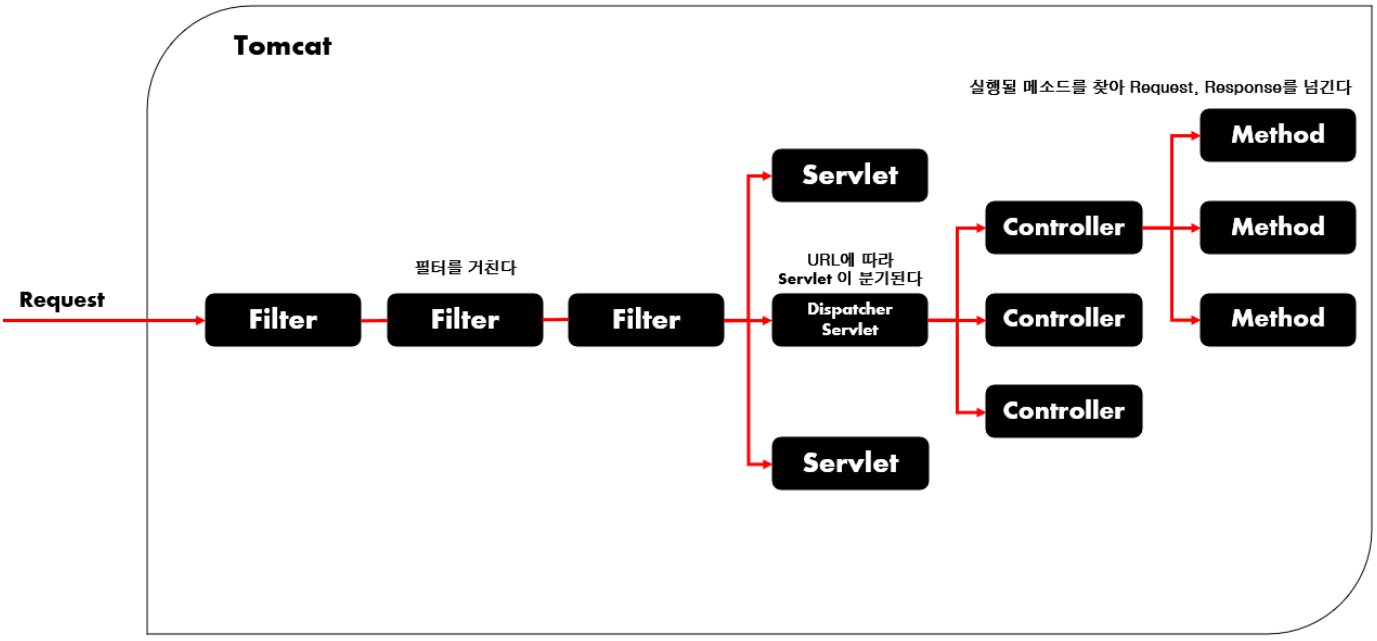
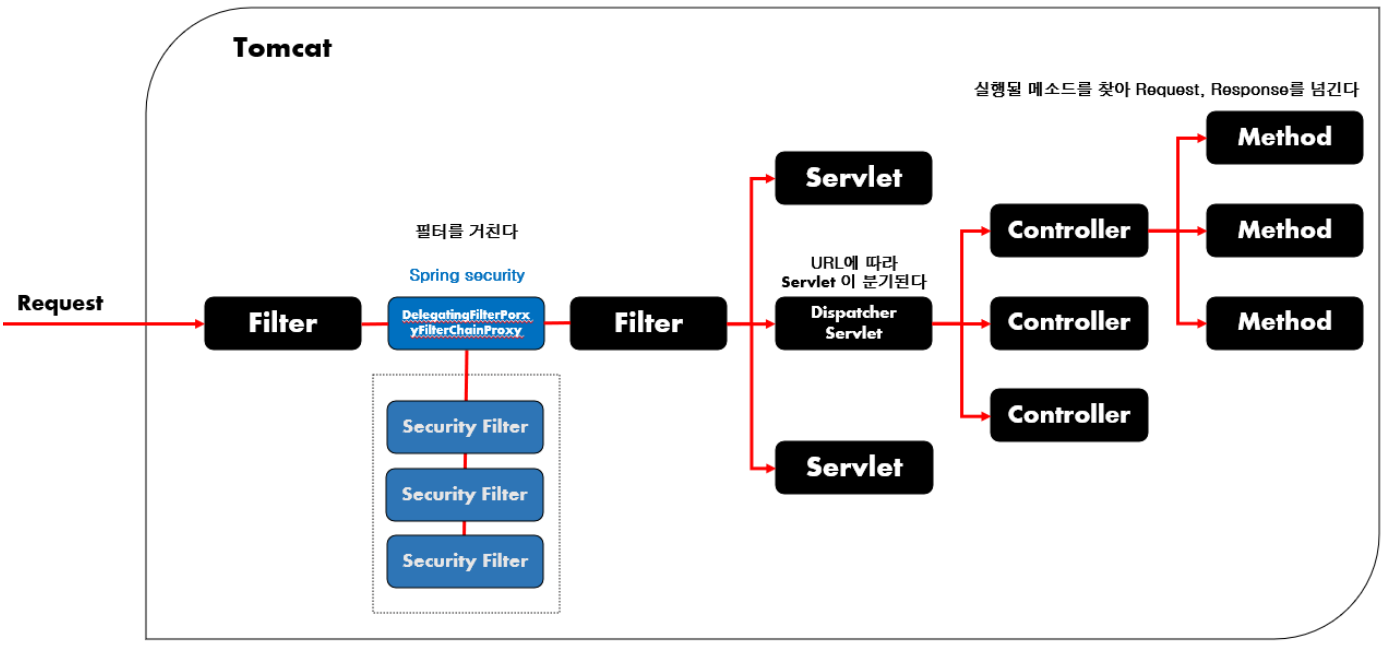
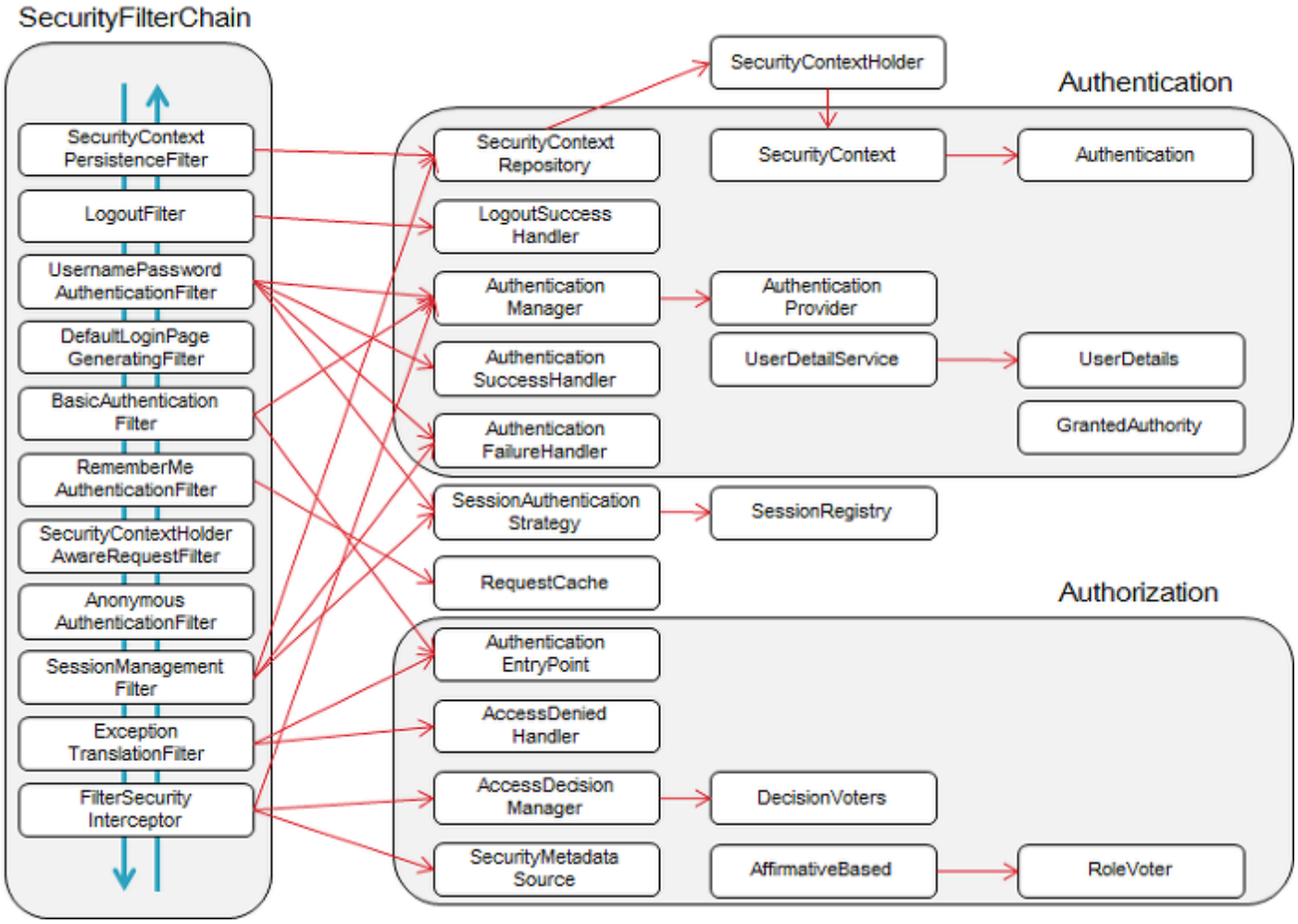
spring SECURITY

1. Authentication(인증) :
   특정 대상이 "누구"인지 확인하는 절차이다.
2. Authorization(권한부여,인가) :
   인증된 주체가 특정한 곳에 접근 권한을 확인하고 허용하는 과정이다.





Spring security는 여러 가지 필터로 이루어져 있고 필터들이 순서대로 동작하게 된다. 이러한 구조를 Filter Chain이라고 한다.
다음과 같이 필터의 종류는 정말 다양하다.



Spring Security는 Spring 기반의 어플리케이션 보안(인증과 권한, 인가 등)을 담당하는 스프링 하위 프레임워크이다. Security는 인증과 권한에 대한 부분을 Filter의 흐름에 따라 처리하고 있다.

- Spring Security의 securityConfig 작성
WebSecurityConfigurerAdapter가 Deprecated 되었으니 SecurityFilterChain를 Bean으로 등록해서 사용해야 하지만 두 가지 방법 모두 가능하며 구현 방법도 비슷하다.

authorizeRequest() (보안 절차를 거치고)
anyRequest() (어떠한 request라도)
authenticated() (인증을 받아야 함)
formLogin() (그 방식은 폼 로그인)

형식으로 설정 클래스가 처리된다. 최종적으로 SecurityFilterChain 타입으로 값을 빌드하여 리턴 해야 한다. 그 외에도 다양한 사용자 정의 보안 설정 적용이 가능합니다.

```
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
                http
                .authorizeRequests()
                .anyRequest().authenticated()
                .formLogin();

        return http.build();
}
```

기본적으로 사용되는 formLogin 인증 API 구조는 다음과 같다.

```
    http.formLogin()
      .login("/login")
      .defaultSuccessUrl("/home")
      .failureUrl("/login")
      .usernameParameter("username")
      .passwordParameter("password")
      .loginProcessingUrl("/login")
      .successHandler(loginSuccessHandler())
      .failereHandler(loginFailureHander())
```

login : 사용자가 정의한 로그인 페이지
defaultSuccessUrl: 로그인 성공 후 이동하는 페이지
failureUrl: 로그인 실패 후 이동하는 페이지
usernameParameter: 폼 태그에 사용되는 아이디 파라미터명
passwordParameter: 폼 태그에 사용되는 비밀번호 파라미터명
loginProcessingUrl: 폼 태그에 사용되는 url
successHandler: 로그인 성공 후 실행되는 handler
failereHandler: 로그인 실패 후 실행되는 handler

- 핸들러 작성 예

```
      .successHandler((request, response, authentication) -> {
                log.info("authentication name = {}", authentication.getName())
                response.sendRedirect("/");
            })
      .failureHandler((request, response, exception) -> {
                log.info("exception.getMessage() = {}", exception.getMessage());
                response.sendRedirect("/login");
            })
```

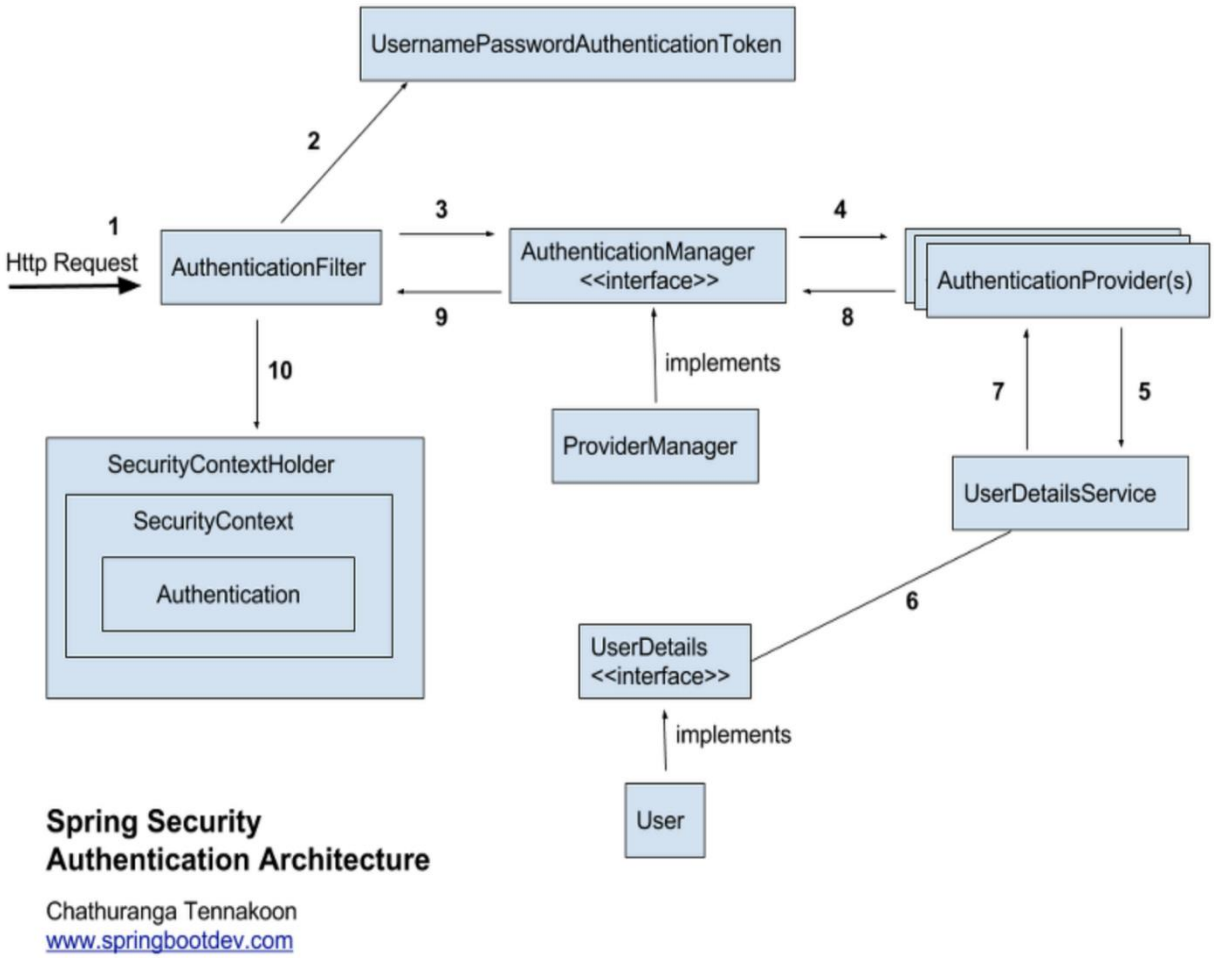
Spring Security는 dispatcherServlet 이전에 필터링하는 역할을 수행하므로, 여기서 지칭하는 핸들러는 Security에 내장된 혹은 사용자가 정의한 핸들러이다.

successHandler를 람다식으로 해당 폼 로그인을 성공하면 인증객체의 이름으로 로그를 찍고, "/"으로 리다이렉트하는 핸들러를 작성한 것이다. 만약 실패할 경우, 필터가 fauilerHandler를 찾아 해당 람다식 핸들러를 실행시킨다.

Login Form 인증 절차

폼 로그인은 다음과 같은 절차로 이루어진다.

2-1. 인증관련 architecture



1. Http Request가 서버로 넘어온다.
2. AuthenticationFilter가 요청을 낚아챈다. AuthenticationFilter에서 Request의 Username, password를 이용하여 UsernamePasswordAuthenticationToken을 생성한다.
3. 토큰을 AuthenticationManager가 받는다.

4. AuthenticationManager는 토큰을 AuthenticationProvider에게 토큰을 넘긴다.
5. AuthenticationProvider는 UserDetailsService로 토큰의 사용자 아이디(username)을 전달하여 DB에 존재하는지 확인한다.
6. UserDetailsService는 DB의 회원정보를 UserDetails 라는 객체로 반환한다.
7~10. AuthenticationProvider는 반환받은 UserDetails 객체와 실제 사용자의 입력정보를 비교한다. 비교가 완료되면 사용자 정보를 가진 Authentication 객체를 SecurityContextHolder에 담은 이후 AuthenticationSuccessHandle를 실행한다.
(실패시 AuthenticationFailureHandler를 실행한다.)

- Spring Security Filte의 종류

SecurityContextPersistenceFilter :
SecurityContextRepository에서 SecurityContext를 가져오거나 저장하는 역할을 한다.

LogoutFilter :
설정된 로그아웃 URL로 오는 요청을 감시하며, 해당 유저를 로그아웃 처리

(UsernamePassword)AuthenticationFilter :
(아이디와 비밀번호를 사용하는 form 기반 인증) 설정된 로그인 URL로 오는 요청을 감시하며, 유저 인증 처리

DefaultLoginPageGeneratingFilter :
인증을 위한 로그인폼 URL을 감시한다.

BasicAuthenticationFilter :
HTTP 기본 인증 헤더를 감시하여 처리한다.

RequestCacheAwareFilter :
로그인 성공 후, 원래 요청 정보를 재구성하기 위해 사용된다.

SecurityContextHolderAwareRequestFilter :
HttpServletRequestWrapper를 상속한 SecurityContextHolderAwareRequestWapper 클래스로 HttpServletRequest 정보를 감싼다. SecurityContextHolderAwareRequestWrapper 클래스는 필터 체인상의 다음 필터들에게 부가정보를 제공한다.

AnonymousAuthenticationFilter :
이 필터가 호출되는 시점까지 사용자 정보가 인증되지 않았다면 인증토큰에 사용자가 익명 사용자로 나타난다.

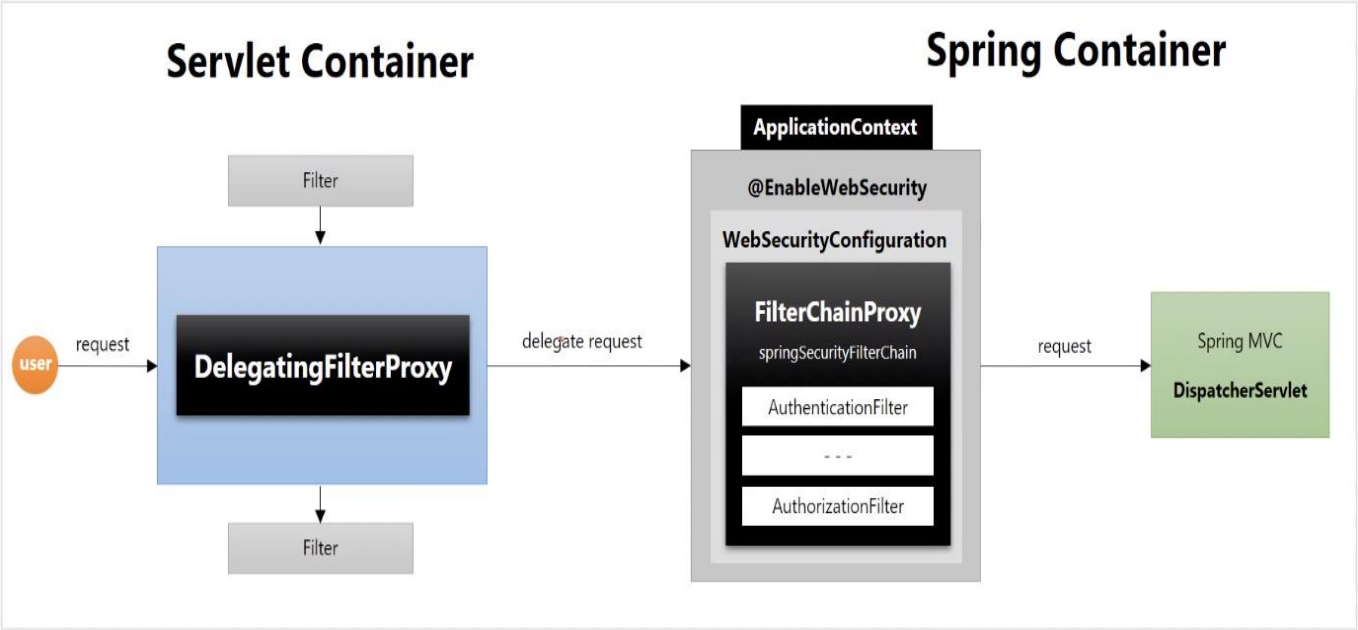
SessionManagementFilter :
인증된 사용자와 관련된 모든 세션을 추적한다.

ExceptionTranslationFilter :
보호된 요청을 처리하는 중에 발생할 수 있는 예외를 위임하거나 전달하는 역할을 한다.

FilterSecurityInterceptor :
AccessDecisionManager 로 권한부여 처리를 위임함으로써 접근 제어 결정을 쉽게 해준다.

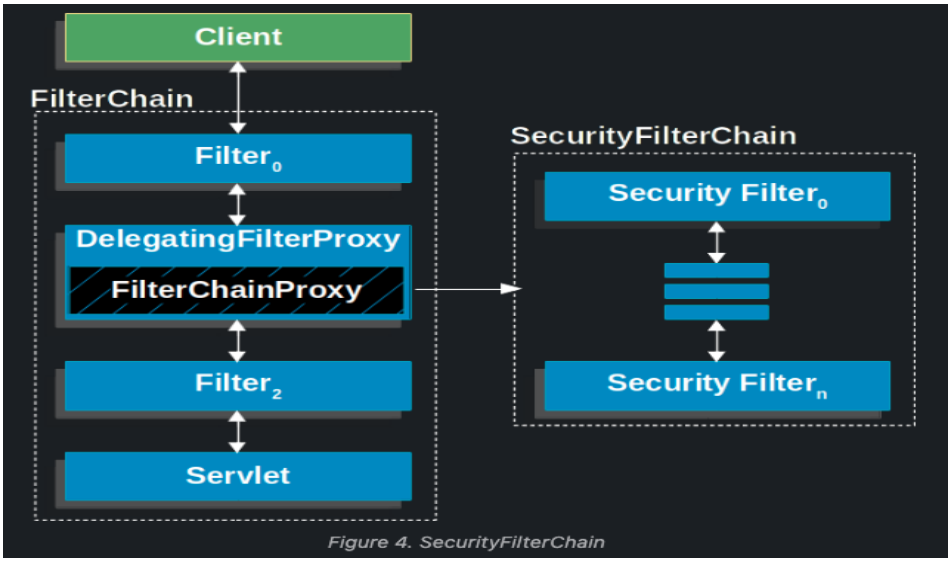
[SecurityFilterChain 다중 필터 관리]

SecurityFilterChain 란?



client가 request 요청을 보내면, Spring Container 이전에 Servlet Container에서 먼저 Filter에 의해 여러 가지 요청이 처리된다. SpringContainer와 Was 서버 간에 request 요청이 연결되어야 하는데, 이를 수행하는 Filter가 DelegatingFilterProxy 이다.

DelegatingFilterProxy는 SpringContainer에 존재하는 FilterChainProxy에게 해당 요청을 위임함으로써, request를 Spring Container에서 처리할 수 있다. 위임 전략을 사용하는 이유는, 스프링에서 제공하는 다양한 기술들을 사용할 수 있기에 다양한 장점이 있기 때문이다.



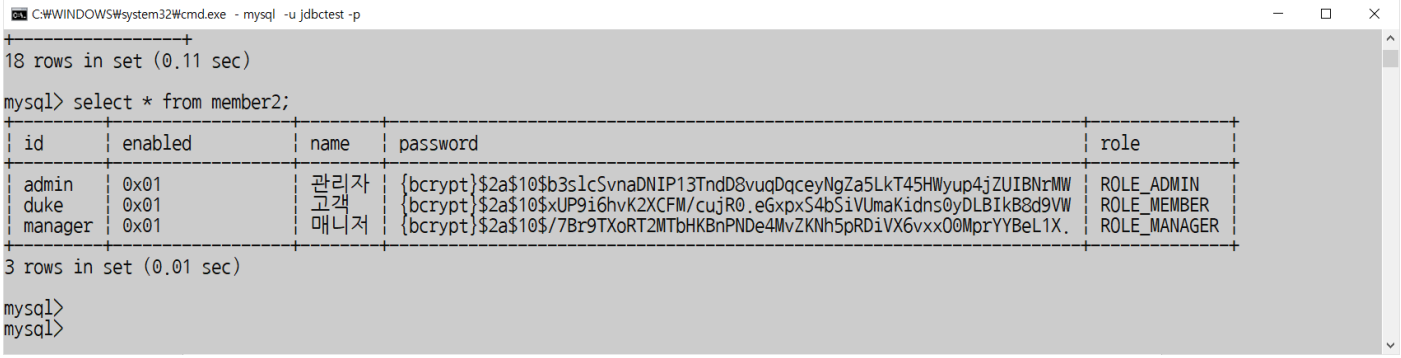
FilterChainProxy로 위임 받은 필터는 SecurityFilterChain이라는 구현체를 가지고 있고 그 안에 다양한 SecurityFilter가 존재한다. SecurityFilter_0 ~ SecurityFilter_n 까지 반복문이 돌면서, 해당 request에 매핑되는 antMatcher를 찾는다면, 필터는 해당 filter의 메소드에 작성된 체인들을 설정 요구사항을 확인하며 처리한다. 최종적으로 filter를 통과하면, DispatcherSerlvet으로 요청이 위임하여 성공 핸들러(사용자 정의 핸들러 혹은 controller)가 실행된다.

```
@Configuration // IoC 빈(bean)을 등록
@EnableWebSecurity // 필터 체인 관리 시작 어노테이션
@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true) // 특정 주소 접근시 권한 및 인증을 위한 어노테이션 활성화
public class SecurityConfig extends WebSecurityConfigurerAdapter{
    @Bean
    public BCryptPasswordEncoder encodePwd() {
        return new BCryptPasswordEncoder();
    }
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable();
        http.authorizeRequests()
                .antMatchers("/user/**").authenticated()
                //.antMatchers("/admin/**").access("hasRole('ROLE_ADMIN') or hasRole('ROLE_USER')")
                //.antMatchers("/admin/**").access("hasRole('ROLE_ADMIN') and hasRole('ROLE_USER')")
                .antMatchers("/admin/**").access("hasRole('ROLE_ADMIN')")
                .anyRequest().permitAll()
                .and()
                .formLogin()
                .loginPage("/login")
                .loginProcessingUrl("/loginProc")
                .defaultSuccessUrl("/");
    }
}
```

```
// UesrDetailService 구현 예
@Service
public class BoardUserDetailsService implements UserDetailsService {

	@Autowired
	private Member2Repository memberRepo;
	@Override
	public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
		Optional<Member2> optional = memberRepo.findById(username);
		if (!optional.isPresent()) {
			throw new UsernameNotFoundException(username + " 사용자 없음..");
		} else {
			Member2 member = optional.get();
			return new SecurityUser(member);
		}
	}
}

// 컨트롤러 구현 예
@Secured("ROLE_MANAGER")
	@GetMapping("/manager")
	public @ResponseBody String manager() {
		return "매니저 페이지입니다.";
	}

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

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

	@PostMapping("/joinProc")
	public String joinProc(User user) {
		System.out.println("회원가입 진행 : " + user);
		String rawPassword = user.getPassword();
		String encPassword = bCryptPasswordEncoder.encode(rawPassword);
		user.setPassword(encPassword);
		user.setRole("ROLE_USER");
		userRepository.save(user);
		return "redirect:/";
	}
```

// 테스트 클래스 사용자 정보(패스워드 암호화) DB 테이블에 저장하는 예

```
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
@DataJpaTest
public class DataCreationTest {
    @Autowired
    private Member2Repository mr;
    @Autowired
    private PasswordEncoder encoder;

    @Test
    @Rollback(false) // rollback이 기본임. DML 문 수행한 후에 rollback 하고싶지 않다면
    @Transactional
    void test() {
        Member2 m = new Member2();
        m.setId("duke");
        m.setPassword(encoder.encode("d123"));
        m.setName("고객");
        m.setRole(Role.ROLE_MEMBER);
        m.setEnabled(true);
        mr.save(m);
        m.setId("manager");
        m.setPassword(encoder.encode("m123"));
        m.setName("매니저");
        m.setRole(Role.ROLE_MANAGER);
        m.setEnabled(true);
        mr.save(m);
        m.setId("admin");
        m.setPassword(encoder.encode("a123"));
        m.setName("관리자");
        m.setRole(Role.ROLE_ADMIN);
        m.setEnabled(true);
        mr.save(m);
        List<Member2> list = mr.findAll();
        list.stream().forEach(System.out::println);
    }
}
```



JWT (Json Web Token)

Cookie / Session / Token 인증 방식 종류

보통 서버가 클라이언트 인증을 확인하는 방식은 대표적으로 쿠키, 세션, 토큰 3가지 방식이 있다.

JWT를 배우기 앞서 우선 쿠키와 세션의 통신 방식을 복습해보고 이들의 각각 특징과 장단점 그리고 왜 토큰 인증 방식을 사용하지는에 대해 간단하게 알아보자. (주입식 암기하지말고 과정을 이해하면 머릿속에 더 오래 남을 것이다!)

Cookie 인증

쿠키는 Key-Value 형식의 문자열 덩어리이다.

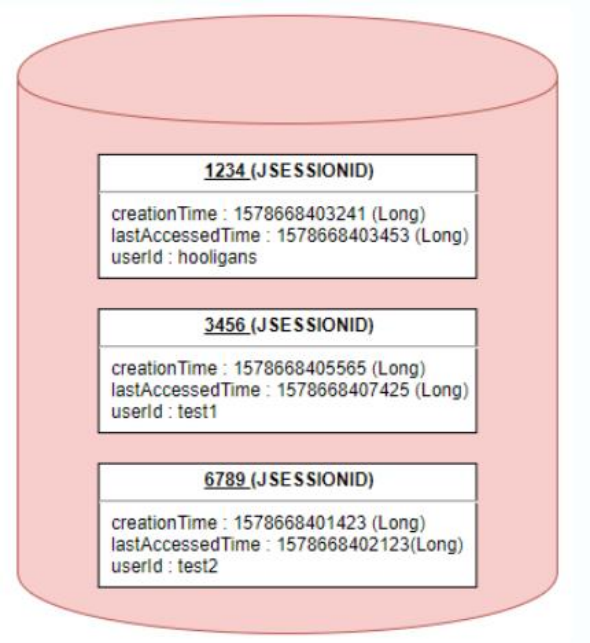
클라이언트가 어떠한 웹사이트를 방문할 경우, 그 사이트가 사용하고 있는 서버를 통해 클라이언트의 브라우저에 설치되는 작은 기록 정보 파일이다. 각 사용자마다의 브라우저에 정보를 저장하니 고유 정보 식별이 가능한 것이다.

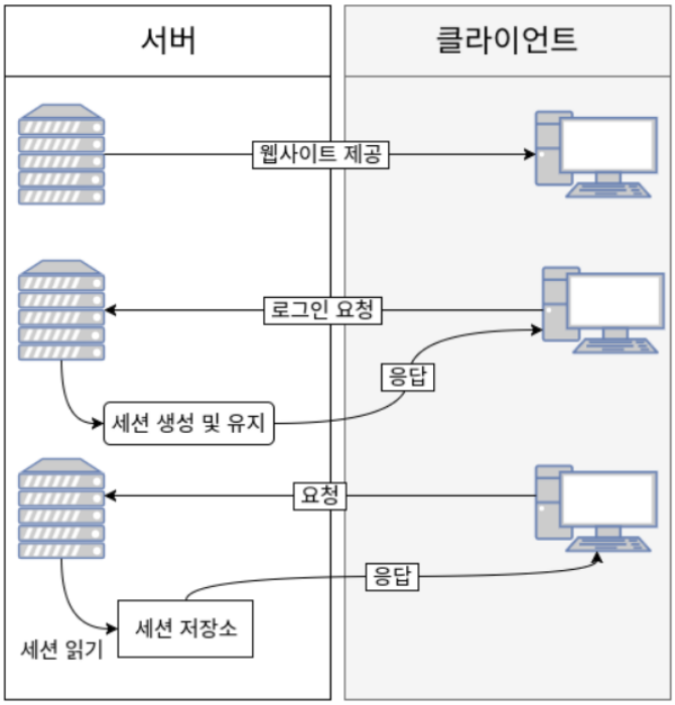
브라우저(클라이언트)가 서버에 요청(접속)을 보낸다.
서버는 클라이언트의 요청에 대한 응답을 작성할 때, 클라이언트 측에 저장하고 싶은 정보를 응답 헤더의 Set-Cookie에 담는다. 이후 해당 클라이언트는 요청을 보낼 때마다, 매번 저장된 쿠키를 요청 헤더의 Cookie에 담아 보낸다.
서버는 쿠키에 담긴 정보를 바탕으로 해당 요청의 클라이언트가 누군지 식별하거나 정보를 바탕으로 추천 광고를 띄우거나 한다.

Cookie 방식의 단점
가장 큰 단점은 보안에 취약하다는 점이다.
요청 시 쿠키의 값을 그대로 보내기 때문에 유출 및 조작 당할 위험이 존재한다.
쿠키에는 용량 제한이 있어 많은 정보를 담을 수 없다.
웹 브라우저마다 쿠키에 대한 지원 형태가 다르기 때문에 브라우저간 공유가 불가능하다.
쿠키의 사이즈가 커질수록 네트워크에 부하가 심해진다.

Session 인증
이러한 쿠키의 보안적인 이슈 때문에, 세션은 비밀번호 등 클라이언트의 민감한 인증 정보를 브라우저가 아닌 서버 측에 저장하고 관리한다. 서버의 메모리에 저장하기도 하고, 서버의 로컬 파일이나 데이터베이스에 저장하기도 한다.

핵심 골자는 민감한 정보는 클라이언트에 보내지 않고 서버에서 모두 관리한다는 점이다.
세션 객체는 Key에 해당하는 SESSION ID와 이에 대응하는 Value로 구성되어 있다.
Value에는 세션 생성 시간, 마지막 접근 시간 및 User가 저장한 속성 등 이 Map 형태로 저장된다.





(1) 유저가 웹사이트에서 로그인하면 세션이 서버 메모리(혹은 데이터베이스) 상에 저장된다.
이때, 세션을 식별하기 위한 Session Id를 기준으로 정보를 저장한다.
(2) 서버에서 브라우저에 쿠키에다가 Session Id를 저장한다.
(3) 쿠키에 정보가 담겨있기 때문에 브라우저는 해당 사이트에 대한 모든 Request에 Session Id를 쿠키에 담아 전송한다.
(4) 서버는 클라이언트가 보낸 Session Id 와 서버 메모리로 관리하고 있는 Session Id를 비교하여 인증을 수행한다.

[Session 방식의 단점]

쿠키를 포함한 요청이 외부에 노출되더라도 세션 ID 자체는 유의미한 개인정보를 담고 있지 않는다.

그러나 해커가 세션 ID 자체를 탈취하여 클라이언트인척 위장할 수 있다는 한계가 존재한다. (이는 서버에서 IP특정을 통해 해결 할 수 있긴 하다)
서버에서 세션 저장소를 사용하므로 요청이 많아지면 서버에 부하가 심해진다.

Token 인증

토큰 기반 인증 시스템은 클라이언트가 서버에 접속을 하면 서버에서 해당 클라이언트에게 인증되었다는 의미로 '토큰'을 부여한다. 이 토큰은 유일하며 토큰을 발급받은 클라이언트는 또 다시 서버에 요청을 보낼 때 요청 헤더에 토큰을 심어서 보낸다. 그러면 서버에서는 클라이언트로부터 받은 토큰을 서버에서 제공한 토큰과의 일치 여부를 체크하여 인증 과정을 처리하게 된다.

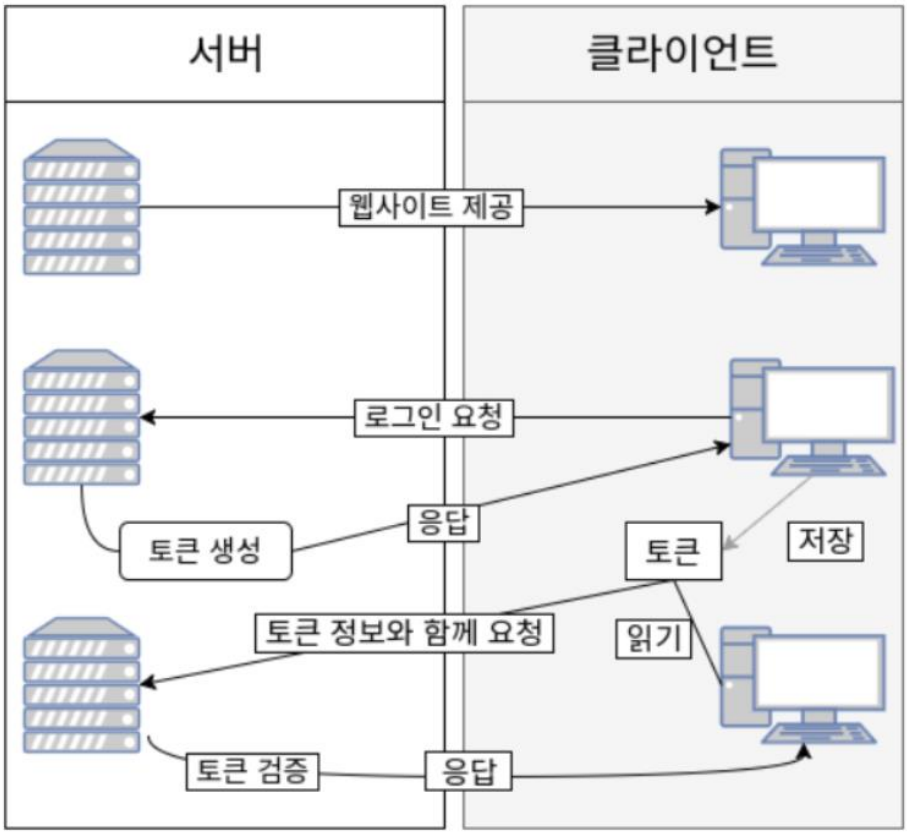
기존의 세션기반 인증은 서버가 파일이나 데이터베이스에 세션정보를 가지고 있어야 하고 이를 조회하는 과정이 필요하기 때문에 많은 오버헤드가 발생한다. 하지만 토큰은 세션과는 달리 서버가 아닌 클라이언트에 저장되기 때문에 메모리나 스토리지 등을 통해 세션을 관리했던 서버의 부담을 덜 수 있다. 토큰 자체에 데이터가 들어있기 때문에 클라이언트에서 받아 위조되었는지 판별만 하면 되기 때문이다. 토큰은 앱과 서버가 통신 및 인증할 때 가장 많이 사용된다. 왜냐하면 웹에는 쿠키와 세션이 있지만 앱에서는 없기 때문이다.

서버(세션) 기반 인증 시스템

서버의 세션을 사용해 사용자 인증을 하는 방법으로 서버측(서버 램 or 데이터베이스)에서 사용자의 인증 정보를 관리하는 것을 의미한다.
그러다 보니, 클라이언트로부터 요청을 받으면 클라이언트의 상태를 계속에서 유지해놓고 사용한다.
(Stateful) 이는 사용자가 증가함에 따라 성능의 문제를 일으킬 수 있으며 확장성이 어렵다는 단점을 지닌다.

토큰 기반 인증 시스템

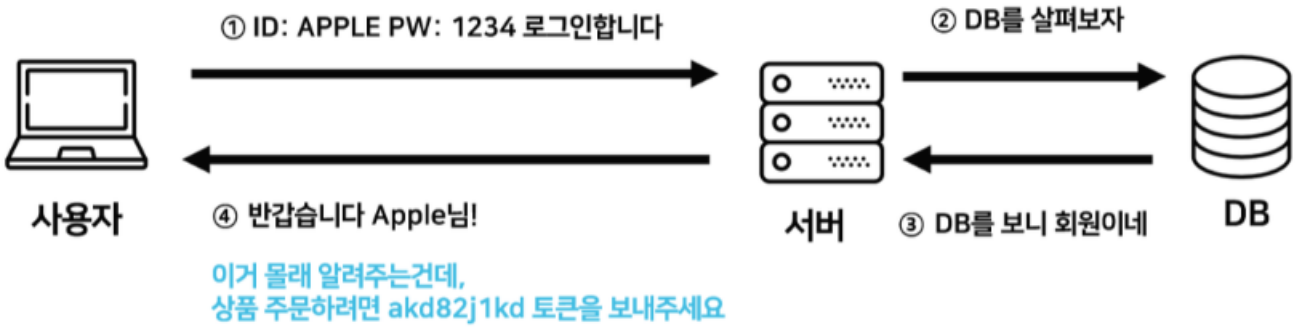
이러한 단점을 극복하기 위해서 "토큰 기반 인증 시스템"이 나타났다.
인증받은 사용자에게 토큰을 발급하고, 로그인이 필요한 작업일 경우 헤더에 토큰을 함께 보내 인증받은 사용자인지 확인한다.
이는 서버 기반 인증 시스템과 달리 상태를 유지하지 않으므로 Stateless 한 특징을 가지고 있다.



(1) 사용자가 아이디와 비밀번호로 로그인을 한다.
(2) 서버 측에서 사용자(클라이언트)에게 유일한 토큰을 발급한다.
(3) 클라이언트는 서버 측에서 전달받은 토큰을 쿠키나 스토리지에 저장해 두고, 서버에 요청을 할 때마다 해당 토큰을 HTTP 요청 헤더에 포함시켜 전달한다.
(4) 서버는 전달받은 토큰을 검증하고 요청에 응답한다.
(5) 토큰에는 요청한 사람의 정보가 담겨있으므로 서버는 DB를 조회하지 않고 누가 요청하는지 알 수 있다.

[Token 방식의 단점]
- 쿠키/세션과 다르게 토큰 자체의 데이터 길이가 길어, 인증 요청이 많아질수록 네트워크 부하가 심해질 수 있다.
- Payload 자체는 암호화되지 않기 때문에 유저의 중요한 정보는 담을 수 없다.
- 토큰을 탈취당하면 대처하기 어렵다. (따라서 사용 기간 제한을 설정하는 식으로 극복한다)

JWT (JSON Web Token) 이란



JWT(JSON Web Token)란 인증에 필요한 정보들을 암호화시킨 JSON 토큰을 의미한다. 그리고 JWT 기반 인증은 JWT 토큰(Access Token)을 HTTP 헤더에 실어 서버가 클라이언트를 식별하는 방식이다

JWT는 JSON 데이터를 Base64 URL-safe Encode 를 통해 인코딩하여 직렬화한 것이며, 토큰 내부에는 위변조 방지를 위해 개인키를 통한 전자서명도 들어있다. 따라서 사용자가 JWT 를 서버로 전송하면 서버는 서명을 검증하는 과정을 거치게 되며 검증이 완료되면 요청한 응답을 돌려준다.

Header 에는 JWT 에서 사용할 타입과 해시 알고리즘의 종류가 담겨있으며, Payload 는 서버에서 첨부한 사용자 권한 정보와 데이터가 담겨있다. 마지막으로 Signature 에는 Header, Payload 를 Base64 URL-safe Encode 를 한 이후 Header 에 명시된 해시함수를 적용하고, 개인키(Private Key)로 서명한 전자서명이 담겨있다.

실제 디코딩된 JWT는 다음과 같은 구조를 지닌다.

[Header]

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

alg : 서명 암호화 알고리즘(ex: HMAC SHA256, RSA)
typ : 토큰 유형

[Payload]
토큰에서 사용할 정보의 조각들인 Claim 이 담겨있다. (실제 JWT 를 통해서 알 수 있는 데이터, 정보의 한 조각)서버와 클라이언트가 주고받는 시스템에서 실제로 사용될 정보에 대한 내용을 담고 있는 섹션이다. key-value 형식으로 이루어진 한 쌍의 정보를 Claim이라고 칭한다.

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

페이로드는 정해진 데이터 타입은 없지만, 대표적으로 Registered claims, Public claims, Private claims 이렇게 세 가지로 나뉜다.

```
{
    "jti": "1000",                      // Registered Claim
    "exp": "1521430000000",             // Registered Claim
    "https://kevin.tistory.com": true, // Public Claim
    "username": "kevin"                 // Private Claim
}
```

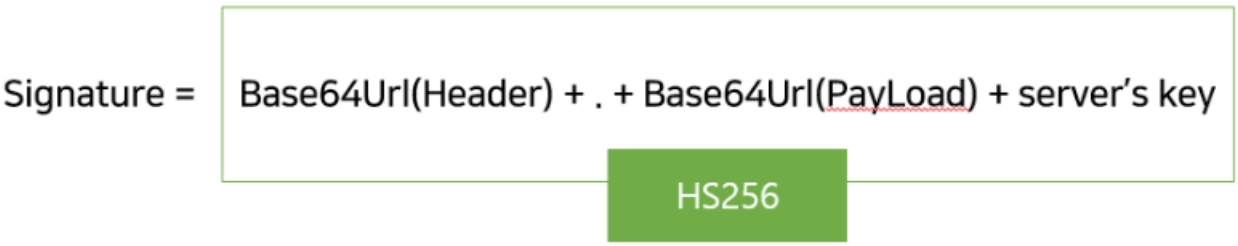
Registed claims : 미리 정의된 클레임.
iss(issuer; 발행자),
exp(expireation time; 만료 시간),
sub(subject; 제목),
iat(issued At; 발행 시간),
jti(JWI ID)
Public claims : 사용자가 정의할 수 있는 클레임 공개용 정보 전달을 위해 사용.
Private claims : 해당하는 당사자들 간에 정보를 공유하기 위해 만들어진 사용자 지정 클레임.
외부에 공개되도 상관없지만 해당 유저를 특정할 수 있는 정보들을 담는다

[Signature]

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

시그니처에서 사용하는 알고리즘은 헤더에서 정의한 알고리즘 방식(alg)을 활용한다.

시그니처의 구조는 (헤더 + 페이로드)와 서버가 갖고 있는 유일한 key 값을 합친 것을 헤더에서 정의한 알고리즘으로 암호화를 한다.



Header와 Payload는 단순히 인코딩된 값이기 때문에 3자가 복호화 및 조작할 수 있지만,
Signature는 서버 측에서 관리하는 비밀키가 유출되지 않는 이상 복호화할 수 없다.
따라서 Signature는 토큰의 위변조 여부를 확인하는데 사용된다.

토큰은 아래처럼 요청 헤더의 Authorization 필드에 담아져 보내진다.

Authorization: <type> <credentials>

[다양한 인증 타입(알고리즘)]
토큰에는 많은 종류가 있고, 서버는 다양한 종류의 토큰을 처리하기 위해 전송받은 type에 따라 토큰을 다르게 처리한다. type에는 아래와 같은 타입들이 존재한다.

- Basic
사용자 아이디와 암호를 Base64로 인코딩한 값을 토큰으로 사용한다. (RFC 7617)

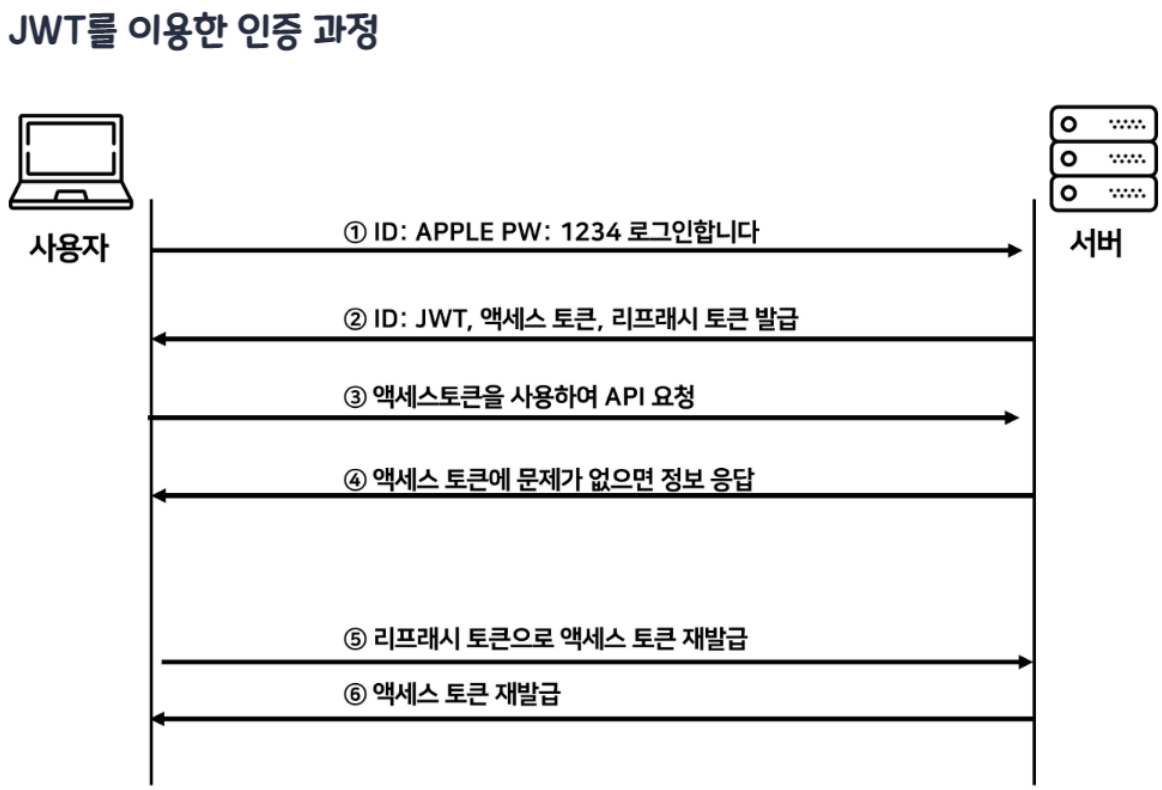
- Bearer
JWT 혹은 OAuth에 대한 토큰을 사용한다. (RFC 6750)

- Digest
서버에서 난수 데이터 문자열을 클라이언트에 보낸다. 클라이언트는 사용자 정보와 nonce를 포함하는 해시값을 사용하여 응답한다 (RFC 7616)

- HOBA
전자 서명 기반 인증 (RFC 7486)

- Mutual
암호를 이용한 클라이언트-서버 상호 인증 (draft-ietf-httpauth-mutual)

- AWS4-HMAC-SHA256
AWS 전자 서명 기반 인증



(1) 사용자가 ID, PW를 입력하여 서버에 로그인 인증을 요청한다.
(2) 서버에서 클라이언트로부터 인증 요청을 받으면, Header, PayLoad, Signature를 정의한다.
Hedaer, PayLoad, Signature를 각각 Base64로 한 번 더 암호화하여 JWT를 생성하고
이를 쿠키에 담아(응답 헤더) 클라이언트에게 발급한다.
클라이언트는 서버로부터 받은 JWT를 로컬 스토리지에 저장한다.
(쿠키나 다른 곳에 저장할 수도 있음)
(3) API를 서버에 요청할 때 Authorization header에 Access Token을 담아서 보낸다.
서버가 할 일은 클라이언트가 Header에 담아서 보낸 JWT가 내 서버에서 발행한 토큰인지 일치

여부를 확인하여 일치한다면 인증을 통과시켜주고 아니라면 통과시키지 않으면 된다.

(4) 인증이 통과되었으므로 페이로드에 들어있는 유저의 정보들을 select해서 클라이언트에 응답한다.

(5) 클라이언트가 서버에 요청을 했는데, 만일 액세스 토큰의 시간이 만료되면 클라이언트는 리프래시 토큰을 이용해서 서버로부터 새로운 엑세스 토큰을 발급 받는다.

[토큰 인증이 신뢰성을 가지는 이유]

유저 JWT: A(Header) + B(Payload) + C(Signature) 일 때 (만일 임의의 유저가 B를 수정했다고 하면 B'로 표시한다.)

다른 유저가 B를 임의로 수정 -> 유저 JWT: A + B' + C
수정한 토큰을 서버에 요청을 보내면 서버는 유효성 검사 시행
유저 JWT: A + B' + C
서버에서 검증 후 생성한 JWT: A + B' + C' => (signature) 불일치

대조 결과가 일치하지 않아 유저의 정보가 임의로 조작되었음을 알 수 있다.
서버는 토큰 안에 들어있는 정보가 무엇인지 아는 게 중요한 것이 아니라 해당 토큰이 유효한 토큰인지 확인하는 것이 중요하기 때문에, 클라이언트로부터 받은 JWT의 헤더, 페이로드를 서버의 key값을 이용해 시그니처를 다시 만들고 이를 비교하며 일치했을 경우 인증을 통과시킨다.

💡 JWT은 서명(인증)이 목적이다.
JWT는 Base64로 암호화를 하기 때문에 디버거를 사용해서 인코딩된 JWT를 1초만에 복호화할 수 있다. 복호화 하면 사용자의 데이터를 담은 Payload 부분이 그대로 노출되어 버린다. 그래서 페이로드에는 비밀번호와 같은 민감한 정보는 넣지 말아야 한다. 토큰의 진짜 목적은 정보 보호가 아닌, 위조 방지이다. 시그니처에 사용된 비밀키가 노출되지 않는 이상 데이터를 위조해도 시그니처 부분에서 바로 걸러지기 때문이다.

[JWT 장점]

- Header와 Payload를 가지고 Signature를 생성하므로 데이터 위변조를 막을 수 있다.
- 인증 정보에 대한 별도의 저장소가 필요없다.
- JWT는 토큰에 대한 기본 정보와 전달할 정보 및 토큰이 검증됐음을 증명하는 서명 등 필요한 모든 정보를 자체적으로 지니고 있다.
- 클라이언트 인증 정보를 저장하는 세션과 다르게, 서버는 무상태(StateLess)가 되어 서버 확장성이 우수해질 수 있다.
- 토큰 기반으로 다른 로그인 시스템에 접근 및 권한 공유가 가능하다. (쿠키와 차이)

- OAuth(Open Authorization)의 경우 Facebook, Google 등 소셜 계정을 이용하여 다른 웹서비스에서도 로그인을 할 수 있다. 모바일 어플리케이션 환경에서도 잘 동작한다. (모바일은 세션 사용 불가능)

- 참고
서버에서 가장 피해야 할 것은 데이터베이스 조회이다.
서버 자체가 죽는 경우도 있지만, 대부분 DB가 터져서 서버도 같이 죽는 경우가 허다하기 때문이다.
이런 점에서, JWT 토큰은 DB조회를 안해도 되는 장점을 가지고 있다는 점이다.
만일 payload에 유저이름과 유저등급 을 같이 두고 보내면, 서버에서는 유저이름을 가지고 DB를 조회해서 유저 등급을 얻지 않아도 바로 원하는 정보를 취할 수 있다.

[JWT 단점]
- Self-contained : 토큰 자체에 정보를 담고 있으므로 양날의 검이 될 수 있다.
- 토큰 길이 : 토큰의 Payload에 3종류의 클레임을 저장하기 때문에, 정보가 많아질수록 토큰의 길이가 늘어나 네트워크에 부하를 줄 수 있다.
- Payload 인코딩 : payload 자체는 암호화 된 것이 아니라 BASE64로 인코딩 된 것이기 때문에, 중간에 Payload를 탈취하여 디코딩하면 데이터를 볼 수 있으므로, payload에 중요 데이터를 넣지 않아야 한다.
- Store Token : stateless 특징을 가지기 때문에, 토큰은 클라이언트 측에서 관리하고 저장한다. 때문에 토큰 자체를 탈취당하면 대처하기가 어렵게 된다.

[JWT의 Access Token / Refresh Token]

JWT도 제 3자에게 토큰 탈취의 위험성이 있기 때문에, 그대로 사용하는 것이 아닌 Access Token, Refresh Token 으로 이중으로 나누어 인증을 하는 방식을 현업에선 취한다.

Access Token 과 Refresh Token은 둘다 똑같은 JWT이다. 다만 토큰이 어디에 저장되고 관리되느냐에 따른 사용 차이일 뿐이다.

Access Token : 클라이언트가 갖고 있는 실제로 유저의 정보가 담긴 토큰으로, 클라이언트에서 요청이 오면 서버에서 해당 토큰에 있는 정보를 활용하여 사용자 정보에 맞게 응답을 진행
Refresh Token: 새로운 Access Token을 발급해주기 위해 사용하는 토큰으로 짧은 수명을 가지는 Access Token에게 새로운 토큰을 발급해주기 위해 사용. 해당 토큰은 보통 데이터베이스에 유저 정보와 같이 기록.

Access Token은 접근에 관여하는 토큰, Refresh Token은 재발급에 관여하는 토큰의 역할로 사용되

는 JWT 이라고 말할 수 있다.

[JWT 인코딩 / 디코딩 해보기]
직접 JWT 토큰을 생성해 연습해보고 싶다면, 아래 공식사이트에서 쉽게 JWT 토큰을 인코딩(생성)하거나 디코딩 할 수 있다.

https://jwt.io/

[JWT를 통한 인증절차]

- 사용자가 로그인을 한다.
- 서버에서는 계정 정보를 읽어 사용자를 확인 후, 사용자의 고유 ID 값을 부여한 후 기타 정보와 함께 Payload 에 집어넣는다.
- JWT 토큰의 유효기간을 설정한다.
  암호화할 Secret key 를 이용해 Access Token 을 발급한다. 클라이언트는 Access Token 을 받아 저장 후, 인증이 필요한 요청마다 토큰을 헤더에 실어 보낸다.
  서버에서는 해당 토큰의 Verify Signature 를 Secret key 로 복호화한 후, 조작 여부, 유효기간을 확인한다. 검증이 완료되었을 경우, Payload 를 디코딩 하여 사용자의 ID 에 맞는 데이터를 가져온다. 보통 Access Token의 유효기간은 매우 짧다. 이유는 보안 문제 때문이다. 그래서 Refresh Token을 따로 발급해주는데, Access Token이 만료되면 새로운 JWT를 발급할 수 있는 토큰이다.

[Refresh Token]

Refresh Token이란 Access Token을 재발급 받기위한 Token이다.

1. 클라이언트에서 로그인한다.
2. 서버는 클라이언트에게 Access Token과 Refresh Token을 발급한다. 동시에 Refresh Token은 서버에 저장된다. 클라이언트는 local 저장소에 두 Token을 저장한다.
3. 매 요청마다 Access Token을 헤더에 담아서 요청한다.
4. 이 때, Access Token이 만료가 되면 서버는 만료되었다는 Response를 하게 된다.
5. 클라이언트는 해당 Response를 받으면 Refresh Token을 보낸다.
6. 서버는 Refresh Token 유효성 체크를 하게 되고, 새로운 Access Token을 발급한다.
7. 클라이언트는 새롭게 받은 Access Token을 기존의 Access Token에 덮어쓰게 된다.

예)
Access Token 만료기간 : 30분 ~ 1시간
Refresh Token 만료기간 : 3일 ~ 1달

Spring Security 에서의 JWT 구현

```
@Configuration
@EnableWebSecurity // 시큐리티 활성화 -> 기본 스프링 필터체인에 등록
public class SecurityConfig {
    @Autowired
    private UserRepository userRepository;
    @Autowired
    private CorsConfig corsConfig;
    @Bean
    SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
                .csrf().disable()
                .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .and()
                .formLogin().disable()
                .httpBasic().disable()
                .apply(new MyCustomDsl()) // 커스텀 필터 등록
                .and()
                .authorizeRequests(authroize -> authroize.antMatchers("/api/v1/user/**")
                .access("hasRole('ROLE_USER') or hasRole('ROLE_MANAGER') or hasRole('ROLE_ADMIN')")
                .antMatchers("/api/v1/manager/**")
                .access("hasRole('ROLE_MANAGER') or hasRole('ROLE_ADMIN')")
                .antMatchers("/api/v1/admin/**")
                .access("hasRole('ROLE_ADMIN')")
                .anyRequest().permitAll())
                .build();
    }

    public class MyCustomDsl extends AbstractHttpConfigurer<MyCustomDsl, HttpSecurity> {
        @Override
        public void configure(HttpSecurity http) throws Exception {
            AuthenticationManager authenticationManager =
                        http.getSharedObject(AuthenticationManager.class);
            http.addFilter(corsConfig.corsFilter())
                .addFilter(new JwtAuthenticationFilter(authenticationManager))
                .addFilter(new JwtAuthorizationFilter(authenticationManager, userRepository));
        }
    }
}
```

```
public interface JwtProperties {
	String SECRET = "둘리"; // 우리 서버만 알고 있는 비밀값(^^)
	int EXPIRATION_TIME = 864000000; // 10일 (1/1000초)
	String TOKEN_PREFIX = "Bearer ";
	String HEADER_STRING = "Authorization";
}


import lombok.RequiredArgsConstructor;

@RequiredArgsConstructor
public class JwtAuthenticationFilter extends UsernamePasswordAuthenticationFilter{

    private final AuthenticationManager authenticationManager;

    // Authentication 객체 만들어서 리턴 => 의존 : AuthenticationManager
    // 인증 요청시에 실행되는 함수 => /login
    @Override
    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
                        throws AuthenticationException {

        System.out.println("JwtAuthenticationFilter : 진입");

        // request에 있는 username과 password를 파싱해서 자바 Object로 받기
        ObjectMapper om = new ObjectMapper();
        LoginRequestDto loginRequestDto = null;
        try {
            loginRequestDto = om.readValue(request.getInputStream(), LoginRequestDto.class);
        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("JwtAuthenticationFilter : "+loginRequestDto);

        // 유저네임패스워드 토큰 생성
        UsernamePasswordAuthenticationToken authenticationToken =
                        new UsernamePasswordAuthenticationToken(
                                            loginRequestDto.getUsername(),
                                            loginRequestDto.getPassword());

        System.out.println("JwtAuthenticationFilter : 토큰생성완료");

        // authenticate() 함수가 호출 되면 인증 프로바이더가 유저 디테일 서비스의
        // loadUserByUsername(토큰의 첫번째 파라메터) 를 호출하고
        // UserDetails를 리턴받아서 토큰의 두번째 파라메터(credential)과
```

```
        // UserDetails(DB값)의 getPassword()함수로 비교해서 동일하면
        // Authentication 객체를 만들어서 필터체인으로 리턴해준다.

        // Tip: 인증 프로바이더의 디폴트 서비스는 UserDetailsService 타입
        // Tip: 인증 프로바이더의 디폴트 암호화 방식은 BCryptPasswordEncoder
        // 결론은 인증 프로바이더에게 알려줄 필요가 없음.
        Authentication authentication =
                        authenticationManager.authenticate(authenticationToken);

        PrincipalDetails principalDetailis = (PrincipalDetails) authentication.getPrincipal();
        System.out.println("Authentication : "+principalDetailis.getUser().getUsername());
        return authentication;
    }


    // JWT Token 생성해서 response에 담아주기
    @Override
    protected void successfulAuthentication(HttpServletRequest request,
                                            HttpServletResponse response, FilterChain chain,
                                            Authentication authResult) throws IOException, ServletException {

        PrincipalDetails principalDetailis = (PrincipalDetails) authResult.getPrincipal();

        String jwtToken = JWT.create()
                .withSubject(principalDetailis.getUsername())
                .withExpiresAt(new Date(System.currentTimeMillis()+JwtProperties.EXPIRATION_TIME))
                .withClaim("id", principalDetailis.getUser().getId())
                .withClaim("username", principalDetailis.getUser().getUsername())
                .sign(Algorithm.HMAC512(JwtProperties.SECRET));

        response.addHeader(JwtProperties.HEADER_STRING, JwtProperties.TOKEN_PREFIX+jwtToken);
    }
}


// 인가
public class JwtAuthorizationFilter extends BasicAuthenticationFilter {
    private UserRepository userRepository;
    public JwtAuthorizationFilter(AuthenticationManager authenticationManager,
                                  UserRepository userRepository) {
        super(authenticationManager);
        this.userRepository = userRepository;
    }
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
```

```
                                FilterChain chain) throws IOException, ServletException {
        String header = request.getHeader(JwtProperties.HEADER_STRING);
        if (header == null || !header.startsWith(JwtProperties.TOKEN_PREFIX)) {
            chain.doFilter(request, response);
            return;
        }
        System.out.println("header : " + header);
        String token = request.getHeader(JwtProperties.HEADER_STRING)
                                                .replace(JwtProperties.TOKEN_PREFIX, "");

        // 토큰 검증 (이게 인증이기 때문에 AuthenticationManager도 필요 없음)
        // 내가 SecurityContext에 집적접근해서 세션을 만들때 자동으로 UserDetailsService에 있는
        // loadByUsername이 호출됨.
        String username = JWT.require(Algorithm.HMAC512(JwtProperties.SECRET)).build().verify(token)
                                .getClaim("username").asString();

        if (username != null) {
            Users user = userRepository.findByUsername(username);
            // 인증은 토큰 검증시 끝. 인증을 하기 위해서가 아닌 스프링 시큐리티가 수행해주는 권한 처리를 위해
            // 아래와 같이 토큰을 만들어서 Authentication 객체를 강제로 만들고 그걸 세션에 저장!
            PrincipalDetails principalDetails = new PrincipalDetails(user);
            Authentication authentication = new UsernamePasswordAuthenticationToken(
                        principalDetails, // 나중에 컨트롤러에서 DI해서 쓸 때 사용하기 편함.
                        null, // 패스워드는 모르니까 null 처리, 어차피 지금 인증하는게 아니니까!!
                        principalDetails.getAuthorities());
                        // 강제로 시큐리티의 세션에 접근하여 값 저장
                        SecurityContextHolder.getContext().setAuthentication(authentication);
            }
            chain.doFilter(request, response);
    }
}
```

[Access Token 과 Refresh Token 생성 예]

```
public Token createAccessToken(String userEmail, List<String> roles) {
    Claims claims = Jwts.claims().setSubject(userEmail); // JWT payload 에 저장되는 정보단위
    claims.put("roles", roles); // 정보는 key / value 쌍으로 저장된다.
    Date now = new Date();

    //Access Token
    String accessToken = Jwts.builder()
                .setClaims(claims) // 정보 저장
                .setIssuedAt(now) // 토큰 발행 시간 정보
                .setExpiration(new Date(now.getTime() + accessTokenValidTime)) // set Expire Time
                .signWith(SignatureAlgorithm.HS256, accessSecretKey)  // 사용할 암호화 알고리즘과
                // signature 에 들어갈 secret값 세팅
                .compact();

    //Refresh Token
    String refreshToken =  Jwts.builder()
                .setClaims(claims) // 정보 저장
                .setIssuedAt(now) // 토큰 발행 시간 정보
                .setExpiration(new Date(now.getTime() + refreshTokenValidTime)) // set Expire Time
                .signWith(SignatureAlgorithm.HS256, refreshSecretKey)  // 사용할 암호화 알고리즘과
                // signature 에 들어갈 secret값 세팅
                .compact();

     return Token.builder().accessToken(accessToken).refreshToken(refreshToken).key(userEmail).build();
}
```

[Spring Security + JWT 예외처리 하기]

JWT를 구현하면 jjwt 라이브러리 내부에서 여러가지 내용으로 예외를 던지고 있다. 그렇게 예외가 던져지게 되고, 내가 별도의 처리를 해주지 않으면 Response로 "Internal Server Error", "Forbbiden"과 같은 값들이 Response가 된다.

직접 예외를 처리하고자 하는 경우 다음 사항을 고려해 볼 수 있다.

(1) Advice 로 구현하여 적용
(2) JWTExceptionFilter 를 구현하여 적용
(3) 후처리 객체인 AuthenticationEntryPoint 사용

- JWTExceptionFilter 를 구현하여 적용

```
@RequiredArgsConstructor
public class JwtExceptionFilter extends OncePerRequestFilter {

    private final ObjectMapper objectMapper;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
                                    FilterChain filterChain) throws ServletException, IOException {
        response.setCharacterEncoding("utf-8");
        log.info("jwtExceptionFilter 실행");
        try{
            filterChain.doFilter(request, response);
        } catch (ExpiredJwtException e){

            log.info("expiredJwtException 터짐");
            Map<String, String> map = new HashMap<>();

            map.put("errortype", "Forbidden");
            map.put("code", "402");
            map.put("message", "만료된 토큰입니다. Refresh 토큰이 필요합니다.");

            log.error("만료된 토큰");
            response.getWriter().write(objectMapper.writeValueAsString(map));

            log.info("생성된 response = {}", response);
        } catch (JwtException e){
            log.info("JwtException 터짐");
            Map<String, String> map = new HashMap<>();

            map.put("errortype", "Forbidden");
            map.put("code", "400");
            map.put("message", "변조된 토큰입니다. 로그인이 필요합니다.");

            log.error("변조된 토큰");
            response.getWriter().write(objectMapper.writeValueAsString(map));
        }
    }
}
```

- 후처리 객체인 AuthenticationEntryPoint 사용

AuthenticationEntryPoint는 Spring Security에서 예외가 발생한 후,
반환되는 AuthenticationException을 감지하여 후처리를 수행해주는 인터페이스이다. 해당 인터페이스를 implement하여, "commence 메소드"를 구현한다.

```
@Slf4j
@Component
public class CustomAuthenticationEntryPoint implements AuthenticationEntryPoint {

    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response,
                         AuthenticationException authException) throws IOException {
        String exception = (String)request.getAttribute("exception");

        if(exception == null) {
            setResponse(response, Code.UNKNOWN_ERROR);
        }
        //잘못된 타입의 토큰인 경우
        else if(exception.equals(Code.WRONG_TYPE_TOKEN.getCode())) {
            setResponse(response, Code.WRONG_TYPE_TOKEN);
        }
        //토큰 만료된 경우
        else if(exception.equals(Code.EXPIRED_TOKEN.getCode())) {
            setResponse(response, Code.EXPIRED_TOKEN);
        }
        //지원되지 않는 토큰인 경우
        else if(exception.equals(Code.UNSUPPORTED_TOKEN.getCode())) {
            setResponse(response, Code.UNSUPPORTED_TOKEN);
        }
        else {
            setResponse(response, Code.ACCESS_DENIED);
        }
    }
    //한글 출력을 위해 getWriter() 사용
    private void setResponse(HttpServletResponse response, Code code) throws IOException {
        response.setContentType("application/json;charset=UTF-8");
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);

        JSONObject responseJson = new JSONObject();
        responseJson.put("message", code.getMessage());
        responseJson.put("code", code.getCode());

        response.getWriter().print(responseJson);
```

```
    }
}
```

// Spring Security 등록

```
@RequiredArgsConstructor
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    private final JwtTokenProvider jwtTokenProvider;
    private final ObjectMapper objectMapper;

    // authenticationManager를 Bean 등록합니다.
    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        http.csrf().disable();
        http.httpBasic().disable()
                .authorizeRequests()// 요청에 대한 사용권한 체크
                    :
                .and()
                .exceptionHandling()
                .authenticationEntryPoint(new CustomAuthenticationEntryPoint())
                .and()
                .addFilterBefore(new JwtAuthenticationFilter(jwtTokenProvider),
                        UsernamePasswordAuthenticationFilter.class)
                .addFilterBefore(new JwtExceptionFilter(objectMapper), JwtAuthenticationFilter.class);
                                        :
```

[CORS 필터 만들기]

```
@Configuration
public class CorsConfig {

    @Bean
    public CorsFilter corsFilter() {
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        CorsConfiguration config = new CorsConfiguration();
        config.setAllowCredentials(true);
        config.addAllowedOrigin("*");
        config.addAllowedHeader("*");
        config.addAllowedMethod("*");

        source.registerCorsConfiguration("/api/**", config);
        return new CorsFilter(source);
    }

}
```

[CSRF(Cross-Site Request Forgery)]

사이트 간 요청 위조는 웹사이트 취약점 공격의 하나로, 사용자가 자신의 의지와는 무관하게 공격자가 의도한 행위를 특정 웹사이트에 요청하게 하는 공격을 말한다. 유명 경매 사이트인 옥션에서 발생한 개인정보 유출 사건에서 사용된 공격 방식 중 하나이다.