

9

세션 9 스프링 시큐리티 활용

작성일 : 2024년 6월 28일

작성자 : 최 강

[스프링 시큐리티란?](#)

[프로젝트 설정 및 의존성](#)

[기본 Controller 작성](#)

[데이터베이스 비활성화](#)

[SecurityConfiging 작성](#)

[커스텀 로그인](#)

[회원 가입 로직 작성](#)

[데이터베이스 기반 로그인 검증 로직](#)

[사용자 아이디 정보 보관 - 세션 방식](#)

[참고자료](#)

[최종 완성 코드](#)

스프링 시큐리티란?

세션 vs 토큰 vs 쿠키? 기초개념 잡아드립니다. 10분 순삭!

#Token #JWT #Cookies #Sessions

개발자라면 이제는 제대로 이해해야합니다.

세션. 토큰. 쿠키. JWT...용어!

<https://www.youtube.com/watch?v=tosLBcAX1vk&t=4s>



웹 사이트를 이용할 때, 특정 url로의 접근 혹은 특정 api로의 접근에 제한을 두고 싶은 경우가 있다.

회원가입을 한 사용자에게만 접근을 허용한다거나 관리자 페이지로의 접근은 **관리자 권한** 을 가진 사용자만 허용하고 싶은 경우가 이와 같은 경우이다.

이런 상황에서 사용하는 것이 바로 스프링 시큐리티 프레임워크로, 스프링 시큐리티 프레임워크를 사용하면 사용자의 **인증(Authentication)** 과 **인가(Authorization)** 을 통해 허가받은 사용자만 접근을 허용할 수 있게 된다.

이번 세션에서는 스프링 시큐리티 프레임워크를 활용하여 세션방식으로 간단한 로그인,회원가입 페이지의 구현과 사용자별 인가처리를 통해 admin 페이지로의 접근을 구현해볼 예정이다.

프로젝트 설정 및 의존성

▼ 필수 의존성

- Spring Web
- Lombok
- Tnymleaf

- Spring Security
- Spring Data JPA
- MySQL Driver

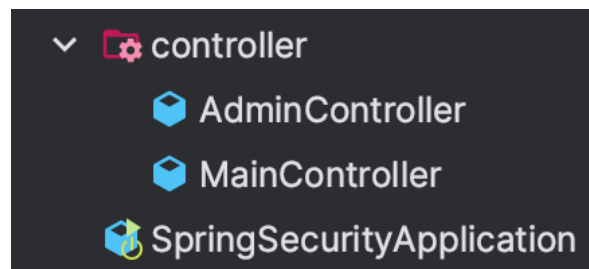
아래와 같이 버전과 의존성을 선택하여 프로젝트 생성을 진행한다.

start.spring.io
https://start.spring.io/

The screenshot shows the Spring Start configuration interface. On the left, under 'Project', 'Language' is set to Java. Under 'Spring Boot', version 3.3.1 is selected. The 'Project Metadata' section includes fields for Group (com.example), Artifact (spring-security), Name (spring-security), Description (Demo project for Spring Boot), and Package name (com.example.spring-security). Packaging is set to Jar, and Java version 17 is selected. On the right, the 'Dependencies' section lists selected dependencies: Spring Web (WEB), Lombok (DEVELOPER TOOLS), Spring Data JPA (SQL), MySQL Driver (SQL), Spring Security (SECURITY), and Thymeleaf (TEMPLATE ENGINES). Each dependency has a brief description.

기본 Controller 작성

controller 패키지를 만들고 MainController와 AdminController를 작성한다.



MainContro | ler.java

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

@Controller
public class MainController {

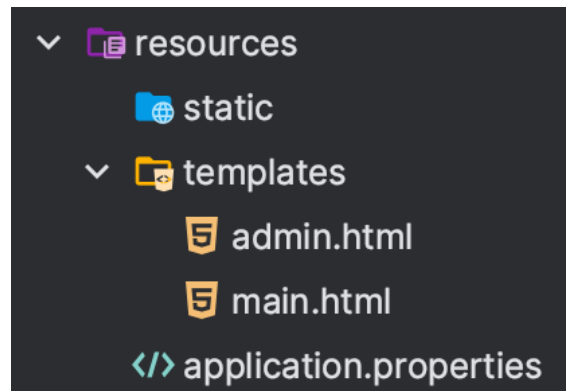
    @GetMapping("/")
    public String mainPage() {
        return "main";
    }
}
```

```
}  
}
```

AdminController.java

```
package com.example.spring_security.controller;  
  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.GetMapping;  
  
@Controller  
public class AdminController {  
  
    @GetMapping("/admin")  
    public String adminPage() {  
        return "admin";  
    }  
}
```

타임리프를 사용하여 화면에 메인 페이지를 보여줄 예정이므로, resources - template 디렉토리 하부에 main.html, admin.html 를 작성한다.



main.html

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <title>Title</title>  
</head>  
<body>  
    This is Main Page!!  
</body>  
</html>
```

admin.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
  This is Admin Page!
</body>
</html>
```

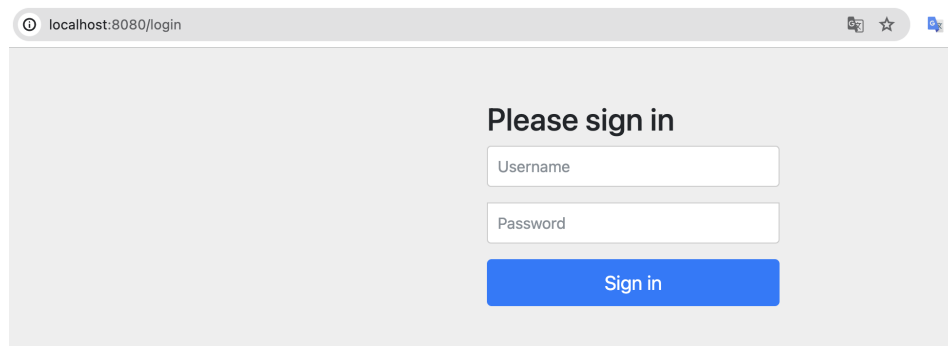
데이터베이스 비활성화

기존에 설치해둔 의존성대로 서버를 실행시키면 데이터베이스 연동이 아직 되어있지 않기 때문에 데이터베이스를 임시로 비활성화 시켜준다.

```
dependencies {
    // implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    implementation 'org.springframework.boot:spring-boot-starter-security'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    compileOnly 'org.projectlombok:lombok'
    // runtimeOnly 'com.mysql:mysql-connector-j'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'org.springframework.security:spring-security-test'
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
}
```

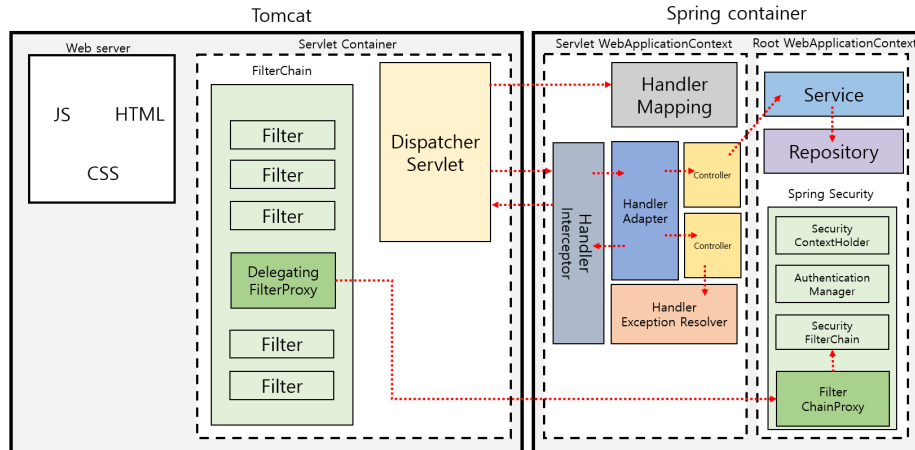
이제 프로젝트를 실행시켜보자

메인페이지가 보여지길 기대하였으나 /login 페이지로 연결되는 것을 확인 할 수 있다.



어떻게 이런일이 일어나는 것일까?

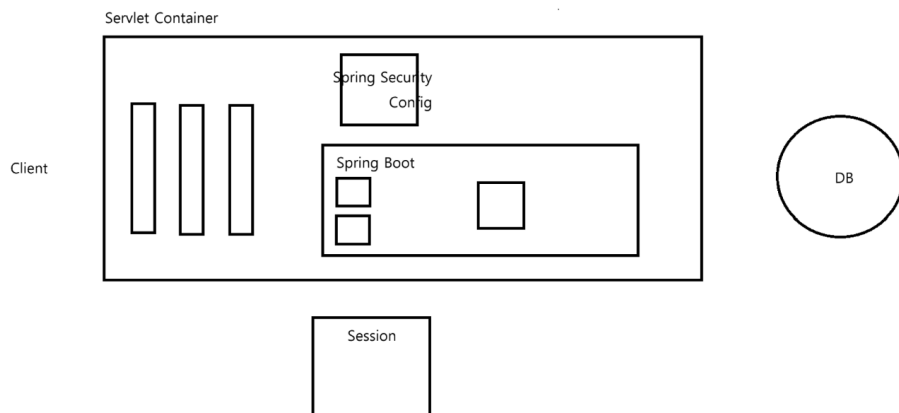
스프링의 구조를 살펴보면 아래와 같다.



사용자의 요청이 발생하면 서블릿 컨테이너의 여러 필터들을 통과한 후 Dispatcher Servlet에 의해 적절한 Controller로 배치되고 이후부터는 우리가 알고있는 흐름대로 Controller - Service - Repository 등의 계층을 거쳐 요청이 진행된다.

Spring Security 프레임워크를 사용하게 되면 FilterChain의 Delegating Filter Proxy에 의해 사용자의 요청이 가로채지며 Security Filter Chain에 의해 사용자 인증(Authentication)과 인가(Authorization)를 거쳐 접근 권한이 있는 사용자에게 접근을 허용하게 되는 것이다.

그림을 더 단순화 해서 그려보면 아래와 같다.



Client, 즉 사용자가 요청을 보내게 되면 Spring Security 프레임워크에 의해 Security Filter가 자동으로 생성되고 이 필터를 거쳐 사용자 인증, 인가 작업이 완료되면 스프링 프레임워크는 사용자 정보를 Session에 저장하게 된다.

앞선 페이지를 살펴보면 로그인을 하라고 되어있는데 사용자가 로그인한 정보와 일치하는 데이터가 데이터베이스에 존재하는지 확인하여 일치하는 경우에만 인증을 허용하게 된다.

하지만 아직 데이터베이스에 회원이 존재하지 않으며 스프링 시큐리티가 회원정보를 확인할 수단이 제공되지 않았으므로 스프링 시큐리티는 임시 회원을 생성해준다.

임시 회원의 username은 `user` 이며, 비밀번호를 서버 로그를 통해 찾을 수 있다.

```

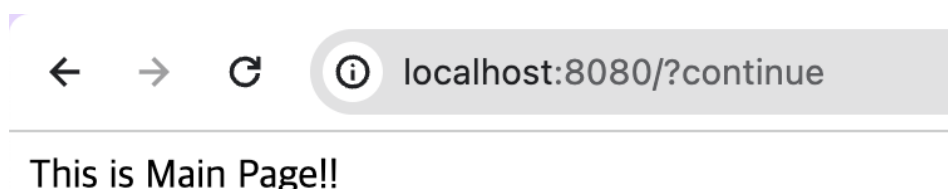
2024-06-28T14:20:10.703+09:00 INFO 34177 --- [spring-security] [
2024-06-28T14:20:10.722+09:00 INFO 34177 --- [spring-security] [
2024-06-28T14:20:10.723+09:00 INFO 34177 --- [spring-security] [
2024-06-28T14:20:10.849+09:00 WARN 34177 --- [spring-security] [

Using generated security password: 912b3345-1ecd-4188-8be0-5909c8bccae1

This generated password is for development use only. Your security confi

```

이와 같은 정보를 로그인 페이지에 입력하고 Sign in 버튼을 클릭하면 메인 페이지로 정상적으로 접근되는 것을 확인 할 수 있다.

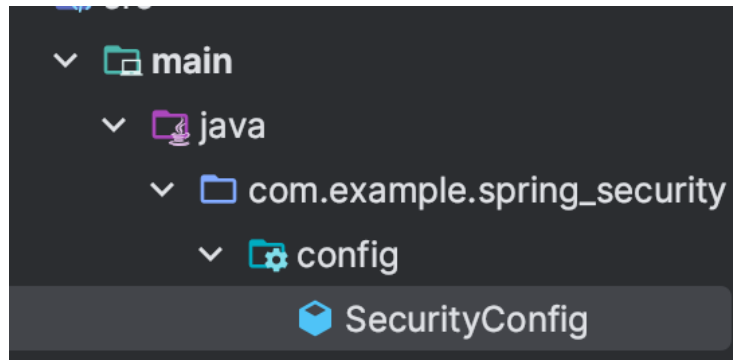


SecurityConfing 작성

별도의 설정을 하지 않으면 스프링 시큐리티 프레임워크는 기본 시큐리티 필터를 생성하여 위와 같은 로그인 페이지를 제공하며 모든 페이지에 대해 인증을 요구하며 인증된 사용자(로그인에 성공한 사용자)에 한해 접근을 허용하게 된다.

이제부터 시큐리티 필터를 커스터마이징하여 특정 페이지로의 접근만 허용하지 않고, 추가로 특정한 권한을 갖춘 사용자에게 한해서만 접근을 허용하도록 인증, 인가 절차를 설정해보자.

이를 위해서 config 디렉토리를 만들고 SecurityFilterChain을 커스텀 빈으로 등록하기 위한 구성 클래스를 작성해야한다.



SecurityConfig.class

```
package com.example.spring_security.config;

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;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    // SecurityFilterChain 커스텀 빈 스프링 빈으로 등록
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception{

        // Spring Security 버전에 따라 구현 방식이 다름
        http
            .authorizeHttpRequests(auth -> auth // 람다 식으로 작성 해야함
                // 메인페이지와 로그인 페이지로의 접근은 모두 허용(인증 필요x)
                .requestMatchers("/", "/login").permitAll()
                // admin 페이지로의 접근은 ADMIN 권한을 가진 경우에만 허용
                .requestMatchers("/admin").hasRole("ADMIN")
                // my 경로 이후의 모든 경로에 대해서 ADMIN, 혹은 USER권한 보유시 접근 가능
                .requestMatchers("/my/**").hasAnyRole("ADMIN", "USER")
                // 위에서 설정하지 않은 모든 요청에 대해서는 인증(authenticated)된 사용자에게 한해 허용
            );
    }
}
```

```

        .anyRequest().authenticated()
    );

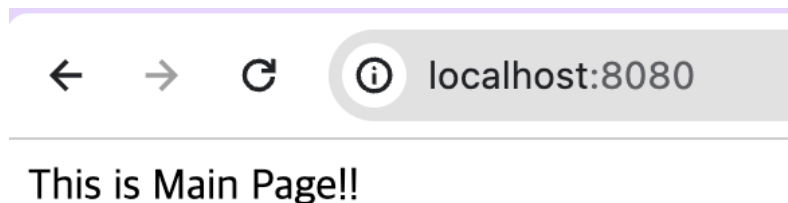
    return http.build();
}
}

```

HttpSecurity에서 제공하는 함수들에 대해 간단히 설명하면 아래와 같다.

- requestMatchers() : 사용자의 요청에 따른 인증,인가 조건을 설정한다.
- permitAll() : 모든 사용자에게 접근을 허용한다. 위 예제에선, 루트 디렉토리와 로그인 페이지는 인증(Authentication)을 수행하지 않은 사용자도 접근을 허용하도록 하였다.
- hasRole(ROLE) : 인증 절차를 마친 사용자에게 대해 ROLE 권한을 가진 사용자의 접근을 허용한다.
- hasAnyRole(ROLE1,ROLE2 ..) : ROLE1, ROLE2 .. 중 하나의 권한이라도 가진 사용자의 접근을 허용한다.
- anyRequest() : requestMatchers()를 통해 설정하지 않은 모든 요청 경로에 대해 설정한다.
- authenticated() : 인증된 사용자라면 모두 허용한다.

위와 같이 구성클래스를 작성한 후 메인 페이지로 접근해보면 이전과 달리 로그인 페이지로 연결되지 않고 정상적으로 접근되는 것을 확인 할 수 있다.



admin 페이지로의 접근을 시도하면 어떻게 될까?



localhost에 대한 액세스가 거부됨

이 페이지를 볼 수 있는 권한이 없습니다.

HTTP ERROR 403

새로고침

로그인 페이지가 연결될 것이라고 기대한바와 달리 액세스가 거부되는 것을 확인 할 수 있다.

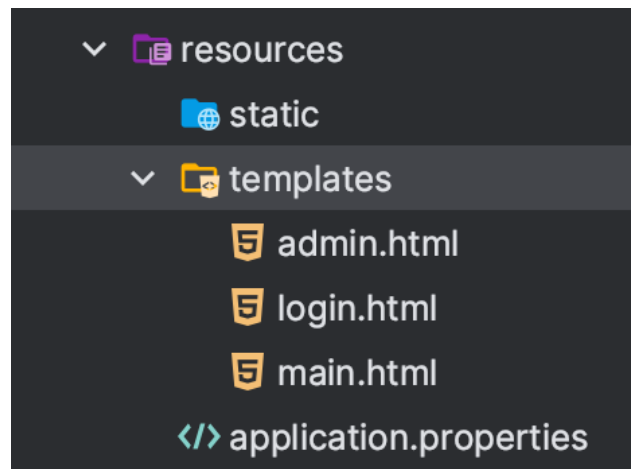
액세스가 거부되는 원인은 현재 요청을 한 사용자가 인증(Authentication)을 수행하지 않았고 그로 인해 사용자의 권한을 확인할 수 없어 인가(Authorization)을 수행할 수 없기 때문이다.

스프링 시큐리티 프레임워크가 디폴트로 생성해주는 SecurityFilterChain을 사용하게 되면 폼 로그인을 제공주나, SecurityConfig 구성 클래스를 사용하여 SecurityFilterChain을 커스텀하여 생성해주었기 때문에 디폴트 로그인 폼으로의 연결이 설정되어있지 않아 인증 절차를 수행할 수 없게 되는 것이다.

커스텀 로그인

앞서 말한 인증을 위해 커스텀 로그인 설정을 작성해보자.

먼저 로그인 페이지로 사용하기 위한 login.html 파일을 작성한다.



login.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Login</title>
```

```

</head>
<body>
  <h1>Login Page</h1>
  <hr>
  <form action="/loginProc" method="post" name="loginForm">
    <input id="username" type="text" name="username" placeholder="id"/>
    <input id="password" type="password" name="password" placeholder="password"/>
    <input type="submit" value="login">
  </form>
</body>
</html>

```

추가로 /login 요청시 로그인 페이지로 연결해주기 위한 LoginController도 작성해준다.

LoginController

```

package com.example.spring_security.controller;

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

@Controller
public class LoginController {

    // /login 요청시 로그인 페이지로 연결
    @GetMapping("/login")
    public String loginPage() {
        return "login";
    }
}

```

The screenshot shows a web browser window with the address bar set to 'localhost:8080/login'. The page content includes a heading 'Login Page' followed by a horizontal line. Below the line is a login form consisting of two text input fields, one labeled 'id' and the other 'password', followed by a button labeled 'login'.

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Login</title>
</head>
<body>
  <h1>Login Page</h1>
  <hr>

```

```

<form action="/loginProc" method="post" name="loginForm">
    <input id="username" type="text" name="username" placeholder="id"/>
    <input id="password" type="password" name="password" placeholder="password"/>
    <input type="submit" value="login">
</form>
</body>
</html>

```

html 코드를 살펴보자, 해당 페이지를 통해 사용자가 아이디와 비밀번호를 입력하고 login 버튼을 누르게되면, form 태그의 action에 의해 /loginProc 에 POST요청으로 사용자의 아이디와 비밀번호가 전달된다. 그러므로 /loginProc 에 대한 매핑 함수를 작성한 뒤 사용자가 입력한 아이디와 비밀번호를 통해 인증과 인가 절차를 수행하는 코드를 작성해야 할 것이다. 이러한 로그인 방식을 **폼 로그인(form login)** 방식이라고 한다.

이제 로그인 페이지를 만들었으니 /admin 로 요청시 인증을 수행하기위해 /login 요청으로 연결시켜주어야 한다.

이를 위해 SecurityConfig 파일의 filterChain 함수를 아래와 같이 수정한다.

```

package com.example.spring_security.config;

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;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    // SecurityFilterChain 커스텀 빈 스프링 빈으로 등록
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception{

        // Spring Security 버전에 따라 구현 방식이 다름
        http

            .authorizeHttpRequests(auth -> auth // 람다 식으로 작성 해야함
                // 메인페이지와 로그인 페이지로의 접근은 모두 허용(인증 필요x)
                .requestMatchers("/", "/login").permitAll()
                // admin 페이지로의 접근은 ADMIN 권한을 가진 경우에만 허용
                .requestMatchers("/admin").hasRole("ADMIN")
                // my 경로 이후의 모든 경로에 대해서 ADMIN, 혹은 USER권한 보유시 접근 가능
                .requestMatchers("/my/**").hasAnyRole("ADMIN", "USER")
                // 위에서 설정하지 않은 모든 요청에 대해서는 인증(authenticated)된 사용자에게 한해 허용
                .anyRequest().authenticated()
            )
        // form 로그인 방식을 사용할 것이다.
        .formLogin(auth -> auth
            .loginPage("/login") // 로그인은 /login에서 수행한다.
            .loginProcessingUrl("/loginProc")
            .permitAll()
        ) // 로그인 프로세싱은 해당 경로로 수행한다.
    }
}

```

```

        // 폼 로그인 방식을 사용하게 되면 csrf 설정이 디폴트로 동작하게 된다.
        // csrf가 동작하면 POST 요청을 보낼 때 csrf 토큰도 함께 보내야 요청이 수행된다.
        // 우리의 개발환경에서는 토큰을 보내주도록 설정하지 않았기 때문에 임시로 비활성화 한다.
        .csrf(auth -> auth.disable());

    return http.build();
}
}

```

폼 로그인 방식은 기본적으로 csrf 공격을 방지하기 위해 csrf 설정이 활성화 되어있다.

하지만 우리의 개발환경에서는 csrf를 위한 코드 설정이 되어있지 않기 때문에 임시로 비활성화 하도록 한다.

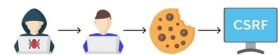
CSRF에 대한 추가 정보를 얻고 싶다면 아래 블로그 글을 참고하길 바란다.

CSRF란, CSRF 동작원리, CSRF 방어방법

CSRF란, CSRF 동작원리, CSRF 방어방법 CSRF란 CSRF란, Cross Site Request Forgery의 약자로, 한글 뜻으로는 사이트간 요청 위조를 뜻합니다. CSRF는 웹 보안 취약점의 일종이며, 사용자가 자신의 의지와는 무관하게 공격자가 의도한 행위(데이터 수정, 삭제, 등록 등)을 특정 웹사이트에 요청하게 하는 공격입니다. 예를 들어, 피해자의 전자 메일 주소

<https://devscb.tistory.com/123>

CSRF - Cross site request forgery attack



회원 가입 로직 작성

로그인을 하기 위해서는 먼저 회원가입이 선행되어 사용자 정보가 데이터베이스에 등록되어 있어야 한다.

데이터베이스 연동

이전에 주석해두었던 데이터베이스 관련 의존성들을 다시 활성화 시킨 후, application.properties를 통해 데이터베이스 연동을 진행해보자 (우측 상단의 코끼리 버튼을 눌러 의존성을 업데이트 해야한다.)

```

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'org.springframework.boot:spring-boot-starter-security'
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.thymeleaf.extras:thymeleaf-extras-springsecurity6'
    compileOnly 'org.projectlombok:lombok'
    runtimeOnly 'com.mysql:mysql-connector-j'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'org.springframework.security:spring-security-test'
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
}

```

application.properties

```
spring.application.name=spring-security

# DATABASE Connection
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/spring-security
spring.datasource.username=root
spring.datasource.password=1234

# JPA
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

logging.level.org.springframework.security=DEBUG
```

(database url은 생성한 데이터베이스 이름에 맞춰 적절히 설정한다.)

암호화 메서드 작성

스프링 시큐리티는 회원가입된 사용자의 비밀번호가 암호화 되어있다는 것을 전제로 한다.

따라서 사용자 정보를 DB에 저장하기 전에 비밀번호를 암호화한 뒤 저장해두어야 나중에 인증 작업을 수행할 수 있다.

암호화 방법에는 여러가지 방법이 있지만, 스프링 시큐리티는 암호화를 위해 BCrypt Password Encoder를 권장하므로 해당 클래스를 Bean으로 등록하여 어디서든 사용가능하도록 할 것이다.

SecurityConfig.java

추가로 SecurityFilterChain 함수의 requestMatchers에 "join"과 "joinProc" 에 대한 경로도 허용해주도록 한다.

```
package com.example.spring_security.config;

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.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    // 비밀번호 암호화를 위한 빈 생성
    @Bean // 빈으로 등록시킴으로서 어디서든 사용할 수 있게된다.
    public BCryptPasswordEncoder bCryptPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }

    // SecurityFilterChain 커스텀 빈 스프링 빈으로 등록
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception{
```

```

// Spring Security 버전에 따라 구현 방식이 다름
http
    .authorizeHttpRequests(auth -> auth // 람다 식으로 작성 해야함
        // 메인페이지와 로그인 페이지로의 접근은 모두 허용(인증 필요x)
        .requestMatchers("/", "/login", "/join", "joinProc").permitAll()
        // admin 페이지로의 접근은 ADMIN 권한을 가진 경우에만 허용
        .requestMatchers("/admin").hasRole("ADMIN")
        // my 경로 이후의 모든 경로에 대해서 ADMIN, 혹은 USER권한 보유시 접근 가능
        .requestMatchers("/my/**").hasAnyRole("ADMIN", "USER")
        // 위에서 설정하지 않은 모든 요청에 대해서는 인증(authenticated)된 사용자에게 한해 허용
        .anyRequest().authenticated()
    )
    // form 로그인 방식을 사용할 것이다.
    .formLogin(auth -> auth
        .loginPage("/login") // 로그인은 /login에서 수행한다.
        .loginProcessingUrl("/loginProc")
        .permitAll()
    ) // 로그인 프로세싱은 해당 경로로 수행한다.

    // 폼 로그인 방식을 사용하게 되면 csrf 설정이 디폴트로 동작하게 된다.
    // csrf가 동작하면 POST 요청을 보낼 때 csrf 토큰도 함께 보내야 요청이 수행된다.
    // 우리의 개발환경에서는 토큰으로 보내주도록 설정하지 않았기 때문에 임시로 비활성화 한다.
    .csrf(auth -> auth.disable());

    return http.build();
}
}

```

이제부터 회원가입을 위한 join.html 과 JoinController를 작성해보자

join.html

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>JOIN</title>
</head>
<body>
    <h1>This is Join Page</h1>
    <form action="/joinProc" method="post" name="joinForm">
        <input type="text" name="username" placeholder="Username"/>
        <input type="password" name="password" placeholder="Password"/>
        <input type="submit" value="Join"/>
    </form>

</body>
</html>

```

JoinController.java

```
package com.example.spring_security.controller;

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

@Controller
public class JoinController {

    // 회원가입 페이지로 이동
    @GetMapping("/join")
    public String joinPage() {
        return "join";
    }

    // 회원가입
    @PostMapping("/joinProc")
    public String joinProcess() {
        return "redirect:/login"; // 로그인 페이지로 리다이렉트
    }
}
```

이후 회원가입 페이지에 대한 접근 설정을 추가해주어야한다. 회원가입 페이지는 모든 사용자에게 접근을 허용하는 것이 일반적이므로 SecurityConfig 의 SecurityFilterChain의 requestMatchers에 "join" 과 "joinProc" 으로의 접근을 모두 허용하도록 설정해야한다.

```
package com.example.spring_security.config;

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.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

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

    // SecurityFilterChain 커스텀 빈 스프링 빈으로 등록
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception{

        // Spring Security 버전에 따라 구현 방식이 다름
        http
```

```

        .authorizeHttpRequests(auth -> auth // 람다 식으로 작성 해야함
            // 메인페이지와 로그인 페이지로의 접근은 모두 허용(인증 필요x)
            .requestMatchers("/", "/login", "/join", "/joinProc").permitAll()
            // admin 페이지로의 접근은 ADMIN 권한을 가진 경우에만 허용
            .requestMatchers("/admin").hasRole("ADMIN")
            // my 경로 이후의 모든 경로에 대해서 ADMIN, 혹은 USER권한 보유시 접근 가능
            .requestMatchers("/my/**").hasAnyRole("ADMIN", "USER")
            // 위에서 설정하지 않은 모든 요청에 대해서는 인증(authenticated)된 사용자에게 한해 허용
            .anyRequest().authenticated()
        )
        .formLogin(auth -> auth.loginPage("/login")
            .loginProcessingUrl("/loginProc")
            .permitAll())
        // csrf 설정 임시 비활성화
        // 실제 서비스를 위해서는 활성화 시켜주어야한다.
        // 이번 세션은 스프링 시큐리티가 무엇인지 학습하는 것이 목표였기 때문에 개발단계의 편의를 위해 비
        .csrf(auth -> auth.disable());

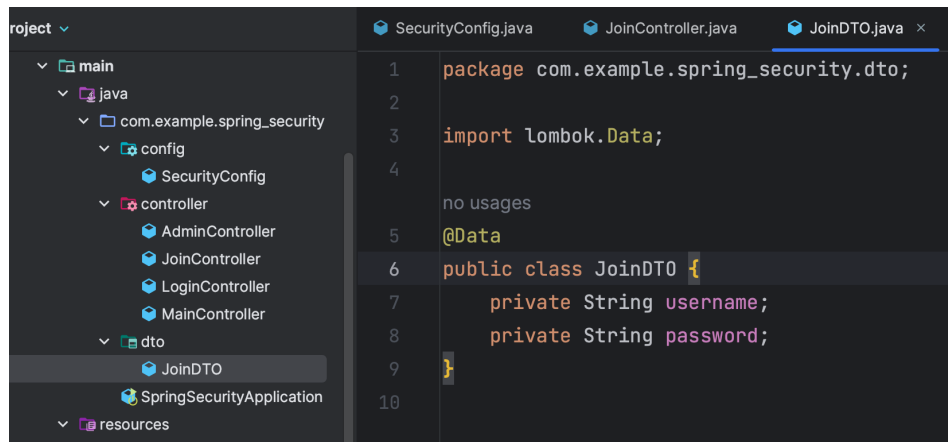
    return http.build();
}
}

```

회원 가입을 위한 JoinDTO, UserEntity, JoinService, UserRepository도 차례로 작성해주도록 한다.

JoinDTO

사용자가 로그인 폼에 입력한 값들을 가져오기 위해 사용하는 DTO클래스, dto 패키지 하단에 작성한다.



```

package com.example.spring_security.dto;

import lombok.Data;

@Data
public class JoinDTO {
    private String username;

```



```

        private String password;
    }

```

UserEntity

사용자 정보를 저장하기 위한 엔티티, entity 패키지 하단에 작성한다.

동일한 이름을 가진 회원이 생성되는 것을 방지하기 위해 unique = true 속성을 부여한다.

```

package com.example.spring_security.entity;

import jakarta.persistence.*;
import lombok.*;

@Entity
@Getter
@Builder
@NoArgsConstructor(access = AccessLevel.PROTECTED)
@AllArgsConstructor(access = AccessLevel.PROTECTED)
@Table(name = "USERENTITY")
public class UserEntity {

    @Id @GeneratedValue
    private Long id;

    @Column(unique = true) // 중복 방지용
    private String username;
    private String password;

    private String role; // 사용자 권한 확인용
}

```

UserRepository

사용자 정보를 DB에 저장하기 위한 레파지토리 클래스, Spring Data JPA 라이브러리를 사용하여 인터페이스만으로 기본적인 CRUD 함수들이 자동 구현된다. repository 패키지 하단에 작성한다.

동일 이름을 가진 회원이 생성되는 것을 방지하기 위한 함수로 existsByUsername(String username) 을 작성한다. 이 함수는 Spring Data JPA에 의해 구체화되며, username에 해당하는 회원이 존재할 경우 true를, 존재하지 않는 경우 false를 리턴한다.

```

package com.example.spring_security.repository;

import com.example.spring_security.entity.UserEntity;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface UserRepository extends JpaRepository<UserEntity, Long> {

    // 회원 이름 중복 방지용
    boolean existsByUsername(String username); // 해당하는 회원이 존재하는지 확인
}

```

```
// UserDetailsService에서 사용하기 위한 유저 검증 함수
Optional<UserEntity> findByUsername(String username);
}
```

JoinService

회원가입을 수행하기 위한 서비스 계층 클래스에 해당한다. service 패키지 하단에 작성한다.

사용자의 비밀번호를 암호화하여 저장하기 위해 BCryptPasswordEncoder를 주입받는다.

userRepository의 existsByUsername()를 호출하여 동일한 이름의 회원이 생성되는 것을 방지한다.

```
package com.example.springsecuritystudy.service;

import com.example.springsecuritystudy.dto.JoinDTO;
import com.example.springsecuritystudy.entity.UserEntity;
import com.example.springsecuritystudy.repository.UserRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.stereotype.Service;

@Service
@RequiredArgsConstructor
public class JoinService {

    private final UserRepository userRepository;

    // 회원가입시 사용자 비밀번호 암호화
    private final BCryptPasswordEncoder passwordEncoder;

    // 회원가입 진행
    public void joinProcess(JoinDTO joinDTO) {

        // 회원 중복 검증
        boolean isUser = userRepository.existsByUsername(joinDTO.getUsername());
        if (isUser) {
            return;
        }

        // 저장할 사용자 생성
        UserEntity user = UserEntity.builder()
            .username(joinDTO.getUsername())
            .password(passwordEncoder.encode(joinDTO.getPassword())) // 비밀번호 암호화
            .role("ROLE_USER") // 권한 부여시 "ROLE_" 접두사를 앞에 붙여주어야 한다.
            .build();

        userRepository.save(user);
    }
}
```

이제 Service, Repository계층의 작성이 완료되었으므로 JoinController의 내용을 아래와 같이 수정하도록 한다.

JoinProcess에서는 JoinDTO를 폼으로부터 전달 받도록 매개변수에 적어준다.

```

package com.example.spring_security.controller;

import com.example.spring_security.dto.JoinDTO;
import com.example.spring_security.service.JoinService;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;

@Controller
@RequiredArgsConstructor
public class JoinController {

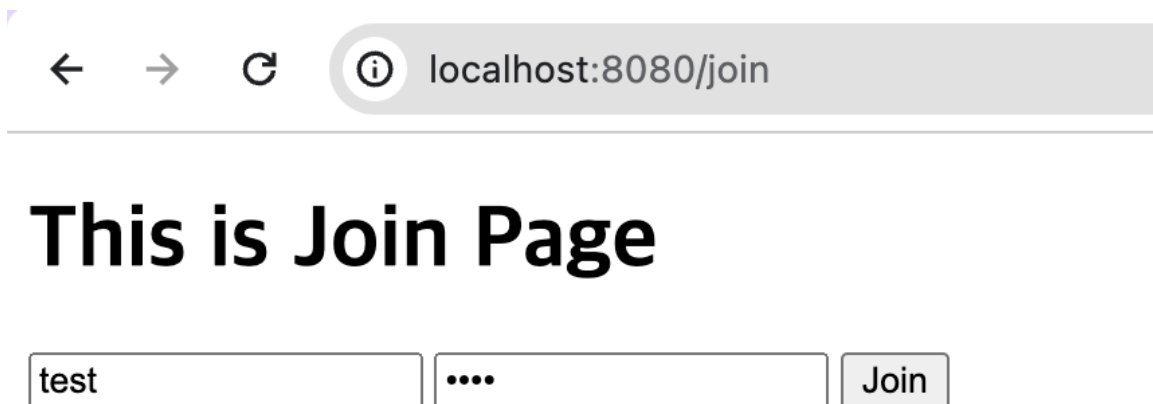
    private final JoinService joinService;

    @GetMapping("/join")
    public String joinPage() {
        return "join";
    }

    // 회원가입
    @PostMapping("/joinProc")
    public String joinProcess(JoinDTO joinDTO) {
        // 회원가입 로직 작성 필요
        joinService.joinProcess(joinDTO); // 회원가입 수행
        return "redirect:/login"; // 회원가입 성공시 로그인 페이지로 연결
    }
}

```

이후 join 페이지로 접근하여 회원가입을 진행해보면 아래와 같이 정상적으로 회원이 저장되는 것을 확인할 수 있다.



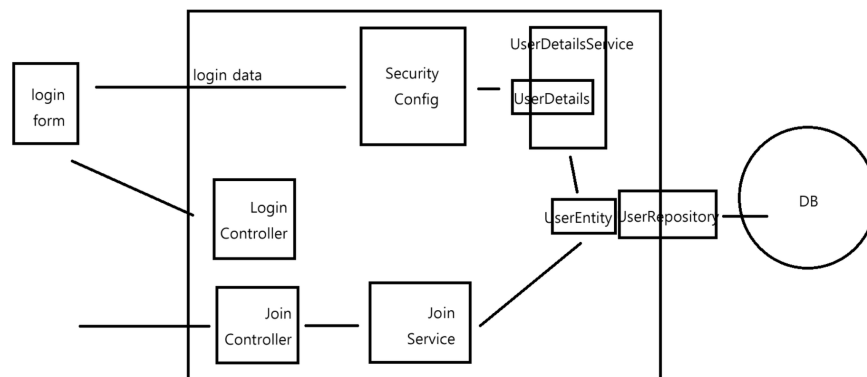
← → ↻ ⓘ localhost:8080/join

This is Join Page

test Join

1 • SELECT * FROM `spring-security`.`userentity`;				
100% 44:1				
Result Grid Filter Rows: Search Edit: Export/Import:				
id	password	role	username	
1	\$2a\$10\$JGWIUN.Hncdsbh7oBevWJO2NVtjdn...	ROLE_USER	test	
NULL	NULL	NULL	NULL	

데이터베이스 기반 로그인 검증 로직



사용자가 인증(Authentication)을 위해 로그인을 시도하면 스프링 시큐리티는 `UserDetailsService` 라는 서비스계층에서 데이터베이스에 접근하여 사용자 정보를 조회하여 검증을 수행한다. `UserDetailsService`는 `UserDetails` 객체(이 안에 DB에서 조회한 사용자 정보가 담김)를 반환하는데, 이는 스프링 시큐리티가 사용자 정보를 저장하고 관리하는 데 사용된다.

기본적으로 스프링 시큐리티는 `User` 엔티티를 사용하여 로그인 과정을 처리한다.

하지만, 사용자 정의 엔티티(이 게시글의 `UserEntity.class`에 해당)를 사용하여 로그인 과정을 처리하고자 할 경우, `UserDetailsService`를 구현한 `CustomUserDetailsService`를 정의해주어야 한다.

`UserDetailsService` 인터페이스는 `loadUserByUsername(String username)` 메서드를 제공한다.

이 메서드는 사용자 이름(username)을 통해 데이터베이스에서 정보를 조회하여 `UserDetails` 객체를 반환하는 역할을 한다.

따라서, 사용자 정의 엔티티를 사용하여 인증 절차를 구현하기 위해서는 `CustomUserDetailsService` 클래스에서 `loadUserByUsername()` 메서드를 오버라이딩하여, 사용자 이름을 통해 사용자 정보를 조회하고, 이를 기반으로 `UserDetails` 객체를 반환하도록 구현해야 한다.

이제부터 이 구현과정을 차근차근 진행해보자

service 디렉토리 밑에 CustomUserDetailService 를 작성한다.

CustomUserDetailService의 이름은 무엇으로 짓든 상관 없으며, UserDetailsService를 구체화하여 스프링 시큐리티가 이 서비스 클래스를 통해 인증/인가를 수행하도록 한다.

UserDetailsService는 loadUserByUsername 함수를 사용하여 UserDetails를 생성하여 사용자 정보를 관리하는데, CustomUserDetailsService는 이를 상속받았으므로 해당 함수를 구체화 해주어야 한다.

매개변수로 전달받은 username(사용자 이름)을 바탕으로 우리가 정의한 커스텀 엔티티(UserEntity)를 정의한 다음, 해당 엔티티로부터 정보를 추출하여 UserDetails를 생성하여 반환해주면 앞서 정의한 SecurityConfig로 전달되어 비로서 사용자의 인증/인가가 진행되는 것이다.

```
package com.example.spring_security.service;

import com.example.spring_security.entity.UserEntity;
import com.example.spring_security.repository.UserRepository;
import lombok.RequiredArgsConstructor;
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 java.util.Optional;

// 스프링 시큐리티에서 인증과 인가 처리를 서비스 계층
// UserDetailsService를 상속받아 구현한다.

@Service
@RequiredArgsConstructor
public class CustomUserDetailService implements UserDetailsService {

    // 실제 사용자 정보를 불러오기 위한 레포지토리 계층
    private final UserRepository userRepository;

    // UserDetailsService 를 구체화한다.
    // username => 사용자 이름에 해당한다.
    // => 데이터베이스에 접근하여 인증(로그인) 절차를 수행한다.
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        Optional<UserEntity> optionalUserEntity = userRepository.findByUsername(username);

        if (optionalUserEntity.isPresent()) {
            UserEntity userEntity = optionalUserEntity.get();

            // 조회된 사용자 정보를 바탕으로 UserDetails를 생성하여 반환한다.
            return new CustomUserDetails(userEntity);
        }

        // 사용자 정보가 없는 경우에 해당
        return null;
    }
}
```

```
}  
}
```

UserDetails를 생성하여 반환하기 위해 CustomUserDetails라는 객체를 정의하도록 한다.

해당 객체는 UserDetails 인터페이스를 구체화하는 것으로 여러가지 함수를 정의해주어야 한다.

또한 CustomUserDetailsService로부터 엔티티 정보를 전달받기 위해 UserEntity를 매개변수로 전달받는 생성자를 추가로 작성해주었다.

CustomUserDetails

```
package com.example.spring_security.dto;  
  
import com.example.spring_security.entity.UserEntity;  
import org.springframework.security.core.GrantedAuthority;  
import org.springframework.security.core.userdetails.UserDetails;  
  
import java.util.ArrayList;  
import java.util.Collection;  
  
public class CustomUserDetails implements UserDetails {  
  
    private UserEntity userEntity;  
  
    public CustomUserDetails(UserEntity userEntity) {  
        this.userEntity = userEntity;  
    }  
  
    // 사용자의 권한을 반환한다. // 데이터베이스에 저장한 ROLE에 해당한다.  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() {  
  
        Collection<GrantedAuthority> collection = new ArrayList<>();  
  
        collection.add(new GrantedAuthority() {  
            @Override  
            public String getAuthority() {  
                return userEntity.getRole(); // 사용자의 권한 리턴  
            }  
        });  
  
        return collection;  
    }  
  
    // 사용자 비밀번호를 반환한다.  
    @Override  
    public String getPassword() {  
        return userEntity.getPassword();  
    }  
  
    // 사용자 username을 반환한다.  
    @Override  
    public String getUsername() {
```

```

        return userEntity.getUsername();
    }
}

```

필수적으로 구현해주어야 하는 함수들은 아래와 같다.

1. `getAuthorities()` : 사용자가 보유한 권한들을 `Collection<GrantedAuthority>` 형태로 반환한다. 엔티티가 보유중인 권한들을 바탕으로 `GrantedAuthority()`를 생성한 뒤 컬렉션에 삽입하고 반환한다.
2. `getPassword()` : 사용자 비밀번호를 반환한다.
3. `getUsername()` : 사용자의 Username을 반환한다

그럼 이제 테스트를 진행해보자.

`SecurityConfig` 코드를 다시 살펴보면 `/admin` 페이지로의 접근에 대해 ADMIN 권한을 보유해야 접근이 가능한것으로 되어있다.

```

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception{
    // HttpSecurity 빈에 아래 설정 추가
    http
        .authorizeHttpRequests(auth -> auth // 람다 식으로 작성 해야함
            // 루트 디렉토리, login 디렉토리는 모두 허용(인증 필요x)
            .requestMatchers("/", "/login", "/loginProc", "/join", "/joinProc").permitAll()
            // admin 페이지는 인증된 사용자가 ADMIN 권한을 갖고 있어야함(인가)
            .requestMatchers("/admin").hasRole("ADMIN")
            // my질문 이후의 모든 경로에 대해 ADMIN 혹은 USER권한 보유시 접근 가능
            .requestMatchers("/my/**").hasAnyRole("ADMIN", "USER")
            // 위에서 설정하지 않은 모든 요청은 인증(Authenticated)을 거쳐야 접근 가능(로그인 필요)
            .anyRequest().authenticated()
        )
    // 로그인 설정 추가
}

```

정상적으로 동작하는지 확인하기 위해 ADMIN 권한을 보유한 새 유저를 생성하고 해당 유저로 admin 페이지로의 접근을 시도해보겠다.

`JoinService`의 `joinProcess()` 함수에서 `role`을 "ROLE_ADMIN"으로 수정한뒤, 회원 생성을 진행한다.

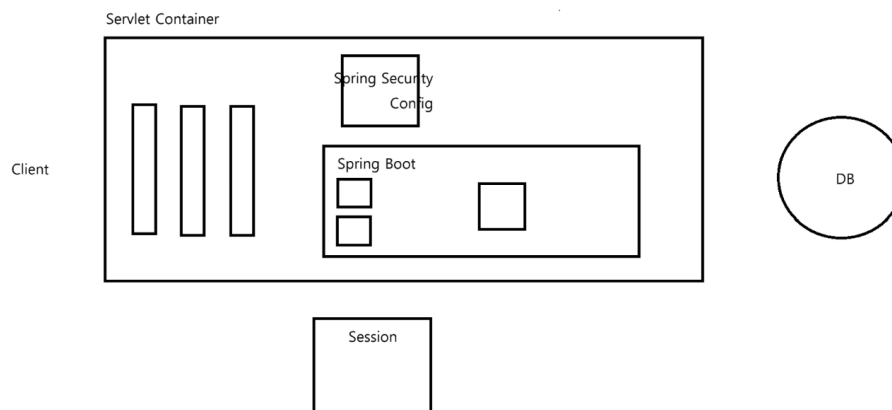
id	password	role	username
1	\$2a\$10\$JGWIUN.Hncdsbh7oBevWJO2NVtjdn...	ROLE_USER	test

이제 admin 페이지로 접근해보자. 그럼 당연하게도 사용자 권한을 확인하기 위해 `/login` 페이지로 리다이렉트 된다.

앞서 회원가입한 ADMIN권한을 가진 계정으로 로그인하여 접근이 되는지 확인해보면 성공적으로 접근되는것을 확인할 수 있다.

만약 로그인에 성공하였지만 권한이 달라 접근할 수 없다면 아래와 같이 white label이 보여지게 된다.

사용자 아이디 정보 보관 - 세션 방식



SecurityConfig에서 정의한 Spring Security Filter Chain에 의해 사용자의 로그인이 이루어지면, 스프링 시큐리티는 사용자 정보 (UserDetails)를 세션에 저장한다.

구체적으로 설명하면 아래와 같다.

1. 로그인 과정:

- 사용자가 로그인 폼을 제출하면, 스프링 시큐리티는 UsernamePasswordAuthenticationFilter를 사용하여 인증 요청을 처리한다.
- 이 필터는 사용자가 입력한 사용자 이름과 비밀번호를 AuthenticationManager를 통해 인증한다.
- AuthenticationManager는 여러 AuthenticationProvider를 사용하여 실제 인증을 수행하며, 이 과정에서 UserDetailsService를 통해 사용자 정보를 조회한다.

2. UserDetails 저장:

- 인증이 성공하면, 스프링 시큐리티는 인증된 사용자 정보를 나타내는 Authentication 객체를 생성한다. 이 객체는 UserDetails를 포함한다.
- 생성된 Authentication 객체는 SecurityContextHolder를 통해 SecurityContext에 저장된다. 기본적으로 SecurityContext는 세션에 저장되므로, 사용자 정보(UserDetails)도 세션에 저장된다.

3. 세션 관리:

- 세션에 저장된 SecurityContext는 사용자가 애플리케이션 내에서 이동할 때마다 인증 상태를 유지하는 데 사용된다. 이를 통해 사용자는 로그인 상태를 유지하고, 권한 검사를 받을 수 있게된다.

스프링 시큐리티를 통해 저장된 사용자 정보는 SecurityContextHolder를 통해서 얻을 수 있다.

아래와 같이 코드를 작성한 뒤 model에 데이터를 담고 html 페이지에서 값을 확인해보자.

MainController

```
package com.example.spring_security.controller;

import org.springframework.security.core.Authentication;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

import java.util.Collection;
import java.util.Iterator;

@Controller
public class MainController {

    @GetMapping
    public String mainPage(Model model) {

        // 사용자 이름(id) 확인
        String id = SecurityContextHolder.getContext().getAuthentication().getName();
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

        // 사용자 권한 확인
```

```

        Collection<? extends GrantedAuthority> authorities = authentication.getAuthorities();
        Iterator<? extends GrantedAuthority> iter = authorities.iterator();
        GrantedAuthority authority = iter.next();
        String role = authority.getAuthority();

        model.addAttribute("id", id);
        model.addAttribute("role", role);

        return "main";
    }
}

```

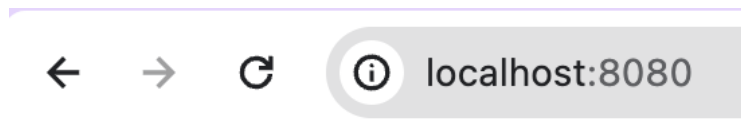
main.html

```

<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Hello</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
<p th:text="'안녕하세요. ' + ${id}" >안녕하세요. 손님</p>
<p th:text="'권한 : ' + ${role}" ></p>
</body>
</html>

```

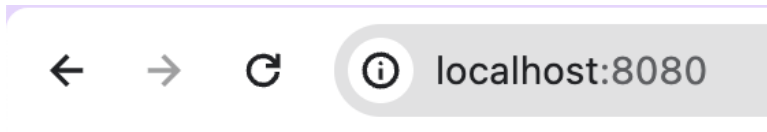
서버를 실행하고 메인페이지로 접속하면 아직 로그인을 하지 않았기 때문에 anonymousUser라고 출력된다.



안녕하세요. anonymousUser

권한 : ROLE_ANONYMOUS

이후 로그인을 하고 다시 메인페이지에 접속하게 되면 아래와 같이 현재 로그인한 사용자의 정보와 권한 정보를 확인할 수 있다.



안녕하세요. admin

권한 : ROLE_ADMIN

이후 새로고침을 아무리 진행하더라도 사용자의 정보는 SecurityContext에 담겨져있기 때문에 사용자 정보가 유지되는것을 확인 할 수 있다.

참고자료

[SpringBoot] Spring Security JWT 인증과 인가 - (1) 회원 가입

이번 포스팅에서는 Spring Boot에서 JWT를 이용한 인증과 인가를 다루기 이전, 사용자를 회원가입시키는 API 로직을 작성하고, 비밀번호를 해시처리하는 작업을 우선적으로 진행해보도록 하겠다. JWT 토큰 인증 방식에 대해서는 아래의 포스팅을 참고하면 좋을 것 같다. <https://sjh9708.tistory.com/46> [Web] 인증과 인가 - JWT 토큰 인증 앞 포스팅에 <https://sjh9708.tistory.com/83>

스프링 시큐리티

스프링 시큐리티 : Spring Security

<https://www.youtube.com/playlist?list=PLKjrxxiBSFCKD9TRKDYn7IE96K2u3C3U>



해당 강의 자료는 유튜브 개발자 유미님의 영상에 약간의 코드를 변형시켜 제작하였습니다.

JWT를 이해하기 위해 반드시 숙달해야 하는 내용이므로 다음주 세션까지 틈틈히 복습하시길 바랍니다. 복습하시면서 이해가 잘 안가는 부분이 있다면 언제든지 운영진들에게 질문해주세요!

최종 완성 코드

GitHub - HSU-Likelion-Backend-12th/spring-security: 9주차 세션 스프링 시큐리티 프레임워크 로그인,회원가입 예제

9주차 세션 스프링 시큐리티 프레임워크 로그인,회원가입 예제. Contribute to HSU-Likelion-Backend-12th/spring-security development by creating an account on GitHub.

<https://github.com/HSU-Likelion-Backend-12th/spring-security>

HSU-Likelion-Backend-12th/spring-security

9주차 세션 스프링 시큐리티 프레임워크 로그인,회원가입 예제

A 1 Contributor 0 Issues 0 Stars 0 Forks