가비아 인턴 개인 과제 수행 회고(2)

👨‍💻 *Endo(안태욱)* *2023-01-10*

🗣️ **안건 관리 기능 개발하기**

가비아 인턴 과제를 수행한 둘째 날로,

첫 날 인증 인가 구현에 이어서 실제 **투표 시스템의 기능**을 구현하였습니다.

사용자와 관리자의 기능이 구별된 만큼 금일은 **관리자의 안건 조작 기능**을 구현하였고,

merge 하기 전 **코드 리뷰**를 통해 부족한 부분을 보완할 수 있었습니다.

첫 날과 다르게 *깃랩 리포지토리 링크*를 가져왔기 때문에

작성한 코드를 살펴보며 복기할 수 있게 되었습니다.

도메인

관리자의 관리 대상인 자원 **"안건"** 도메인에 대해 살펴보겠습니다.

```
@NoArgsConstructor(access = AccessLevel.PROTECTED)
@Getter
@Entity
public class Agenda {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long ID;

    @Column(nullable = false)
    private String title;

    @Enumerated(EnumType.STRING)
    @Column(nullable = false, length = 50)
    private AgendaType type;

    private int positiveRights = 0;

    private int negativeRights = 0;

    private int invalidRights = 0;

    @Column(nullable = false)
    private LocalDateTime startsAt;

    @Setter
    @Column(nullable = false)
    private LocalDateTime endsAt;

    private Agenda(String title, AgendaType type, LocalDateTime startsAt, LocalDateTime endsAt) {
        this.title = title;
```

```
        this.type = type;
        this.startsAt = startsAt;
        this.endsAt = endsAt;
    }

    public static Agenda of(AgendaCreateRequestDto requestDto) {
        return new Agenda(requestDto.getTitle(), requestDto.getType(),
                requestDto.getStartsAt(), requestDto.getEndsAt());
    }

    public static Agenda create(String title, AgendaType type, LocalDateTime startsAt, LocalDateTime endsAt) {
        return new Agenda(title, type, startsAt, endsAt);
    }
}
```

우선 안건 생성에 관해,

ID는 자동으로 생성되고 투표 수는 **0**으로 초기화가 고정이기 때문에 나머지 4개의 요소를 빌더 패턴을 사용하지 않고 **생성자 메서드를 정의**하는 방향으로 설계하였습니다.

위와 같이 설계하면 **안건 생성 DTO를 통한 생성**과 **테스트 코드 작성** 시에 유리하고, 간단하게 안건 객체를 생성할 수 있기 때문입니다.

하지만 **기본 생성자가 요구**되기 때문에(**프록시 객체 생성**을 위해) 기본 생성자를 **protected 레벨로 정의**하였습니다.

안건의 상태에 관해,

안건의 **종류(일반, 선착순)**은 있지만 안건의 **상태**를 정의하는 *enum*이 없는데, 그 이유는 안건의 종료(유효) 여부를 투표를 보내는 시간과 안건의 종료 시간을 비교하여 결정하기 때문입니다.

그렇기 때문에 관리자가 기존의 종료 시간 전에 **안건을 종료**하면 **기존 종료 시간을 앞당기는 방식으로 동작**하게 됩니다.

서비스

```
@RequiredArgsConstructor
@Transactional(readOnly = true)
@Service
public class AgendaService {
    private final AgendaRepository agendaRepository;
    public Page<Agenda> getAgendas(Pageable pageable) {
        return agendaRepository.findAll(pageable);
    }

    public Agenda getAgenda(long agendaId) {
        return getAgendaWithId(agendaId);
    }

    @Transactional
    public Agenda createAgenda(AgendaCreateRequestDto agendaCreateRequestDto) {
```

```
        return agendaRepository.save(Agenda.of(agendaCreateRequestDto));
    }

    @Transactional
    public void removeAgenda(long agendaId) {
        Agenda agenda = getAgendaWithId(agendaId);
        agendaRepository.delete(agenda);
    }

    @Transactional
    public Agenda terminate(long agendaId) {
        Agenda agenda = getAgendaWithId(agendaId);
        agenda.setEndsAt(LocalDateTime.now());
        return agenda;
    }

    private Agenda getAgendaWithId(Long agendaId) {
        Optional<Agenda> findAgenda = agendaRepository.findById(agendaId);
        validateAgenda(findAgenda);
        return findAgenda.get();
    }

    private void validateAgenda(Optional<Agenda> agenda) {
        if (agenda.isEmpty()) {
            throw new EntityNotFoundException("해당 Id에 해당하는 안건이 존재하지 않습니다.");
        }
    }
}
```

리턴 방식에 관해,

클라이언트로 **엔티티의 직접적인 노출을 지양**하기 위해 **요청에 적합한 Dto로 변환**하여 반환하는데,

변환 작업을 **서비스**와 **컨트롤러** 중 어디에서 해야 하는 지에 대한 고민 끝에

요청으로 들어온 Dto가 실제 리포지토리에 들어가기 전인 **서비스**에서

Dto to Entity 변환이 이루어지고,

반한 될 개체가 실제 클라이언트에게 반환 되기 직전인 **컨트롤러**에서

Entity to Dto 로의 변환이 이루어지도록 구현하였는데,

모든 변환을 한 계층에서 수행해야 하는 지에 대해서는 좀 더 고민하고

리팩토링 단계에서 보완하는 방식으로 진행 할 예정입니다.

ID를 통한 안건 조회에 관해,

실제 요청한 **ID를 가진 안건의 존재 여부를 확인하는 로직**을 두 개의 메서드로 **분리**하여

ID와 함께 들어온 **요청을 처리하도록 구현**하였습니다.

컨트롤러

다음은 안건 컨트롤러 입니다.

```
@RequiredArgsConstructor
@RequestMapping("/api")
@RestController
public class AgendaController {

    private final AgendaService agendaService;

    @GetMapping("/agendas")
    public Page<AllAgendaResponseDto> agendas(
            @PageableDefault(size = 10, sort = "startsAt", direction = Sort.Direction.ASC) Pageable pageable) {
        return agendaService.getAgendas(pageable).map(AllAgendaResponseDto::from);
    }

    @ResponseStatus(HttpStatus.CREATED)
    @PostMapping("/agendas")
    public SimpleAgendaResponseDto create(@RequestBody AgendaCreateRequestDto agendaCreateRequestDto) {
        Agenda savedAgenda = agendaService.createAgenda(agendaCreateRequestDto);
        return SimpleAgendaResponseDto.from(savedAgenda);
    }

    @GetMapping("/agendas/{id}")
    public SimpleAgendaResponseDto agenda(@PathVariable("id") Long agendaId) {
        return SimpleAgendaResponseDto.from(agendaService.getAgenda(agendaId));
    }

    @DeleteMapping("/agendas/{id}")
    public void delete(@PathVariable("id") Long agendaId) {
        agendaService.removeAgenda(agendaId);
    }

    @PatchMapping("/agendas/{id}/terminate")
    public SimpleAgendaResponseDto end(@PathVariable("id") Long agendaId) {
        return SimpleAgendaResponseDto.from(agendaService.terminate(agendaId));
    }
}
```

전체 안건 조회에 관해,

전체 안건 조회는 기본적으로 **페이징**을 제공하기 위해 **JPA의 Pageable 처리 방식**을 사용하였습니다.

이를 통해 **페이징** 외에도 페이징 된 **안건들의 정렬 기능**을 사용하여 기본적으로 **투표 시작 시간이 빠른 안건부터 조회**할 수 있도록 구현하였습니다.

안건 종료에 관해,

해당 요청을 *GET* 메서드로 받는 것으로 기존에 구현하였었는데,

코드 리뷰에서 메서드와 *URL*의도에 대해 말씀해 주셔서 많은 고민을 할 수 있었습니다.

CRUD 외에 요청은 어쩔 수 없이 자원이 아닌 행위가 들어가는 *URL*을 사용하였었는데,

메서드는 그 만큼도 고민하지 않았다는 사실을 깨달았고,

안건의 상태를 변경하지만 일부(종료 시간)만 변경함을 요청하므로

`PATCH /agendas/{id}/terminate`

우선 위와 같이 구현을 하였는데,

해당 부분에 대해서는 **추가적인 고민이 필요**할 것 같습니다.

Security 설정

인가 별 동작 구분을 위해 Security 설정을 추가하였습니다.

```
// SecurityConfig.java
@Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
                .httpBasic().disable()
                .csrf().disable()
                .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .and()
                .authorizeHttpRequests()
                .requestMatchers("/api/sign-up").permitAll()
                .requestMatchers("/api/login").permitAll()
                .requestMatchers(HttpMethod.POST, "/api/agendas").hasRole(Role.ADMIN.name())
                .requestMatchers(HttpMethod.DELETE, "/api/agendas/**").hasRole(Role.ADMIN.name())
                .requestMatchers("/api/agendas/*/terminate").hasRole(Role.ADMIN.name())
                .anyRequest().authenticated()
                .and()
                .addFilterBefore(new JwtAuthenticationFilter(jwtTokenProvider), UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }
```

requestMatchers 추가

안건 **생성과 삭제, 종료는 관리자만 수행**할 수 있으므로 정의했던 *Role* 기준으로

관리자 *Role*을 소유한 클라이언트에 대해서만 **요청을 허용**하도록 설정을 추가하였습니다.

결론

단순히 어떤 자원을 다룰 수 있는 서버에서 **MVC 패턴**으로 나눠서 구현할 때,

자원을 다루는 **로직** 외에도 신중하게 생각해야 하는 부분이 많다는 것을 느꼈습니다.

실제 코드의 효율이나 가독성 만큼 중요한 것은

자신의 판단 기준을 정하고 항상 그 기준에 따라 구현해야 한다는 부분인 것 같습니다.

(자신의 기준이 있다면 모자라더라도 그 기준을 보완해 나가면 되기 때문입니다)

또한, 처음으로 코드 리뷰를 받아봤는데

혼자서 코드를 작성할 때에 느끼지 못했던 다양한 개선점들을 발견할 수 있음을 느꼈습니다.

다음은 사용자의 투표 수행을 구현하고,

Aiden 님이 말씀하신 **Swagger 를 통한 API 명세 작성**과,

Woody 님이 말씀하신 **jasypt를 이용한 property 암호화**를 적용 할 예정입니다.