Spring Boot Security

강사 강태광

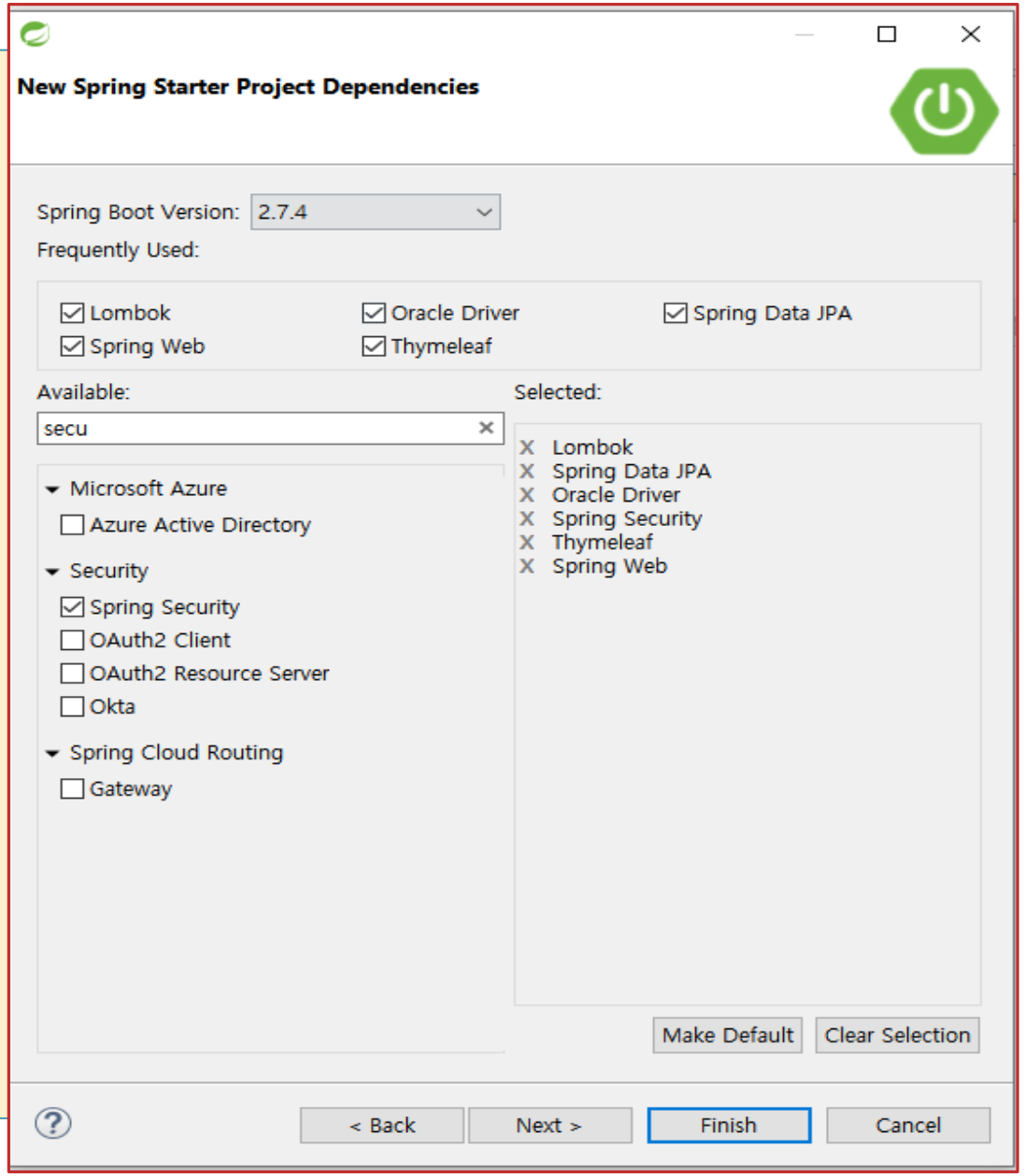
1-1. Spring boot Security 개발 설정

1. 왼쪽 화면과 같이 개발 설정

2. Spring Security 주요 architecture
 1) Authentication(인증)
  - 사용자가 누구인지 확인
  - 사용자의 인증정보를 저장하는 Token
   ① principal : 사용자Id 또는 User객체
   ② credentials: 사용자 비밀번호
   ③ authorities : 인증된 사용자 권한 목록
   ④ details: 인증 부가 정보
   ⑤ authenticated : 인증 여부

  - 인증후 **securityContext**에 저장 ,
   전역적으로 참조[Authentication 객체가
   저장되는 저장소
  - 인증 완료되면 **HttpSession**에 저장
   Application 전반에 걸쳐 참조 가능

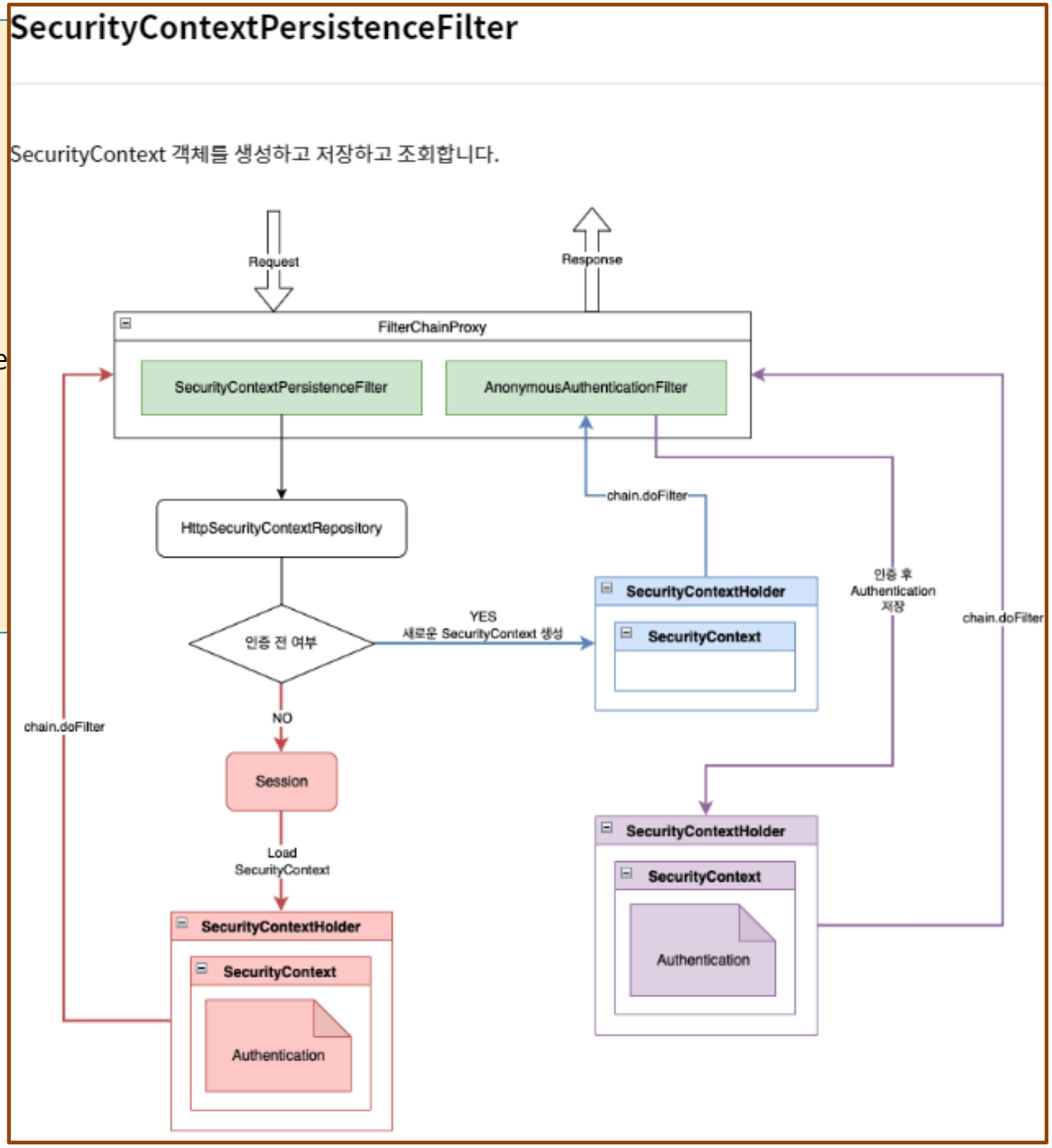
 2) Authorization(인가)
  - 사용자가 요청을 실행할수 있는 권한이 있는지
   확인하는 절차



1-2-1. 환경 및 yml 설정

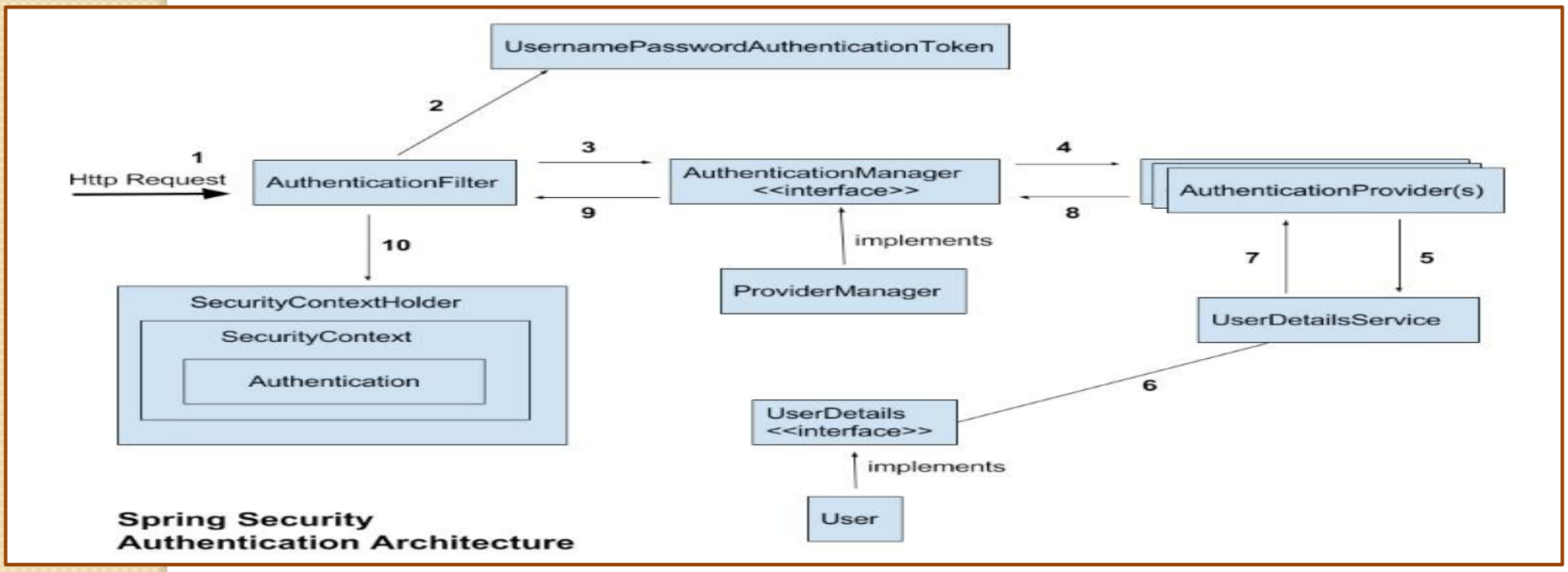
1. 환경설정 (DB→oracle)

```
server:
  port: 8781
# Oracle Connect
spring:
  datasource:
    driver-class-name: oracle.jdbc.driver.
                       OracleDriver
    url:  jdbc:oracle:thin:@localhost:1521/xe
    username: scottsecurity
    password: tiger
# JPA
  jpa:
    show-sql: true
    hibernate:
      ddl-auto: none
```

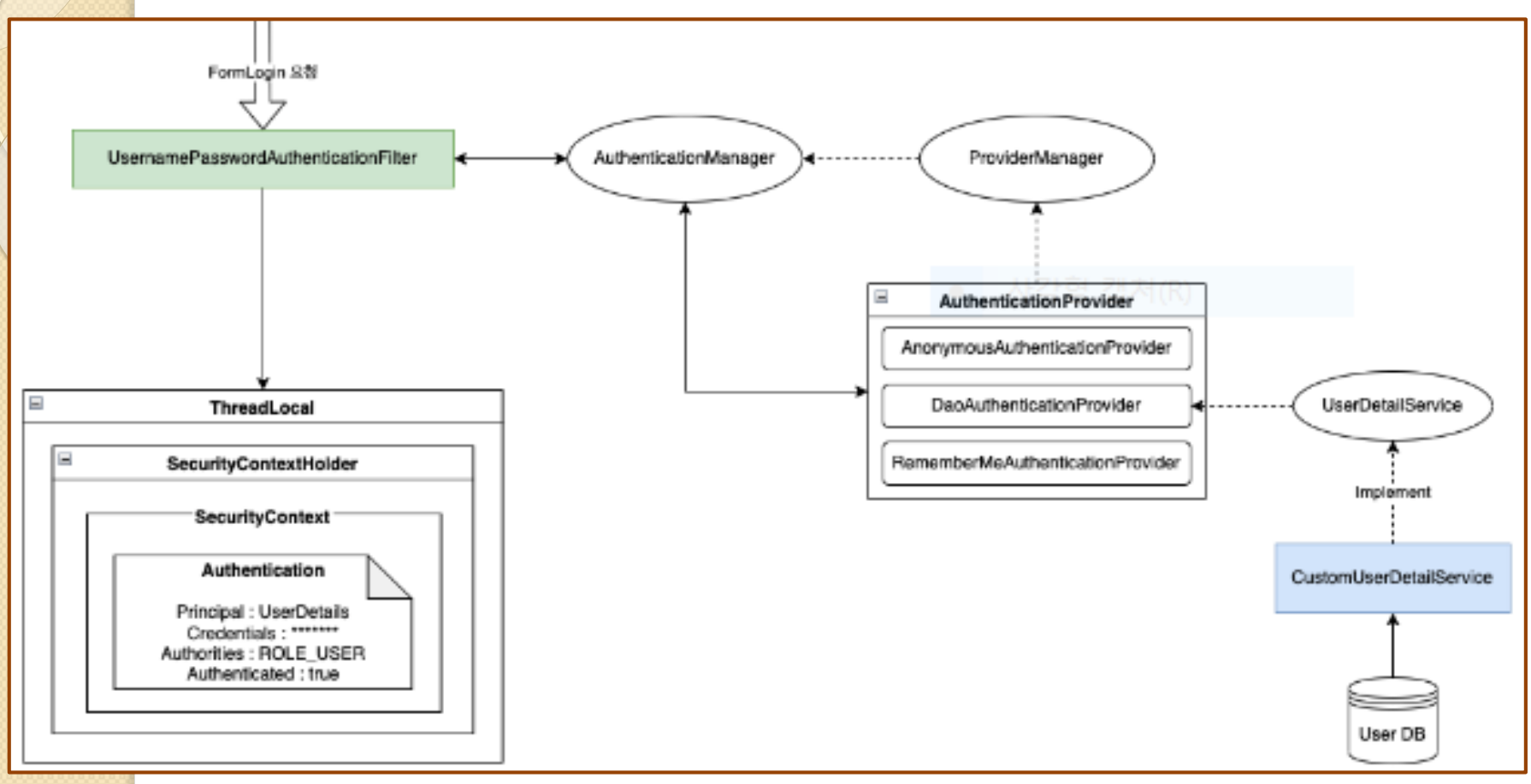


1-2-2. security Process

1. security Process 흐름
① 사용자가 입력한 사용자 정보를 가지고 인증을 요청.(Request)
② AuthenticationFilter가 이를 가로채 UsernamePasswordAuthenticationToken(인증용 객체)를 생성
③ 필터는 요청을 처리하고 AuthenticationManager의 구현체 ProviderManager에 Authentication과 UsernamePasswordAuthenticationToken을 전달.
④ AuthenticationManager는 검증을 위해 AuthenticationProvider에게 Authentication과 UsernamePasswordAuthenticationToken을 전달.
⑤ 이제 DB에 담긴 사용자 인증정보와 비교하기 위해 **UserDetailsService**에 **사용자 정보**를 넘겨줌. **DB에서 찾은 사용자 정보인 UserDetails 객체**를 만든다.
⑥ **AuthenticationProvider**는 UserDetails를 넘겨받고 비교.
⑦ 인증이 완료되면 권한과 사용자 정보를 담은 Authentication 객체가 반환.
⑧ AuthenticationFilter까지 Authentication정보를 전달.
⑨ Authentication을 SecurityContext에 저장.



1-3-1. Authentication flow



1. Form Login요청
2. **UserNamePasswordAuthenticationFilter**에서 **AuthenticationProvider**에게 인증을 위임
3. **AuthenticationProvide**에서 **User**객체를 조회해 **userDetails Type**으로 가져옴
4. **AuthenticationProvide**에서 **userDetails 와** 권한정보를 인증후 Token 객체를 생성 해 **AuthenticationManager**로 전달
5. **UsernamePasswordAuthenticationFilter**로 **Authentication 객체**가 전달되고 **securityContext에** 인증객체를 저장한 뒤 응답

1-3-2. Authentication Process

1. 익명 사용자
 ① 새로운 **securityContext 객체** 생성 후, **securityContextHolder**에 저장
 ② AnonymousAuthenticationFilter에서 AnonymousAuthenticationFilter Token 객체를 **securityContext**에 저장

2. 인증시
 ① 새로운 **securityContext** 객체 생성 후, **securityContextHolder**에 저장
 ② **UserNamePasswordAuthenticationFilter**에서 **인증 성공후, securityContext에 UserNamePasswordAuthentication**객체 저장
 ③ 인증 최종 완료후 Session에 **securityContext**를 저장

3. 인증후
 ① Session에 **SecurityContext를 꺼내어 , securityContextHolder에 저장**
 ② **SecurityContext**안에 **Authentication** 객체 존재시, 계속 인증을 유지

1-4-1. SecurityConfig 환경 작업

1. SecurityConfig
 ① IoC 빈(bean)을 등록 : **@Configuration**
 ② 필터 체인 관리 시작 어노테이션 : @EnableWebSecurity
 ③ 특정 주소 접근시 권한 및 인증을 위한 어노테이션 활성화
  - prePostEnabled = **true, securedEnabled = true**
2. BCryptPasswordEncoder Bean 등록
 1) 암호화 모듈 추가이유
  - 스프링 시큐리티를 활용하기 위해서는 DB에 해쉬로 암호화된 패스워드를 저장
   - BCryptPasswordEncoder에 encode 메서드를 통해 해쉬암호화를 사용하여 패스워드를 저장
  - BCrypt strong hashing function

 2) SecurityFilterChain
  ① 스프링 버전이 올라가면서 WebSecurityConfigurerAdapter 에도 deprecate
  ② SecurityFilterChain으로 연결 정의

 3) SecurityFilterChain **스프링시큐리티 설정**
  ① **antMatchers :** 특정 리소스에 대해서 권한을 설정
  ② **permitAll :** antMatchers 설정한 리소스의 접근을 인증절차 없이 허용
  ③ **anyRequest :** **모든 리소스를 의미**하며 접근허용 리소스 및 인증후 특정 레벨의 권한을 가진 사용자만
  접근가능한 리소스를 설정하고 그외 나머지 리소스들은 무조건 인증을 완료해야 접근 가능
  ④ **formLogin :** 로그인 페이지와 기타 로그인 처리 및 성공 실패 처리를 사용
  ⑤ **loginPage :** 사용자가 따로 만든 로그인 페이지를 사용하려고 할때 설정
  설정하지 않으면 디폴트 URL이 "/login", 때문에 "/login"로 호출하면 스프링 제공하는 기본 로그인페이지가 호출
  ⑥ **loginProcessingUrl :** 로그인 즉 인증 처리를 하는 URL을 설정.
  "/login-process" 가 호출되면 인증처리를 수행하는 필터가 호출

  ⑦ **defaultSuccessUrl :** 정상적으로 인증성공 했을 경우 이동하는 페이지를 설정

1-4-2. SecurityFilterChain 처리 설정

1. SecurityConfig 안에 SecurityFilterChain filterChain(HttpSecurity http) bean 설정

```
@Bean
protected SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http.csrf().disable();
http.authorizeRequests()
        .antMatchers("/user/**").authenticated()  // /user/** 은 인증 필요 --> 인증만 되면 들어갈수 있음
        .antMatchers("/manager/**").access("hasRole('ROLE_ADMIN') or hasRole('ROLE_MANAGER')")
        //.antMatchers("/admin/**").access("hasRole('ROLE_ADMIN') and hasRole('ROLE_USER')")
        .antMatchers("/admin/**").access("hasRole('ROLE_ADMIN')")
        .anyRequest().permitAll()
            .and()
        .formLogin()
        //.loginPage("/loginForm")
         // login이라는 주소가 호출이 되면, Security가 낚아 채서 login 진행
        // Controller에  /login 만들지 않아도 됨
        .loginProcessingUrl("/login")
        // .loginProcessingUrl("/loginProc")
        .failureUrl("/loginFail")
        .defaultSuccessUrl("/");

return http.build();
}
```

1-5. Error 처리 설정

1. SecurityController 에서 ErrorController 를 implements

```
@Controller
public class SecurityController implements ErrorController
```

2. ErrorController 안에서 URL(/error) 정의

```
 @GetMapping("/error")
 public String error() {
    System.out.println("error 403 start...");
    return "error";
 }
```

3. error.html 생성하여 구현

위의 두과정을 생략해도 error.html은 기본적으로 적용됨

1-6. 암호화 처리

1. SecurityConfig **에서** BCryptPasswordEncoder**를 Bean으로 등록**

2. SecurityController에서 암호화 하고자 한다면 아래 적용

```
@Autowired
private BCryptPasswordEncoder bCryptPasswordEncoder;
```

3.코딩구현

```
String encPassword = bCryptPasswordEncoder.encode(rawPassword);
```

4. 암호화의 개념 및 유형

1) 암호화의 개념
- 정보를 노출시키지 않기 위해 특정 알고리즘을 이용하여 암호화된 형태로 변형
- 암호화 알고리즘은 수학적인 과정을 통해 특정한 정보를 의미가 없는 문자로 나열

2) 방법
- 혼돈: 메시지 원문의 내용을 짐작하기 어렵게 만드는 것을 의미
- 확산: 암호화 알고리즘의 패턴 추론을 어렵게 만드는 역할
- 치환(Substitution) : 문자열을 다른 문자열로 이동 및 교체
- 이동(Transposition) : 원칙에 따라 문자열 순서를 무작위로 바꿔 이동

3) 암호화 유형

① 대칭 암호화 : 발신자와 수신자가 동일한 개인 암호화 키를 사용하여 암호화된 메시지를 디코딩하고 인코딩(DES,SEED,AES,Aria..) , 비밀키

② 비대칭 암호화: 개인정보를 보호하기 위해 대칭 암호화보다는 더 고차원적이고 안전한 방법 공개키는 발신자와 수신자 모두 동일한 키를 가지고 있지만 개인키는 각 당사자가 고유한 키 (RSA,Rabin, **ELGamal**)

③ **해쉬(Hash)** : 임의길이를 해시함수(해시 알고리즘)를 이용하여 고정된 길이의 암호화된 문자열
MD5, SHA-1, HAS-180은 사용하면 안된다. SHA-256, SHA-512등을 사용하기를 권고함.
참고로 SHA-512가 보안이 더 좋음
bcrypt : 패스워드를 위해 탄생해서 아주 강력한 해시 알고리즘이 적용됨.
Spring security에서 사용

1-7-1. Owasp 10 예방

1. Owasp 대표적 해킹 기법인 XSS와 CSRF 예방 처리의 Spring Security 사용 권장
   1) XSS(Cross-Site Scripting)
      - 크로스 사이트 스크립팅(사이트 간 스크립팅)은 SQL injection과 함께 웹 상에서 가장 기초적인 취약점 공격 방법의 일종으로, 악의적인 사용자가 공격하려는 사이트에 스크립트를 넣는 기법
   2) CSRF (Cross-Site Request Fogery)
      - Site간 요청 위조(또는 크로스 사이트 요청 위조, CSRF, XSRF)는 웹 사이트 취약점 공격의 하나로, 사용자가 자신의 의지와는 무관하게 공격자가 의도한 행위(수정, 삭제, 등록 등)를 특정 웹사이트에 요청하게 하는 공격

XSS의 방지 대책	CSRF의 방지 대책
1. 쿠키에 중요한 정보를 담지 않고 서버에 중요정보를 저장하는 방법 2. 정보를 암호화 하는 방법 3. httponly 속성 on (자바스크립트의 document.cookie를 이용해서 쿠키에 접속하는 것을 막는 옵션으로, 쿠키를 훔쳐가는 행위를 막기 위한 방법이다.) 4. Secure coding (str_replace <치환함수로 XSS방지 대책이지만 미흡한 부분이 있다 -> url encoding> htmlspecialchars <html 엔터티로 변환해줌> ..등 보안과 관련된 함수를 사용한다.) ▶ Encoding : 특정 플랫폼에서 문자를 표현하기 위한 규약	1. Referrer 검증 (Back -end 단에서 request 의 referrer을 확인하여 domain이 일치하는지 검증하는 방법) 2. Security Token 사용 (Reffer검증이 불가한 환경일 시, Security Token을 활용한다. 사용자 세션에 임의의 난수 값을 저장하고, 사용자의 요청마다 해당 난수 값을 포함 시켜 전송한다. 이후 Back -end 단에서 요청을 받을 때마다 세션에 저장된 토큰 값과 요청 파라미터에 전달되는 토큰 값이 일치한지 검증하는 방법) 두 방법 모두 같은 도메인 내에 XSS취약점이 있다면 CSRF 공격에 취약 해질 수 있다. 출처 : https://dlsdn73.tistory.com/634

1-7-2. Owasp 10 예방

1. Owasp 대표적 해킹 기법인 XSS와 CSRF 예방 처리의 Spring Security 사용 권장
   1) XSS(Cross-Site Scripting) / CSRF (Cross-Site Request Fogery) 비교

항목	XSS(Cross-Site Scripting)	CSRF(Cross-Site Request Forgery)
개요	악성 스크립트가 클라이언트에서 실행	권한을 도용당한 클라이언트가 가짜 요청을 서버에 전송
공격 대상	클라이언트	서버
목적	쿠키・세션 갈취, 웹사이트 변조 등	권한 도용