object
    boolean supports(Class<? Extends Object> authentication);
}
```

UserDetailsService

- UserDetailsService 인터페이스
  - 사용자 정보 조회 서비스

```
public interface UserDetailsService {
    // locate the user based on the username
    // return a user record
    // throws UsernameNotFoundException if the user could not be found
    // or the user has no GrantedAuthority
    // throws DataAccessException if the user could not be found for
    // repository-specific reason
    UserDetails loadUserByUsername(String username)
       throw UsernameNotFoundException, DataAccessException;
}
```

UserDetailsService

- CustomUserDetailsService 구현 예시

```
@Service
public class CustomUserDetailsService implements UserDetailsService {
  public UserDetails loadUserByUsername(String username) {
    return userRepository.findByUsername(username)
      .map(user -> new CustomUserDetails(user))
      .orElseThrow(() -> new UsernameNotFoundException("사용자 없음"));
  }
}
```

UserDetails

- UserDetails 인터페이스
  - 인증 이후 사용자의 권한과 계정 상태 정보 제공

```
public interface UserDetails extends Serializable {
    Collection<GrantedAuthority> getAuthorities();
    String getPassword();
    String getUsername();

    boolean isAccountNonExpired();
    boolean isAccountNonLocked();
    boolean isCredentialsNonExpired();
    boolean isEnabled();
}
```

UserDetails

- CustomUserDetails 구현 예시

```
@RequiredArgsConstructor
public class CustomUserDetails implements UserDetails {
  private final User user;
  // getAuthorities, getUsername 등 구현
  public String getUsername() { return user.getUsername(); }
  public String getPassword() { return user.getPassword(); }
..
}
```

Password Encoding

- BCryptPasswordEncoder
  - Spring Security 프레임워크에서 제공하는 클래스 중 하나로 PasswordEncoder (비밀번호 암호화) 인터페이스를 구현한 클래스
  - BCrypt hashing function를 사용해서 비밀번호를 인코딩해주는 메서드와 사용자의 의해 제출된 비밀번호와 저장소에 저장되어 있는 비밀번호의 일치 여부를 확인해주는 메서드를 제공
  - 생성자의 인자 값(version, strength, SecureRandom instance)을 통해서 해시의 강도를 조절할 수 있음
- BCryptPasswordEncoder는 Spring Security 5.4.2부터 3개의 메서드 제공
  - String encode(CharSequence rawPassword)
  - boolean matches(CharSequence rawPassword, String encodePassword)
  - default boolean upgradeEncoding(String encodePassword)

Password Encoding

- BCryptPasswordEncoder 설정 예시

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

public class CustomUserDetails implements UserDetails {
..
// 패스워드 저장시 자동 암호화
    public CustomUser encodePassword(PasswordEncoder passwordEncoder) {
        this.password = passwordEncoder.encode(this.password);
        return this;
    }
}
```

RememberMe Authentication

- RememberMe는 장기 로그인 기능 제공
  - RememberMeAuthenticationFilter
  - RememberMeServices
  - RememberMeAuthenticationProvider

```
public interface RememberMeServices {
    Authentication autoLogin(HttpServletRequest request,
                                        HttpServletResponse respons);
    void loginFail(HttpServletRequest request,
                        HttpServletResponse respons);
    void loginSucess(HttpServletRequest request,
                            HttpServletResponse respons,
                            Authentication successfulAuthentication);
}
```

RememberMe

- RememberMe 설정 예시

```
http.rememberMe()
        .key("uniqueAndSecret")
        .tokenValiditySeconds(86400);
```

Login/Logout

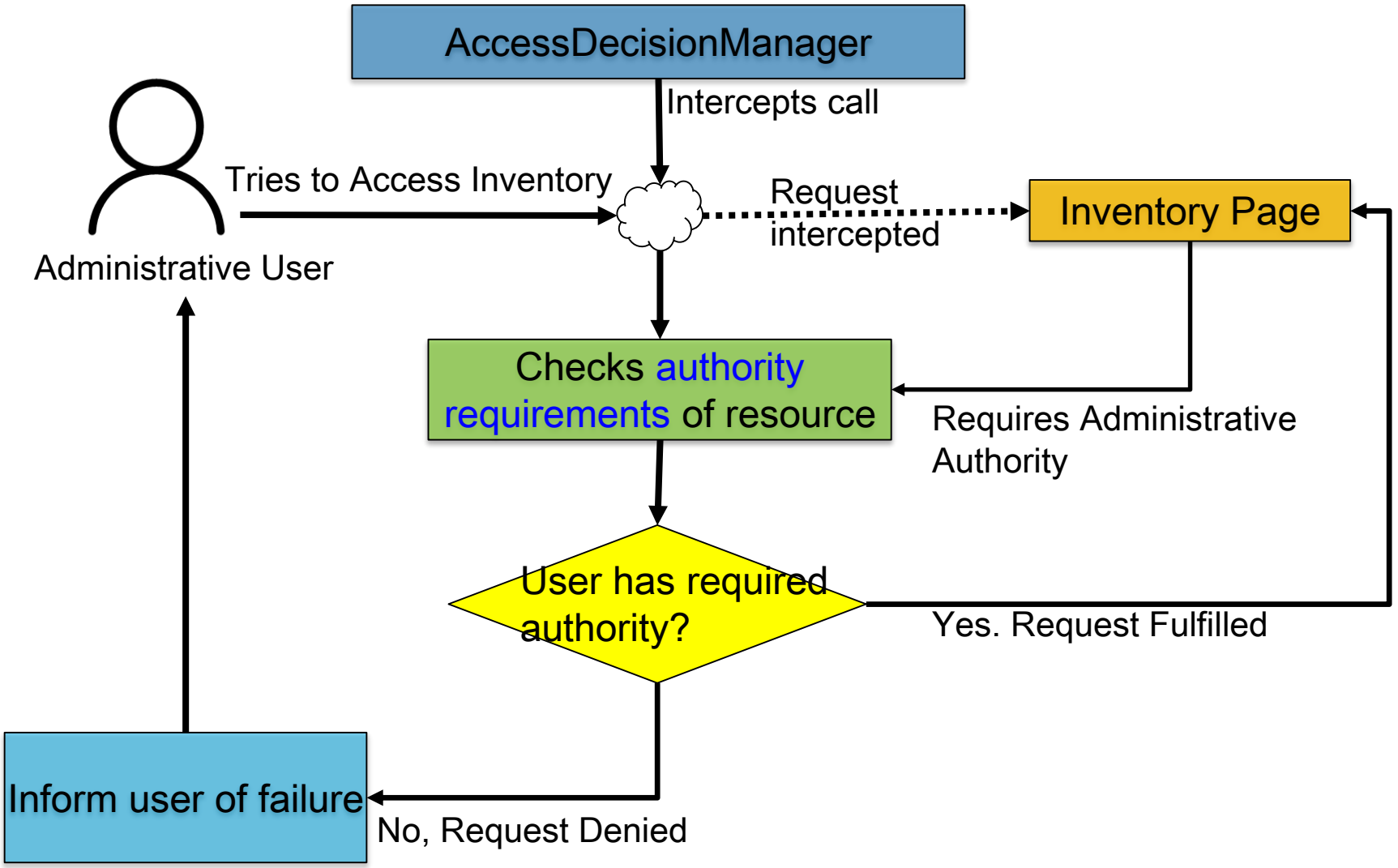
- 폼 로그인/로그아웃 인증 실패/성공 처리 설정 예시
  - 로그인

```
http.formLogin()
      .loginPage("/login")
      .defaultSuccessUrl("/home")
      .failureUrl("/login?error")
      .permitAll();
```

  - 로그아웃

```
http.logout()
      .logoutUrl("/logout")
      .logoutSuccessUrl("/login")
      .invalidateHttpSession(true);
```

Authorization



Authorization

- 인가 (Authorization)
  - 인증된 사용자에게 특정 Resource에 대한 접근(Access) 권한(Permission) 혹은 역할(Role)을 부여하거나 거부하는 과정
  - 예를 들어, 파일 공유 시스템에서 관리자는 /admin/** 경로 접근 가능, 일반 사용자는 차단됨
  - 실패 시 에러코드
    - 403 (Forbidden) 발생은 인가 실패 (인증은 성공적이나 이 사용자는 접근 권한이 없다는 뜻)
    - 401 (Unauthorized) 발생은 인증 자체가 실패한 경우 (로그인 필요)

Authorization

- 인가의 주요 요소
  - AccessDecisionManager
    - 접근 허용 여부를 결정 (Voter 방식)하는 주요 전략 인터페이스
  - Role/Authority
    - 사용자의 역할 또는 권한 (예: ROLE_USER, ROLE_ADMIN)
  - GrantedAuthority
    - 권한 표현하는 핵심 인터페이스
  - SimpleGrantedAuthority
    - GrantedAuthority 인터페이스의 구현체이며, 문자열 기반 권한 표현 (예: ROLE_ADMIN)
  - Authentication 객체
    - 인증 후, 사용자의 권한 목록 getAuthorities()를 포함
  - RBAC (Role-Based Access Control) 지원
    - 권한은 코드 또는 설정 파일에서 정의하여 접근 제어 수행
  - 접근 제어 방법
    - @PreAuthorize, hasRole(), hasAuthority() 등 메서드 수준 인가
    - .requestMatchers().hasRole() 등 URL 패턴 기반 인가

AccessDecisionManager

- AccessDecisionManager 인터페이스
  - AccessDecisionManger는 Voter를 통해 접근 허용 여부를 결정
  - AffirmativeBased, ConsensusBased, UnanimousBased

```
public interface AccessDecisionManager {
        void decide(Authentication authentication, // 사용자
                        Object object, // 접근 자원
                        Collection<ConfigAttribute> configAttributes) // 보안 설정
                throw AccessDeniedException, InsufficientAurhenticationException;
        boolean supports(ConfigAttribute attribute);
        boolean supports(Class<?> clazz);
}
```

URL-Based Authorization

- URL 기반 인가 설정 예시

```
http.authorizeHttpRequests()
    //Ant 패턴을 사용 가능, 요청이 /admin 하위 경로 ADMIN 권한 필요
    .requestMatchers("/admin/**").hasRole("ADMIN")
    //Ant 패턴을 사용 가능, 요청이 /user 하위 경로 USER/ADMIN 권한 필요
    .requestMatchers("/user/**").hasAnyRole("USER", "ADMIN")
    // 위에서 정의한 규칙 외 모든 엔드포인트 요청은 인증을 필요
    .anyRequest().authenticated();
```

  - hasRole("ADMIN")은 **내부적으로 hasAuthority("ROLE_ ADMIN")**
  - 요청 URL이 /admin/** 앤드 포인트일 경우 ADMIN 권한 필요
  - 요청 URL이 /user/** 앤드 포인트일 경우 여러 역할 중 USER 또는 ADMIN 하나라도 있으면 통과

URL-Based Authorization

```
http.authorizeHttpRequests(authorize -> authorize
        //요청이 /user 엔드포인트 요청인 경우 USER 권한 필요
        .requestMatchers("/user"). hasAuthority("ROLE_USER")
        //Ant 패턴을 사용 가능, 요청이 /mypage 하위 경로 USER 권한 필요
        .requestMatchers( "/mypage/**").hasRole("USER")
        //정규 표현식 사용 가능
        .requestMatchers(RegexRequestMatcher.regexMatcher("/resource/[A-
Za-z0-9]+")).hasAuthority("ROLE_USER")
        //HTTP METHOD 를 옵션으로 설정 가능
        .requestMatchers(HttpMethod.GET, "/**").hasAuthority("ROLE_READ")
        // POST 방식의 모든 엔드포인트 요청은 write 권한을 필요
        .requestMatchers(HttpMethod.POST).hasAuthority("ROLE_WRITE")
        //원하는 RequestMatcher를 직접 사용 가능
        .requestMatchers(new
AntPathRequestMatcher("/manager/**")).hasAuthority("MANAGER")
        //admin/ 이하의 모든 요청은 ADMIN 과 MANAGER 권한을 필요
        .requestMatchers("/admin/**").hasAnyAuthority("ROLE_ADMIN","ROLE_
MANAGER")
        // 위에서 정의한 규칙 외 모든 엔드포인트 요청은 인증을 필요
        .anyRequest().authenticated());
```

HttpSecurity.authorizeHttpRequests()

- 요청 기반 권한 부여
  - authorizeHttpRequests()는 사용자의 자원 접근을 위한 요청 엔드 포인트와 접근에 필요한 권한을 매핑시키기 위한 규칙을 설정
  - authorizeHttpRequests()를 통해 요청과 권한 규칙이 설정되면 내부적으로 AuthorizationFilter가 요청에 대한 권한 검사 및 승인 작업을 수행
  - anyRequest().authenticated()는 모든 앤드 포인트 요청에 대해 인증(authentication)이 필요함을 설정

requestMatchers()

- requestMatchers()
  - requestMatchers 메소드는 HTTP 요청의 URL 패턴, HTTP 메소드, 요청 파라미터 등을 기반으로 어떤 요청에 대해서는 특정 보안 설정을 적용하고 다른 요청에 대해서는 적용하지 않도록 세밀하게 제어 가능
  - 예를 들어 특정 API 경로에만 CSRF 보호를 적용하거나, 특정 경로에 대해 인증을 요구하지 않도록 설정 가능
  - 이를 통해 애플리케이션의 보안 요구 사항에 맞춰서 유연한 보안 정책을 구성 가능

API	정의
authenticated	인증된 사용자의 접근을 허용
fullyAuthenticated	아이디와 패스워드로 인증된 사용자의 접근을 허용, rememberMe 인증 제외
anonymous	익명사용자의 접근을 허용
rememberMe	기억하기를 통해 인증된 사용자의 접근을 허용
permitAll	요청에 승인이 필요하지 않는 공개 엔드포인트이며 세션에서 Authentication을 검색하지 않음
denyAll	요청은 어떠한 경우에도 허용되지 않으며 세션에서 Authentication을 검색하지 않음
access	요청이 사용자 정의 AuthorizationManager를 사용하여 액세스를 결정 (표현식 문법 사용)
hasAuthority	사용자의 Authentication에는 지정된 권한과 일치하는 GrantedAuthority가 있어야 함
hasRole	hasAuthority의 단축키로 ROLE_ 또는 기본접두사로 구성 **(ROLE_ 을 제외해야 함)**
hasAnyAuthority	사용자의 Authentication에는 지정된 권한 중 하나와 일치하는 GrantedAuthority가 있어야 함
hasAnyRole	hasAnyAuthority의 단축키로 ROLE_ 또는 기본 접두사로 구성 (ROLE_ 을 제외해야 함)

Method-Level Security

- Method-Level Security
  - 메서드 호출 전/후에 권한 검사를 수행하여 접근 제어를 구현
  - 주로 서비스 계층(Service Layer)에서 사용
- 메서드 보안 활성화 설정
  - @EnableMethodSecurity(prePostEnabled = true)은 @PreAuthorize, @PostAuthorize, @PostFilter, @Secured 등을 사용할 수 있도록 활성화
- 메서드 보안 적용
  - @Secured 메소드 실행 이전, 지정된 권한(문자열 기반)을 가진 사용자만 접근 허용
  - @PreAuthorize 메서드 실행 이전, 권한 검사(SpEL 기반 표현식)
  - @PostAuthorize 메서드 실행 이후, 반환값 기반 권한 검사
  - @PostFilter 메서드 실행 이후, 반환되는 컬렉션 필터링 수행

Method-Level Security

- Method-Level Security 예제

```
@Secured(“ROLE_CUSTOMER_READ”)
public Member getProjectsByName(Member member) { ... }

@PreAuthorize("hasRole('ADMIN')")
public String deleteMember(Long id) { ... }

@PreAuthorize(“hasRole(‘USER’)”)
@PostAuthorize(“hasPermission(returnObject. ‘read’)”)
public Member getMemberByName(String name) {
}

@PreAuthorize(“hasRole(‘USER’)”)
@PostFilter(“hasPermission(filterObject. ‘read’)”)
public List<Member> getMembers() {
}
```

WebSecurityCustomizer

- 정적 자원 보완 제외
  - 정적 자원이란 인증/인가 처리와 무관한 공용 리소스 (예: CSS, JavaScript, 이미지, 폰트, ico 등)
  - 성능 향상을 위해 보안 필터 체인에서 제외할 필요가 있음

```
@Bean
public WebSecurityCustomizer webSecurityCustomizer() {
        return (web) -> web.ignoring()
            .requestMatchers("/css/**", "/js/**", "/images/**", "/favicon.ico");
}
```

Session vs Stateless Authentication

- 세션 기반 인증
  - 사용자 로그인 시, 서버 측에서 세션(Session) 생성
  - 인증 정보는 서버 메모리 or 세션 저장소에 저장됨
  - 이후 요청마다 **세션 ID (JSESSIONID 쿠키)**를 통해 인증 상태 유지
  - 전통적 웹 어플리케이션, Form 인증에 적합
  - 서버상태 유지 필요

클라이언트 ——(로그인 요청)——▸ 서버
서버 ——(JSESSIONID 발급)——▸ 클라이언트
클라이언트 ——(요청 + JSESSIONID 쿠키)——▸ 서버 (세션 확인)

Session vs Stateless Authentication

- Stateless 인증
  - 인증 정보는 **JWT** 등 토큰에 포함하며, 서버에 상태 저장 없음
  - 매 요청마다 인증 정보를 HTTP Header에 포함시켜 전송
  - REST API, 모바일 백엔드, JWT 기반 인증 시스템에 사용
  - 서버 무상태 유지 및 확장성 높음
  - 토큰 탈취 시 보안 위험

클라이언트 ──(로그인 요청)──▸ 서버
서버 ──(JWT 발급)──▸ 클라이언트
클라이언트 ──(요청 + Authorization Header)──▸ 서버 (토큰 검증)

SessionManagement

- 세션 관리 설정
  - SessionCreationPolicy.STATELESS 세션 생성 안 함 REST API, JWT 인증 방식에 사용
  - SessionCreationPolicy.ALWAYS 항상 세션 생성
  - **SessionCreationPolicy.IF_REQUIRED (default)** 필요 시 세션 생성
  - SessionCreationPolicy.NEVER securit는 세션 사용 안 함
  - SessionCreationPolicy.INVALID_SESSION_URL("/login") 세션 만료 시 이동할 URL 설정

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http.sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS) // 세션 생성 안함
        .and()
        .authorizeHttpRequests(auth -> auth
        .anyRequest().authenticated());
    return http.build();
  };
```

CORS Configuration

- 동일 출처 정책 (Same-Origin Policy)
  - 브라우저 보안 정책 중 하나
  - 웹 페이지가 로드 된 출처(origin)와 다른 출처의 리소스를 불러오거나 조작하는 것을 제한
  - 보안상의 이유로 Cross-Site 요청 공격(CSRF 등) 방지 목적
  - Same Origin 기준은 **프로토콜(http/https) + 도메인(example.com) + 포트(3000)** 모두 동일해야 함
- Cross-Origin 문제
  - 서로 **다른 출처**에서 리소스를 요청하는 경우, **브라우저는 요청 차단**
  - 특히, XMLHttpRequest, fetch() 등 JavaScript를 통한 Ajax 호출 시 문제가 발생
  - 예를 들어, React 프론트엔드(localhost:3000) -> Spring Boot 백엔드(localhost:8080) 요청

CORS Configuration

- 해결방법은 CORS (Cross-Origin Resource Sharing)
  - 서버가 브라우저에게 **"특정 출처의 요청을 허용한다 "**고 명시적으로 선언
  - 이를 위해 HTTP 응답 헤더에 아래와 같은 CORS 관련 헤더(Access-Control-Allow-Origin) 포함
  - CORS 설정 없이는 브라우저가 요청 자체를 차단

Access-Control-Allow-Origin: http://localhost:3000
Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS
Access-Control-Allow-Headers: Content-Type, Authorization

CORS Configuration

- 다음과 같은 방식으로 CORS 설정 허용
  - @CrossOrigin 으로 컨트롤러/메소드 수준에서 허용
  - WebMvcConfigurer 전역 설정
  - http.cors().configurationSource(...)로 보안 필터와 통합

```
@RestController
@CrossOrigin(origins = "http://localhost:3000", allowedHeaders = "*") //
ApiContriller 클래스 내의 모든 HTTP 요청 처리에 대해 CORS 요청을 허용
public class ApiController {
    @GetMapping("/data")
    public Data getData() {
        return new Data();
    }
}
```

CORS Configuration

```
@Configuration
public class CorsConfig implements WebMvcConfigurer {
  @Override
  public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/**")
      .allowedOriginPatterns("http://localhost:3000") // Spring Boot 2.4+ 이후
      .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")
      .allowedHeaders("*") // 모든 헤더 허용
      .allowCredentials(true)  // 자격 증명(쿠키/인증) 허용 - allowedOriginPatterns("*")
와 함께 쓰면 동작하지 않음에 유의
      .maxAge(3600);
  }
}
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  http.cors(Customizer.withDefaults()) // WebMvcConfigurer 설정 사용
       .csrf(AbstractHttpConfigurer::disable) // crsf disable for REST
...
       return http.build();
}
```

CORS Configuration

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  http.cors(cors -> cors.configurationSource(request -> {
            CorsConfiguration config = new CorsConfiguration();
            config.setAllowedOriginPatterns(List.of("http://localhost:3000"));
            config.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE",
"OPTIONS"));
            config.setAllowedHeaders(List.of("*"));
            config.setAllowCredentials(true); // 인증 정보 포함 허용
            config.setMaxAge(3600L);
            return config;
          }))
      .csrf(AbstractHttpConfigurer::disable)
      .authorizeHttpRequests(auth -> auth
          .requestMatchers("/api/auth/**").permitAll()
          .anyRequest().authenticated());
  return http.build();
}
```

CSRF Protection

- CSRF (Cross-Site Request Forgery) 사이트간 요청 위조
  - CSRF 공격이란, 사용자가 인증된 세션을 보유한 상태에서, 악의적인 사이트가 사용자의 브라우저를 조작하여 서버에 의도치 않은 요청을 전송하도록 유도하는 공격
  - 공격자는 사용자의 인증 쿠키를 활용하여, 사용자 모르게 데이터 변경 요청(예를 들어, 결제, 비밀번호 변경 등)을 서버에 전송
- Spring Security는 기본적으로 CSRF 보호 활성화 (default)
  - 즉, CSRF protection을 통해 상태를 변화시킬 수 있는 POST, PUT, DELETE 요청으로부터 보호함
  - GET, HEAD, OPTIONS, TRACE는 일반적으로 보호 대상 아님

CSRF Protection

- HTML Form 기반 요청 시 CSRF 토큰 필요
  - Spring Security는 <input type="hidden" name="_csrf" value="..."> 를 HTML form에 자동 삽입
  - 서버는 세션 또는 쿠키를 통해 CSRF 토큰 값을 저장하고 검증
  - Form 기반 POST 요청 시, 아래와 같은 CSRF 토큰이 포함되어야 위조 요청을 방지하게 됨

```
<form method="post" action="/transfer">
<input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
<button type="submit">Transfer</button>
</form>
```

- REST API에서는 CSRF 비활성화가 일반적
  - RESTful API는 stateless 한 특성이므로 세션 기반 CSRF 보호가 적합하지 않음
  - 대신 JWT나 OAuth2 등 토큰 기반 인증 방식 사용
  - 아래와 같이 CSRF 보호를 명시적으로 비활성화

```
http.csrf().disable()
```

JWT Token

- JWT(JSON Web Token)
  - Header + Payload + Signature 로 구성된 Base64 인코딩 문자열
  - Header – 서명 알고리즘 및 타입 명시

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

  - Payload – 인증 정보 및 Claim 포함 (예를 들아, 사용자 ID, 권한 등)

```
{
  "sub": "user123",
  "role": "ROLE_USER",
  "iat": 1715270000,
  "exp": 1715273600
}
```

  - Signature – 비밀 키를 이용한 서명 (HMAC SHA-256 등)

```
HMACSHA256(
  base64UrlEncode(header) + "." + base64UrlEncode(payload),
  secret
)
```

JWT Token

- JWT 인증 흐름 (Stateless)
  - 사용자 로그인
    - 사용자가 자격 증명으로 인증 수행
    - 서버는 사용자 정보를 기반으로 JWT를 생성하여 클라이언트에게 전달
  - 요청 시 JWT 전달
    - 클라이언트는 **Authorization: Bearer <token>** 헤더로 **JWT 포함**
  - 서버는 JWT 검증
    - Signature, Expiration, Claim 확인
    - 인증 객체(SecurityContext)에 등록하여 인증 처리

JWT Token

- 사용자 인증 정보를 포함하여 **Stateless 인증** 구현 JWT 토큰 생성 예시

```
@Component
public class JwtTokenProvider {

    public String createToken(String username, String role) {
        Date now = new Date();
        Date expiry = new Date(now.getTime() + 3600000); // 1시간
        return Jwts.builder()
            .setSubject(username)
            .claim("role", role)
            .setIssuedAt(now)
            .setExpiration(expiry)
            .signWith(SignatureAlgorithm.HS256, secretKey)
            .compact();
    }
}
```

JWT Token

- JWT 인증 필터
  - JWT 인증은 세션이 없는 구조이므로 별도의 필터가 사용자 요청에 포함된 토큰을 파싱하고 인증 객체를 설정해야 함

```
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {
    private final JwtTokenProvider jwtTokenProvider;
    private final MemberDetailsService memberDetailsService;
    @Override
    protected void doFilterInternal(HttpServletRequest request,
            HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {
        String token = jwtTokenProvider.resolveToken(request);
        if (token != null && jwtTokenProvider.validateToken(token)) {
            String username = jwtTokenProvider.getUsernameFromToken(token);
            UserDetails userDetails = memberDetailsService.loadUserByUsername(username);
            UsernamePasswordAuthenticationToken authentication =
                new UsernamePasswordAuthenticationToken(userDetails, null,
userDetails.getAuthorities());
            SecurityContextHolder.getContext().setAuthentication(authentication);
        }
        filterChain.doFilter(request, response);                    }       }
```

JWT Token

- Spring SecurityConfig에 JWT 인증 필터 등록
  - **UsernamePasswordAuthenticationFilter 앞에** JWT 필터 등록

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

OAuth2

- OAuth2 (Open Authorization 2.0)
  - 제3자 어플리케이션이 사용자 리소스에 접근할 수 있도록 위임된 권한을 부여하는 프레임워크
  - 사용자 비밀번호 없이 인증 위임이 가능
  - 실생활 사례 (Google 로그인, 네이버 로그인, 카카오 로그인)
- OAuth2 구성요소와 역할

구성요소	역할
Resource Owner	리소스 소유자 (사용자)
Client	인증 위임받는 앱 (e.g. React, Android 앱)
Authorization Server	인증 및 **Access Token** 발급 서버 (Google, Naver)
Resource Server	보호된 자원 API (e.g. /api/user, /api/profile)

OAuth2

- OAuth2 인증 흐름 (Authorization 기준)
  - 사용자 -> 클라이언트 -> 인증 서버에 리다이렉션
  - 사용자가 로그인 및 권한 동의
  - 인증 서버가 Authorization Code 발급
  - 클라이언트는 Authorization Code -> Access Token 교환
  - 클라이언트는 Access Token으로 API(Resource Server) 호출

Spring Security OAuth2 Client

- Spring Security Google 로그인으로 JWT 발급까지 하려면, 먼저 Google API Console에서 OAuth2 클라이언트 등록
  1. Google Cloud Console 접속
     https://console.cloud.google.com/apis/credentials
  2. OAuth2.0 Client ID 생성
     - ApplicationType: Web Application
     - Authorized redirect URI:
       http://localhost:8080/login/oauth2/code/google
  3. Client 정보 확인
     - CLIENT_ID & CLIENT_SECRET

Spring Security OAuth2 Client

- application.properties 설정
  - Google Cloud Console에서 발급 받은 CLIENT_ID & _SECRET

```
# OAuth2 (Google)
spring.security.oauth2.client.registration.google.client-
id=YOUR_GOOGLE_CLIENT_ID
spring.security.oauth2.client.registration.google.client-
secret=YOUR_GOOGLE_CLIENT_SECRET
spring.security.oauth2.client.registration.google.redirect-
uri={baseUrl}/login/oauth2/code/{registrationId}
spring.security.oauth2.client.registration.google.scope=profile,email
spring.security.oauth2.client.registration.google.client-name=Google

spring.security.oauth2.client.provider.google.authorization-
uri=https://accounts.google.com/o/oauth2/v2/auth
spring.security.oauth2.client.provider.google.token-
uri=https://oauth2.googleapis.com/token
spring.security.oauth2.client.provider.google.user-info-
uri=https://www.googleapis.com/oauth2/v3/userinfo
spring.security.oauth2.client.provider.google.user-name-attribute=sub
```

Spring Security OAuth2 Client

- Spring Security에 설정

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http.oauth2Login(login -> login
            .loginPage("/login")
            .failureUrl("/login?error") // 로그인 실패 시 다시 login.html 렌더링
            .userInfoEndpoint(userInfo ->
userInfo.userService(customOAuth2UserService))
            .successHandler(oAuth2LoginSuccessHandler) // JWT 발급
            );
        return http.build();
    }
}
```

Spring Security OAuth2 Client

```
@Service @RequiredArgsConstructor
public class CustomOAuth2UserService extends DefaultOAuth2UserService {
    private final MemberRepository memberRepository;
    private final JwtTokenProvider jwtTokenProvider;
    @Override
    public OAuth2User loadUser(OAuth2UserRequest userRequest) throws
OAuth2AuthenticationException {
        OAuth2User oAuth2User = super.loadUser(userRequest);
        String email = oAuth2User.getAttribute("email");
        String name = oAuth2User.getAttribute("name");
        Member member = memberRepository.findByUserId(email)
            .orElseGet(() -> memberRepository.save(Member.builder()
                ...
                .memberType(MemberType.GOOGLE).build()));
        String token = jwtTokenProvider.createToken(member.getUserId(), List.of("ROLE_USER"));
        Map<String, Object> attributes = new HashMap<>(oAuth2User.getAttributes());
        attributes.put("jwt", token);
        return new DefaultOAuth2User(
            Collections.singleton(new SimpleGrantedAuthority("ROLE_USER")),
            attributes,
            "sub"
        );
    }       }
```

Spring Security OAuth2 Client

```
@Component
@RequiredArgsConstructor
public class OAuth2LoginSuccessHandler implements AuthenticationSuccessHandler {
    private final JwtTokenProvider jwtTokenProvider;

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request,
                                        HttpServletResponse response,
                                        Authentication authentication) throws IOException {
        OAuth2User oAuth2User = (OAuth2User) authentication.getPrincipal();
        String email = oAuth2User.getAttribute("email");

        String jwt = jwtTokenProvider.createToken(email, List.of("ROLE_USER"));

        // 로그인 후 Swagger로 리다이렉트
        //response.sendRedirect("/swagger-ui/index.html");
    }
}
```

Spring Security OAuth2 Client

- 사용자 정의 로그인 페이지에 Google 로그인 버튼 추가

```
<div class="login-box">

    <a th:href="@{/oauth2/authorization/google}">
        <button class="google-btn">Google 로그인</button>
    </a>

</div>
```

- Google 로그인 흐름
  - 사용자 -> login.html에서 Google 로그인 버튼 클릭
    -> "/oauth2/authorization/google“
    -> [Google OAuth2 인증] -> redirect "/login/oauth2/code/google“
    -> Spring Security가 access token 및 사용자 정보 요청
    -> CustomOAuth2UserService -> 사용자 등록 + JWT 생성
    -> OAuth2LoginSuccessHandler -> JWT & 로그인 완료

Spring Security Exception Handling

- 인증 및 인가 실패 처리
  - 인증 실패 AuthenticationEntryPoint
  - 인가 실패 AccessDeniedHandler

```
http.exceptionHandling()
    .accessDeniedPage("/access-denied")
    .authenticationEntryPoint(customEntryPoint);
```

Spring Security Test

- 보안테스트 필요성
  - 로그인 실패/성공 시나리오
  - 인가된 사용자만 리소스 접근 가능 확인
- 통합 테스트
  - 실제 Security 설정과 함께 MVC 요청 실행
- 단위 테스트
  - @WithMockUser를 사용하여 인증된 컨텍스트 주입

```
@WithMockUser(username = "user1", roles = {"USER"})
@Test
void testAccessWithUserRole() throws Exception {
    mockMvc.perform(get("/user/profile"))
            .andExpect(status().isOk());
}
```

Spring Security Test

```
@SpringBootTest
@AutoConfigureMockMvc
public class SecureApiIntegrationTest {
    @Autowired
    private MockMvc mockMvc;
    @Test
    void testUnauthenticatedAccessDenied() throws Exception {
        mockMvc.perform(get("/admin/dashboard"))
                .andExpect(status().isForbidden());
    }
    @Test
    void testWithSecurityContext() {
        UsernamePasswordAuthenticationToken auth =
            new UsernamePasswordAuthenticationToken("admin", "password",
                List.of(new SimpleGrantedAuthority("ROLE_ADMIN")));

        SecurityContextHolder.getContext().setAuthentication(auth);
        // 테스트 로직 수행
    }
}
```

pom.xml for Spring Security

```
<dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
                <groupId>org.springframework.security</groupId>
                <artifactId>spring-security-test</artifactId>
                <scope>test</scope>
        </dependency>
    ...
```

pom.xml for Spring Security

```
<!-- JWT -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId> <!-- or jjwt-gson -->
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
```

pom.xml for Spring Security

```
        <!-- Google OAuth2 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-oauth2-client</artifactId>
        </dependency>
        <!-- Swagger UI with Springdoc for Spring Boot 3.x+ -->
        <dependency>
            <groupId>org.springdoc</groupId>
            <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
            <version>2.8.8</version>
        </dependency>
…
</dependencies>
```

Configuration

```
@Configuration
@EnableWebSecurity // Spring Security 설정을 활성화하여 SecurityFilterChain 사용가능
public class SecurityConfig {
    @Bean
    public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder();  }
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.csrf().disable() // csrf 토큰 비활성화
            .authorizeHttpRequests((authorize)-> authorize.requestMatchers("/", "/login",
"/signup", "/members").permitAll()
            .requestMatchers("/delete/**", "/changepw/**").authenticated() // 보호
.anyRequest().authenticated())
            .formLogin((form) ->form.loginPage("/login")
                                    .loginProcessingUrl("/login") // Security가 로그인 진행
                                    .defaultSuccessUrl("/").permitAll()
            .logout((logout) -> logout
                                .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
                                .logoutSuccessUrl("/")
        return http.build();
```

UserDetails 구현

```
@Entity@Data@AllArgsConstructor
public class Member implements UserDetails {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;
    private String userId;     private String password;     private String email; …
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() { return List.of(new
SimpleGrantedAuthority("ROLE_USER"); }
    @Override
    public String getPassword() { return password; }
    @Override
    public String getUsername() { return username; }
    @Override
    public boolean isAccountNonExpired() { return true; }
    @Override
    public boolean isAccountNonLocked() { return true; }
    @Override
    public boolean isCredentialsNonExpired() { return true; }
    @Override
    public boolean isEnabled() { return true; }
}
```

UserDetailsService 구현

```
public interface MemberRepository extends JpaRepository<Member, Long> {
    Optional<Member> findByUserId(String userId); // userId로 조회
    void deleteByUserId(String userId); // userId로 삭제
}

@Service
@RequiredArgsConstructor
public class MemberService implements UserDetailsService {
    private final MemberRepository memberRepository;

    @Override
    public UserDetails loadUserByUsername(String username) {
        return memberRepository.findByUserId(username)
                .orElseThrow(() -> new IllegalArgumentException((username)));
    }
...
}
```