


가비아 인턴 개인 과제 수행 회고(1)

 Endo(안태욱)

2023-01-09



G사 주주 총회 전자 투표 시스템 개발하기

가비아 인턴 과제를 본격적으로 수행한 첫 날로,

ERD(Entity Relation Diagram) 와 API 명세 작성, **Jwt** 를 활용한 인증, 인가를 구현하였습니다.

그 동안 **Jwt** 를 활용한 인증을 직접 구현해 본 경험이 없어서,

개발하기 전 주말에 어느 정도 익혀 놓고 구현하였는데,

분명 근거 없이 또는 정확한 동작 방식을 이해하지 못하고 코드를 작성한 부분이 있기 때문에,

금번은 작성했던 코드를 복기 하며 회고 할 계획입니다.

도메인

우선 도메인부터 살펴보겠습니다.

```
@Getter
@Builder
@Entity
public class Member implements UserDetails {

    @Id
    @Column(updatable = false, unique = true, nullable = false)
    private String memberId;

    @Column(nullable = false)
    private String password;

    @Enumerated(EnumType.STRING)
    private Role role;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new SimpleGrantedAuthority(role.name()));
    }

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

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

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

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }
}
```

```

    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

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

```

```

public enum Role {
    USER,
    ADMIN
}

```

실제 작성한 엔티티 이름은 User 인데, Member와 이름만 다를 뿐 동일한 의미입니다.

하지만 금일 작성해보니 **Spring Security** `packageorg.springframework.security.core.userdetails` 에 User 구현체가 이미 존재하여, 패키지를 구분하여 선언해야 하는 불편함이 있었습니다.. (추후 이름 변경 고려)

여기서 사용자 도메인을 *UserDetails* 인터페이스를 구현하였는데, *UserDetails* 는 Spring Security 에서 **사용자의 정보를 다루는 인터페이스**로써 해당 구현체를 통해 **사용자의 정보를 식별하고 사용**하게 됩니다.

오버라이딩 함수

- **getAuthorities**
 - 계정 권한을 리턴하는 메서드
 - 한 사용자가 하나의 권한을 가지도록 구현하였으므로 List.of 를 통해 하나의 원소가 포함된 컬렉션을 반환
- **getPassword**
 - 사용자의 **비밀 번호**를 반환
- **getUsername**
 - 사용자를 구별하는 값으로 **고유해야 함**
 - 사용자 테이블의 PK값인 ID를 반환
- **isAccountNonExpired, isCredentialsNonExpired**
 - 사용자의 계정과 비밀번호의 만료 여부
 - **만료 없음**으로 구현
- **isAccountNonLocked**
 - 사용자의 계정 잠금 여부
 - **잠금 없음**으로 구현
- **isEnabled**
 - 사용자 계정의 활성화 여부
 - **활성화**로 리턴

▼ Role.java

사용자의 역할은 주주와 관리자로 분류하였습니다.

토큰 발급

다음은 **Jwt** 토큰 발급을 위한 토큰 발급 클래스입니다.

```
@Slf4j
@Component
public class JwtTokenProvider {

    private final Key key;

    public JwtTokenProvider(@Value("${spring.jwt.secret}") String secretKey) {
        byte[] keyBytes = Decoders.BASE64.decode(secretKey);
        this.key = Keys.hmacShaKeyFor(keyBytes);
    }

    public TokenInfo generateToken(Authentication authentication) {
        String authorities = authentication.getAuthorities().stream()
            .map(GrantedAuthority::getAuthority)
            .collect(Collectors.joining(","));

        long now = (new Date()).getTime();

        Date accessTokenExpiresIn = new Date(now + 3600000);

        String accessToken = Jwts.builder()
            .setSubject(authentication.getName())
            .claim("auth", authorities)
            .setExpiration(accessTokenExpiresIn)
            .signWith(key, SignatureAlgorithm.HS256)
            .compact();

        return TokenInfo.builder()
            .grantType("Bearer")
            .accessToken(accessToken)
            .build();
    }

    public Authentication getAuthentication(String accessToken) {
        Claims claims = parseClaims(accessToken);

        if (claims.get("auth") == null) {
            throw new RuntimeException("권한 정보가 없는 토큰입니다.");
        }

        Collection<? extends GrantedAuthority> authorities =
            Arrays.stream(claims.get("auth").toString().split(","))
                .map(SimpleGrantedAuthority::new).toList();

        UserDetails principal = new User(claims.getSubject(), "", authorities);
        return new UsernamePasswordAuthenticationToken(principal, "", authorities);
    }

    public boolean validateToken(String token) {
        try {
            Jwts.parserBuilder().setSigningKey(key).build().parseClaimsJws(token);
            return true;
        } catch (SecurityException | MalformedJwtException e) {
            log.info("Invalid JWT Token", e);
        } catch (ExpiredJwtException e) {
            log.info("Expired JWT Token", e);
        } catch (UnsupportedJwtException e) {
            log.info("Unsupported JWT Token", e);
        } catch (IllegalArgumentException e) {
            log.info("JWT claims string is empty", e);
        }
        return false;
    }

    private Claims parseClaims(String accessToken) {
        try {
            return Jwts.parserBuilder().setSigningKey(key).build().parseClaimsJws(accessToken).getBody();
        } catch (ExpiredJwtException e) {
            return e.getClaims();
        }
    }
}
```

```
}
}
```

우선 클래스 내 변수로 사용되는 key는 `java.security.Key` 타입으로 선언되며, 서버에서 토큰의 암호화, 복호화에 사용하는 비밀 키를 의미합니다.

이는 application.yml 내 선언되어 있으며

```
jwt:
  secret: VlwEyVBsYt9V7zq57TejMnVUyZbLYcfPQye08f7MGVA9XkHa
```

이후 HS256 알고리즘을 사용하기 위해, 256비트 이상으로 설정하였습니다.

```
String accessToken = Jwts.builder()
    .setSubject(authentication.getName())
    .claim("auth", authorities)
    .setExpiration(accessTokenExpiresIn)
    .signWith(key, SignatureAlgorithm.HS256)
    .compact();
```

토큰은 다음과 같이 사용자의 정보를 기반으로 생성이 되는데, 등록 클레임인 토큰 제목은 사용자의 고유한 ID를 사용합니다.

또한, 사용자 인증 정보를 문자열로 완성하여 "auth" 이름의 비공개 클레임으로 등록하였습니다.

```
public Authentication getAuthentication(String accessToken) {
    Claims claims = parseClaims(accessToken);

    if (claims.get("auth") == null) {
        throw new RuntimeException("권한 정보가 없는 토큰입니다.");
    }

    Collection<? extends GrantedAuthority> authorities =
        Arrays.stream(claims.get("auth").toString().split(","))
            .map(SimpleGrantedAuthority::new).toList();

    UserDetails principal = new User(claims.getSubject(), "", authorities);
    return new UsernamePasswordAuthenticationToken(principal, "", authorities);
}
```

`getAuthentication` 메서드는 **Jwt** 토큰을 복호화하여 토큰에 들어있는 정보를 꺼내는 메서드입니다.

이제 소개할 **Jwt** 인증 필터에서 사용됩니다.

클레임에서 권한 정보를 가져와서 *Authentication* 을 생성하여 리턴하는 방식입니다.

필터

다음은 Jwt 인증 필터입니다.

```
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends GenericFilter {
```

```

private final JwtTokenProvider jwtTokenProvider;

@Override
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {
    String token = resolveToken((HttpServletRequest) request);

    if (token != null && jwtTokenProvider.validateToken(token)) {
        Authentication authentication = jwtTokenProvider.getAuthentication(token);
        SecurityContextHolder.getContext().setAuthentication(authentication);
    }
    chain.doFilter(request, response);
}

private String resolveToken(HttpServletRequest request) {
    String bearerToken = request.getHeader("Authorization");
    if (StringUtils.hasText(bearerToken) && bearerToken.startsWith("Bearer ")) {
        return bearerToken.substring(7);
    }
    return null;
}
}

```

서버에 요청 시 *UsernamePasswordAuthenticationFilter* 전에 만나는 필터로,
요청 헤더의 “**Authorization**” 값을 찾아 인증을 시도합니다.

토큰과 사용자 정보가 유효하다면, *SecurityContextHolder* 에 **인증 정보를 등록**하고 다음 필터로
요청을 전달합니다.

토큰의 정보를 읽을 때 Bearer 토큰인지 확인하는 과정 또한 포함됩니다.

필터 체인

다음은 *SecurityFilterChain* 설정 입니다. (**Spring Security 5.4** 적용)

```

@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfig {

    private final JwtTokenProvider jwtTokenProvider;

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .httpBasic().disable()
            .csrf().disable()
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            .and()
            .authorizeHttpRequests()
            .requestMatchers("/members/join").permitAll()
            .requestMatchers("/members/login").permitAll()
            .anyRequest().authenticated()
            .and()
            .addFilterBefore(new JwtAuthenticationFilter(jwtTokenProvider), UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }
}

```

우선 *Spring Security Filter* 를 등록하기 위해 `@EnableWebSecurity` 어노테이션을 등록하였습니다.

```
http.httpBasic().disable().csrf().disable()
```

현재 개발하는 서버는 Rest 기반 API 서버기 때문에, basic auth 및 csrf 보안을 사용하지 않습니다.

▼ 왜 basic auth 를 사용하지 않을까?

직접 토큰을 활용한 인증을 구현하였기 때문

▼ 왜 csrf 보안을 사용하지 않을까?

session 기반 인증과 다르게 stateless 하기 때문에 csrf 취약점으로부터 어느 정도 안전하기 때문

또한, 로그인과 회원 가입 url 은 권한 없이 접근할 수 있도록 허용하였고,

UsernamePasswordAuthenticationFilter 필터 전에 **Jwt** 인증 필터를 추가하여

Jwt 인증이 먼저 수행되도록 구현하였습니다.

서비스

마지막으로 서비스 입니다.

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

    private final MemberRepository memberRepository;
    private final PasswordEncoder passwordEncoder;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        return memberRepository.findByMemberId(username)
            .map(this::createUserDetails)
            .orElseThrow(() -> new UsernameNotFoundException("해당하는 유저를 찾을 수 없습니다."));
    }

    private UserDetails createUserDetails(Member member) {

        return User.builder()
            .username(member.getUsername())
            .password(member.getPassword())
            .roles(List.of(member.getRole().toString()).toArray(new String[0]))
            .build();
    }
}
```

`org.springframework.security.core.userdetails.UserDetailsService` 는

Spring Security 에서 유저의 정보를 가져오는 인터페이스로써, `loadUserByUsername` 메서드의 오버라이딩을 의무화 합니다.

사용하는 리포지토리에서 식별값으로 객체를 가져오는 메서드를 사용하고,

이를 **Security** 의 **User** 로 변환하여 리턴 해주면 됩니다.

사용자의 회원 가입과 로그인 요청을 처리하는 서비스 입니다.

```

@Service
@Transactional(readOnly = true)
@RequiredArgsConstructor
public class MemberService {

    private final AuthenticationManagerBuilder authenticationManagerBuilder;
    private final JwtTokenProvider jwtTokenProvider;
    private final MemberRepository memberRepository;
    private final PasswordEncoder passwordEncoder;

    @Transactional
    public Member create(MemberJoinRequestDto memberJoinRequestDto) {

        Member member = Member
            .builder()
            .memberId(memberJoinRequestDto.getMemberId())
            .password(passwordEncoder.encode(memberJoinRequestDto.getPassword()))
            .role(memberJoinRequestDto.getRole())
            .build();
        return memberRepository.save(member);
    }

    @Transactional
    public TokenInfo login(String memberId, String password) {
        UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(memberId, password);

        Authentication authentication = authenticationManagerBuilder.getObject().authenticate(authenticationToken);

        return jwtTokenProvider.generateToken(authentication);
    }
}

```

회원 가입 요청 Dto 를 통해 **사용자 객체를 생성 및 저장**하고 생성된 객체를 반환하는데,
프로젝트 코드에서는 실제 객체가 아닌 **응답 Dto 를 통해 반환**합니다. (엔티티 노출 방지)

로그인 요청 시,
로그인 정보가 **유효한지 확인**하고, 유효할 경우 **상응하는 Jwt 토큰 객체를 반환**합니다.

결론

Rest 기반 API 서버에서 Jwt 를 활용한 인증, 인가를 알게 되었지만,
추후 토큰 만료시 사용되는 **refresh token** 이나 **비밀번호 만료**에 대한 개념도 추가 학습을
진행할 예정입니다.

또한, 개인 과제 프로젝트 첫 개발에 있어서 **gitlab** 사용법을 더 익혀야 하고,
Backer 님의 도움으로 **템플릿 사용**도 경험해 볼 수 있었습니다.