ERD 분석

상태	진행 중
다중 선택	데이터베이스 백엔드 스프링
생성 일시	@2025년 1월 14일 오후 10:30

ERD 분석하기

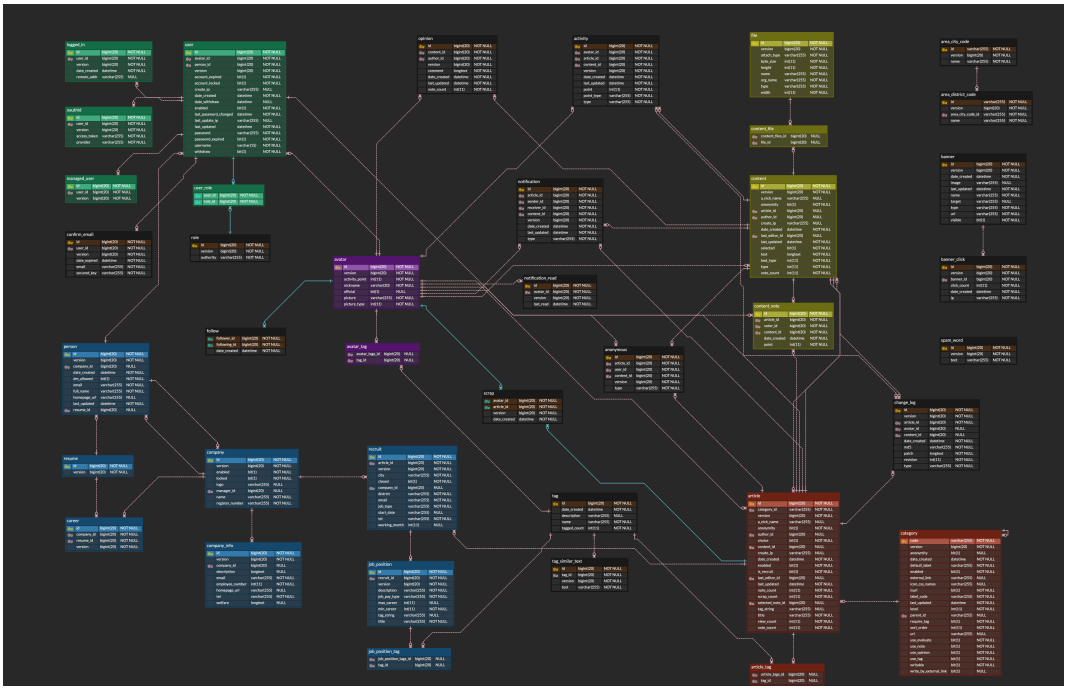
- 테이블 설계에 앞서서 데이터 모델링 개념을 살펴보고
- ERD 다이어그램 샘플들을 살펴보자.

---

https://inpa.tistory.com/entry/DB-%F0%9F%93%9A-%EB%8D%B0%EC%9D%B4%ED%84%B0-%EB%AA%A8%EB%8D%B8%EB%A7%81-1N-%EA%B4%80%EA%B3%84-%F0%9F%93%88-ERD-%EB%8B%A4%EC%9D%B4%EC%96%B4%EA%B7%B8%EB%9E%A8

https://inpa.tistory.com/entry/DB-%F0%9F%93%9A-%EC%A0%9C-1-2-3-%EC%A0%95%EA%B7%9C%ED%99%94-%EC%97%AD%EC%A0%95%EA%B7%9C%ED%99%94

1. OKKY



Q. bit 타입?

- `BIT`는 **1비트 또는 더 작은 공간**을 사용하여 데이터를 저장합니다.
- 일반적으로 `0`과 `1` 값으로 **부울(Boolean) 타입**을 나타냅니다.
- 하나의 컬럼에서 여러 비트를 묶어 저장할 수도 있습니다(예: `BIT(8)`은 최대 8개의 플래그를 표현 가능).
- SQL 표준에서는 `BOOLEAN` 타입이 지원되지 않거나 별도로 구현되지 않는 경우 `BIT`를 활용합니다.

다른 대안은 없어?

데이터 타입	MySQL 저장 크기	가독성	Spring JPA 매핑 용이성	상태 확장성	공간 효율성	추천 상황
BIT	1비트	낮음	보통	낮음	매우 높음	단순 플래그 (TRUE/FALSE).
BOOLEAN	1바이트 (TINYINT)	높음	매우 높음	보통	보통	플래그와 호환성을 모두 고려.
TINYINT	1바이트	보통	보통	높음	보통	플래그와 상태값 확장이 필요한 경우.
CHAR(1)	1바이트	높음	보통	높음	낮음	사람이 읽기 쉬운 값(Y/N) 필요 시.
ENUM	1바이트 이상	매우 높음	매우 높음	매우 높음	보통	상태값이 명확히 정의될 때.

결론

- **단순 플래그(`is_active`, `is_admin`)**
  - **Spring JPA와 MySQL 호환성**: `BOOLEAN` (내부적으로 `TINYINT(1)` 처리) 추천.
  - 공간 최적화 필요: `BIT`.
- **읽기 쉬운 플래그가 필요한 경우**
  - `CHAR(1)`을 활용해 `Y/N` 또는 `T/F` 형태로 저장
- **다양한 상태 값이 필요한 경우**
  - Enum 타입이 적합: Spring JPA에서 `@Enumerated`를 활용하여 `ENUM` 매핑.

Q. logged_in 테이블은 왜 따로 관리되나요?

> loggeed_in : user → N : 1

만약 로그인 상태를 `user` 테이블에 통합하려고 한다면 다음과 같은 문제가 발생할 수 있습니다:

1. **중복 데이터**:
   - 한 사용자가 여러 디바이스에서 동시에 로그인하면, `user` 테이블에 이 정보를 담기 어려움.
   - `user` 테이블에 단일 로그인 상태만 저장되므로 세션 관리가 불가능.
2. **이력 관리 어려움**:
   - 사용자의 이전 로그인 정보(`last_login_time`)를 덮어쓰게 되므로, 과거 데이터는 손실됩니다.
3. **성능 문제**:
   - 사용자 정보와 로그인 정보를 동시에 조회하거나 업데이트해야 하므로 테이블에 더 많은 부하가 걸림.
   - 예: 사용자가 로그인할 때마다 `user` 테이블의 `is_logged_in` 값을 변경하면, 잠재적으로 잠금 현상(Lock Contention)을 유발

▼ 더 자세히 **"단일 로그인 상태 저장 방식의 문제점"**을 물어봤다.

(1) 다중 로그인 상태를 구분할 수 없음

단일 로그인 상태 저장 방식은 **사용자가 여러 디바이스에서 로그인할 경우 이를 구분할 수 없습니다.**

- 예: Alice가 PC와 스마트폰에서 로그인했다고 가정하면, `is_logged_in` 컬럼은 마지막으로 로그인한 디바이스의 상태만 반영됩니다.

- 이전 디바이스(예: PC)의 세션 정보는 덮어쓰여져 **세션 충돌**이 발생.

(2) 세션별 정보 저장 불가

단일 로그인 상태를 저장할 경우 세션별로 관리해야 할 정보를 저장할 수 없습니다.

- `user` 테이블에 `session_token`이나 `device_info`를 추가한다고 해도, **한 번에 하나의 세션 정보**만 저장 가능.
- 다중 디바이스 로그인을 지원하거나 특정 디바이스만 로그아웃하려면 **별도의 구조**가 필요.

(3) 이력 추적 어려움

단일 로그인 상태 저장 방식에서는 이전 로그인 기록이 덮어쓰이므로, 사용자가 **언제, 어디서 로그인했는지**를 추적하기 어렵습니다.

- 보안상 문제가 발생할 경우, 어떤 디바이스에서 의심스러운 활동이 있었는지 파악할 수 없음.
- 감사 로그와 같은 기능을 제공하기 어려움.

(4) 로그아웃 처리의 제약

특정 디바이스에서만 로그아웃하거나, **전체 디바이스에서 로그아웃**과 같은 기능을 제공하기 어렵습니다.

- 단일 `is_logged_in` 컬럼으로 관리하면, 특정 디바이스의 세션을 유지한 채 다른 디바이스에서 로그아웃을 수행할 방법이 없습니다.

Q. version 컬럼?

- `version` 컬럼은 **데이터 동시성 제어**와 **이력 관리**를 목적으로 사용됩니다.
- 낙관적 락을 구현하여, 다중 사용자 환경에서 데이터 무결성을 보장.
- Spring JPA에서는 `@Version` 애너테이션을 통해 쉽게 구현 가능.
- `version` 컬럼을 활용하면 데이터 충돌 방지와 함께 수정 이력 추적도 가능하여, 안전하고 효율적인 데이터 관리를 지원합니다.

▼ 낙관적 락을 어떻게 방지하는 지 더 자세히 찾아봤다.

https://sabarada.tistory.com/175

→ ... **같은 row에 대해서 각기 다른 2개의 수정 요청이 있었지만 1개가 업데이트 됨에 따라 version이 변경되었기 때문에 뒤의 수정 요청은 반영되지 않게 되었습니다.** 이렇게 낙관적락은 version과 같은 별도의 컬럼을 추가하여 충돌적인 업데이트를 막습니다. version 뿐만 아니라 hashcode 또는 timestamp를 이용하기도 합니다.

Q. user_role 연결 테이블은 왜 식별관계를 사용했을까?

(1) 식별 관계를 사용한 이유

1. **다대다 관계 관리의 표준 방식**
   - 다대다 관계를 구현할 때, 연결 테이블(Bridge Table)은 보통 양쪽 부모 테이블의 기본키를 조합한 복합 기본키를 가집니다.
   - 이는 두 엔터티 간의 관계를 정확히 나타낼 수 있는 효율적이고 직관적인 방식입니다.
2. **관계의 강한 의존성 표현**
   - `map_user_role`의 레코드는 반드시 특정 `user`와 특정 `role`에 종속됩니다.
     - 예: 사용자가 삭제되면, 해당 사용자의 역할 정보도 삭제되어야 함.
     - 역할이 삭제되면, 해당 역할에 관련된 모든 사용자 관계도 삭제되어야 함.

- 식별 관계를 사용함으로써, 이러한 강한 종속성을 자연스럽게 구현할 수 있습니다.

3. **무결성 유지**
   - 식별 관계를 통해 `user_id`와 `role_id`가 항상 유효한 부모 엔터티를 참조하도록 보장됩니다.
   - 예: `user`나 `role`에 없는 ID가 `map_user_role`에 들어가는 것을 방지.
4. **데이터 참조 간소화**
   - 식별 관계를 사용하면 `map_user_role`의 기본키가 `user_id`와 `role_id`로 설정되므로, 추가적인 식별자를 생성할 필요가 없습니다.
   - 이를 통해 데이터를 관리하고 조회하는 쿼리가 간결해집니다.

(2) 비식별 관계를 사용할 경우의 문제점

- 비식별 관계를 사용하려면 `map_user_role`에 독립적인 기본키(예: `id`)를 추가해야 합니다.
- 이렇게 하면 테이블의 기본키와 부모 엔터티 간의 관계가 약해지고, 관계 무결성을 명확하게 보장하지 못할 수 있습니다.
- 또한, `id`를 추가로 생성하는 것은 불필요한 오버헤드일 수 있습니다.

▼ jpa에서는 복합키를 어떻게 구현할 수 있나요?

https://cs-ssupport.tistory.com/482

- ManyToMany 매핑이 가능하다....

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToMany
    @JoinTable(
        name = "map_user_role", // 중간 테이블 이름
        joinColumns = @JoinColumn(name = "user_id"), // 현재 엔터티의 외래키
        inverseJoinColumns = @JoinColumn(name = "role_id") // 반대쪽 엔터티의 외래키
    )
    private Set<Role> roles = new HashSet<>();
}
```

- but... 연결테이블의 추가적인 컬럼을 넣고 싶다면 문제가 되기에 테이블로 풀어주자.

```
@Entity
public class MapUserRole {
    @EmbeddedId
    private MapUserRoleId id;

    @ManyToOne
    @MapsId("userId")
    private User user;

    @ManyToOne
    @MapsId("roleId")
    private Role role;
```

```
    private LocalDateTime assignedDate;
}

@Embeddable
public class MapUserRoleId implements Serializable {
    private Long userId;
    private Long roleId;

    // equals, hashCode 오버라이드 필수
}
```

- 아직 안 끝났다... 그래서 연결테이블을 식별관계/비식별관계 무엇으로 설정할까?

> JPA를 사용하는 현업에서는 **대부분의 경우 연결 테이블을 비식별 관계로 설정**하며, 특별히 필요한 경우에만 식별 관계를 사용하는 것이 일반적입니다.

1. 유연성: 비식별 관계는 비즈니스 요구사항 변경에 더 유연하게 대응할 수 있습니다
2. 단순성: 비식별 관계는 주로 대리 키를 사용하여 단일 컬럼으로 기본 키를 구성할 수 있어 매핑이 더 간단합니다[2] [3].
3. 성능: 식별 관계를 사용하면 복합 키가 필요한 경우가 많아 SQL 성능이 저하될 수 있습니다.
4. JPA 지원: JPA는 @GeneratedValue와 같은 편리한 대리 키 생성 방법을 제공하여 비식별 관계 사용을 쉽게 만듭니다.
5. 조인 효율성: 필수적 비식별 관계를 사용하면 내부 조인을 사용할 수 있어 조인 효율성이 높아집니다.

▼ 아직도 안 끝났다..2 그래서 비식별관계의 무결성 관리 → 제약조건 적극 활용.

1. 애플리케이션 로직 구현
2. 데이터베이스 레벨에서 CHECK 제약 조건을 사용하여 데이터의 유효성을 검증합니다.
3. 외래 키 제약 조건: 비식별 관계에서도 외래 키를 사용하여 참조 무결성을 유지할 수 있습니다.
4. NULL 제약 조건
5. 유니크 제약 조건
6. 트리거 사용: 데이터베이스 트리거를 활용하여 데이터 변경 시 자동으로 무결성 검사를 수행할 수 있습니다.
7. 정기적인 데이터 감사: 주기적으로 데이터를 검토하고 정합성을 확인하는 프로세스를 구축합니다.