ChatGPT

게시판 시스템의 세분화된 권한 제어 아키텍처 제안

Spring Boot + JPA + Spring Security 환경에서 **게시물 단위로 사용자(사원) 및 팀별 읽기/수정/삭제 권한**을 구현하려면, 기본 역할(ROLE) 기반 보안만으로는 한계가 있습니다. 아래 다섯 가지 설계 방안을 제시하며, 각 접근 방법의 구조와 용도, 장단점, 핵심 코드 예시를 설명합니다. 필요한 경우 현재 구조의 개선 방향도 언급합니다.

1. Spring Security ACL(Access Control List) 활용

구조: Spring Security ACL을 사용하면 도메인 객체(`Post`)별로 권한(ACE)을 부여하고 검사할 수 있습니다. ACL은 `acl_sid`, `acl_object_identity`, `acl_entry` 등 전용 테이블을 두고, 각 게시물 인스턴스에 대해 권한 목록을 저장합니다. 게시물 생성 시 AclService를 통해 `ObjectIdentity`를 만들고, `acl.insertAce(...)`로 사용자/팀별 `READ/WRITE/DELETE` 권한을 추가합니다. 이후 메서드나 리소스 접근 시 Spring Security의 `hasPermission()` 등을 통해 ACL 검사를 수행합니다.
- **적용 예:** 복잡한 권한 관리가 필요하고, 게시물이 매우 많거나 권한 로직이 자주 변경되는 대규모 시스템에서 유용합니다. 관리자, 편집자, 일반 사용자 등 역할 외에 **개별 객체 수준의 세밀한 권한**을 부여하기에 적합합니다 [1].
- **장점:** Spring Security에서 공식 지원하는 객체 수준 보안 기능이므로 안정적이며, **객체별 권한 부여/철회**가 명시적입니다. 사용자(SID)와 역할(그룹) 모두를 주체로 권한을 부여할 수 있습니다. 또한 ACL 상속, 소유자 개념 등 고급 기능을 활용할 수 있습니다.
- **단점:** 초기 설정과 학습 비용이 높으며, **전용 테이블과 조인 연산**이 많아 성능 이슈가 있을 수 있습니다. 설계와 운영이 복잡하며, 기본 CRUD 로직과 ACL 동기화 코드가 필요합니다. 작은 규모나 단순 권한 요구에는 과할 수 있습니다.
- **구현 스니펫:** 게시물 생성/공유 시 ACL 엔트리를 추가하는 예시입니다.

```
// ACL 서비스 주입
@Autowired
private MutableAclService aclService;

public void grantReadPermission(Post post, User user) {
  // 게시물에 대한 ObjectIdentity 생성
  ObjectIdentity oid = new ObjectIdentityImpl(Post.class, post.getId());
  MutableAcl acl = (MutableAcl) aclService.createAcl(oid);
  // 사용자에게 READ 권한 부여
  acl.insertAce(acl.getEntries().size(), BasePermission.READ, new PrincipalSid(user.getUsername()),
true);
  aclService.updateAcl(acl);
}
```

이후 메서드나 컨트롤러에서 `@PreAuthorize("hasPermission(#post, 'READ')")` 등의 어노테이션으로 접근을 제한할 수 있습니다.

2. 별도 권한(Authorization) 엔티티 테이블 도입

구조: `Post`와 `Employee`, `Team` 사이의 다대다 대신 **권한 엔티티**를 별도로 설계합니다. 예를 들어 `PostPermission` 엔티티를 만들어 `post`, `user(사원)`, `team`, `permissionType(READ/EDIT/DELETE)` 필드를 둡니다. 한 게시물에 대해 특정 사용자나 팀에게 개별 권한을 부여/저장할 수 있습니다. 기존 `Post` 엔티티에서는 `readableEmployees`/`readableTeams` 대신 이 테이블 조회로 권한을 확인합니다.

- **적용 예:** 현재 ManyToMany 필드 구조가 복잡해지거나 권한 유형이 증가할 때 유용합니다. 권한 로직을 데이터 모델에 명확히 드러내고 DB 레벨에서 관리하려는 경우 적합합니다.
- **장점:** 권한 구조가 **명시적이고 유연**합니다. 권한 부여 이력을 관리하거나, `PermissionType`을 확장(예: 공유권한 등 추가)하기 편리합니다. JPA로 엔티티를 관리하므로 비즈니스 로직에서 쉽게 쿼리/수정할 수 있습니다.
- **단점:** 직접 권한 검사 로직을 구현해야 합니다. 권한을 확인할 때 추가 조회가 필요하며, 쿼리 작성이 복잡해질 수 있습니다. 작은 프로젝트에서는 오버헤드일 수 있습니다.
- **구현 스니펫:** `PostPermission` 엔티티 예제입니다.

```
@Entity
public class PostPermission {
  @Id @GeneratedValue
  private Long id;

  @Enumerated(EnumType.STRING)
  private PermissionType permission; // READ, EDIT, DELETE 등

  @ManyToOne
  private Post post;

  @ManyToOne
  private Employee employee;  // 사원 권한 대상

  @ManyToOne
  private Team team;       // 팀 권한 대상

  // ... 생성자, getter/setter 생략 ...
}
```

이 구조로 `PostPermissionRepository`를 작성해, 특정 사용자/팀과 게시물, 권한 유형으로 조회/생성/삭제할 수 있습니다. 예를 들어, 게시물 수정 시 현재 로그인한 사용자가 해당 게시물에 대해 `EDIT` 권한이 있는지 `PostPermissionRepository.findByPostAndEmployeeAndPermission(...)` 등을 통해 확인합니다.

3. 메서드 보안(@PreAuthorize) + 커스텀 PermissionEvaluator

구조: Spring Security의 메서드 보안 기능을 활용하여, `@PreAuthorize`나 `@PostAuthorize` 어노테이션과 커스텀 `PermissionEvaluator`를 사용합니다. 서비스나 컨트롤러 메서드에 `@PreAuthorize("hasPermission(#post, 'WRITE')")` 식으로 권한을 검사하고, `PermissionEvaluator`가 실제 로직을 구현합니다. 예를 들어 현재 인증된 사용자 객체와 `Post` 엔티티를 받아서, 사용자가 게시물 작성자이거나 `readableEmployees` / `readableTeams`에 포함되는지 검사합니다.
- **적용 예:** 권한 검사 로직을 서비스 레이어에 분리하고, 표현식 기반으로 제어하고 싶을 때 유용합니다. 작은 규모나 복잡한 프레임워크 도입 없이 **객체 단위 권한 확인**을 직접 코드로 구현하는 경우에 적합합니다.
- **장점:** Spring Security와의 통합이 쉽고 직관적입니다. `@PreAuthorize`를 통해 메서드 단위로 권한을 선언적(declarative)으로 지정할 수 있어 가독성이 높습니다. 전용 `PermissionEvaluator`를 통해 원하는 검증 로직을 자유롭게 구현할 수 있습니다 [2].
- **단점:** 권한 검사가 코드 실행 시점에 동적으로 이루어지므로, 잘못 사용하면 런타임 오류가 발생할 수 있습니다. 권한 확인 로직이 분산되기 쉽고, 성능 면에서 매 요청마다 객체 비교/조회 비용이 들어갑니다. 보안상 누락 없이 모든 필요한 메서드에 어노테이션을 적용해야 합니다.
- **구현 스니펫:** `PermissionEvaluator` 예제와 사용 모습입니다.

```
@Component
public class  PostPermissionEvaluator implements PermissionEvaluator {
  @Override
  public boolean hasPermission(Authentication auth, Object target, Object perm) {
    if (!(target instanceof Post)) return false;
    Post post = (Post) target;
    String op = (String) perm;  // "VIEW", "EDIT", "DELETE" 등
    Employee currentUser = ((MyUserDetails) auth.getPrincipal()).getEmployee();

    if ("VIEW".equals(op)) {
      // 작성자이거나 허용된 팀/사원 목록에 포함되면 true
      return post.getWriter().equals(currentUser)
        || post.getReadableEmployees().contains(currentUser)
        || currentUser.getTeams().stream().anyMatch(post.getReadableTeams()::contains);
    }
    if ("EDIT".equals(op) || "DELETE".equals(op)) {
      // 수정/삭제 권한 검사 (예: 작성자만 허용 or 별도 로직)
      return post.getWriter().equals(currentUser);
    }
    return false;
  }
  // hasPermission(Authentication auth, Serializable targetId, String targetType, Object perm) 메서드도 구현 가능
}

// 사용 예 (Service 또는 Controller):
@PreAuthorize("hasPermission(#post, 'VIEW')")
public Post viewPost(@P("post") Post post) { ... }
```

위 예제는 메서드 호출 전 권한을 확인하는 방식으로, 권한 여부를 반환합니다. `@EnableMethodSecurity` 설정이 필요합니다.

4. JPA Repository/Specification 수준에서 필터링

구조: 데이터 접근 계층(JPA Repository)에서 현재 사용자/팀에 따라 조회되는 게시물 리스트를 필터링합니다. 예를 들어 `PostRepository`에 JPQL/Criteria를 작성하여 `WHERE p.writer = :user OR :user MEMBER OF p.readableEmployees OR :team IN (p.readableTeams)` 등의 조건으로 쿼리합니다. 이렇게 하면 읽기 가능한 게시물만 DB 수준에서 가져옵니다. 수정/삭제 시에도 비슷한 조건으로 해당 게시물을 찾거나 예외를 발생시켜 권한을 제어할 수 있습니다.
- **적용 예: 게시물 목록 조회** 시점에 권한 필터링을 적용하면 클라이언트에 노출될 글을 자동 제한할 수 있습니다. 다만, 단일 게시물 수정/삭제는 추가 로직이 필요하지만, 리스트에는 효과적입니다.
- **장점:** JPA 차원에서 권한 조건이 적용되므로 불필요한 데이터 로딩을 줄일 수 있습니다. DB 인덱스를 활용한 효율적 조회가 가능합니다. 비즈니스 로직 코드에서 별도 권한 검증을 반복하지 않아도 되며, 조건을 변경하면 쿼리만 수정하면 됩니다.
- **단점:** 읽기(List)에는 유리하지만, 수정/삭제 권한 검증에는 별도의 코드가 필요합니다. JPQL이나 Specification이 복잡해질 수 있으며, 인원/팀이 많은 경우 조인 비용이 커질 수 있습니다. 또한 실시간 팀/사원 정보 변화 반영에 주의해야 합니다.
- **구현 스니펫:** 현재 사용자와 팀을 입력받아 접근 가능한 게시물만 조회하는 예시입니다.

```
public interface  PostRepository extends JpaRepository<Post, Long> {
  @Query("SELECT p FROM Post p LEFT JOIN p.readableEmployees e LEFT JOIN p.readableTeams t
" +
      "WHERE p.writer = :user OR e = :user OR t IN :teams")
  List<Post> findAccessiblePosts(@Param("user") Employee user, @Param("teams") List<Team>
teams);
}
```

호출 예: `postRepo.findAccessiblePosts(currentUser, currentUser.getTeams());`
이 외에 JPA `Specification`을 사용해 동적으로 조건을 조합할 수도 있습니다.

5. 정책 기반 접근 제어(ABAC/외부 정책 엔진)

구조: 정책 엔진(Policy Engine)이나 ABAC(Attribute-Based Access Control) 기법을 도입하여 권한 로직을 분리합니다. 예를 들어 OPA(Open Policy Agent) 같은 외부 서비스나 jCasbin 등을 사용해 권한 정책을 선언적으로 관리합니다. 게시물 소유자, 허용된 팀/사원 정보를 OPA에 입력값으로 제공하면, OPA가 Rego 언어로 작성한 정책대로 `allow` / `deny`를 결정합니다. Spring Security Filter나 AOP에서 OPA 정책을 호출해 결과에 따라 접근을 허용하거나 차단합니다.
- **적용 예:** 기업 정책이나 규정에 따라 권한이 자주 바뀌거나, 서비스 간 공통 정책을 중앙에서 관리해야 할 때 유용합니다. ABAC 방식은 **사용자 속성(userId, 팀), 리소스 속성(post의 readable 리스트, 작성자)** 등을 조합하여 동적 평가합니다 [3]. 예를 들어 "사용자가 게시물 작성자이거나, 게시물 허용팀에 속하면 읽기 허용" 같은 규칙을 정책으로 정의할 수 있습니다.
- **장점:** 권한 로직이 코드에서 분리되어 정책으로 중앙 집중화되므로, 정책 변경 시 애플리케이션 코드 수정 없이 대응할 수 있습니다. 복잡한 비즈니스 규칙(예: 시간, 위치, 환경 변수 등)도 정책에 포함 가능합니다. 대규모 시스템에서 **일관된 권한 관리**가 가능합니다.
- **단점:** 외부 컴포넌트 도입 및 관리가 필요해 복잡도가 증가합니다. 시스템 간 네트워크 호출 비용이나 정책의 복잡성으로 인한 성능 저하를 고려해야 합니다. 작은 프로젝트에는 오버헤드일 수 있으며, 정책 문법 학습이 필요합니다.
- **구현 스니펫:** OPA에서 게시물 권한 정책을 정의하는 예시(Rego 언어)입니다.

```
package post

default allow = false

# 게시물 작성자는 모든 권한 허용
allow {
 input.userId == data.posts[input.postId].writerId
}
# 읽기 권한: 사용자나 사용자 소속 팀이 허용 목록에 있으면 허용
allow {
 input.userId == data.posts[input.postId].readableEmployees[_]
}
allow {
 data.posts[input.postId].readableTeams[_] == input.userTeamId
}
```

위 정책에 따라 Spring Boot 애플리케이션에서 `input = { "postId":123, "userId":456, "userTeamId":789 }`를 OPA에 전달하여 `allow` 여부를 판단합니다. 정책 엔진 사용 시, Spring Security와 통합하거나 직접 HTTP 클라이언트로 호출하여 권한 검증을 수행할 수 있습니다 [3].

개선 방향 요약

현재 구조(`Post`가 `readableTeams`, `readableEmployees`를 ManyToMany로 보유)에서는 읽기 권한만 정의되어 있고, **수정/삭제 권한 필드가 없습니다**. 제안된 방안 중에서는 **별도 권한 엔티티 도입(안 2)**이나 **Spring ACL(안 1)**이 이 구조를 확장하기 적합합니다. 또한 `Post` 생성/수정 시 권한 정보를 함께 받아 저장하도록 UI/API를 보완해야 합니다. 예를 들어 `PostDTO`에 권한 관련 입력 필드를 추가하고, Service 레이어에서 ACL 또는 권한 엔티티를 갱신합니다.

Spring Security만으로는 현재 `Authentication` 객체 수준의 권한만 제공되므로, 객체 레벨 권한 검사를 위해선 위와 같은 설계가 필요합니다. 각 제안은 요구사항과 팀 역량, 시스템 규모에 따라 선택할 수 있습니다.

참고 자료: Spring Security ACL을 활용한 도메인 객체 보안 예제 [1], Method Security와 PermissionEvaluator 가이드 [2], ABAC/정책 기반 접근 제어 소개 [3] 등을 참조했습니다.

---

[1] [Spring] Spring Security ACL Tutorial
https://doongjun.tistory.com/92

[2] Spring Security PermissionEvaluator
https://codingnomads.com/spring-security-permissionevaluator

[3] Externalized Authorization using OPA and Spring Security
https://sultanov.dev/blog/externalized-authorization-using-opa-and-spring-security/