

## 10 장 스프링 시큐리티

---

스프링 프레임워크를 이용해서 웹 어플리케이션을 개발할 때 빠질 수 없는 것이 사용자의 인증/인가를 처리하는 스프링 시큐리티의 접목입니다.

이 장에서는 스프링 시큐리티를 이용해서 로그인/아웃 처리 방식을 개발하고, 자동 로그인, 접근 제한등의 처리를 어떻게 하는지 알아보도록 합니다. 이 장에서 학습하는 내용은 다음과 같습니다.

- 스프링 시큐리티의 설정
- 인증처리와 사용자 데이터베이스 처리
- 자동로그인 처리
- 어노테이션을 이용한 접근 제한 처리

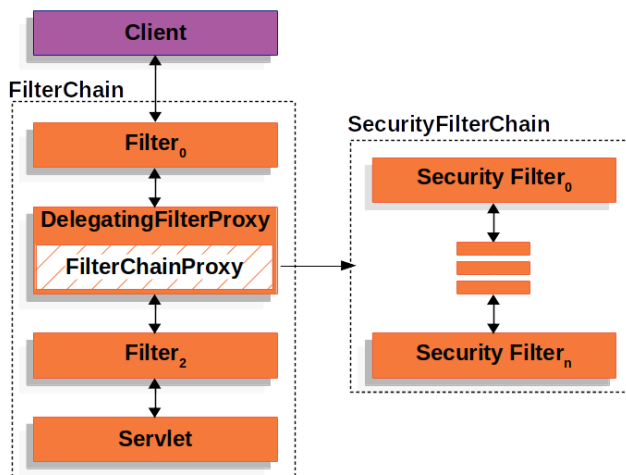
## 10.1 스프링 시큐리티 역할

이 장에서는 스프링 시큐리티를 이용한 인증/인가 처리를 진행합니다. 거의 모든 웹 어플리케이션의 경우 흔히 로그인이라고 하는 사용자 인증을 이용합니다. 과거에 서블릿 기반을 이용하는 경우 인증에 대한 처리는 흔히 세션이라고 부르는 HttpSession 을 이용하거나 쿠키(Cookie)를 이용해서 이러한 문제를 해결합니다.

스프링 시큐리티는 기본적으로 HttpSession 을 이용해서 사용자의 인증/인가를 처리합니다. 스프링 시큐리티는 기본적인 흐름이 이미 만들어져 있는 구조에 인증과 관련된 부분만을 개발해서 추가해 주는 것만으로 이러한 처리가 가능합니다.

예제에서 사용하는 스프링 시큐리티는 엄밀하게 말하면 스프링 웹 시큐리티입니다. 스프링 시큐리티의 경우 전체 보안 프레임워크로, 인증(authentication), 인가(authorization), 암호화, 세션 관리 등을 제공하는 Spring 기반의 보안 라이브러리이고 스프링 웹 시큐리티는 이 기능의 일부만을 사용하는 것입니다(책에서 스프링 시큐리티라는 용어는 엄밀하게는 스프링 웹 시큐리티라고 보면 됩니다.).

스프링 웹 시큐리티는 HTTP 요청에 대해서 여러 개의 필터를 만들어서 동작합니다.



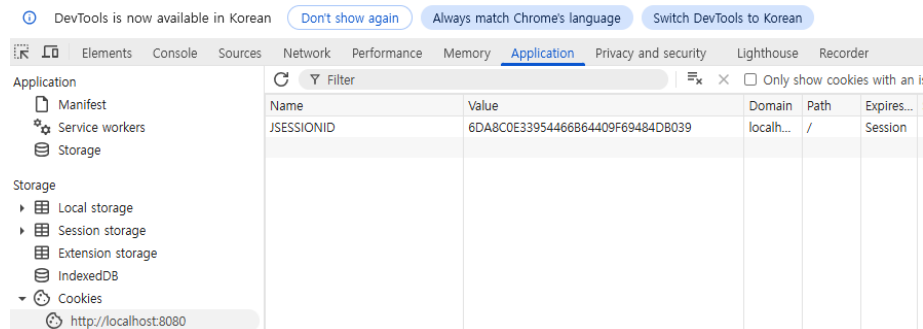
(출처: <https://docs.spring.io/spring-security/reference/6.2/servlet/architecture.html>)

스프링 웹 시큐리티는 위의 그림과 같이 기본적인 필터들이 있지만 추가적인 필터를 추가하거나 제거할 수도 있어서 개발자가 원하는 흐름을 구성할 수 있습니다.

## 스프링 웹 시큐리티와 HttpSession

스프링 시큐리티의 장점 중 하나는 다양한 방식으로 사용자의 인증/인가 처리를 수행할 수 있다는 것입니다. 예제의 경우 기본적으로 제공되는 방식으로 사용자의 정보를 처리하게 되는데 이 때 사용되는 방식은 서블릿에서 사용하는 HttpSession 입니다.

HttpSession 을 이용해서 로그인 정보를 처리하기 때문에 브라우저에서 사용하는 세션 쿠키가 중요합니다.

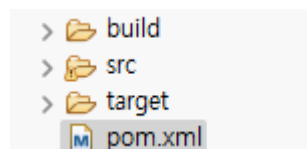


개발 과정에서 로그 아웃이 필요한 경우에는 위의 화면에서 'JSESSIONID' 쿠키를 삭제해서 더 이상 HttpSession 이 사용되지 않도록 처리하면 됩니다(로그아웃은 조금 뒤쪽에서 다루기 때문에 개발 단계에서 새로 로그인을 해야 하는 경우에는 'JSESSIONID'를 삭제하면 됩니다.).

## 10.2 스프링 시큐리티 설정

스프링 프레임워크는 의존성 주입을 지원하는 핵심적인 라이브러리에 필요에 따라 여러 라이브러리들을 추가해서 개발할 수 있습니다. 대부분의 경우에는 동일한 버전으로 맞추는 것이 일반적이지만 스프링 시큐리티의 경우 버전업이 빠른 편이라 별도의 버전을 쓰는 경우가 많습니다. 예제에서 사용하는 버전은 6.2.4 버전을 이용하도록 합니다.

pom.xml 에 라이브러리를 추가합니다.



```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-config</artifactId>
  <version>6.2.4</version>
</dependency>
```

```

<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-web</artifactId>
  <version>6.2.4</version>
</dependency>

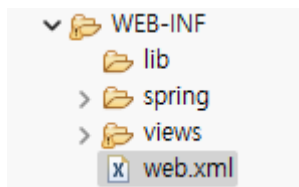
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-taglibs</artifactId>
  <version>6.2.4</version> <!-- 또는 사용하는 Spring Security 버전에 맞게 -->
</dependency>

```

## Java 설정을 위한 @Configuration

스프링 프레임워크의 설정 방법은 XML 기반의 설정과 Java 기반 설정으로 나누어 질 수 있는데 스프링 시큐리티 6 버전을 사용하는 경우는 Java 설정을 이용할 것을 권장합니다.

web.xml 에는 스프링 시큐리티가 동작할 때 사용하는 필터를 위한 설정을 아래와 같이 추가합니다.



...생략

```

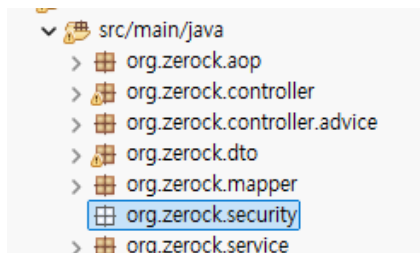
<servlet-mapping>
  <servlet-name>appServlet</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>

<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

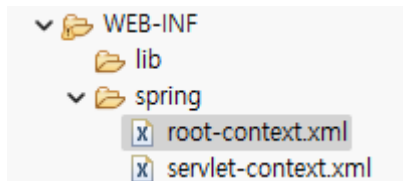
</web-app>

```

시큐리티에 대한 설정은 별도의 security 라는 패키지를 이용해서 구성합니다.



root-context.xml 에는 org.zerock.security 패키지를 scan 하도록 설정합니다.



...생략

```
<mybatis-spring:scan base-package="org.zerock.mapper"/>

<context:component-scan base-package="org.zerock.aop"></context:component-scan>

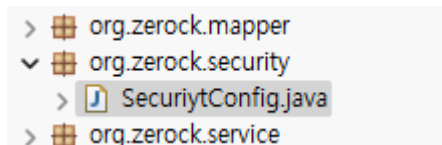
<context:component-scan base-package="org.zerock.security"></context:component-scan>

<aop:aspectj-autoproxy/>

<tx:annotation-driven/>

</beans>
```

security 패키지내에는 SecurityConfig 클래스를 작성합니다. SecurityConfig 클래스 내에는 @EnableWebSecurity 어노테이션과 @Configuration 어노테이션을 추가합니다. @Configuration 은 XML 설정 대신에 Java 설정을 사용하기 위한 어노테이션이고 @EnableWebSecurity 의 경우 프로젝트가 실행될 때 스프링 시큐리티 관련 필터들을 활성화시키는 역할을 합니다.



```
package org.zerock.security;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.web.SecurityFilterChain;

import lombok.extern.log4j.Log4j2;

@Configuration
@Log4j2
@EnableWebSecurity
public class SecuriytConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

        Log.info("-----security config-----");

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

정상적으로 설정이 완료되었다면 서버의 로그에는 log.info()로 작성된 부분이 동작하는 것을 확인할 수 있습니다.

```
748) DEBUG HikariPool-1 - Added connection org.mariadb.jdbc.Connection@35ae7c72
IG HikariPool-1 - After adding stats (total=2/10, idle=2/2, active=0, waiting=0)
INFO -----security config-----
tyFilterChain.<init>(DefaultSecurityFilterChain.java:54) INFO Will secure any request with
tionContext(ContextLoader.java:288) INFO Root WebApplicationContext initialized in 1661 ms
```

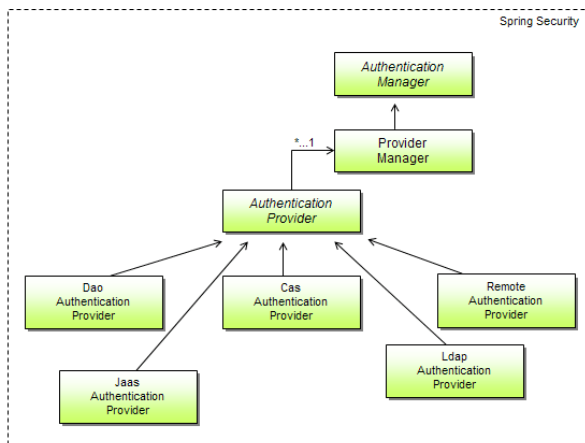
## 10.3 인증(authentication) 과 인가(authorization)

스프링 시큐리티에서 가장 중요한 개념은 사용자가 자신을 증명하는 인증(authentication)과 서버 내부에서 인증 과정을 통과한 사용자에게 권한을 부여하는 인가(authorization)입니다.

‘인증’은 사용자가 자신을 스스로 증명하는 행위이므로 흔히 로그인에 필요한 사용자의 아이디와 비밀번호 등을 전달하는 것을 의미합니다. 개발시에 주의해야 하는 점은 스프링 시큐리티에서는 사용자의 아이디는 username 이라는 용어를 이용하고 비밀번호는 password 라는 단어를 사용한다는 점입니다. 스프링 시큐리티에서는 사용자는 User 라는 용어로 사용되므로 개발시에 주의해야 합니다.

### 인증 매니저와 인증 제공자

스프링 시큐리티 내부에서 인증을 처리하는 과정은 인증 매니저(AuthenticationManager)와 인증 제공자(AuthenticationProvider)를 통해서 이루어집니다.



(출처- <https://docs.gigaspaces.com/latest/security/introducing-spring-security.html>)

인증 매니저는 여러 개의 인증 제공자를 관리하는 존재입니다. 따라서 실제 동작은 인증 제공자를 통해서 이루어지는데 위의 그림에서 볼 수 있듯이 여러 종류의 인증 제공자를 가질 수 있습니다. 일반적으로 인증 처리는 인증 제공자를 정해진 방식으로 개발해서 이를 추가하는 방식으로 개발합니다.

## UserDetailsService 인터페이스

인증 제공자가 여러 종류의 구현체를 가지는 방법은 각 구현체가 동일하게 UserDetailsService 라는 인터페이스를 구현하고 있기 때문입니다. UserDetailsService 는 loadByUsername( )이라는 단 하나의 메소드를 가지고 있는데 개발자는 이를 이용해서 사용자에게 대한 정보를 반환합니다.

**Package** org.springframework.security.core.userdetails

**Interface** UserDetailsService

**All Known Subinterfaces:**  
UserDetailsManager

**All Known Implementing Classes:**  
CachingUserDetailsService, InMemoryUserDetailsManager, JdbcDaoImpl, JdbcUserDetailsManager, LdapUserDetailsService, LdapUserDetailsService

---

public interface **UserDetailsService**

Core interface which loads user-specific data.

It is used throughout the framework as a user DAO and is the strategy used by the DaoAuthenticationProvider.

The interface requires only one read-only method, which simplifies support for new data-access strategies.

**See Also:**  
DaoAuthenticationProvider, UserDetails

---

**Method Summary**

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method	Description
UserDetails	loadUserByUsername(String <sup>Ⓔ</sup> username)	Locates the user based on the username.

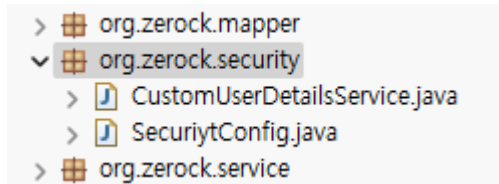
UserDetailsService 의 loadByUsername( )의 반환 타입은 UserDetails 라는 인증과 인가에 필요한 정보를 알 수 있도록 구성되는 인터페이스입니다. 아래 화면은 UserDetails 인터페이스 내부에 선언된 메서드들로 username, password 와 함께 권한(Authority)을 사용하는 것을 볼 수 있습니다.

**Method Summary**

All Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type	Method	Description	
Collection <sup>Ⓔ</sup> extends GrantedAuthority	getAuthorities()	Returns the authorities granted to the user.	
String <sup>Ⓔ</sup>	getPassword()	Returns the password used to authenticate the user.	
String <sup>Ⓔ</sup>	getUsername()	Returns the username used to authenticate the user.	
default boolean	isAccountNonExpired()	Indicates whether the user's account has expired.	
default boolean	isAccountNonLocked()	Indicates whether the user is locked or unlocked.	
default boolean	isCredentialsNonExpired()	Indicates whether the user's credentials (password) has expired.	
default boolean	isEnabled()	Indicates whether the user is enabled or disabled.	

## UserDetailsService 의 구현과 로그인

작성된 security 패키지에 CustomUserDetailsService 라는 클래스를 작성하고 UserDetailsService 를 구현하는 코드를 아래와 같이 작성합니다.



```
package org.zerock.security;

import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import lombok.extern.log4j.Log4j2;

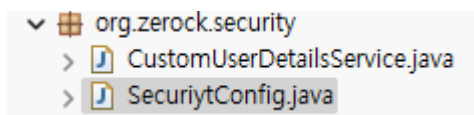
@Service
@Log4j2
public class CustomUserDetailsService implements UserDetailsService{

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        Log.info("-----loadUserByUsername-----" , username);
        // TODO Auto-generated method stub
        return null;
    }
}
```

CustomUserDetailsService 는 사용자의 인증 과정에서 동작하게 됩니다.

스프링 시큐리티의 설정을 담당하고 있는 SecurityConfig 클래스에 formLogin()이라는 기능을 추가합니다(스프링 5 버전과 6 버전의 가장 큰 차이 중 하나가 바로 설정에서 람다식을 이용하는 설정으로 변경된 점입니다.).



```
package org.zerock.security;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.web.SecurityFilterChain;

import lombok.extern.log4j.Log4j2;

@Configuration
```



```

@Log4j2
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

        log.info("-----security config-----");

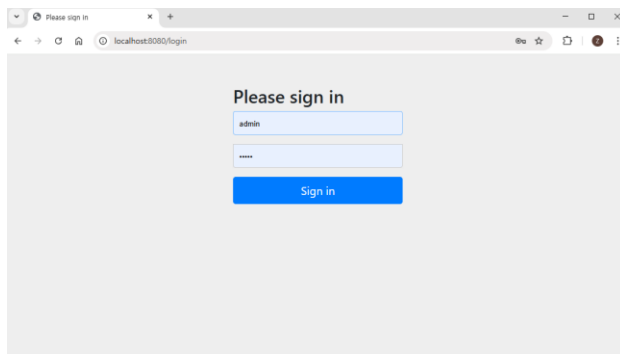
        http.formLogin(config -> {

        });

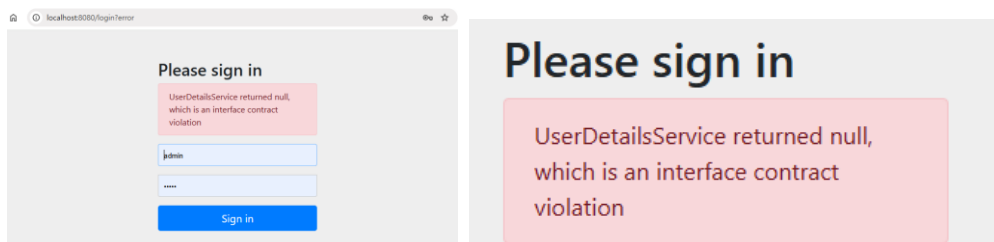
        return http.build();
    }
}

```

formLogin()설정이 추가된 후에 프로젝트를 실행하면 스프링 시큐리티가 기본적으로 제공하는 '/login'이라는 경로가 생성됩니다.



'/login' 화면에서는 아직 정상적으로 로그인에 되지 않지만 임의의 값을 이용해서 로그인을 시도해 보면 아래와 같은 실패 화면을 보게 됩니다.



서버의 내부에서는 CustomUserDetailsService 클래스의 로그가 기록된 것을 확인할 수 있습니다.

```

CustomUserDetailsService.loadUserByUsername(CustomUserDetailsService.java:18) INFO -----loadUserByUsername-----
AuthenticationFilter org.springframework.security.web.authentication.AbstractAuthenticationProcessingFilter.doFilter(AbstractAuthen
AuthServiceException: UserDetailsService returned null, which is an interface contract violation
AuthenticationProvider.retrieveUser(DaoAuthenticationProvider.java:105) ~[spring-security-core-6.2.4.jar:6.2.4]

```

위의 화면에서 에러 메시지를 자세히 보면 CustomUserDetailsService 의 메서드가 null 을 반환한 것이 문제라는 것을 알 수 있습니다.

loadUserByUsername( )에서 UserDetails 타입을 반환하는 코드를 추가해 봅니다. UserDetails 는 인터페이스이므로 이를 모두 구현할 수도 있지만 org.springframework.security.core.userdetails.User 클래스를 이용해서 구현하는 것이 편리합니다.

```
@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

    Log.info("-----loadUserByUsername-----" , username);

    UserDetails user = User.builder()
        .username(username)
        .password("1111")
        .roles("USER") //ROLE_USER
        .build();

    return user;
}
```

사용자의 정보는 크게 username(아이디), password(패스워드) 와 함께 권한데이터를 같이 반환합니다. 스프링 시큐리티에서 사용하는 권한은 기본적으로 'ROLE\_'로 시작하는 코드 값을 사용하는데 위와 같이 'USER'로 지정하면 'ROLE\_USER'라는 권한의 이름이 됩니다.

## PasswordEncoder

위의 설정을 이용하기 위해서 서버를 재시작하고 브라우저에서 다시 로그인을 시도하면 아래와 같이 500 에러가 발생하는 것을 볼 수 있습니다. 에러의 원인은 'PasswordEncoder'라는 것을 지정하지 않았기 때문입니다.



## HTTP 상태 500 – 내부 서버 오류

타임 예외 보고

**메시지** There is no PasswordEncoder mapped for the id "null"

**설명** 서버가, 해당 요청을 충족시키지 못하게 하는 예기치 않은 조건을 맞닥뜨렸습니다.

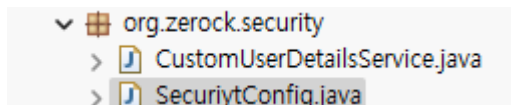
**예외**

```
java.lang.IllegalArgumentException: There is no PasswordEncoder mapped for the id "null"
    org.springframework.security.crypto.password.DelegatingPasswordEncoder$UnmappedIdPasswordEncoder.matches(DelegatingPasswordEncoder.java:289)
    org.springframework.security.crypto.password.DelegatingPasswordEncoder.matches(DelegatingPasswordEncoder.java:237)
    org.springframework.security.authentication.dao.DaoAuthenticationProvider.additionalAuthenticationChecks(DaoAuthenticationProvider.java:86)
    org.springframework.security.authentication.dao.AbstractUserDetailsAuthenticationProvider.authenticate(AbstractUserDetailsAuthenticationProvider.java:182)
    org.springframework.security.authentication.ProviderManager.authenticate(ProviderManager.java:201)
    org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter.attemptAuthentication(UsernamePasswordAuthenticationFilter.java:231)
    org.springframework.security.web.authentication.AbstractAuthenticationProcessingFilter.doFilter(AbstractAuthenticationProcessingFilter.java:221)
    org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:374)
    org.springframework.security.web.authentication.logout.LogoutFilter.doFilter(LogoutFilter.java:107)
```

스프링 시큐리티는 기본적으로 password에 대해서는 해시화(암호화)를 할 수 있도록 설정되어야 합니다.

다행히 스프링 시큐리티 라이브러리 내부에는 BCryptPasswordEncoder라는 클래스를 제공합니다.

SecurityConfig 내에 @Bean을 이용해서 PasswordEncoder 타입을 구현한 객체를 반환하도록 설정합니다.



```
package org.zerock.security;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
```

```
import lombok.extern.log4j.Log4j2;
```

```
@Configuration
```

```
@Log4j2
```

```
@EnableWebSecurity
```

```
public class SecurityConfig {
```

```
    @Bean
```

```
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
```

```
        log.info("-----security config-----");
```

```
        http.formLogin(config -> {
```

```
        });
```

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

```
    }
```

```
    @Bean
```

```

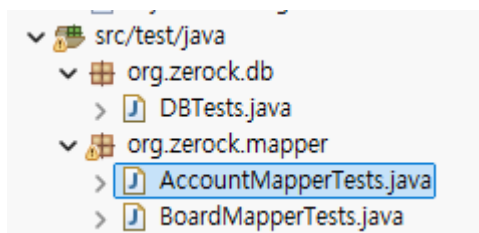
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}

```

BCryptPasswordEncoder 는 비밀번호 암호화를 위해서 개발된 알고리즘을 구현한 것으로 '단방향'암호화 처리됩니다. '단방향'이라는 의미는 쉽게 말해서 암호화된 문자열을 다시 원본으로 되돌릴 수 없는 방식입니다. 복호화 대신 특정한 문자열로 암호화된 문자열이 나올 수 있는지에 대한 확인만 가능하기 때문에 암호화된 값이 노출되어도 원래의 단어를 알 수 없습니다.

BCryptPasswordEncoder 에 대해서는 반드시 테스트 코드를 통해서 동작 여부를 확인하고 이를 현재 구현 중인 CustomUserDetailsService 에 사용해야만 합니다.

test 폴더에 나중에 만들 AccountMapperTests 클래스를 추가합니다.



```

package org.zerock.mapper;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import lombok.extern.log4j.Log4j2;

@ExtendWith(SpringExtension.class)
@ContextConfiguration("file:src/main/webapp/WEB-INF/spring/root-context.xml")
@Log4j2
public class AccountMapperTests {

    @Autowired
    private PasswordEncoder encoder;

    @Test
    public void testEncoding() {

        String pw = "1111";

        String enPw = encoder.encode(pw);

        Log.info(enPw);

        Log.info("-----");

        boolean match = encoder.matches(pw, enPw);
    }
}

```

```

    Log.info(match);
}
}

```

작성된 테스트 코드의 내용을 보면 '111'이라는 문자열을 PasswordEncoder 를 이용해서 암호화 하고 암호화된 결과가 '1111'이라는 글자로 만들어질 수 있는지 확인하는 것입니다.

테스트 코드를 여러 번 실행해 보면 매번 다른 값으로 암호화되는 것을 확인할 수 있습니다. 아래의 결과는 3 번 테스트 코드를 실행했을 때 매번 다르게 문자열이 만들어지는 것을 확인할 수 있습니다.

```

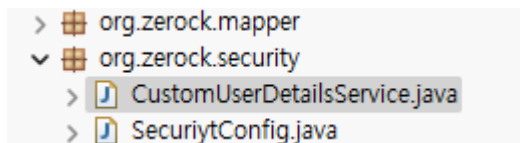
INFO $2a$10$enwJ3wifBzdzTQ7EucGAUOD8P7k9eYBtPeIC5oaD63sVKS0buCnLW
INFO -----
INFO true

INFO $2a$10$EPGguk22b31bM1YxtnjTauUU8Lvq1DyOV11uhps7B65WBfLXza/i2
INFO -----
INFO true

INFO $2a$10$0NQtfGchYMOQKt2VBEVPOBNezYuk.BfDwOaO.zJPQjzvpa1Sbh7q
INFO -----
INFO true
.....

```

테스트 코드에서 '1111'에 대한 암호화된 값을 이용해서 현재 작성중인 CustomUserDetailsService 에 반영합니다.



```

@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

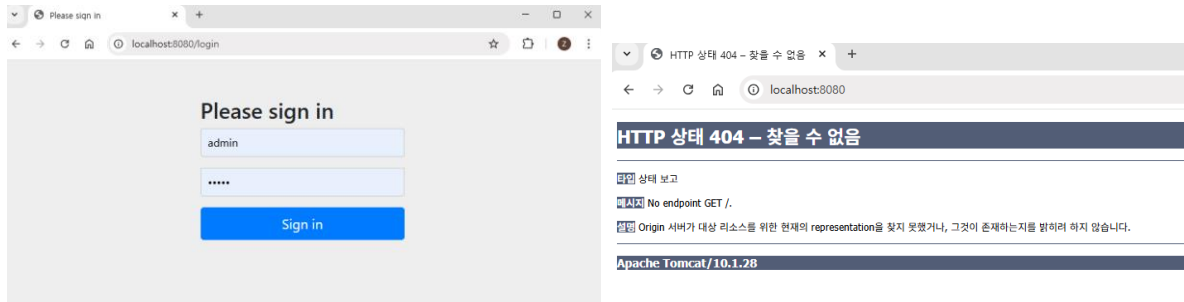
    Log.info("-----loadUserByUsername-----" , username);

    UserDetails user = User.builder()
        .username(username)
        .password("$2a$10$0NQtfGchYMOQKt2VBEVPOBNezYuk.BfDwOaO.zJPQjzvpa1Sbh7q")
        .roles("USER") //ROLE_USER
        .build();

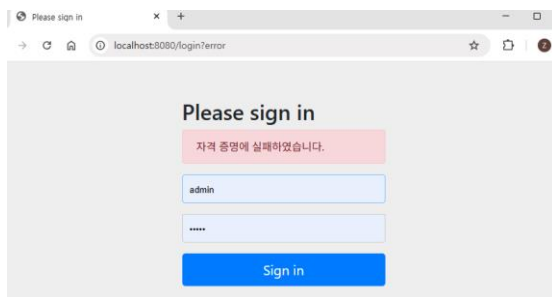
    return user;
}

```

이제 화면에서 패스워드가 '1111'로 로그인을 시도하면 이전과 다르게 '/' 경로로 이동하는 것을 볼 수 있습니다.

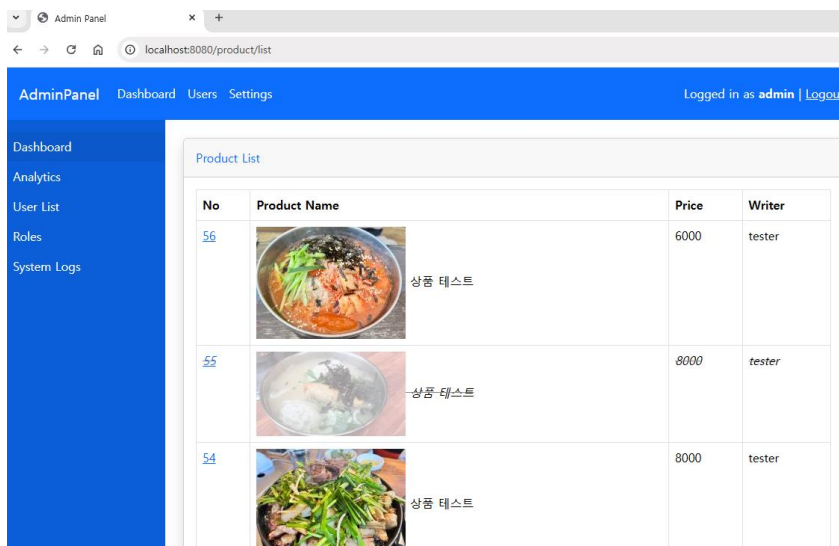


만일 '1111'이 아닌 다른 값으로 로그인을 시도하면 '자격 증명에 실패하였습니다'라는 에러메시지를 볼 수 있게 됩니다.



## CSRF 설정

사용자의 로그인이 처리된 후에 이전에 개발한 상품이나 게시물에 대한 GET 방식의 호출에는 아무런 문제가 없는 것을 확인할 수 있습니다.



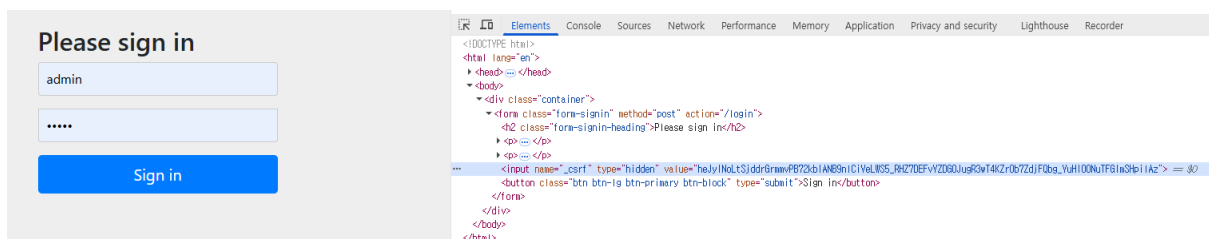
하지만 GET 방식 이외에 POST 방식 등의 호출은 정상적으로 이루어지지 않는 문제가 발생하는데 이것은 CSRF 토큰의 설정의 문제입니다.

CSRF 토큰을 이해하기 위해서는 우선 'CSRF'라는 용어 자체에 대해서 알아야 합니다. 'CSRF'는 'Cross-Site Request Forgery'의 약어로 '사이트간 요청 위조'라는 다소 생소한 용어입니다.

CSRF 공격은 현재 로그인한 사용자가 할 수 있는 요청을 '자신도 모르게 요청'하도록 하는 공격 방식입니다. 예를 들어 단순히 화면에서 '이벤트 당첨'과 같은 버튼을 클릭했을 뿐인데 자신도 모르게 자신이 로그인한 다른 사이트에 뭔가를 요청하도록 하는 공격 방식입니다.

CSRF 공격을 막기 위해서 스프링 웹 시큐리티는 기본적으로 POST,PUT,DELETE,PATCH 방식의 호출에 대해서 현재의 사용자만이 사용할 수 있는 별도의 문자열을 생성해서 이 값이 일치할 경우에만 요청을 처리하는데 이를 CSRF 토큰이라고 합니다.

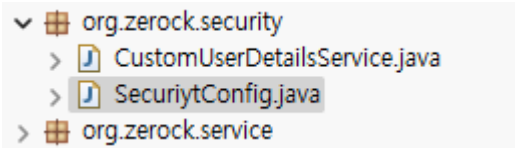
예를 들어 '/login'을 호출했을 경우에 만들어지는 화면에서도 <input type='hidden'>으로 만들어진 CSRF 토큰 값이 만들어 진 것을 확인할 수 있습니다.



만일 현재의 코드에서 CSRF 토큰을 전송하도록 수정한다면 JSP 에서 아래와 같이 `$_csrf.parameterName`등을 이용해서 추가할 수 있습니다(실제 서비스에서는 CSRF 토큰을 이용하는 것이 안전합니다.).

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

CSRF 토큰을 사용하고 싶지 않은 경우에는 SecurityConfig 의 설정을 통해서 조정할 수 있습니다.



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

    Log.info("-----security config-----");

    http.formLogin(config -> {

    });

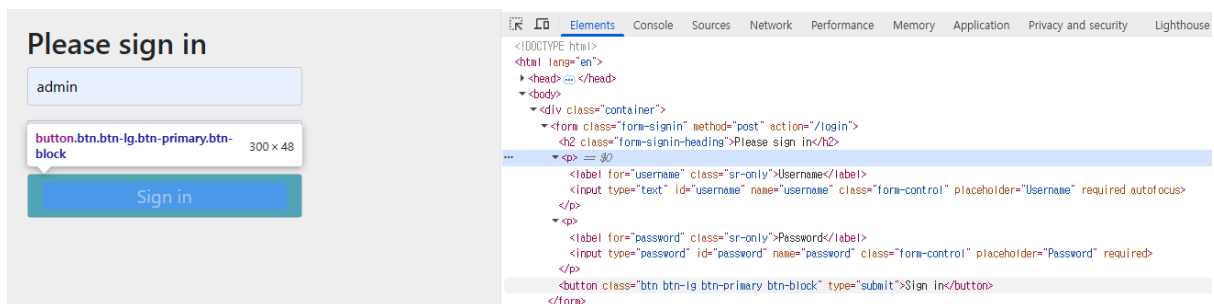
    http.csrf(config -> {

        config.disable();

    });

    return http.build();
}
```

위의 설정이 적용된 후에 다시 '/login'을 확인해 보면 CSRF 토큰을 위해서 만들어지는 <input type='hidden'..> 부분이 생성되지 않는 것을 확인할 수 있습니다.



## 10.4 사용자 권한 체크

현재까지 작성된 코드에서 로그인한 사용자는 'USER'라는 권한을 가지는 사용자입니다. 이것을 이용해서 특정한 권한이 있는 사용자만이 특정한 기능을 수행할 수 있도록 제어할 수 있습니다.

스프링 시큐리티에서는 사용자의 권한을 체크하는 부분을 표현식(expression)을 이용해서 작성할 수 있고 이 설정은 XML 을 이용하거나 권한 체크를 위해서 사용하는 @PreAuthorize 와 같은 어노테이션을 이용하는 방식이 있습니다.

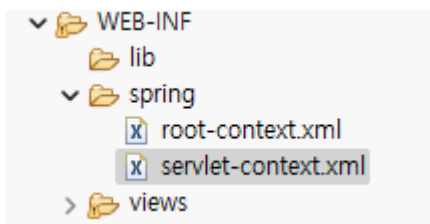
게시물 관리에서 권한 체크가 필요하다면 아래와 같이 정리할 수 있습니다.



경로	권한	표현식
/board/list	모든 사용자 접근 가능	permitAll( )
/board/read/11	로그인한 사용자만 접근 가능	hasRole('USER') 혹은 isAuthenticated( )

## @PreAuthorize 를 위한 설정

컨트롤러에서 어노테이션을 이용해서 권한을 체크하고 사용하기 위해서는 설정을 담당하는 servlet-context.xml 설정의 조정이 필요합니다.



servlet-context.xml 의 설정에는 security 와 관련된 네임스페이스를 추가합니다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"

  xmlns:security="http://www.springframework.org/schema/security"

  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/mvc
    https://www.springframework.org/schema/mvc/spring-mvc.xsd
    http://www.springframework.org/schema/context
    https://www.springframework.org/schema/context/spring-context.xsd

    http://www.springframework.org/schema/security
    https://www.springframework.org/schema/security/spring-security.xsd
  ">
...

```

servlet-context.xml 의 마지막 부분에 <security...> 를 아래와 같이 추가합니다.

```
<mvc:resources mapping="/images/**"
  location="file:C:/upload/" />

<bean id="multipartResolver"
  class="org.springframework.web.multipart.support.StandardServletMultipartResolver"/>

<security:global-method-security pre-post-annotations="enabled"/>

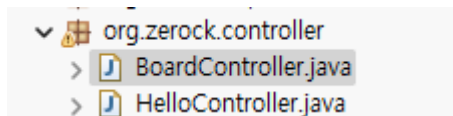
```

</beans>

설정된 pre-post-annotations 의 역할은 메서드의 실행 전에 권한을 체크하는 @PreAuthorize 와 실행 후 체크하는 @PostAuthorize 어노테이션을 사용할 수 있도록 허용할 것인지를 결정합니다.

## 표현식 적용

추가적인 설정 없이 BoardController 의 게시물 조회를 처리하는 read( )에 @PreAuthorize 를 적용해 봅니다. read( )는 사용자가 로그인한 사용자들만이 접근할 수 있도록 isAuthenticated( )를 적용해 봅니다.



```
@PreAuthorize("isAuthenticated()")
@GetMapping("read/{bno}")
public String read( @PathVariable("bno")Long bno, Model model ) {

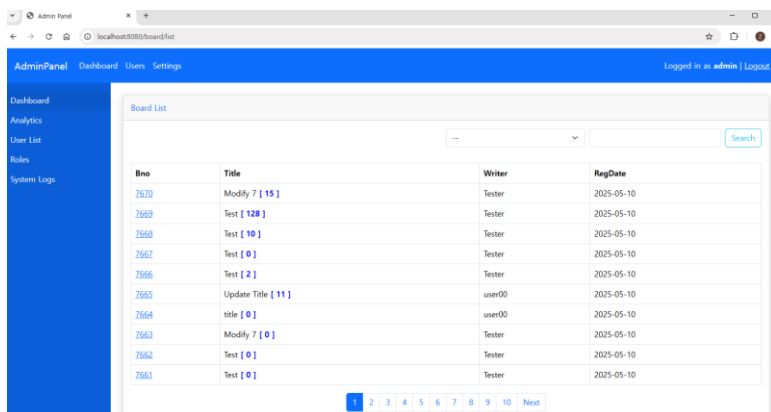
    Log.info("-----");
    Log.info("board read");

    BoardDTO dto = boardService.read(bno);

    model.addAttribute("board", dto);

    return "/board/read";
}
```

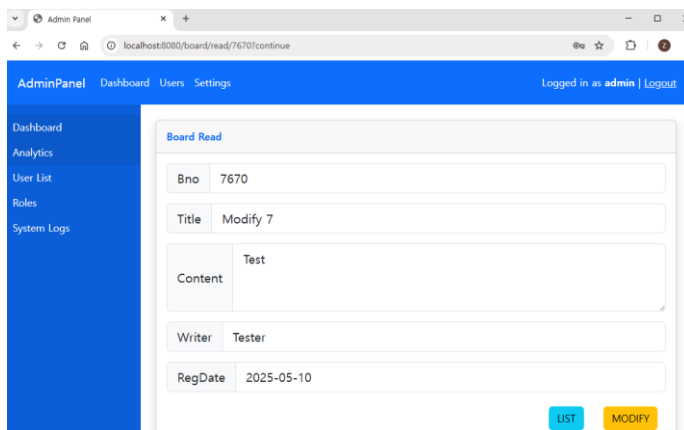
적용 후 프로젝트를 재시작하고 로그인이 안 된 상태에서 '/board/list'경로를 이용해서 게시물 목록을 먼저 확인합니다.



로그인이 안 된 상태에서는 특정 게시물을 조회할 수 없어야 정상입니다. 특정 게시물을 선택한 순간 로그인 화면으로 이동하게 됩니다.



로그인이 정상적으로 실행되면 원래 사용자가 원했던 경로로 자동으로 이동하게 됩니다. 스프링 웹 시큐리티는 로그인에 필요한 화면으로 이동할 때 해당 경로를 저장하고 로그인 후에는 이를 다시 호출하는 기능을 가지고 있습니다.



자주 사용하는 표현식은 다음과 같습니다.

- `permitAll()` - 모두 허용
- `denyAll()` - 모두 차단
- `isAuthenticated()` - 인증된 사용자만
- `isAnonymous()` - 인증되지 않은 사용자만
- `hasRole()`, `hasAnyRole()` - 특정 권한 혹은 여러 권한 중 하나라도 해당 권한이 있는 사용자만
- `hasAuthority()`, `hasAnyAuthority()` - 특정 권한 혹은 여러 권한 중 하나 (Role 과 달리 'ROLE\_'를 사용하지 않는 권한)

## 파라미터 값의 일부와 권한 체크

@PreAuthorize 내부에 작성하는 표현식은 SpEL(Spring Expression Language) 규격에 맞는 문자열인데 이를 이용하면 산술적인 계산식부터 조건식과 같이 다양한 연산이 가능합니다. 이 중에서 가장 유용한 표현식은 메서드에 전달되는 파라미터와 현재 사용자에 대한 정보를 표현식을 통해서 연산 처리하는 것입니다.

예를 들어 게시물의 경우 현재 사용자와 게시물의 작성자(writer)가 일치하는 경우에만 동작이 가능하도록 설정한다면 다음과 같이 지정할 수 있습니다.

```
@PostMapping("modify")
@PreAuthorize("authentication.name == #boardDTO.writer")
public String modifyPOST( BoardDTO boardDTO ) {

    Log.info("-----");
    Log.info("board modify post");

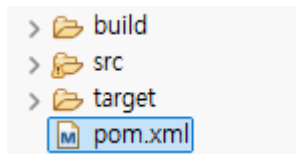
    boardService.modify(boardDTO);

    return "redirect:/board/read/" + boardDTO.getBno();
}
```

표현식의 내부에서 사용하는 authentication 은 시큐리티에서는 현재 사용자의 인증 정보를 의미하는 객체이고 name 은 인증 과정에서 사용되는 username 값을 의미합니다.

'#'을 이용하면 전달되는 파라미터의 설정을 이용할 수 있는데 이를 사용하려면 메서드의 파라미터의 이름을 표현식에서 인식할 수 있도록 설정을 조정해 주어야 합니다.

pom.xml 의 마지막에 존재하는 <plugin>설정의 <compilerArgs>를 추가해야 합니다.

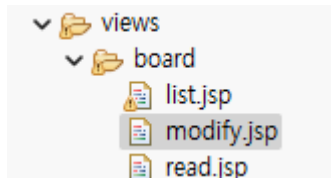


```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.13.0</version>
      <configuration>
        <release>17</release>

        <compilerArgs>
          <arg>-parameters</arg>
        </compilerArgs>

      </configuration>
    </plugin>
    ...생략
  </plugins>
</build>
```

동작 확인을 위해서 WEB-INF/views/board/modify.jsp 에서 writer 값을 전달하도록 조정합니다.



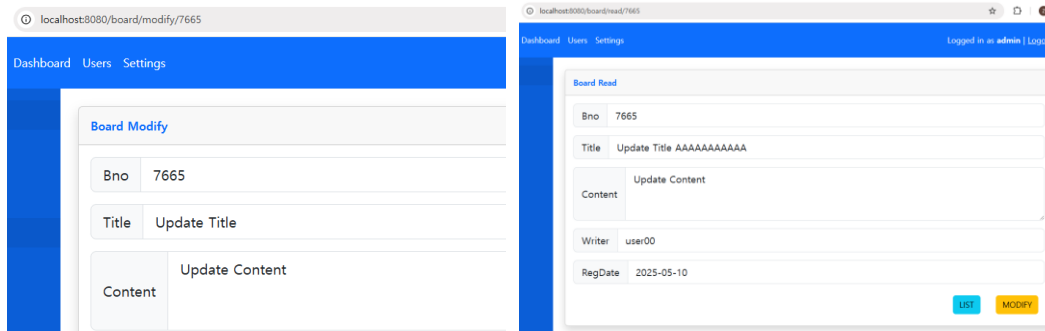
```
<div class="mb-3 input-group input-group-lg">
  <span class="input-group-text">Writer</span>
  <input type="text" class="form-control" name="writer" value="<c:out value='${board.writer}'>" readonly>
</div>
```

프로젝트를 실행해서 게시물의 목록에서 테스트하려는 게시물의 작성자를 확인합니다.

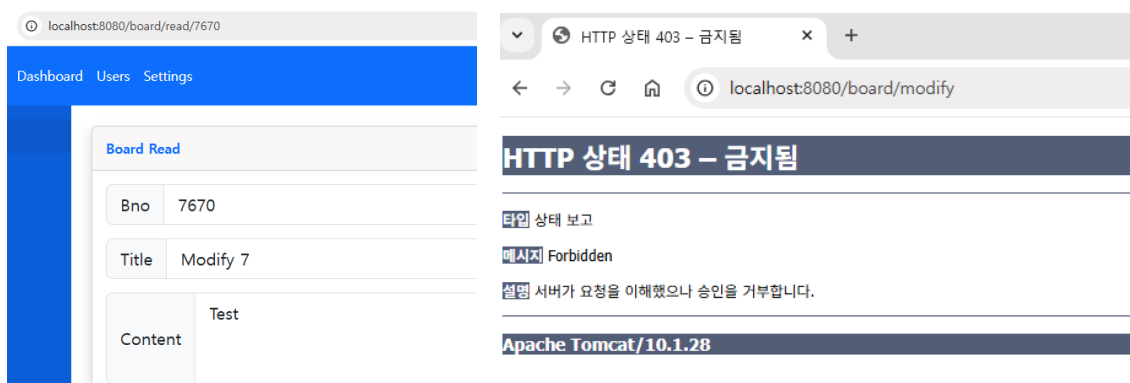
Board List			
<div style="text-align: right;"> -- <span>▼</span> <input type="text"/> <input type="button" value="Search"/> </div>			
Bno	Title	Writer	RegDate
<a href="#">7670</a>	Modify 7 [ 15 ]	Tester	2025-05-10
<a href="#">7669</a>	Test [ 128 ]	Tester	2025-05-10
<a href="#">7668</a>	Test [ 10 ]	Tester	2025-05-10
<a href="#">7667</a>	Test [ 0 ]	Tester	2025-05-10
<a href="#">7666</a>	Test [ 2 ]	Tester	2025-05-10
<a href="#">7665</a>	Update Title [ 11 ]	user00	2025-05-10
<a href="#">7664</a>	title [ 0 ]	user00	2025-05-10

위의 화면에서 7665 번은 user00 이라는 사용자가 작성한 게시물이므로 '/login'을 이용해서 'user00'으로 로그인합니다.

로그인 후에 '/board/modify/7665'를 이용해서 수정/삭제 화면으로 이동해서 수정을 시도하면 정상적으로 수정된 후에 조회 화면으로 이동하게 됩니다.



만일 다른 사용자가 작성한 게시글을 수정하게 되면 403(Forbidden) 에러가 발생하게 됩니다.



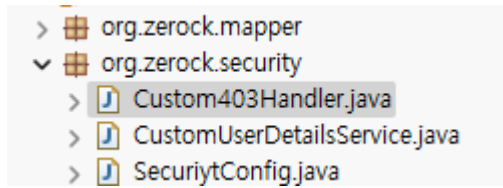
## 403(Forbidden) 처리

403 에러는 흔히 '접근 제한'이라고 부르기도 하는데 말 그대로 현재 사용자가 해당 작업을 수행할 수 있는 권한이 없는 경우에 발생합니다.

'접근 제한'과 '인증되지 않은' 사용자의 개념은 차이가 있으므로 주의해야 합니다. 접근 제한은 '인증된 사용자'이지만 해당 작업을 할 수 있는 권한이 부족한 경우'를 의미합니다. 반면에 '인증이 되지 않은 사용자'는 권한을 따지는 전에 로그인 자체가 안된 사용자이므로 '403'에러가 아니라 '로그인 화면'으로 이동이 일어나게 됩니다.

일반적인 경우 위의 화면과 같이 403 에러 화면이 그냥 보여지는 것이 바람직하지 않기 때문에 403 에러가 발생할 경우 다른 방식의 처리를 위해서 AccessDeniedHandler 라는 인터페이스가 있으므로 이를 이용해서 403 에러를 조정할 수 있습니다.

security 패키지에 Custom403Handler 클래스를 추가합니다.



```
package org.zerock.security;

import java.io.IOException;

import org.springframework.security.access.AccessDeniedException;
import org.springframework.security.web.access.AccessDeniedHandler;

import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.extern.log4j.Log4j2;

@Log4j2
public class Custom403Handler implements AccessDeniedHandler{

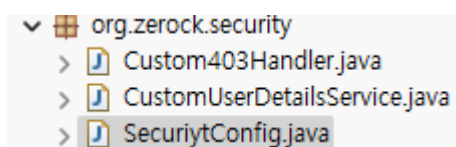
    @Override
    public void handle(HttpServletRequest request, HttpServletResponse response,
        AccessDeniedException accessDeniedException) throws IOException, ServletException {

        log.info("-----AccessDeniedHandler-----");

    }
}
```

AccessDeniedHandler 인터페이스는 handle()이라는 하나의 추상 메서드를 가지고 있는데 여기에 HttpServletRequest 와 HttpServletResponse 를 이용해서 직접 화면을 구성할 수 있습니다.

작성된 Custom403Handler 는 SecurityConfig 를 이용해서 설정합니다.



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

    log.info("-----security config-----");

    http.formLogin(config -> {

    });

    http.csrf(config -> {

        config.disable();

    });

    http.exceptionHandling(handler -> {
```

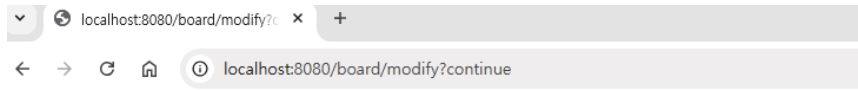
```

        handler.accessDeniedHandler(new Custom403Handler());
    });

    return http.build();
}

```

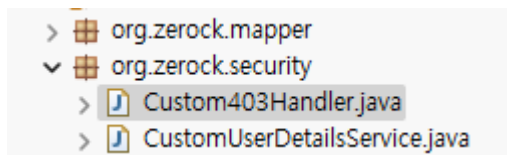
위의 설정이 반영된 후에 다시 프로젝트를 재시작하고 로그인과 게시물의 수정을 시도하면 Custom403Handler 가 동작하는 것을 확인할 수 있고 이전과 달리 빈 화면을 보게 됩니다.



```

.handle(Custom403Handler.java:21) INFO -----AccessDeniedHandler-----
org.springframework.security.web.header.writers.HstsHeaderWriter.writeHeaders(HstsHeaderWriter.java:245) TRACE Trying to
org.springframework.security.web.FilterChainProxy.getFilters(FilterChainProxy.java:245) TRACE Trying to
org.springframework.security.web.FilterChainProxy.doFilterInternal(FilterChainProxy.java:223) DEBUG Security
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:37) DEBUG Security
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:37) DEBUG Security
org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainProxy.java:37) DEBUG Security

```



예제에서는 403 에러가 발생할 경우 '/sample/access-denied' 경로를 호출하도록 수정합니다.

```

package org.zerock.security;

import java.io.IOException;

import org.springframework.security.access.AccessDeniedException;
import org.springframework.security.web.access.AccessDeniedHandler;

import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.extern.log4j.Log4j2;

@Log4j2
public class Custom403Handler implements AccessDeniedHandler{

    @Override
    public void handle(HttpServletRequest request, HttpServletResponse response,
        AccessDeniedException accessDeniedException) throws IOException, ServletException {

        log.info("-----AccessDeniedHandler-----");

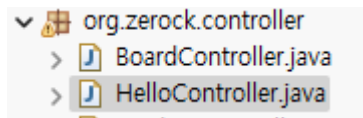
        response.sendRedirect(request.getContextPath() + "/sample/access-denied");
    }
}

```



```
}
}
```

controller 패키지에는 '/sample/access-denied'를 처리하도록 HelloController 에 아래의 코드를 작성합니다.



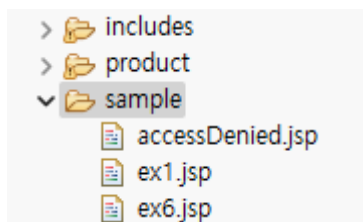
```
@Controller
@RequiredArgsConstructor
@ToString
@Log4j2
@RequestMapping("/sample")
public class HelloController {

    private final HelloService helloService;

    @GetMapping("access-denied")
    public String accessDenied() {

        return "/sample/accessDenied";
    }
}
...생략
```

accessDenied.jsp 파일을 작성합니다.



```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>

<!DOCTYPE html>
<html>
<head>
    <title>403 Forbidden</title>
    <!-- Bootstrap 5 CDN -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>
<body class="bg-light">

<div class="container vh-100 d-flex justify-content-center align-items-center">
    <div class="text-center">
        <h1 class="display-1 text-danger fw-bold">403</h1>
        <h2 class="mb-3">접근이 거부되었습니다</h2>
        <p class="lead mb-4">이 페이지에 접근할 권한이 없습니다.<br>필요한 권한이 있는지 확인하거나 관리자에게 문의하세요.</p>
    </div>
```

```
</div>
```

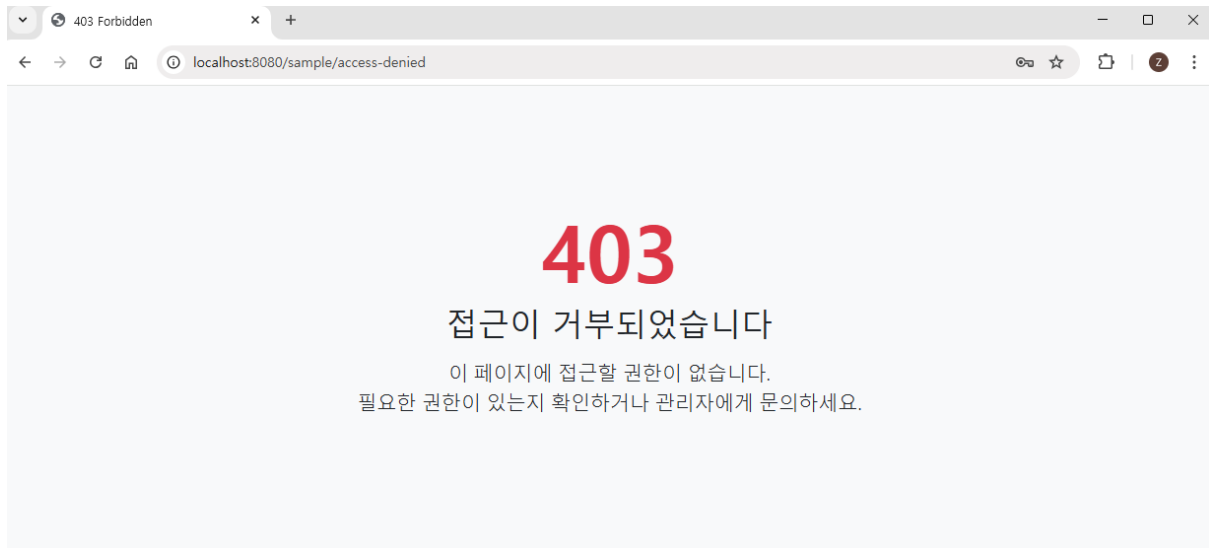
```
<!-- 선택: Bootstrap 아이콘 사용 시 -->
```

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.10.5/font/bootstrap-icons.css" rel="stylesheet">
```

```
</body>
```

```
</html>
```

정상적으로 적용되었다면 403 화면은 아래와 같은 모습으로 출력되는 것을 확인할 수 있습니다.

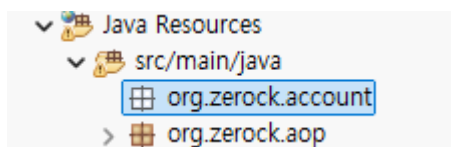


## 10.5 사용자 계정 처리

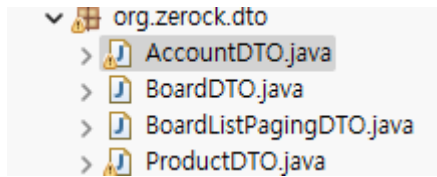
기본적인 시큐리티의 설정과 동작여부를 확인했다면 실제 데이터베이스를 이용해서 사용자의 정보를 가져올 수 있도록 구성할 필요가 있으므로 클래스와 데이터베이스, MyBatis 를 구성하도록 합니다. 다른 예제들과 달리 이번 예제는 데이터베이스가 아닌 클래스를 설계를 우선적으로 처리해 보도록 합니다.

### 클래스 설계와 적용

예제에서 사용자는 Account 라는 용어를 이용해서 사용하고 이를 위한 account 패키지를 추가합니다.



예제에서 사용하는 dto 는 CustomUserDetailsService 에서 반환되어야 하기 때문에 AccountDTO 클래스는 UserDetails 인터페이스를 구현하도록 클래스를 추가합니다.



UserDetails 에는 여러 개의 추상메서드가 존재하기 때문에 아래와 같은 코드가 작성됩니다.

```
package org.zerock.dto;

import java.util.Collection;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import lombok.Data;

@Data
public class AccountDTO implements UserDetails{

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public String getPassword() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public String getUsername() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public boolean isAccountNonExpired() {
        // TODO Auto-generated method stub
        return false;
    }

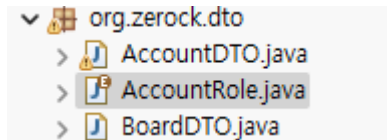
    @Override
    public boolean isAccountNonLocked() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean isEnabled() {
        // TODO Auto-generated method stub
        return false;
    }
}
```

```
}
```

인터페이스의 추상 메소드의 구현은 조금 뒤에 구현하도록 구성하고 계정이 가져야 하는 권한에 대한 부분은 enum 을 이용해서 AccountRole 이라는 이름으로 작성합니다. AccountRole 은 단순히 일반 사용자와 관리자 등을 구분하기 위한 문자열로 예제에서는 USER, MANAGER, ADMIN 으로 구분합니다.



```
package org.zerock.dto;

public enum AccountRole {

    USER, MANAGER, ADMIN;

}
```

AccountDTO 에는 데이터베이스에서 관리된 정보들을 추가하고 UserDetails 인터페이스의 추상 메소드도 구현합니다. 특히 주의해야 하는 부분은 AccountRole 을 스프링 시큐리티에서 인식할 수 있도록 GrantedAuthority 라는 타입으로 바꿔주는 부분입니다. 이 작업은 문자열로 권한을 만들 수 있는 org.springframework.security.core.authority.impl.GrantedAuthority 라는 것을 활용합니다.

```
package org.zerock.dto;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.stream.Collectors;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import lombok.Data;

@Data
public class AccountDTO implements UserDetails{

    private String uid;

    private String upw;

    private String uname;

    private String email;

    private List<AccountRole> roleNames;

    public void addRole(AccountRole role) {

        if(roleNames == null) {
```

```

        roleNames = new ArrayList<>();
    }
    roleNames.add(role);
}

public void clearRoles() {

    roleNames.clear();
}

@Override
public Collection<? extends GrantedAuthority> getAuthorities() {

    if(roleNames == null || roleNames.size() == 0) {
        return List.of();
    }

    return roleNames.stream()
        .map(accountRole -> new SimpleGrantedAuthority("ROLE_" + accountRole.name()))
        .collect(Collectors.toList());

}

@Override
public String getPassword() {
    return upw;
}

@Override
public String getUsername() {
    return uname;
}

@Override
public boolean isAccountNonExpired() {
    //만료되지 않았음
    return true;
}

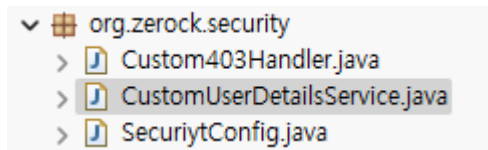
@Override
public boolean isAccountNonLocked() {
    //잠긴 계정 아님
    return true;
}

@Override
public boolean isCredentialsNonExpired() {
    //인증정보 활용 가능함
    return true;
}

@Override
public boolean isEnabled() {
    // 사용 가능
    return true;
}
}

```

작성중인 AccountDTO 가 적절한지 확인하기 위해서 CustomUserDetailsService 를 이용해서 확인해 봅니다.  
로그인을 수행하면 해당 사용자는 USER 와 MANAGER 권한을 가지도록 고정된 코드를 작성합니다.



```
package org.zerock.security;

import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;
import org.zerock.dto.AccountDTO;
import org.zerock.dto.AccountRole;

import lombok.extern.log4j.Log4j2;

@Service
@Log4j2
public class CustomUserDetailsService implements UserDetailsService{

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

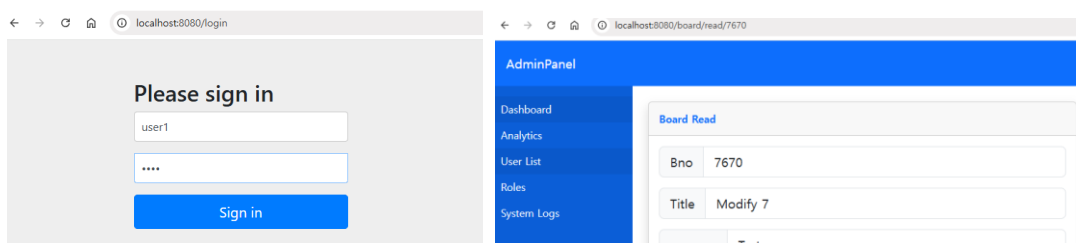
        log.info("-----loadUserByUsername-----" , username);

        AccountDTO accountDTO = new AccountDTO();

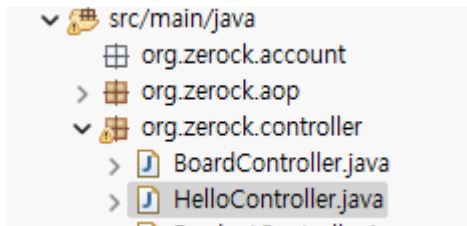
        accountDTO.setUid(username);
        //패스워드는 1111
        accountDTO.setUpw("$2a$10$0NQtffGchYMOQKt2VBEVPOBNezYuk.BfDwOa0.zJPQjzvpa1Sbh7q");
        accountDTO.addRole(AccountRole.USER);
        accountDTO.addRole(AccountRole.MANAGER);

        return accountDTO;
    }
}
```

로그인 후에 AccountDTO 는 'ROLE\_USER, ROLE\_MANAGER' 권한을 가지기 때문에 로그인 후에 게시물의 조회가 가능합니다.



권한 체크가 정상적으로 동작하는지 확인하기 위해서 HelloController 의 ex1()에는 'ROLE\_ADMIN' 권한이 필요하도록 수정하고 로그인했을 때 403(접근 제한)이 발생하는지 확인해 봅니다.



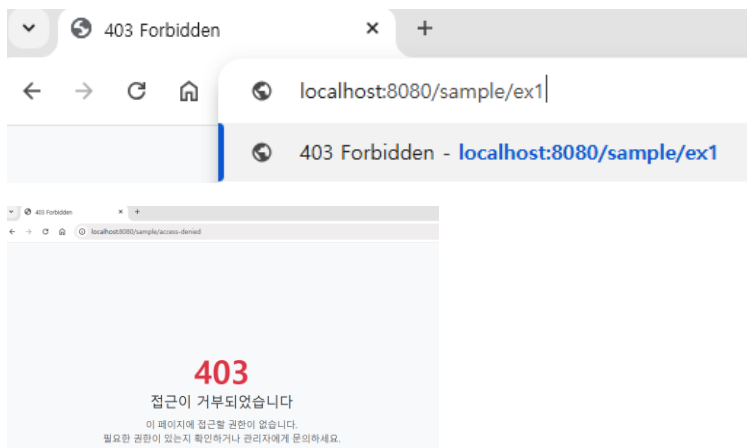
```
@PreAuthorize("hasRole('ADMIN')")
@GetMapping("/ex1")
public void ex1() {

    Log.info("/sample/ex1");

    helloService.hello1();

}
```

현재 사용자는 'ROLE\_ADMIN' 권한은 존재하지 않기 때문에 403 에러가 발생하는 것이 정상입니다.



## 영속 계층 처리

코드를 통해서 적절한 AccountDTO 를 반환하면 정상적으로 동작하는 것을 확인했다면 이제 데이터베이스 내 테이블을 생성하고 이를 처리하는 Mapper 를 작성하도록 합니다.

테이블은 tbl\_account 테이블과 tbl\_account\_roles 로 구분해서 작성합니다. tbl\_account\_roles 테이블의 경우 단독으로 조회할 경우는 없고 uid 를 이용해서만 조회되기 때문에 별도의 PK 를 설정하지 않을 수 있습니다.

```
CREATE TABLE tbl_account (
    uid VARCHAR(50) PRIMARY KEY,
    upw VARCHAR(100) NOT NULL,
    uname VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    enabled BOOLEAN DEFAULT TRUE,
    createdAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

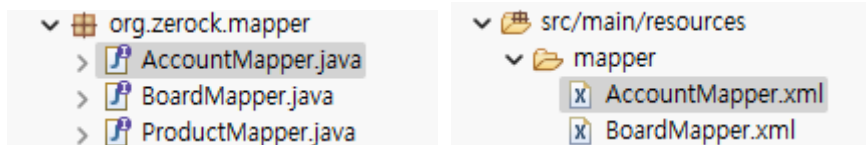
```

        updatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
    );

    CREATE TABLE tbl_account_roles (
        uid VARCHAR(50) NOT NULL,
        rolename VARCHAR(50) NOT NULL,
        FOREIGN KEY (uid) REFERENCES tbl_account(uid) -- 동일 계정에 동일한 역할 중복 방지
    );

```

mapper 패키지에는 AccountMapper 인터페이스를 추가하고, XML 매퍼 파일도 준비합니다.



## 계정 등록

새로운 계정의 등록은 tbl\_account 테이블의 insert와 한 번에 여러 개를 insert 해야 하는 tbl\_account\_roles를 처리하기 위해서 두 개의 메서드로 분리합니다.

```

package org.zerock.mapper;

import org.zerock.dto.AccountDTO;

public interface AccountMapper {

    int insert(AccountDTO accountDTO);

    int insertRoles(AccountDTO accountDTO);
}

```

AccountMapper.xml에서의 구현은 <foreach>를 이용해서 반복 처리가 가능하도록 작성합니다.

```

<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "https://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="org.zerock.mapper.AccountMapper">

    <insert id="insert">

        INSERT INTO tbl_account (uid, upw, uname, email) VALUES ( #{uid}, #{upw}, #{uname}, #{email} )

    </insert>

    <insert id="insertRoles">

        INSERT INTO tbl_account_roles (uid, rolename)
        VALUES

        <foreach collection="roleNames" item="role" separator="," >
            (#{uid}, #{role})
        </foreach>

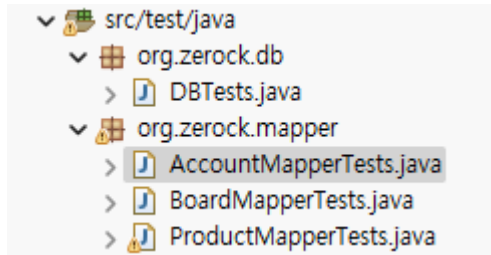
    </insert>

```



</mapper>

작성된 insert()와 insertRoles()는 앞에서 만들어둔 AccountMapperTests 내에 테스트 코드를 이용해서 실제 데이터베이스에 추가합니다. 이 과정에서 패스워드는 반드시 PasswordEncoder를 이용해서 encode()한 결과를 넣어주어야 합니다.



```
package org.zerock.mapper;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.test.annotation.Commit;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.transaction.annotation.Transactional;
import org.zerock.dto.AccountDTO;
import org.zerock.dto.AccountRole;

import lombok.extern.log4j.Log4j2;

@ExtendWith(SpringExtension.class)
@ContextConfiguration("file:src/main/webapp/WEB-INF/spring/root-context.xml")
@Log4j2
public class AccountMapperTests {

    @Autowired
    private PasswordEncoder encoder;

    @Autowired
    private AccountMapper accountMapper;

    @Test
    @Transactional
    @Commit
    public void testInsert() {

        for(int i = 1; i <= 100; i++) {
            AccountDTO accountDTO = new AccountDTO();

            accountDTO.setUid("user" + i);
            accountDTO.setUpw(encoder.encode("1111"));
            accountDTO.setUname("User" + i);

            accountDTO.setEmail("user"+i+"@aaa.com");

            accountDTO.addRole(AccountRole.USER);

            if(i >= 80) {
                accountDTO.addRole(AccountRole.MANAGER);
            }
        }
    }
}
```

```

        if(i >= 90) {
            accountDTO.addRole(AccountRole.ADMIN);
        }

        accountMapper.insert(accountDTO);
        accountMapper.insertRoles(accountDTO);

    } //end for
}

@Test
public void testEncoding() {
    ...
}

```

테스트 코드는 'user1' 부터 'user100'까지 총 100 개의 계정을 생성합니다. 100 개의 계정 중에 'user79'까지는 일반 사용자이고 'user90' 부터 'user100'까지는 모든 권한을 가지도록 생성됩니다. 예를 들어 'user99'의 경우 tbl\_account\_roles 테이블에 3 개의 데이터가 추가됩니다.

```
SELECT * FROM tbl_account_roles ORDER BY uid DESC;
```

	uid	rolename
1	user99	ADMIN
2	user99	MANAGER
3	user99	USER

## 계정 조회

계정 조회는 tbl\_account 테이블과 tbl\_account\_roles 테이블을 조인처리해서 한 번에 권한 데이터까지 처리하도록 구성합니다.

데이터베이스에서 특정 계정의 정보와 권한들을 조회하는 쿼리를 작성하면 다음과 같습니다.

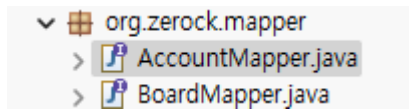
```

SELECT
    ac.uid, ac.upw, ac.uname, ac.email, ar.rolename
FROM
    tbl_account ac INNER JOIN tbl_account_roles ar ON ar.uid = ac.uid
WHERE ac.uid = 'user100'
;

```

	uid	upw	uname	email	rolename
1	user100	\$2a\$10\$FeT.xMPUH6hOYbOvBFnZbO...	User100	user100@aaa.com	USER
2	user100	\$2a\$10\$FeT.xMPUH6hOYbOvBFnZbO...	User100	user100@aaa.com	MANAGER
3	user100	\$2a\$10\$FeT.xMPUH6hOYbOvBFnZbO...	User100	user100@aaa.com	ADMIN

상품과 상품 이미지들과 유사하게 <resultMap>을 이용해서 처리합니다. 우선 AccountMapper 인터페이스에 selectOne( )을 선언합니다.



```
package org.zerock.mapper;

import org.zerock.dto.AccountDTO;

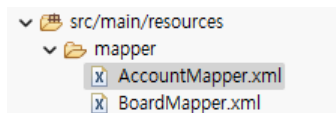
public interface AccountMapper {

    ...생략

    AccountDTO selectOne( @Param("uid") String uid);

}
```

AccountMapper.xml 에서는 쿼리 결과로 발생하는 rolename 이라는 컬럼을 enum 타입인 AccountRole 타입으로 변환하는 부분을 org.apache.ibatis.type.EnumTypeHandler 를 이용해서 처리합니다.



```
<resultMap type="AccountDTO" id="selectMap">

    <id property="uid" column="uid"/>
    <result property="upw" column="upw"/>
    <result property="uname" column="uname"/>
    <result property="email" column="email"/>

    <collection property="roleNames" ofType="AccountRole">

        <result column="rolename" typeHandler="org.apache.ibatis.type.EnumTypeHandler"
        javaType="AccountRole"/>

    </collection>

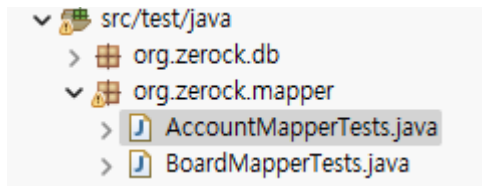
</resultMap>

<select id="selectOne" resultMap="selectMap">

SELECT
    ac.uid, ac.upw, ac.uname, ac.email, ar.rolename
FROM
    tbl_account ac INNER JOIN tbl_account_roles ar ON ar.uid = ac.uid
WHERE ac.uid = #{uid}

</select>
```

작성한 selectOne( )에 대하여 테스트 코드를 작성해서 확인합니다.



```
@Test
public void testSelectOne() {

    String uid = "user100";

    AccountDTO accountDTO = accountMapper.selectOne(uid);

    Log.info(accountDTO);

    Log.info(accountDTO.getRoleNames());

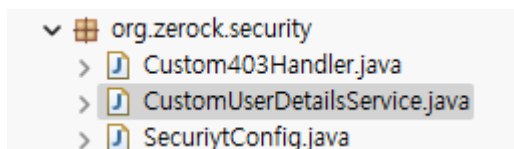
}
```

테스트 코드의 실행 결과를 보면 AccountDTO 내의 roleNames 의 처리가 정상적으로 된 것을 확인할 수 있습니다.

```
DEBUG ==> Preparing: SELECT ac.uid, ac.upw, ac.uname, ac.email, ar.rolename FROM tbl_account ac INNER JC
DEBUG ==> Parameters: user100(String)
DEBUG <== Total: 3
INFO AccountDTO(uid=user100, upw=$2a$10$FeT.xMPUH6hOYbOvBFnZbO1sO74V7InM1KocNFEkem3/dI9m1YqoC, uname=User10
INFO [USER, MANAGER, ADMIN]
```

## CustomUserDetailsService 수정

기존의 CustomUserDetailsService 는 작성된 코드를 이용해서 로그인이 처리되었기 때문에 AccountMapper 의 selectOne( )기능을 CustomUserDetailsService 에서 활용하도록 수정해 주어야 합니다.



기존 코드에 의존성 주입을 이용해서 AccountMapper 를 추가하고 @RequiredArgsConstructor 를 추가합니다.

```
package org.zerock.security;

import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;
import org.zerock.dto.AccountDTO;
import org.zerock.dto.AccountRole;
import org.zerock.mapper.AccountMapper;

import lombok.RequiredArgsConstructor;
import lombok.extern.log4j.Log4j2;
```

```

@Service
@Log4j2
@RequiredArgsConstructor
public class CustomUserDetailsService implements UserDetailsService{

    private final AccountMapper accountMapper;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        log.info("-----loadUserByUsername-----" , username);

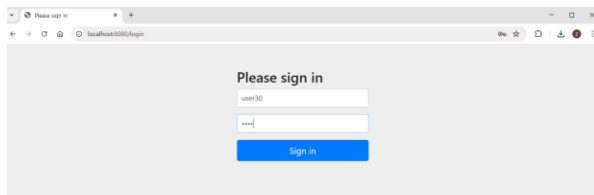
        AccountDTO accountDTO = accountMapper.selectOne(username);

        if(accountDTO == null) {
            throw new UsernameNotFoundException("Account Not Found");
        }

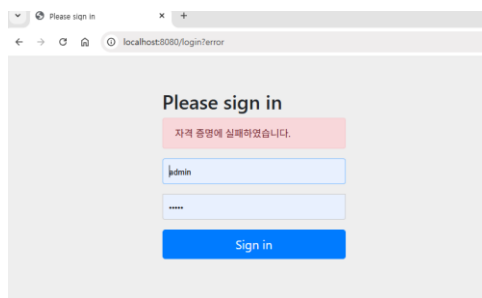
        return accountDTO;
    }
}

```

변경된 코드를 이용해서 프로젝트를 재시작하고 데이터베이스에 존재하는 계정을 이용해서 로그인을 시도해 봅니다.



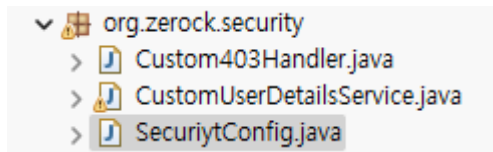
존재하는 계정의 경우 로그인 후에 '/' 경로로 이동하는 것을 확인할 수 있지만 계정이 존재하지 않거나 비밀번호가 틀린 경우에는 로그인 화면에서 에러가 발생합니다.



## 10.6 커스텀 로그인/로그아웃

로그인 기능이 데이터베이스와 연동된 것을 확인했다면 스프링 시큐리티가 생성하는 로그인 화면이 아닌 개발자가 직접 만든 화면을 이용하도록 수정해 봅니다.

로그인 관련 설정은 SecurityConfig 에서 작성하는 메서드 내에서 formLogin( )에 대한 설정으로 추가할 수 있습니다.



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

    log.info("-----security config-----");

    http.formLogin(config -> {

    });
}
```

...생략

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

formLogin( ) 설정은 전달되는 파라미터의 이름 등의 설정 외에도 로그인 페이지의 경로를 지정할 수 있습니다. 예제에서는 로그인 화면의 경로를 '/account/login'으로 지정합니다.

```
http.formLogin(config -> {

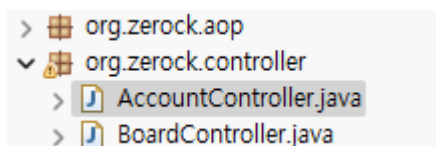
    config.loginPage("/account/login");

});
```

이 설정을 반영하면 더 이상 기존에 사용했던 '/login'은 사용할 수 없게 됩니다.



controller 패키지에 AccountController 를 추가하고 '/account/login'을 GET 방식으로 처리합니다.



```

package org.zerock.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;

import lombok.extern.log4j.Log4j2;

@Controller
@Log4j2
@RequestMapping("/account")
public class AccountController {

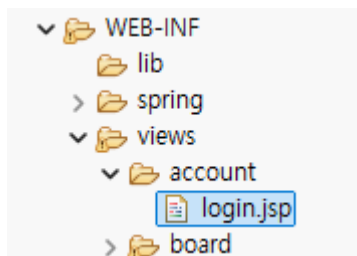
    @GetMapping("login")
    public void loginGET() {

    }

}

```

WEB-INF/views 에는 account 폴더와 login.jsp 를 작성합니다.



login.jsp 에서 스프링 시큐리티가 사용하는 username, password 를 name 속성값으로 이용하는 화면을 구성합니다.

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>로그인</title>
    <style>
        body {
            font-family: sans-serif;
            display: flex;
            justify-content: center;
            align-items: center;
            min-height: 100vh;
            background-color: #f4f4f4;
        }

        .login-container {
            background-color: #fff;
            padding: 30px;
            border-radius: 8px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
            width: 300px;
            text-align: center;
        }

        h2 {
            margin-bottom: 20px;
            color: #333;
        }
    </style>
</head>
<body>
    <h2>로그인</h2>
    <div class="login-container">
        <div>
            <input type="text" value="아이디" />
            <input type="password" value="비밀번호" />
            <input type="button" value="로그인" />
        </div>
    </div>
</body>
</html>

```

```

.form-group {
  margin-bottom: 15px;
  text-align: left;
}

label {
  display: block;
  margin-bottom: 5px;
  color: #555;
  font-size: 0.9em;
}

input[type="text"],
input[type="password"] {
  width: calc(100% - 12px);
  padding: 10px;
  border: 1px solid #ddd;
  border-radius: 4px;
  box-sizing: border-box;
  font-size: 1em;
}

button[type="submit"] {
  background-color: #007bff;
  color: white;
  padding: 10px 15px;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  font-size: 1em;
  width: 100%;
}

button[type="submit"]:hover {
  background-color: #0056b3;
}
</style>
</head>
<body>
  <div class="login-container">
    <h2>로그인</h2>
    <form method="post">
      <div class="form-group">
        <label for="username">사용자 이름:</label>
        <input type="text" id="username" name="username" required>
      </div>
      <div class="form-group">
        <label for="password">비밀번호:</label>
        <input type="password" id="password" name="password" required>
      </div>
      <button type="submit">로그인</button>
    </form>
  </div>
</body>
</html>

```

login.jsp 코드에서 특이한 점은 <form>태그의 action 을 지정하지 않은 점입니다. <form>태그에서 action 속성값을 지정하지 않으면 현재 브라우저의 경로가 action 값이 되므로 엄밀하게 말하면 action 속성값은 '/account/login'이 됩니다.



브라우저를 통해서 확인해 보면 POST 방식의 처리 없이도 정상적으로 로그인이 가능한 것을 확인할 수 있습니다.



## 로그인 성공 후 처리

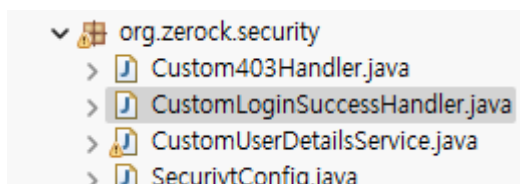
스프링 시큐리티를 이용할 때 로그인 페이지를 보게 되는 경우는 크게 다음과 같이 2 가지 경우가 있습니다.

- 직접 로그인 경로를 호출하는 경우
- 특정한 경로를 호출했는데 권한이 필요해서 로그인 화면으로 리다이렉트 되는 경우

직접 로그인 경로를 호출하는 경우에는 로그인이 성공한 후에 '/' 경로로 이동하게 되고(위의 화면) 특정 경로를 호출했을 경우에는 로그인 성공 후에는 원래 접근하려고 했던 경로로 이동하게 됩니다(예를 들어 '/board/read/33'과 같은 경로에서 로그인 페이지로 이동했다면 로그인 후에 '/board/read/33'으로 이동하게 됩니다.).

스프링 시큐리티에서는 AuthenticationSuccessHandler 라는 인터페이스를 구현해서 로그인 성공 후에 이를 제어할 수 있습니다. 예제에서는 로그인에 성공하면 '/board/list'로 가도록 구현해 봅니다.

security 패키지에 CustomLoginSuccessHandler 클래스를 추가합니다.



```
package org.zerock.security;

import java.io.IOException;

import org.springframework.security.core.Authentication;
import org.springframework.security.web.authentication.AuthenticationSuccessHandler;
import org.springframework.security.web.savedrequest.HttpSessionRequestCache;
import org.springframework.security.web.savedrequest.SavedRequest;
```

```

import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.extern.log4j.Log4j2;

@Log4j2
public class CustomLoginSuccessHandler implements AuthenticationSuccessHandler {

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response,
        Authentication authentication) throws IOException, ServletException {

        Log.info("authentication: " + authentication);

        // 세션에서 SavedRequest 확인
        SavedRequest savedRequest = new HttpSessionRequestCache().getRequest(request, response);

        Log.info(savedRequest);

        response.sendRedirect("/board/list");
    }
}

```

코드에서는 로그인 후에 정상적으로 '/board/list'로 이동하도록 작성되어 있습니다. /account/login 을 통해서 로그인을 수행하면 아래와 같은 로그를 확인할 수 있습니다.

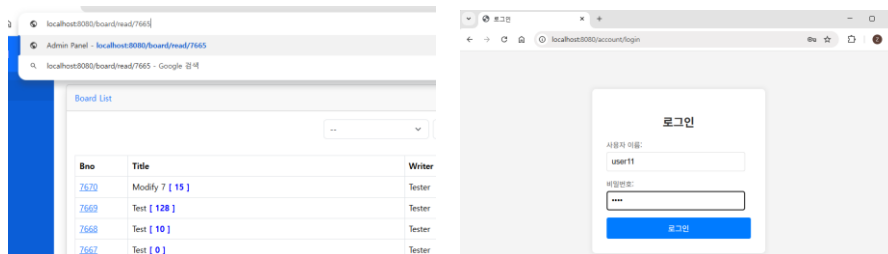
```

(CustomLoginSuccessHandler.java:25) INFO authentication: UsernamePasswordAuthenticationToken
(CustomLoginSuccessHandler.java:30) INFO null

```

## 원래 경로로 리다이렉트

코드내에 SavedRequest 는 로그인이 필요해서 로그인 화면으로 이동한 경우 null 이 아닌 상태가 됩니다. 예를 들어 '/board/read/7665' 경로는 로그인한 사용자만이 볼 수 있는 화면이고 로그인이 되지 않은 상태에서는 로그인 화면으로 리다이렉트 처리되는데 이 상태에서 로그인을 수행하면 위의 화면과 다른 로그를 보게 됩니다.



```

(CustomLoginSuccessHandler.java:25) INFO authentication: UsernamePasswordAuthenticationToken [Principal=AccountDTO
(CustomLoginSuccessHandler.java:30) INFO DefaultSavedRequest [http://localhost:8080/board/read/7665?continue]

```

SavedRequest 는 이처럼 기존 경로를 보관하고 있기 때문에 이를 이용해서 로그인 성공 후 다시 이동할 수 있습니다. 이를 위해서 코드를 아래와 같이 수정합니다.

```
@Override
public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response,
    Authentication authentication) throws IOException, ServletException {

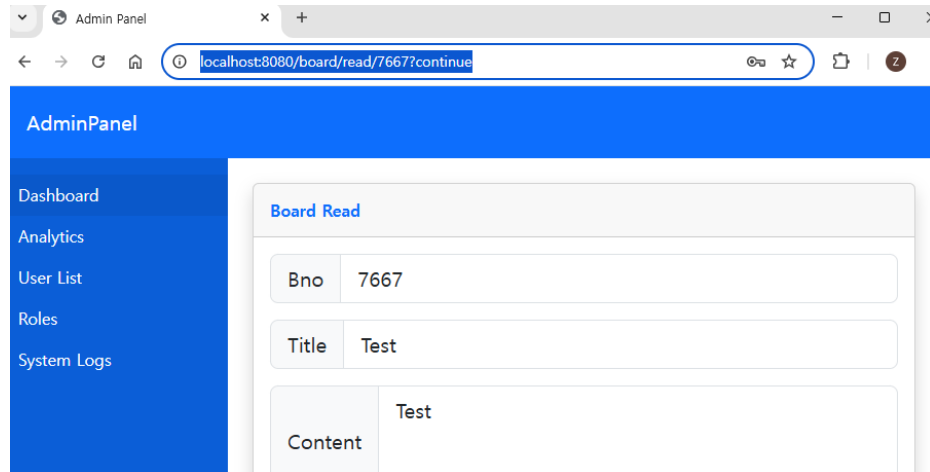
    Log.info("authentication: " + authentication);

    // 세션에서 SavedRequest 확인
    SavedRequest savedRequest = new HttpSessionRequestCache().getRequest(request, response);

    Log.info(savedRequest);

    if(savedRequest != null) {
        response.sendRedirect(savedRequest.getRedirectUrl());
    }else {
        response.sendRedirect("/board/list");
    }
}
```

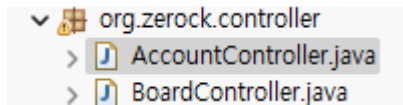
위의 코드가 적용된 후에는 로그인 후에 기존에 로그인에 필요했던 경로로 이동하는 것을 확인할 수 있습니다(이 경우 추가적으로 continue 라는 쿼리스트링이 추가됩니다.).



## 로그아웃 처리

스프링 시큐리티의 로그아웃은 '/logout'이라는 경로를 호출하는 것으로 이루어집니다. 다만 주의해야 할 점은 로그아웃은 기본적으로 POST 방식으로 동작한다는 점입니다. 스프링 시큐리티에서 로그아웃은 CSRF 토큰을 이용하는데 만일 예제와 같이 CSRF 토큰을 사용하지 않는 경우라면 '/logout'역시 GET 방식으로 동작할 수 있습니다.

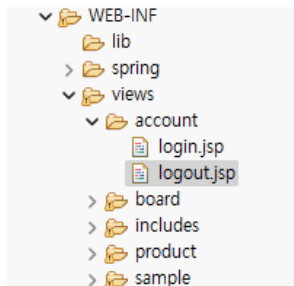
예제에서는 CSRF 토큰을 사용하지 않기 때문에 GET 방식으로 호출이 가능하지만 사용자 경험을 고려해서 '/account/logout'이라는 경로를 이용해서 로그아웃 확인 화면을 작성합니다.



```
@GetMapping("logout")
public void logoutGET() {

}
```

JSP 페이지에서는 로그아웃 버튼을 작성해서 사용자가 '/logout'을 호출할 수 있도록 조정합니다.



```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>로그아웃</title>
    <style>
        body {
            font-family: sans-serif;
            display: flex;
            justify-content: center;
            align-items: center;
            min-height: 100vh;
            background-color: #f4f4f4;
        }

        .logout-container {
            background-color: #fff;
            padding: 30px;
            border-radius: 8px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
            width: 300px;
            text-align: center;
        }

        h2 {
            margin-bottom: 20px;
            color: #d9534f; /* 경고 색상 계열 */
        }

        p {
            color: #555;
            margin-bottom: 20px;
        }
    </style>
</head>
<body>
    <div class="logout-container">
        <h2>로그아웃</h2>
        <p>로그아웃 버튼을 클릭하면 로그아웃됩니다.</p>
    </div>
</body>
</html>
```

```

}

.button-group {
  display: flex;
  gap: 10px;
}

.logout-button,
.cancel-button {
  flex: 1;
  padding: 10px 15px;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  font-size: 1em;
}

.logout-button {
  background-color: #d9534f; /* 경고 색상 */
  color: white;
}

.logout-button:hover {
  background-color: #c9302c;
}

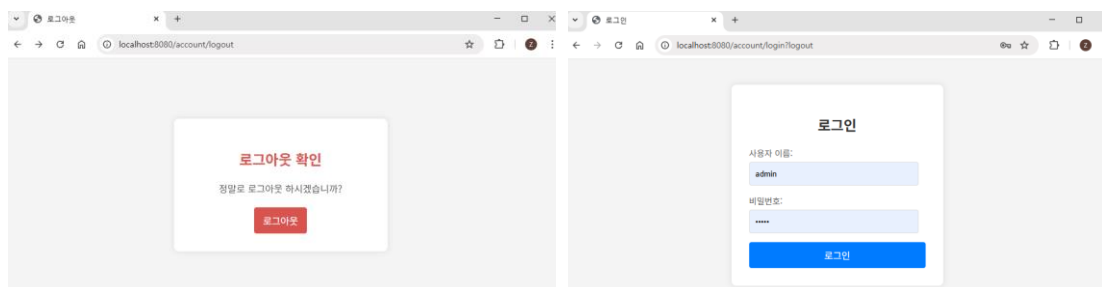
.cancel-button {
  background-color: #f0f0f0;
  color: #333;
  border: 1px solid #ccc;
}

.cancel-button:hover {
  background-color: #e0e0e0;
}
</style>
</head>
<body>
  <div class="logout-container">
    <h2>로그아웃 확인</h2>
    <p>정말로 로그아웃 하시겠습니까?</p>

    <form action="/logout" method="post">
      <button class="logout-button">로그아웃</button>
    </form>
  </div>
</body>
</html>

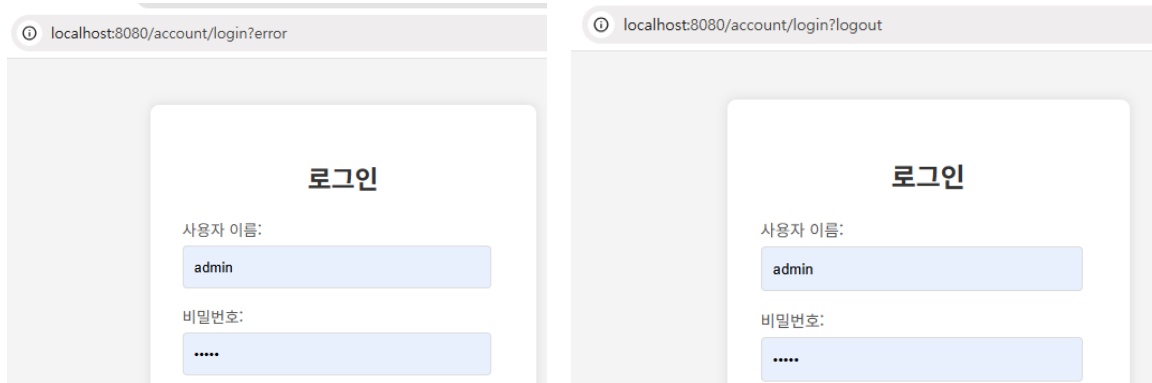
```

브라우저를 통해서 '/account/logout'을 확인해 봅니다. 로그아웃이 실행된 후에는 '/account/login?logout'으로 이동하는 것을 확인할 수 있습니다.



## 로그인에러와 로그아웃 메시지

로그인 과정에서 발생하는 에러와 로그아웃 메시지는 로그인 경로에 '?error' 와 같이 쿼리스트링으로 전달됩니다.



이를 컨트롤러 혹은 JSP 에서 처리해 주도록 합니다. login.jsp 를 아래와 같이 수정합니다.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>

<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>로그인</title>
    <style>
        body {
            font-family: sans-serif;
            display: flex;
            justify-content: center;
            align-items: center;
            min-height: 100vh;
            background-color: #f4f4f4;
        }

        .login-container {
            background-color: #fff;
            padding: 30px;
            border-radius: 8px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
            width: 300px;
            text-align: center;
        }

        h2 {
            margin-bottom: 20px;
            color: #333;
        }

        .form-group {
            margin-bottom: 15px;
            text-align: left;
        }

        label {
            display: block;
```

```

        margin-bottom: 5px;
        color: #555;
        font-size: 0.9em;
    }

    input[type="text"],
    input[type="password"] {
        width: calc(100% - 12px);
        padding: 10px;
        border: 1px solid #ddd;
        border-radius: 4px;
        box-sizing: border-box;
        font-size: 1em;
    }

    button[type="submit"] {
        background-color: #007bff;
        color: white;
        padding: 10px 15px;
        border: none;
        border-radius: 4px;
        cursor: pointer;
        font-size: 1em;
        width: 100%;
    }

    button[type="submit"]:hover {
        background-color: #0056b3;
    }

    .message {
        margin-bottom: 15px;
        padding: 10px;
        border-radius: 4px;
        font-size: 0.9em;
    }

    .error {
        background-color: #fdecea;
        color: #d9534f;
        border: 1px solid #e74c3c;
    }

    .logout-success {
        background-color: #d4edda;
        color: #155724;
        border: 1px solid #c3e6cb;
    }
}
</style>
</head>
<body>
    <div class="login-container">
        <h2>로그인</h2>

        <c:if test="${param.error != null}">
            <div class="message error">
                로그인 실패: 사용자 이름 또는 비밀번호를 확인하세요.
            </div>
        </c:if>

        <c:if test="${param.logout != null}">
            <div class="message logout-success">
                성공적으로 로그아웃되었습니다.
            </div>
        </c:if>

        <form method="post">

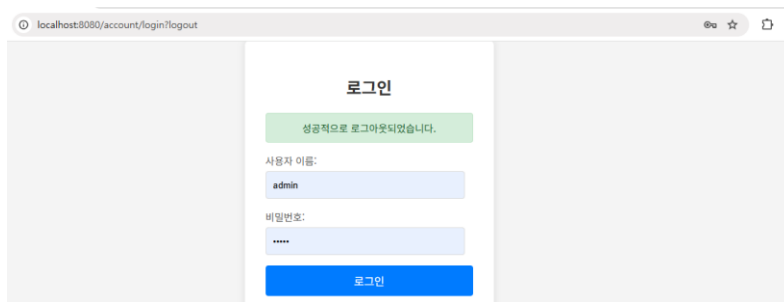
```

```

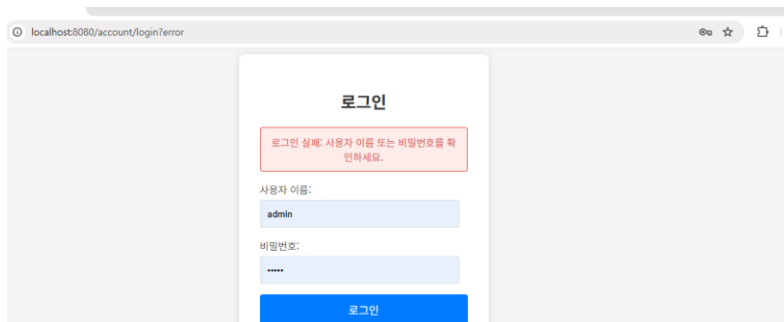
<div class="form-group">
  <label for="username">사용자 이름:</label>
  <input type="text" id="username" name="username" required>
</div>
<div class="form-group">
  <label for="password">비밀번호:</label>
  <input type="password" id="password" name="password" required>
</div>
<button type="submit">로그인</button>
</form>
</div>
</body>
</html>

```

수정된 login.jsp 는 로그아웃시에는 아래와 같은 모습으로 출력됩니다.



로그인의 실패시에는 다른 메시지가 출력됩니다.



## 10.7 Remember-me 자동로그인

스프링 웹 시큐리티가 제공하는 또다른 편리한 기능 중에 하나는 'Remember-me'라고 불리는 자동 로그인기능입니다. 'Remember-me'는 흔히 '자동로그인'이라고 하기도 하는데 예제에서는 '자동로그인'이라는 용어를 사용하도록 하겠습니다.

추가적인 설정이 없다면 스프링 시큐리티의 자동로그인은 쿠키를 사용합니다. 쿠키에 만료시간을 지정하면 지정된 기한까지의 서버 호출시에는 해당 쿠키가 같이 전송된다는 점을 이용해서 해당 사용자가 이전에도



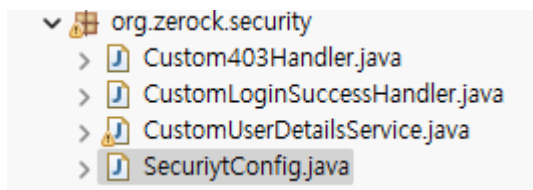
방문한 적이 있다는 사실을 기억하거나 사용자가 계속 유지해야 하는 정보를 쿠키를 통해서 유지하는 방식입니다.

쿠키는 동작 원리 상 매번 브라우저와 서버 통신 과정에서 노출될 수 있기 때문에 과거에는 안전상의 문제등으로 꺼려하는 경우가 많았습니다만 모바일 기기의 특징상 매번 로그인을 하는 입력 과정이 너무 불편하기 때문에 쿠키를 이용한 자동 로그인이 더 많이 사용되게 되었습니다.

## 자동로그인 설정

스프링 시큐리티의 자동로그인은 크게 화면에서 자동로그인 여부를 파라미터로 전달하는 것과 이에 대한 간단한 설정으로 구현이 가능합니다.

우선 `SecuriytConfig` 에 자동로그인 관련 설정을 추가합니다. 자동로그인의 설정은 `rememberme()`를 통해서 설정하는데 이 때 로그인 화면의 변화를 살펴보기 위해서 잠시 로그인 부분은 주석처리해 줍니다.



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

    Log.info("-----security config-----");

    http.formLogin(config -> {
//        config.loginPage("/account/login");
//        config.successHandler(new CustomLoginSuccessHandler());
    });

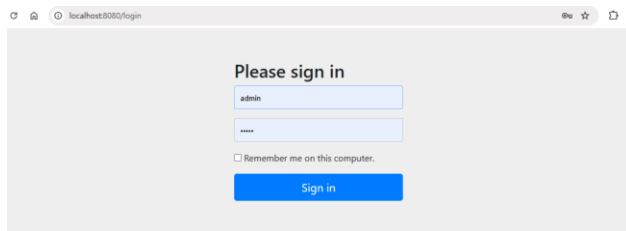
    http.rememberMe(config -> {
    });

    http.csrf(config -> {
        config.disable();
    });

    http.exceptionHandling(handler -> {
        handler.accessDeniedHandler(new Custom403Handler());
    });

    return http.build();
}
```

변경된 설정을 적용하고 프로젝트를 재시작한 후에 기존의 '/login' 경로를 호출하면 화면 아래쪽에 체크박스가 하나 생성되는 것을 확인할 수 있습니다.

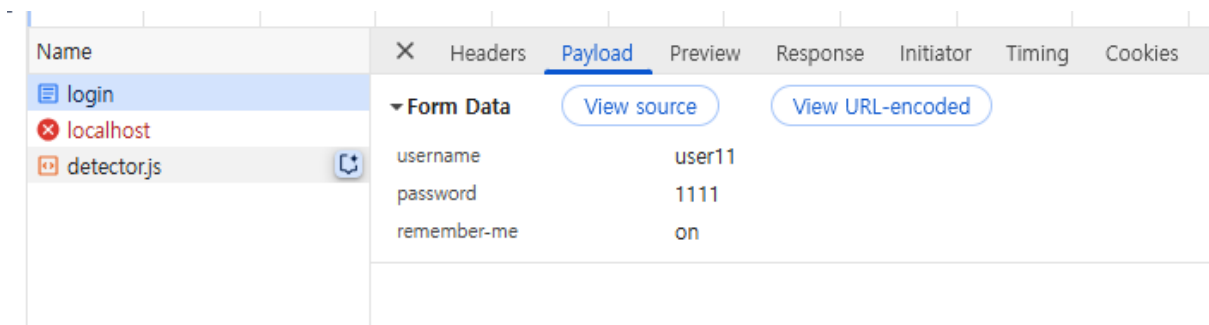


체크박스를 체크한 후에 정상적으로 로그인을 시도하면 브라우저내에 'remember-me' 이름의 쿠키가 생성되는 것을 확인할 수 있습니다.



화면에서는 name 속성값이 'remember-me'라는 이름으로 <input>태그가 하나 추가되는 것을 확인할 수 있습니다.

자동로그인을 선택한 후에 전송되면 POST 방식으로 전송되는 값에 'remember-me'와 'on' 값이 전송되는 것을 확인할 수 있습니다.

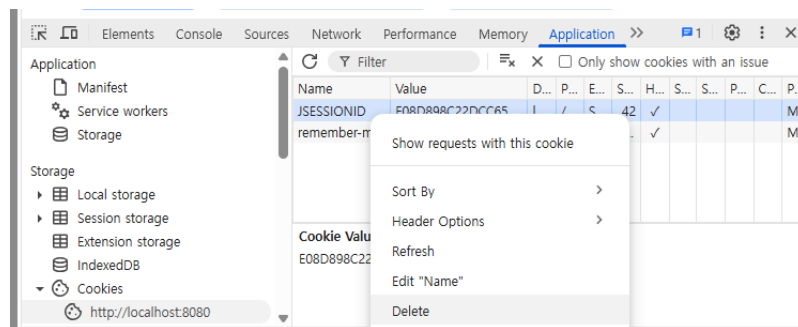


로그인이 성공했다면 브라우저로 'Set-Cookie' 헤더가 전송되면서 'remember-me'라는 이름의 쿠키가 2 주 동안 유지되도록 전송되는 것을 확인할 수 있습니다. 쿠키의 값은 암호화된 문자열입니다.

Location	/
Pragma	no-cache
Set-Cookie	JSESSIONID=031AA29788F28E25D836F9A86F64415F; Path=/; HttpOnly
Set-Cookie	remember-me=VXNlqjExOjE3NDk0NDY5NDAsODk6U0hBMjU2OjZlYjExYzZmYTgkZTdmY2M2NGFIMzNlNGVhYmJ3YWEyZDI4NDNlMGVlY2VlMzA; Max-Age=1209600; Expires=Mon, 09 Jun 2025 05:29:00 GMT; Path=/; HttpOnly
X-Content-Type-Options	nosniff
X-Frame-Options	DENY
X-Xss-Protection	0

Application								
Filter								
Name	Value	Domain	Path	Expires...	Size	HttpO...	Sec	
JSESSIONID	DAA678D134CF6E980A1B075D899B74BA	localh...	/	Session	42	✓		
remember-me	VXNlclExOjE3NDk0NDY0MTExMjY6U0hBMjU2...	localh...	/	2025-...	134	✓		

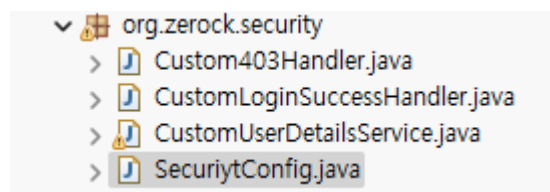
쿠키가 생성된 것을 확인한 후에 해당 브라우저를 종료하고 다시 실행한 후에 '/board/read/7665'와 같이 로그인에 필요한 경로를 호출하거나 'JSESSIONID'를 삭제한 후에 호출했을 때에도 로그인 처리가 된 것과 동일한 결과가 생성되는지 확인합니다.



## 로그인 화면 조정

스프링 시큐리티가 기본으로 제공하는 로그인 화면이 아닌 커스터마이징된 로그인 화면을 이용한다면 `<input name='remember-me'>` 인 태그를 추가해야만 합니다.

우선 `SecuriytConfig` 의 `formLogin()` 을 커스터마이징된 상태로 복원합니다.



```
http.formLogin(config -> {
    config.loginPage("/account/login");
    config.successHandler(new CustomLoginSuccessHandler());
});
```

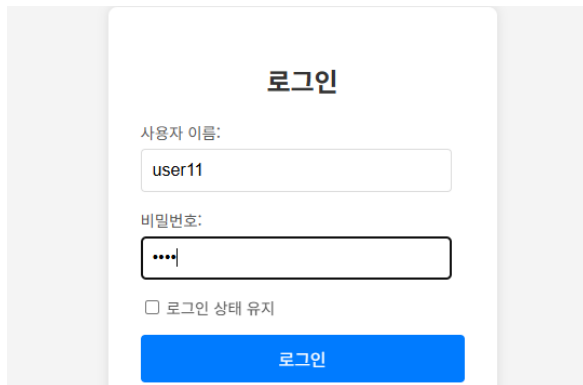
WEB-INF/views/account/login.jsp 를 수정합니다.

```

<form method="post">
  <div class="form-group">
    <label for="username">사용자 이름:</label>
    <input type="text" id="username" name="username" required>
  </div>
  <div class="form-group">
    <label for="password">비밀번호:</label>
    <input type="password" id="password" name="password" required>
  </div>
  <div class="form-group">
    <label>
      <input type="checkbox" name="remember-me">
      로그인 상태 유지
    </label>
  </div>
  <button type="submit">로그인</button>
</form>

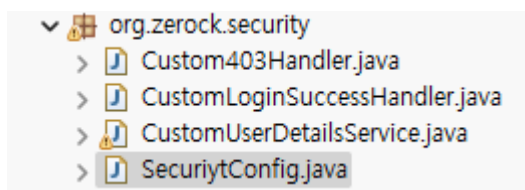
```

로그인 화면에 '로그인 상태 유지' 체크 박스가 보이는지 확인하고 로그인을 시도해 봅니다.



## 자동로그인 쿠키값 조정

자동로그인을 통해서 만들어지는 쿠키는 2 주일동안 유지됩니다. 개발자는 쿠키의 만료 기한과 암호화할 때의 키(key)값 등을 조정할 수 있습니다(별도의 설정이 없다면 키(key)값은 'SpringSecured'라는 값을 이용합니다.).



만일 30 일 동안 유지되어야 하는 쿠키로 만들고 싶다면 아래와 같이 tokenValiditySeconds( )를 통해서 설정을 조정할 수 있고 암호화의 키(key)값 역시 조정이 가능합니다.

```

http.rememberMe(config -> {
    //config.key("my key");
    config.tokenValiditySeconds(60*60*24*30);
});

```

## 데이터베이스를 이용하는 쿠키 보관

자동로그인 쿠키는 문자열이기 때문에 해당 쿠키가 탈취당할 경우 이에 대한 대비가 필요합니다. 스프링 시큐리티는 데이터베이스를 이용해서 만들어진 자동로그인 쿠키값을 저장할 수 있는데 이를 통해서 사용자가 언제 로그인 했는지, 마지막에 언제 사용되었는지 등에 대한 정보를 기록할 수 있습니다.

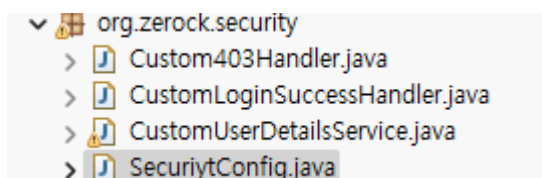
예를 들어 사용자의 자동로그인 쿠키가 공격자에 의해서 탈취된 상황에서 사용자가 다시 로그인을 수행하면 새로 만들어진 쿠키의 값은 데이터베이스로 보관됩니다. 이 과정에서 새로운 쿠키가 만들어지고 데이터베이스는 새로 만들어진 값을 보관하게 됩니다. 공격자가 가지고 있는 쿠키는 이전의 쿠키이므로 데이터베이스를 이용해서 체크하면 공격자가 가진 쿠키 값으로는 자동로그인이 되지 않는 방식입니다.

데이터베이스에 자동로그인 쿠키를 사용하기 위해서는 별도의 테이블을 작성해 주어야 합니다. 물론 이 테이블의 이름과 SQL 등 역시 개발자가 커스터마이징 할 수 있지만 가장 간단하게 구현하고자 한다면 1)지정된 이름의 테이블을 생성하고 2)스프링 시큐리티의 설정을 조정하는 것만으로도 충분합니다.

데이터베이스내에 persistent\_logins 라는 이름의 테이블을 생성합니다(자동으로 테이블을 생성할 수 있는 방법도 있습니다만 만일 테이블이 존재하는 상태에서 실행하면 프로젝트 실행에 문제가 발생하므로 추천하지는 않습니다.).

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

SecurityConfig 는 데이터베이스를 사용하기 위해 DataSource 를 주입 받고, rememberMe( )설정에 tokenRepository 라는 존재를 지정합니다. tokenRepository 의 값이 되는 것은



```
package org.zerock.security;  
  
import javax.sql.DataSource;
```

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.rememberme.JdbcTokenRepositoryImpl;
import org.springframework.security.web.authentication.rememberme.PersistentTokenRepository;

import lombok.extern.log4j.Log4j2;

@Configuration
@Log4j2
@EnableWebSecurity
@EnableMethodSecurity(prePostEnabled = true)
public class SecuriytConfig {

    @Autowired
    private DataSource dataSource;

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

        Log.info("-----security config-----");

        ...생략

        http.rememberMe(config -> {

            config.key("my key");

            config.tokenRepository(persistentTokenRepository());
            config.tokenValiditySeconds(60*60*24*30);

        });

        ...생략

        return http.build();
    }

    ...생략

    @Bean
    public PersistentTokenRepository persistentTokenRepository() {
        JdbcTokenRepositoryImpl tokenRepository = new JdbcTokenRepositoryImpl();
        tokenRepository.setDataSource(dataSource);
        // tokenRepository.setCreateTableOnStartup(true); // 테이블 자동 생성하기 - 추천하지 않음
        return tokenRepository;
    }
}

```

브라우저를 이용해서 자동로그인을 이용하도록 '로그인 상태 유지'를 체크한 후에 로그인을 실행합니다.

데이터베이스를 확인하면 persistent\_logins 테이블에 새로운 레코드가 추가된 것을 확인할 수 있습니다.

```
119 SELECT * FROM persistent_logins;
120
```

persistent_logins (1r × 4c)				
#	username	series	token	last_used
1	User11	GdUcIDK5Ukr+bZ2w8RwVn...	sbgYKKvygVwN2cwEGBn4/A==	2025-05-28 19:09:15

마지막의 last\_used 컬럼은 remember-me 쿠키를 이용한 시간이 기록됩니다. 예를 들어 브라우저를 종료한 후에 다시 로그인이 필요한 경로를 호출하면 컬럼의 값이 변경된 것을 확인할 수 있습니다.

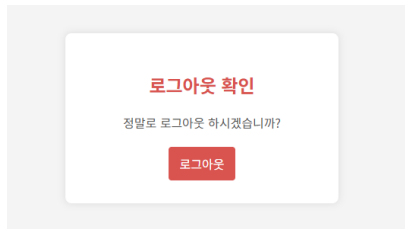
persistent_logins (1r × 4c)				
#	username	series	token	last_used
1	User11	GdUcIDK5Ukr+bZ2w8RwVn...	LVJ9cywj/4fORx0OMrC38A==	2025-05-28 19:17:54

## 로그아웃과 자동로그인

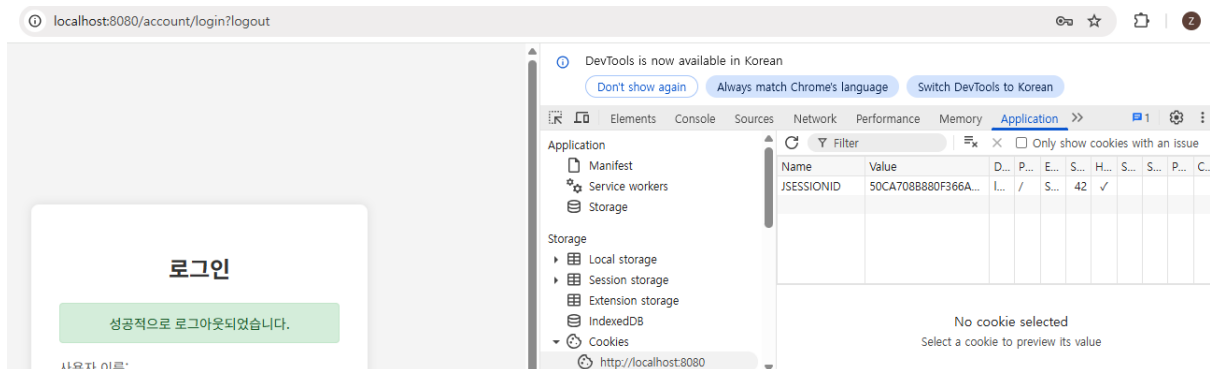
스프링 시큐리티의 자동로그인은 약간의 설정만으로 로그인 정보를 유지할 수 있을 뿐만 아니라 로그아웃시에 신경써야 하는 쿠키의 삭제 역시 자동으로 처리합니다(설정을 통해서 특정한 이름의 쿠키들을 삭제하도록 설정할 수도 있습니다.).

로그인한 상태에서 JSESSIONID 와 remember-me 쿠키가 존재하는지 확인하고 '/account/logout'을 이용해서 로그아웃을 시도해 봅니다.

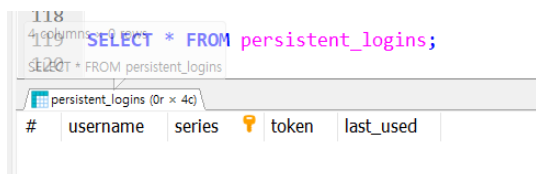
Name	Value	D...	P...	E...	S...	H...	S...	P...
JSESSIONID	B9431E647C68511FD...	L...	/	S...	42	✓		
remember-me	R2RVY0IESzVa3lIMk...	L...	/	2...	93	✓		



정상적으로 로그아웃이 되면 remember-me 쿠키가 삭제된 것을 확인할 수 있습니다.



persistent\_logins 테이블에서도 삭제된 것을 확인할 수 있습니다.



만일 추가적으로 다른 쿠키도 삭제하고 싶다면 아래와 같은 설정을 추가할 수 있습니다.

```
http.logout(config -> {
    config.deleteCookies("JSESSIONID", "remember-me");
});
```

## 10.8 인증 정보의 활용

개발 과정에서는 JSP 나 컨트롤러에서 현재 사용자의 정보를 처리해야 하는 경우가 종종 발생합니다. 예를 들어 화면에서 자신이 작성한 게시물에 대해서만 특정한 버튼을 보여주거나 보여지는 메뉴가 달라지는 경우가 발생할 수 있습니다.

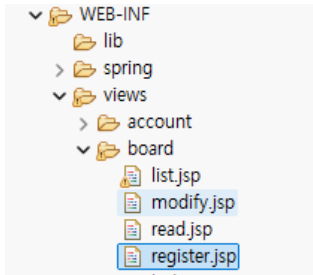


## JSP 에서 인증 정보 활용

JSP 화면에서 인증 정보를 활용하기 위해서 앞에서 설정한 spring-security-taglibs 라이브러리가 반드시 필요합니다. 게시물의 CRUD 등의 작업을 어떻게 처리하는지 살펴보는 것이 도움이 될 것입니다.

### 게시물의 등록

게시물 작성하는 'WEB-INF/board/register.jsp'에서 작성자 부분에 인증 정보를 처리하면 다음과 같습니다.



```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
```

```
<%@include file="/WEB-INF/views/includes/header.jsp" %>
```

JSP 내에 아래와 같은 태그 선언을 추가합니다.

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
```

사용자의 인증 정보는 <sec...>로 시작해서 인증 정보를 확인할 수 있습니다.

```
<div class="row justify-content-center">
  <div class="col-lg-12">
    <div class="card shadow mb-4">
      <div class="card-header py-3">
        <h6 class="m-0 font-weight-bold text-primary">Board Register</h6>
      </div>

      <sec:authentication property="principal"/>
      ...생략
```

위의 코드를 적용하면 화면에서는 현재 인증된 사용자의 정보를 화면에서 아래와 같이 확인할 수 있습니다.

AdminPanel

Dashboard  
Analytics  
User List  
Roles  
System Logs

Board Register

AccountDTO(uid=user11,  
upw=\$2a\$10\$hmFQYK2Ay9okcfl3q8aqUOQ9e6v3VzGQWL8cMWtLG383O.Jwas2,  
uname=User11, email=user11@aaa.com, roleNames=[USER])

Title

Content

Writer

Submit

게시글 작성화면에서는 '작성자(Writer)' 부분은 사용자가 편집할 수 없고 현재 로그인한 사용자만 읽기전용으로 출력합니다.

```
<div class="mb-3">
  <label class="form-label">Writer</label>
  <input type="text"
    name="writer"
    class="form-control"
    value="<sec:authentication property='principal.uid'/">" readonly>
</div>
```

화면에서는 현재 로그인한 사용자의 uid 값이 출력되는 것을 확인할 수 있습니다.

Board Register

Title

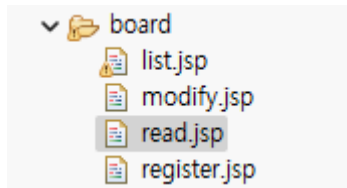
Content

Writer

Submit

## 게시물의 조회

게시물의 조회는 우선 로그인한 사용자만이 접근할 수 있도록 처리되어 있는 상태인데 여기에 현재 게시글의 작성자와 'ROLE\_ADMIN' 권한을 가진 사용자는 '수정/삭제' 화면으로 이동할 수 있도록 화면을 조정할 필요가 있습니다.



```
<%@ include file="/WEB-INF/views/includes/header.jsp" %>

<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>

<div class="row justify-content-center">
```

...생략

```
<div class="float-end">

    <a href='/board/List' class="btn">
        <button type="button" class="btn btn-info btnList"> LIST </button>
    </a>

    <sec:authentication property="principal" var="secInfo" />
    <sec:authentication property="authorities" var="roles" />

    <c:if test="${!board.delFlag && (secInfo.uid == board.writer || fn:contains(roles,
'ROLE_ADMIN'))}">
        <a href='/board/modify/${board.bno}' class="btn">
            <button type="button" class="btn btn-warning btnModify">MODIFY</button>
        </a>
    </c:if>
```

위의 코드를 적용한 후에 다른 사용자가 작성한 게시글을 조회하면 'MODIFY'버튼은 보이지 않고 본인이 작성한 게시글에 대해서만 버튼이 생성됩니다.

Board List

-- ▾

Bno	Title	Writer
<a href="#">7671</a>	USER11번이 작성한 글 [ 0 ]	user11
<a href="#">7670</a>	Modify 7 [ 15 ]	Tester
<a href="#">7669</a>	Test [ 128 ]	Tester

Board Read

Bno: 7669

Title: Test

Content: Test

Writer: Tester

RegDate: 2025-05-10

LIST

Board Read

Bno: 7671

Title: USER11번이 작성한 글

Content: USER11번이 작성한 글

Writer: user11

RegDate: 2025-05-28

LIST MODIFY

만일 로그인한 사용자가 'ADMIN' 권한을 가졌다면 수정이 가능합니다(화면을 위해 사용자 정보를 출력하였습니다.).

Board Read

AccountDTO(uid=**user99**,  
upw=\$2a\$10\$D2DAv3dx0R27.2bYoC4QOF35s/urC1rtCIUQZ4og86sl2sGsXVe,  
uname=User99, email=user99@aaa.com, roleNames=[USER, MANAGER, ADMIN])

Bno

7670

Title

Modify 7

Content

Test

Writer

Tester

RegDate

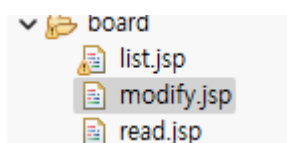
2025-05-10

LIST

MODIFY

## 게시물 수정/삭제

수정과 삭제 역시 현재 게시글의 작성자와 ADMIN 권한을 가진 사용자 만이 가능하다고 가정하면 modify.jsp 는 다음과 같이 변경됩니다.



```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ include file="/WEB-INF/views/includes/header.jsp" %>
```

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
```

...생략

```
<div class="float-end">
    <button type="button" class="btn btn-info btnList">LIST</button>

    <sec:authentication property="principal" var="secInfo" />
    <sec:authentication property="authorities" var="roles" />

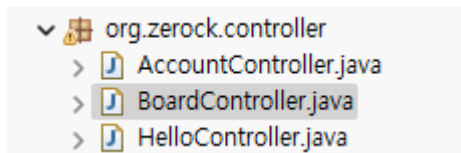
    <c:if test="${!board.delFlag && (secInfo.uid == board.writer || fn:contains(roles,
'ROLE_ADMIN'))}">
        <button type="button" _class="btn btn-warning btnModify">MODIFY</button>
        <button type="button" class="btn btn-danger btnRemove">REMOVE</button>
    </c:if>
</div>
```

## 컨트롤러에서 인증 정보 활용

컨트롤러에서는 @PreAuthorize 에서 사용하는 표현식을 이용해서 현재 사용자의 인증 정보를 체크할 수 있지만 간혹 컨트롤러의 내부에서 사용자의 인증 정보를 변수로 사용해야 하는 상황이 발생할 수도 있습니다.

이러한 상황에서는 org.springframework.security.core.Authentication 을 이용해서 직접 인증 정보를 컨트롤러의 파라미터로 처리할 수 있습니다.

예를 들어 게시물을 조회할 때 현재 사용자의 인증 정보를 사용해 보도록 BoardController 의 read()를 수정해서 Authentication 과 내부의 Principal 을 로그로 기록해 봅니다.



```
@PreAuthorize("isAuthenticated()")
@GetMapping("read/{bno}")
public String read(Authentication authentication, @PathVariable("bno")Long bno, Model model ) {

    Log.info("-----");
    Log.info(authentication);
    Log.info(authentication.getPrincipal());
    Log.info("board read");

    BoardDTO dto = boardService.read(bno);

    model.addAttribute("board", dto);

    return "/board/read";
}
```

프로젝트 실행 후에 read()를 실행한 후 로그를 확인하면 다음과 같은 로그가 기록되는 것을 확인할 수 있습니다.

```
INFO -----
INFO UsernamePasswordAuthenticationToken [Principal=AccountDTO(uid=user11, upw=$2a$10$hmFQYK2Ay9okcfL3q8aqUOQ9e6v3VzGQWLi8cMwtLG3
INFO AccountDTO(uid=user11, upw=$2a$10$hmFQYK2Ay9okcfL3q8aqUOQ9e6v3VzGQWLi8cMwtLG3830..Jwas2, uname=User11, email=user11@aaa.com,
INFO board read
```

로그를 보면 예제에서 사용하는 AccountDTO 는 authentication.getPrincipal()를 통해서 사용할 수 있다는 것을 알 수 있습니다.

만일 AccountDTO 타입으로 사용하고 싶다면 다음과 같이 캐스팅(casting)을 통해서 할 수 있습니다.

```

@PreAuthorize("isAuthenticated()")
@GetMapping("read/{bno}")
public String read(Authentication authentication, @PathVariable("bno")Long bno, Model model ) {

    Log.info("-----");
    Log.info(authentication);
    Log.info("board read");

    AccountDTO accountDTO = (AccountDTO)authentication.getPrincipal();

    Log.info("-----AccountDTO-----");
    Log.info(accountDTO);
    Log.info("-----AccountDTO-----");

    BoardDTO dto = boardService.read(bno);

    model.addAttribute("board", dto);

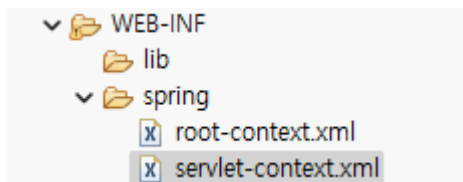
    return "/board/read";
}

```

## @AuthenticationPrincipal 을 이용한 자동 타입 변환

Authentication 을 이용해서 사용자의 정보를 알아 낼 수도 있고 AccountDTO 로 타입을 바꿔서 사용할 수도 있긴 하지만 약간의 코드가 필요합니다. org.springframework.security.core.annotation.AuthenticationPrincipal 은 이러한 상황에서 인증 정보를 주어진 타입으로 강제로 변환하게 하는 어노테이션으로 이를 활용하면 조금 더 간결한 코드를 작성할 수 있습니다.

@AuthenticationPrincipal 을 사용하기 위해서는 프로젝트내 servlet-context.xml 의 설정을 변경해 주어야만 합니다.



```

<mvc:annotation-driven>
    <mvc:argument-resolvers>
        <bean
class="org.springframework.security.web.method.annotation.AuthenticationPrincipalArgumentResolver"/>
    </mvc:argument-resolvers>
</mvc:annotation-driven>

```

앞에서 작성된 read( )의 파라미터와 코드의 일부분을 아래와 같이 수정하고 프로젝트를 재실행 합니다.

```

@PreAuthorize("isAuthenticated()")
@GetMapping("read/{bno}")
public String read( @AuthenticationPrincipal AccountDTO accountDTO, @PathVariable("bno")Long bno,
Model model ) {

    Log.info("-----");
    Log.info("board read");

    Log.info("-----AccountDTO-----");
    Log.info(accountDTO);
    Log.info("-----AccountDTO-----");

    BoardDTO dto = boardService.read(bno);

    model.addAttribute("board", dto);

    return "/board/read";
}

```

변경된 설정이 적용되었고 사용자가 게시물을 조회하면 다음과 같은 결과를 확인할 수 있습니다.

```

INFO -----
INFO board read
INFO == From SecurityContextHolder ==
INFO Principal class: org.zerock.dto.AccountDTO
INFO Principal toString: AccountDTO(uid=user11, upw=$2a$10$hmFQYK2Ay9okcfL3q8aqUOQ9e6v3VzGQWLi8cMwtLG3830..Jwas2, uname=User11, ei
INFO -----AccountDTO-----
INFO AccountDTO(uid=user11, upw=$2a$10$hmFQYK2Ay9okcfL3q8aqUOQ9e6v3VzGQWLi8cMwtLG3830..Jwas2, uname=User11, email=user11@aaa.com,
INFO -----AccountDTO-----

```