

## JPA 연관관계 없는 조인(Non-relationship Join) 개념

JPA에서는 일반적으로 엔티티 간 연관관계(@OneToMany, @ManyToOne 등)가 있을 때 조인을 쉽게 할 수 있습니다. 하지만 실무에서는 외래키로 직접 연결되어 있지 않거나, 엔티티 연관관계가 없는 두 테이블을 조인해야 하는 경우가 자주 있습니다. 이를 "연관관계 없는 조인" 또는 "세타 조인(Theta Join)"이라고 부릅니다.<sup>[1] [2] [3]</sup>.

영어 원문: "In JPA, you can join tables even if there is no foreign key between them by specifying the desired columns in JPQL, similar to SQL."

한글 해석: "JPA에서는 SQL처럼 원하는 컬럼을 JPQL에 지정함으로써 외래키가 없어도 테이블을 조인할 수 있습니다."<sup>[1]</sup>

## 연관관계 없는 조인 방식

### 1. JPQL에서의 연관관계 없는 조인

- JPQL에서는 ON 절을 사용하여 연관관계가 없는 엔티티 간에도 조인이 가능합니다.
- 외래키가 없어도, 조인 조건을 명시적으로 작성하면 됩니다.

#### 예시 코드 (JPQL)

```
// Player와 Team이 연관관계가 없다고 가정
SELECT p FROM Player p LEFT JOIN Team t ON p.name = t.name
```

- 위 쿼리는 Player와 Team이 연관관계가 없어도, 이름이 같은 경우 조인합니다.<sup>[4]</sup>.

### 2. Querydsl에서의 연관관계 없는 조인

- Querydsl에서는 join(엔티티).on(조건) 형태로 SQL과 유사하게 조인 조건을 명시할 수 있습니다.
- 연관관계가 없어도 자유롭게 조인할 수 있으며, 외부 조인도 지원합니다.

#### 예시 코드 (Querydsl)

```
// 예시: Academy와 Teacher가 연관관계가 없다고 가정
List<AcademyTeacher> result = queryFactory
    .select(Projections.fields(AcademyTeacher.class,
        academy.name.as("academyName"),
        teacher.name.as("teacherName")
    ))
    .from(academy)
    .join(teacher).on(academy.id.eq(teacher.academyId))
    .fetch();
```

- 위 코드는 Academy와 Teacher 엔티티가 직접적인 연관관계가 없어도, id 값이 일치하는 경우 조인합니다<sup>[2]</sup>.

## 또 다른 예시 (외부 조인)

```
List<Tuple> result = queryFactory
    .select(member, team)
    .from(member)
    .leftJoin(team).on(team.name.eq(member.username))
    .fetch();
```

- 연관관계가 없는 엔티티 간 외부 조인 시 반드시 on절로 조인 조건을 명시해야 합니다<sup>[3]</sup>.

## 주의사항 및 실무 팁

- 연관관계가 없는 조인을 사용할 때는 항상 on 절로 명확한 조인 조건을 작성해야 합니다. 그렇지 않으면 예외가 발생할 수 있습니다<sup>[3]</sup>.
- 결과를 엔티티로 반환하는 것이 아니라, DTO(Projection)로 조회하는 것이 일반적입니다. 연관관계가 없으므로 새로운 DTO로 결과를 받아야 합니다<sup>[5]</sup>.
- Querydsl, JPQL 모두 연관관계 없는 조인을 지원하지만, Querydsl이 타입 안정성, 가독성, 유지보수 측면에서 더 선호되는 경우가 많습니다<sup>[5]</sup>.

## 용어 정리

용어	설명
연관관계 조인	엔티티에 @OneToMany, @ManyToOne 등으로 직접 매핑된 필드를 통한 조인
연관관계 없는 조인	엔티티 간 매핑이 없고, SQL의 on절 또는 where절을 통해 조건을 명시하는 조인
세타 조인	연관관계가 없는 두 테이블을 where절 조건으로 조인하는 방식 (Cross Join)

## 결론

- JPA/Querydsl에서는 연관관계가 없는 엔티티(테이블) 간에도 조인이 가능하며, SQL처럼 원하는 컬럼 기준으로 자유롭게 조인할 수 있습니다.
- 실무에서는 복잡한 연관관계 대신, 필요한 경우에만 명시적으로 조인을 사용하고 결과를 DTO로 매핑하는 패턴이 많이 활용됩니다<sup>[6] [5]</sup>.
- Querydsl의 join().on() 패턴을 사용하면 연관관계가 없는 조인도 손쉽게 구현할 수 있습니다<sup>[2] [3] [8]</sup>.

## 참고 예시 코드 (Querydsl, 주석 포함)

```
// 회원과 게시판이 연관관계가 없다고 가정
public List<PointRankingResponse> findTopRankPoint(long count) {
    return queryFactory
        .select(Projections.constructor(PointRankingResponse.class,
            memberPoint.id,           // 회원 포인트 ID
```

```

        memberPoint.member.id,    // 회원 ID
        memberPoint.point,        // 포인트
        board.member.id.count()   // 게시글 수
    ))
    .from(memberPoint)
    .leftJoin(board).on(memberPoint.member.id.eq(board.member.id)) // 연관관계 없는 조인
    .groupBy(board.member.id, memberPoint.point, memberPoint.id)
    .orderBy(memberPoint.point.desc())
    .fetch();
}

```

- 위 예시는 회원 포인트와 게시글 수를 연관관계 없이 조인하여 DTO로 반환하는 실전 패턴입니다 [\[9\]](#).

## 요약

JPA에서도 연관관계가 없는 테이블 간 조인이 가능하며, JPQL의 ON절 또는 Querydsl의 join().on()을 활용해 SQL과 유사하게 조인 조건을 직접 지정하면 됩니다. 결과는 보통 DTO로 매핑합니다.

(영어 원문: "In JPA, you can join tables even if there is no foreign key between them by specifying the desired columns in JPQL, similar to SQL." / 한글 해석: "JPA에서는 SQL처럼 원하는 컬럼을 JPQL에 지정함으로써 외래키가 없어도 테이블을 조인할 수 있습니다.") [\[1\]](#) [\[2\]](#) [\[3\]](#) [\[5\]](#) [\[9\]](#)



# 연관관계가 있으면 jpa 메소드를 사용하고 연관관계가 없으면 jpql이나 querydsl로 직접 조인해주는거야?

네, 맞습니다.

JPA에서 **연관관계가 설정되어 있으면 엔티티의 필드와 JPA 메서드(예: find, get, 엔티티 내 네비게이션 조회 등)를 통해 객체지향적으로 조인된 데이터를 쉽게 조회**할 수 있습니다.

예를 들어, @ManyToOne, @OneToMany, @OneToOne 등으로 엔티티 간에 연관관계를 맺어두면, 단순히 엔티티의 필드 접근만으로도 연관된 데이터를 조회할 수 있습니다 [\[10\]](#) [\[11\]](#).

이런 경우에는 굳이 JPQL이나 Querydsl의 조인문을 직접 작성하지 않아도 됩니다.

반면에, **연관관계가 없는 엔티티끼리 조인이 필요할 때는 JPQL이나 Querydsl 등으로 직접 조인 조건을 명시해서 쿼리를 작성해야 합니다.**

JPA 2.1 이상에서는 JPQL의 JOIN ON 구문, Querydsl에서는 join(엔티티).on(조건) 형태로 SQL과 비슷하게 조인 조건을 직접 지정할 수 있습니다 [\[12\]](#) [\[13\]](#).

이 방식은 엔티티 간에 직접적인 연관관계가 없거나, 일시적으로 특정 조건으로만 데이터를 조합해서 보고 싶을 때 주로 사용합니다.

## 정리

상황	사용 방법
엔티티 간 연관관계가 있을 때	JPA 메서드 및 엔티티 필드 네비게이션으로 조회 (객체지향 방식)
엔티티 간 연관관계가 없을 때	JPQL의 JOIN ON, Querydsl의 join().on() 등으로 직접 조인 조건 작성

## 예시 코드 (Querydsl, 연관관계 없는 조인)

```
// Academy와 Teacher가 연관관계가 없을 때
queryFactory
    .select(Projections.fields(AcademyTeacher.class,
        academy.name.as("academyName"),
        teacher.name.as("teacherName")
    ))
    .from(academy)
    .join(teacher).on(academy.id.eq(teacher.academyId))
    .fetch();
```

- 영어 원문:  
"In recent versions, Querydsl supports joining entities without relationships using `join(entity).on(condition)`, similar to SQL."
- 한글 해석:  
"최근 버전에서는 Querydsl이 SQL처럼 `join(entity).on(condition)` 방식으로 연관관계 없는 엔티티 간 조인을 지원합니다." [12]

즉, 연관관계가 있으면 JPA의 객체지향적인 방식으로, 없으면 JPQL/Querydsl 등으로 직접 조인 조건을 명시해서 조회하면 됩니다.

✱

## 연관관계가 없는 테이블끼리 @JoinColumn 하게 되면 어떻게 돼

연관관계가 없는 테이블끼리 @JoinColumn을 사용하면, JPA가 정상적으로 동작하지 않습니다.

### 이유와 동작 원리

- @JoinColumn은 외래키(Foreign Key) 컬럼을 통해 두 엔티티(테이블)가 실제로 연관관계가 있을 때 사용합니다.
- 즉, @JoinColumn은 반드시 @ManyToOne, @OneToOne, @OneToMany, @ManyToMany 등과 함께 사용되어야 하며, 두 엔티티가 외래키로 연결되어 있어야 합니다 [14] [15] [16].

### 만약 연관관계가 없는 테이블에 @JoinColumn을 사용하면?

- JPA는 내부적으로 해당 엔티티의 필드를 외래키로 인식하려고 시도합니다.
- 실제 DB에 외래키 제약조건이 없거나, 엔티티 간에 @ManyToOne 등 관계 어노테이션 없이 @JoinColumn만 사용하면, JPA는 매핑 정보를 제대로 해석하지 못하고 예외가 발생하거나, 의도한 대로 동작하지 않습니다 [17] [15].
- 예를 들어, 단순히 @JoinColumn만 선언하면 JPA는 어떤 엔티티와의 관계인지 알 수 없기 때문에 매핑 자체가 불가능합니다.

### 정리

- 연관관계가 없는 테이블끼리는 @JoinColumn을 사용할 수 없습니다.

- 반드시 연관관계 어노테이션(@ManyToOne 등)과 함께 사용해야 하며, 실제 DB에도 외래키가 존재해야 정상적으로 동작합니다.<sup>[14] [15] [16]</sup>.
- 연관관계가 없는 테이블을 조인하려면 JPQL이나 Querydsl에서 직접 on절로 조인 조건을 명시해서 쿼리를 작성해야 합니다.<sup>[17] [18]</sup>.

## 예시 코드 (잘못된 사용)

```
// 연관관계 어노테이션 없이 @JoinColumn만 사용 (잘못된 예)
@Entity
public class A {
    @Id
    private Long id;

    @JoinColumn(name = "B_ID") // 관계 어노테이션이 없음
    private Long bId; // 단순히 외래키 값을 저장하는 필드
}
```

- 위 코드는 JPA가 bId 필드를 외래키로 인식하지 못하고, 단순한 컬럼으로 처리합니다. 실제로 연관 엔티티를 참조하지 않으므로, 객체지향적인 연관관계 매핑이 되지 않습니다.

## 올바른 사용 예시

```
@Entity
public class A {
    @Id
    private Long id;

    @ManyToOne
    @JoinColumn(name = "B_ID")
    private B b; // 엔티티 B와의 연관관계가 명확히 선언됨
}
```

- 이렇게 해야 JPA가 두 엔티티의 관계를 인식하고, 외래키를 통한 조인 및 연관관계 매핑이 가능합니다.

## 요약

- 연관관계가 없는 테이블끼리는 @JoinColumn을 사용할 수 없으며, JPA가 정상적으로 매핑하지 못합니다.
- 반드시 연관관계 어노테이션과 함께 사용해야 하며, 연관관계가 없으면 JPQL/Querydsl로 직접 조인 조건을 작성해야 합니다.<sup>[14] [17] [15] [16]</sup>.

# 연관관계가 db에는 설정되어있지 않지만 실제 컬럼끼리 조인할 수 있어서 @ManyToOne 관계를 설정해서 지연로딩을 설정했을 경우 예상 동작을 알려줘

힌트: 제대로 동작하지 않음

DB에 실제 외래키(FK) 제약조건이 없지만, JPA 엔티티에서 컬럼 값이 조인될 수 있다는 이유로 @ManyToOne과 지연로딩(FetchType.LAZY)을 설정하면, **정상적으로 동작하지 않거나 예기치 않은 문제가 발생할 수 있습니다.**

## 예상 동작 및 문제점

- JPA는 @ManyToOne 어노테이션이 붙은 필드를 "연관관계 매핑"으로 인식하고, 내부적으로 외래키를 통한 객체 매핑을 시도합니다.
- 하지만 DB에 외래키 제약조건이 실제로 없으면, JPA는 단순히 컬럼 값만 보고 조인을 시도할 뿐, 데이터 정합성이나 참조 무결성을 보장하지 못합니다<sup>[19] [20] [21]</sup>.
- 엔티티를 조회할 때 해당 연관 엔티티는 프록시 객체(Proxy)로 반환되며, 실제 데이터를 사용할 때 (예: `getUser().getName()` 등) 추가 쿼리가 실행되어 데이터를 가져오려고 시도합니다<sup>[22]</sup>.
- 만약 연관된 값이 존재하지 않거나, 잘못된 값이 들어 있다면 프록시 초기화 시 예외가 발생하거나 null이 반환될 수 있습니다.

## 구체적 현상 예시

- 연관관계가 없는 상태에서 @ManyToOne을 사용하면, DB에는 외래키 제약이 없으므로, 잘못된 값이 들어가도 JPA는 이를 알지 못합니다.
- 프록시 객체를 통해 실제 엔티티를 로딩할 때, 해당 값이 없으면 `LazyInitializationException`, `EntityNotFoundException` 등이 발생할 수 있습니다.
- 만약 DB에 값이 존재하면 정상적으로 동작할 수 있지만, 데이터 정합성은 전혀 보장되지 않습니다.
- 실제로는 연관관계가 없는 테이블이므로, JPA의 객체 그래프 탐색이나 연관관계 기반 쿼리 최적화 기능을 제대로 활용할 수 없습니다.

## 예시 코드

```
@Entity
public class Photo {
    @Id
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "user_id") // 실제 FK 제약 없음
    private User user;
}
```

- 위 코드에서 Photo.user\_id와 User.id가 DB에서 FK로 연결되어 있지 않다면, user 필드는 단순히 값만 매핑되고, 실제로 User 엔티티가 없는 값이 들어가도 오류가 발생하지 않습니다.
- user를 사용할 때 프록시 객체가 반환되며, 실제 User 테이블에 해당 id가 없으면 예외가 발생할 수 있습니다.<sup>[22]</sup>.

## 결론 및 권장 사항

- DB에 외래키 제약이 없다면 JPA의 연관관계 매핑(@ManyToOne 등)은 데이터 정합성, 참조 무결성, 객체 그래프 탐색 등에서 신뢰할 수 없습니다.
- 이런 경우에는 JPA의 연관관계 매핑을 사용하지 말고, 단순히 컬럼 값(Long userId 등)으로만 관리하거나, 필요할 때 JPQL/Querydsl로 직접 조인해서 데이터를 조회하는 것이 안전합니다.
- 만약 반드시 연관관계 매핑이 필요하다면, DB에도 외래키 제약조건을 추가하는 것이 가장 안전한 방법입니다.

영어 원문: "The problem is that I want to be able to put a value in the column that doesn't necessarily have a corresponding row in the foreign table, which wouldn't be permitted by a foreign key constraint."

한글 해석: "문제는 외래키 제약이 없으므로 실제로 참조 테이블에 없는 값도 컬럼에 넣을 수 있다는 점입니다. 이는 외래키 제약이 있으면 허용되지 않습니다."<sup>[20]</sup>

### 요약:

DB에 외래키 제약이 없는 상태에서 @ManyToOne과 LAZY를 사용하면,

- 데이터 정합성 보장 불가
- 프록시 객체가 반환되어 실제 접근 시 예외 발생 가능
- 객체 그래프 탐색 등 JPA의 장점을 활용할 수 없음
- 실무에서는 이런 상황을 피하고, 컬럼 값으로만 관리하거나 직접 쿼리로 조인하는 것이 안전합니다.



## DB에 외래키 제약이 없는 상태에서 @ManyToOne과 LAZY를 사용하면

LAZY를 설정한 컬럼이 EAGER 처럼 실제 동작해 해당 내용과 관련해서 자료 찾아줘

DB에 외래키 제약이 없는 상태에서 @ManyToOne(fetch = FetchType.LAZY)를 사용하면, LAZY로 설정해도 실제로는 EAGER처럼 동작하는 현상이 발생할 수 있습니다.

### 왜 이런 현상이 발생하는가?

- JPA의 fetch 전략에서 LAZY는 "프록시 객체를 반환하고, 실제로 엔티티가 사용될 때 쿼리를 날린다"는 의미입니다.
- 하지만 LAZY는 "힌트(hint)"일 뿐, JPA 구현체(Hibernate 등)가 내부적으로 프록시를 생성할 수 없는 상황에서는 EAGER처럼 동작할 수 있습니다.<sup>[23]</sup>.

- 특히 DB에 외래키 제약이 없거나, 연관관계 매핑이 애매한 경우에는 프록시 객체로 감쌀 수 없는 상황이 발생합니다.
- 이럴 때 JPA는 연관 엔티티를 즉시 조회(EAGER)해서 반환하는 쿼리를 실행할 수 있습니다.

## 관련 자료 및 설명

영어 원문:

"The LAZY strategy is a hint to the persistence provider runtime that data should be fetched lazily when it is first accessed."

한글 해석:

"LAZY 전략은 데이터가 처음 접근될 때 지연해서 로딩하라는 퍼시스턴스 제공자에게 주는 힌트일 뿐입니다." [23]

- 즉, LAZY는 반드시 보장되는 동작이 아니라 상황에 따라 구현체가 EAGER로 동작시킬 수도 있습니다.

특히,

- **@OneToOne, @ManyToOne** 연관관계에서 외래키가 null이 될 수 있거나, DB에 외래키 제약이 없는 경우
- 프록시 객체로 감쌀 수 없는 상황이 발생하면
- JPA는 실제 엔티티를 즉시 조회(EAGER)해서 반환할 수 있습니다 [24] [25].

영어 원문:

"When the owner of the relationship is not the one being called, lazy loading does not work and eager loading is triggered. This is because a proxy cannot wrap null, so a query is executed to check if the referenced object is null or not."

한글 해석:

"연관관계의 주인이 아닌 쪽에서 호출하면 지연 로딩이 동작하지 않고 즉시 로딩이 동작합니다. 이는 프록시가 null을 감쌀 수 없기 때문에, 참조 객체가 null인지 확인하기 위해 쿼리가 실행되는 것입니다." [24] [25]

## 결론

- DB에 외래키 제약이 없고, JPA 연관관계 매핑만 있는 경우 LAZY로 설정해도 실제로는 EAGER처럼 동작할 수 있습니다.
- 이는 JPA 구현체가 프록시 객체를 생성할 수 없는 상황에서 발생하며, 이 경우 연관 엔티티를 즉시 조회하는 쿼리가 실행됩니다.
- 실무에서는 외래키 제약이 없는 상태에서 연관관계 매핑과 LAZY를 혼용하는 것을 피하는 것이 좋습니다.

## 참고 요약

상황	예상 동작
외래키 제약 있음 + LAZY	정상적으로 프록시 객체 반환, 실제 사용 시 쿼리 실행
외래키 제약 없음 + LAZY	프록시 객체 생성 불가, EAGER처럼 즉시 쿼리 실행



## 관련 참고자료

- "LAZY 전략은 힌트일 뿐, 구현체가 상황에 따라 EAGER로 동작시킬 수 있다." [23]
- "프록시가 null을 감쌀 수 없어 즉시 로딩 쿼리가 실행된다." [24] [25]



## 질문 요약

- **EAGER & findById**: 쿼리가 join으로 한 번에 조회됨
- **EAGER & findAll**: 각각 추가 쿼리가 발생(N+1 문제)
- 이 차이가 왜 발생하는가?

## 답변

### 1. EAGER & findById 쿼리: join으로 한 번에 조회되는 이유

- findById로 특정 엔티티(PK 기준) 하나만 조회할 때, JPA 구현체(Hibernate)는 연관된 엔티티(예: Team의 members)를 즉시로딩(EAGER) 조건에 따라 \*\*join 쿼리(LEFT OUTER JOIN 등)\*\*로 한 번에 가져오는 최적화를 적용합니다.
- 이 경우, Team과 Member를 한 번의 쿼리로 모두 조회할 수 있으므로 추가 쿼리가 발생하지 않습니다.

### 2. EAGER & findAll 쿼리: 각각 추가 쿼리가 발생하는 이유

- findAll로 여러 엔티티(예: 여러 Team)를 한 번에 조회할 때, JPA는 각 Team 엔티티를 먼저 모두 select합니다.
- 그리고 EAGER 조건에 따라 각 Team의 members를 각각 추가 쿼리로 조회합니다.  
예를 들어 Team이 3개면, Team 전체를 select하는 쿼리 1번 + Team별 members를 select하는 쿼리 3번이 실행됩니다.
- 이 현상은 바로 **N+1 문제**입니다.  
1개의 메인 쿼리(N=Team 개수) + N개의 추가 쿼리(각 Team의 members).

### 3. 왜 이런 차이가 발생하는가?

- JPA는 1:N 관계에서 컬렉션(예: List<Member>)을 join으로 한 번에 가져오면, 데이터가 중복되어 비효율적이거나 메모리 사용이 급증할 수 있기 때문에 기본적으로 추가 쿼리 방식(각 Team별 select)을 선택합니다[26].
- 반면, PK로 단일 엔티티를 조회할 때는 데이터 중복 문제가 없으므로 join을 사용해도 효율적이어서 한 번에 가져옵니다.
- 즉, 여러 엔티티를 한 번에 조회(findAll)할 때는 join을 사용하지 않고, 각 엔티티별로 추가 쿼리를 날리는 것이 JPA의 기본 동작입니다.

## 4. 공식 문서 및 블로그의 설명 요약

"Team 엔티티를 가져올 때, EAGER 로딩이기 때문에 Team의 데이터를 즉시 가져오기 위해 속한 대상 엔티티인 Member 엔티티를 완성시키기 위해서 앞서 봤던 것처럼 Join을 수행하지 않고 추가 쿼리를 날려서 N+1 문제가 똑같이 발생하는 것을 볼 수 있습니다."

"TeamRepository.findById(1L);

위의 쿼리 실행 결과를 보면, Team 엔티티를 조회할 때 Member 테이블을 left join을 사용하여 한 번에 가져오는 것을 알 수 있습니다." [26]

### 결론

- **findById**: 단일 엔티티 조회 시에는 join 쿼리로 한 번에 연관 엔티티까지 조회함 → 추가 쿼리 없음.
- **findAll**: 여러 엔티티 조회 시에는 각 엔티티별로 추가 쿼리(N+1)가 발생함 → join을 사용하지 않음.
- 이는 JPA가 1:N 관계에서 데이터 중복 및 비효율을 피하기 위해 기본적으로 선택한 전략이다 [26].

### 참고: N+1 문제 해결법

- **fetch join** 또는 **@EntityGraph**를 사용하여 join으로 한 번에 조회 가능
- **@BatchSize** 또는 **subselect** 등도 활용 가능

[26]

✱

## id로 찾을때 하나밖에 안나오거나 하니 한번에 조인해서 가져오는거고

findAll의 경우 실제 사용전까진 로딩안한다는 jpa 정책에 따라 추후에 로딩해오는건가 eager이여도 ?

네, 질문하신 내용을 정리하면 다음과 같습니다.

- **id로 찾을때(findById)**: 하나의 엔티티만 조회하므로 JPA 구현체(Hibernate 등)는 EAGER 관계를 join 쿼리로 한 번에 조회하는 최적화를 적용할 수 있습니다. 이때는 Team과 Member를 한 번의 쿼리로 모두 가져오기도 합니다 [27].
- **findAll로 전체를 조회할때**: 여러 엔티티(예: 여러 Team)를 한 번에 조회하므로, EAGER라도 연관된 컬렉션(예: members)은 실제로 사용하기 전까지 프록시 객체로 남아 있다가, 사용 시점에 추가 쿼리(N+1)가 발생합니다. 즉, EAGER라도 findAll에서는 컬렉션 필드를 바로 join해서 한 번에 가져오지 않고, 각각 추가 쿼리를 실행하게 됩니다 [28] [27].

## 왜 이런 차이가 발생하는가?

- JPA의 EAGER 전략은 "즉시 로딩"이지만, 1:N 컬렉션 관계에서 여러 엔티티를 한 번에 join으로 모두 가져오면 데이터 중복, 비효율, 메모리 사용 증가 등의 문제가 발생할 수 있습니다.
- 그래서 findAll처럼 여러 엔티티를 조회할 때는 연관된 컬렉션을 프록시 객체로 남겨두고, 실제로 사용할 때 추가 쿼리로 가져오는 방식(N+1)이 기본 동작입니다<sup>[28]</sup> <sup>[27]</sup>.
- 즉, EAGER라고 해도 findAll에서는 내부적으로 "실제 사용 시점"에 추가 쿼리가 나가도록 구현되어 있습니다.
- findById처럼 단일 엔티티 조회에서는 join으로 한 번에 가져와도 데이터 중복 문제가 없으니, 이 경우만 join 최적화가 적용됩니다<sup>[27]</sup>.

## 참고 자료에서의 설명

"EAGER 로딩처럼 한번에 추가 쿼리가 3개가 나가는 것이 아니라, for문을 돌면서 Member의 닉네임을 출력하는, 즉 Member 엔티티가 사용되는 시점에 추가 쿼리가 발생하여 추가 쿼리 발생 후 다음 추가 쿼리 발생 전에 출력이 이루어지는 것을 볼 수 있습니다.

이렇게 데이터를 사용하는 시점에 불러오는 것이 LAZY 로딩, 지연 로딩입니다."

([ksh-coding.tistory.com/146](https://ksh-coding.tistory.com/146))<sup>[27]</sup>

"EAGER 사용 + findAll() = 항상 N+1 발생"

([stir.tistory.com/173](https://stir.tistory.com/173))<sup>[28]</sup>

## 결론

- **findById**: EAGER 관계가 join으로 한 번에 조회됨(데이터 중복 문제 없음)
- **findAll**: EAGER라도 컬렉션 연관관계는 실제 사용 전까지 프록시로 남아 있다가, 사용할 때 추가 쿼리(N+1) 발생
- 즉, EAGER라고 해도 findAll에서는 실제 사용 시점에 추가 쿼리가 나가는 것이 JPA의 기본 동작입니다.

이 차이를 이해하고, 대량 조회 시에는 fetch join이나 @EntityGraph 등을 통해 N+1 문제를 해결하는 것이 실무에서 중요합니다.



## findAll에서는 그럼 연관관계 컬럼을 메인테이블 전부 조회 후 그제서야 조회하는 형태로 eager 이여도 저런 형태로 동작하는거야?

네, 맞습니다.

findAll()을 사용할 때 EAGER로 설정된 연관관계 컬럼(@ManyToOne, @OneToMany 등)은 메인 테이블(예: Team) 전체를 먼저 조회한 뒤, 각 엔티티별로 연관된 엔티티(예: Member)를 추가 쿼리로 조회하는 형태로 동작합니다. 즉, EAGER라고 해도 findAll에서는 연관관계 컬럼을 한 번에 join해서 모두 가져오지 않고, 각각 추가 쿼리를 실행하게 됩니다.

## 동작 과정 요약

1. **findAll()** 실행 시, 메인 테이블(**Team** 등) 전체를 **select**로 조회
2. 각 **Team** 엔티티별로 **EAGER** 연관관계인 **Member**를 추가 쿼리로 조회 (**N+1 문제 발생**)
  - 예: Team이 3개면, Team 전체 select 1번 + 각 Team별 member select 3번 = 총 4번 쿼리

## 왜 이런 동작이 발생하는가?

- JPA는 EAGER 옵션이더라도, findAll처럼 여러 엔티티를 한 번에 조회할 때 join을 사용하지 않고, 각 엔티티별로 연관관계 데이터를 추가 쿼리로 조회합니다.
- 이는 join으로 한 번에 조회할 경우, 데이터 중복이나 메모리 사용 급증 등 비효율이 발생할 수 있기 때문입니다<sup>[29] [30]</sup>.
- 그래서 JPA 기본 동작은 메인 테이블을 모두 조회한 뒤, 각 엔티티의 EAGER 연관관계를 추가 쿼리로 가져오는 방식입니다.

## 실제 예시 및 자료 인용

- "findAll()을 한 순간 select t from Team t 이라는 JPQL 구문이 생성되고 해당 구문을 분석한 select \* from team 이라는 SQL이 생성되어 실행된다.  
... team과 연관되어 있는 member도 로딩을 해야 한다. ... team 인스턴스들 개수에 맞게 select \* from member where team\_id = ? 이라는 SQL 구문이 생성된다. (N+1 발생)"<sup>[29]</sup>
- "Team 엔티티를 가져올 때, EAGER 로딩이기 때문에 Team의 데이터를 즉시 가져오기 위해 속한 대상 엔티티의 데이터를 추가 쿼리로 가져온다. (N+1 문제)"<sup>[30]</sup>

## 결론

- EAGER여도 findAll에서는 메인 테이블 전체를 먼저 조회하고, 연관관계 컬럼은 추가 쿼리로 각각 조회하는 방식으로 동작합니다.
- 이로 인해 N+1 문제가 발생할 수 있으며, 이를 해결하려면 fetch join, @EntityGraph 등을 활용해야 합니다.

## 참고: N+1 문제 해결 방법

- JPQL의 fetch join 사용
- @EntityGraph 어노테이션 사용

### 요약:

findAll에서는 EAGER여도 연관관계 컬럼을 join으로 한 번에 가져오지 않고, 메인 테이블 전체를 조회한 후 각 엔티티별로 추가 쿼리를 실행하는 것이 기본 동작입니다<sup>[29] [30]</sup>.

## 질문 요약

주테이블에서 상대테이블에 @JoinColumn을 걸 때, 상대테이블의 PK가 아닌 유니크 컬럼(예: unique index가 걸린 컬럼)과 조인을 걸고

@ManyToOne(fetch = FetchType.LAZY)로 설정했음에도 실제로는 EAGER처럼 동작하는 이유

## 답변

### 1. JPA의 연관관계 기본 원리

- JPA에서 @ManyToOne 연관관계는 보통 주테이블(FK) → 상대테이블(PK)로 매핑합니다.
- 이때 LAZY 전략이면, 실제로 연관 엔티티를 사용할 때까지 프록시 객체만 반환하고, DB 조회는 미룹니다.<sup>[31] [32]</sup>.
- 하지만, 조인 대상 컬럼이 PK가 아니라 unique 컬럼일 때는 동작 방식이 달라질 수 있습니다.

### 2. PK가 아닌 unique 컬럼과 조인 시 동작 변화

- JPA는 연관관계 매핑 시, 상대 테이블의 PK가 아닌 unique 컬럼으로 @JoinColumn(referencedColumnName="unique\_col")을 지정해도 DB 레벨에서는 외래키 제약이 아닌 단순한 unique 인덱스만 존재합니다.<sup>[33]</sup>.
- 이 경우, JPA 구현체(Hibernate 등)는 프록시 객체로 감쌀 수 없는 상황이 발생할 수 있습니다.
- 특히, 프록시 객체는 "PK값"을 기반으로 동작하는데, PK가 아닌 컬럼을 기준으로 조인하면 프록시 객체 생성이나 식별에 제약이 생깁니다.

### 3. 실제로 EAGER처럼 동작하는 이유

- **프록시 객체의 한계:**  
프록시 객체는 엔티티의 PK값을 알아야 생성 및 식별이 가능합니다.  
PK가 아닌 unique 컬럼을 사용할 경우, 프록시 객체가 제대로 동작하지 않아,  
JPA는 연관 엔티티를 프록시로 감싸지 않고, 즉시 실제 엔티티를 조회(EAGER)하는 쿼리를 실행합니다.<sup>[32]</sup>.
- **JPA 구현체의 정책:**  
JPA 명세상 fetch=LAZY는 "힌트"일 뿐, 실제 동작은 구현체(Hibernate 등)가 결정합니다.  
프록시로 감쌀 수 없는 상황(대표적으로 PK가 아닌 컬럼 조인)에서는 즉시 로딩(EAGER)로 동작할 수 있습니다.
- **결과적으로:**  
LAZY로 설정해도, unique 컬럼 조인 상황에서는 실제로 연관 엔티티를 즉시 조회하는 쿼리가 실행되어 EAGER처럼 보이게 됩니다.

### 4. 공식 자료 및 참고 설명

- "프록시 객체는 실제 엔티티 객체를 상속하고, 실제 엔티티와 동일한 인터페이스를 구현하여 동일하게 동작하는데, 실제 객체가 아닌 참조 객체이다."  
→ 프록시는 PK 기반으로 동작, PK가 아니면 프록시 사용 불가<sup>[34]</sup>.

- "LAZY 전략은 힌트일 뿐, 프록시 객체를 생성할 수 없는 상황에서는 EAGER로 동작할 수 있다."  
→ 조인 대상이 PK가 아니면, 프록시 불가 → 즉시 로딩<sup>[32]</sup>.

## 결론

- JPA에서 @ManyToOne + @JoinColumn(referencedColumnName="unique\_col")로 PK가 아닌 unique 컬럼을 조인하면,  
프록시 객체의 한계로 인해 fetch=LAZY 전략이 무시되고, 실제로는 EAGER처럼 동작하여 즉시 쿼리가 실행됩니다.
- 이는 JPA/Hibernate가 프록시 생성을 위해 PK값이 필요하지만, unique 컬럼만으로는 프록시를 안전하게 생성할 수 없기 때문입니다.

## 요약

- **PK가 아닌 unique 컬럼 조인 + LAZY**
  - 프록시 객체 생성 불가
  - JPA가 즉시 로딩(EAGER)로 동작
  - 실제로는 연관 엔티티를 바로 쿼리해서 가져옴

"프록시 객체는 엔티티의 PK값을 알아야 생성할 수 있는데, PK가 아닌 컬럼을 기준으로 조인하면 프록시 객체를 만들 수 없어, JPA는 즉시 로딩(EAGER)으로 동작하게 된다."

(영어: "Proxy objects in JPA require the entity's primary key to be created. When joining on a non-PK column, proxies cannot be created, so JPA falls back to eager loading.")

(한글: "JPA의 프록시 객체는 엔티티의 PK값이 있어야 생성할 수 있습니다. PK가 아닌 컬럼으로 조인할 경우 프록시 객체를 만들 수 없어, JPA는 즉시 로딩으로 동작하게 됩니다.")



## PK가 아닌 unique 컬럼과 @ManyToOne LAZY 설정 시 예상 동작 분석

질문하신 상황에서의 예상 동작을 공식 문서와 자료를 바탕으로 자세히 설명드리겠습니다.

### JPA 공식 명세에 따른 기본 원칙

JPA 공식 문서에 따르면, **PK가 아닌 컬럼에 대한 참조는 선택적 기능(optional feature)**입니다<sup>[35]</sup>.

영어 원문: "Support for referenced columns that are not primary key columns of the referenced table is optional. Applications that use such mappings will not be portable."

한글 해석: "참조 테이블의 기본 키가 아닌 컬럼에 대한 참조 지원은 선택적입니다. 이러한 매핑을 사용하는 애플리케이션은 이식성이 보장되지 않습니다."

## 예상 동작 시나리오

## 1. 프록시 객체 생성 실패로 인한 EAGER 동작

```
@Entity
public class Order {
    @Id
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "customer_name", referencedColumnName = "name")
    private Customer customer; // Customer.name은 unique이지만 PK가 아님
}

@Entity
public class Customer {
    @Id
    private Long id;

    @Column(unique = true)
    private String name; // PK가 아닌 unique 컬럼
}
```

### 예상 동작:

- LAZY로 설정했지만 실제로는 **EAGER**처럼 즉시 로딩이 발생
- Order 조회 시 Customer도 함께 조회하는 JOIN 쿼리 실행
- 프록시 객체가 아닌 실제 Customer 엔티티 반환

## 2. 프록시 객체 생성이 불가능한 이유

JPA 구현체(Hibernate)가 프록시 객체를 생성할 수 없는 근본적 이유들 <sup>[36]</sup> <sup>[37]</sup>:

- **영속성 컨텍스트의 제약**: 영속성 컨텍스트는 PK 기반으로 엔티티를 관리하는데, PK가 아닌 컬럼으로는 엔티티 식별이 제한적 <sup>[35]</sup>
- **지연 로딩 지원의 한계**: 지연 로딩은 PK 기반으로 설계되어 있어, PK가 아닌 컬럼으로는 효과적인 지연 로딩이 어려움 <sup>[35]</sup>
- **프록시 객체의 식별자 요구사항**: 프록시 객체는 엔티티의 PK값을 알아야 생성할 수 있는데, PK가 아닌 컬럼만으로는 이를 충족할 수 없음 <sup>[38]</sup>

## 3. N+1 문제 발생 가능성

실무 경험에 따르면, PK가 아닌 컬럼과 조인할 경우 <sup>[36]</sup>:

- **즉시 로딩으로 전환**: LAZY 설정이 무시되고 EAGER처럼 동작
- **추가 쿼리 발생**: 연관 엔티티를 조회하기 위한 별도 쿼리 실행
- **성능 저하**: 의도치 않은 추가 쿼리로 인한 성능 문제

## 4. 공식 해결 방안

JPA 공식 문서와 실무 권장사항<sup>[36]</sup> <sup>[39]</sup>:

1. **@BatchSize 사용**: N+1 문제 완화를 위한 배치 로딩
2. **연관관계 제거**: 엔티티 연관관계 대신 비즈니스 로직에서 관리
3. **PK로 변경**: 가능하다면 연관관계를 PK 기반으로 변경
4. **fetch join 활용**: 필요시 명시적으로 join 쿼리 사용

### 예시 코드 (주석 포함)

```
// 문제가 되는 매핑 - LAZY가 의도대로 동작하지 않음
@Entity
public class OrderItem {
    @Id
    private Long id;

    // PK가 아닌 unique 컬럼과 조인 - 프록시 생성 불가
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "product_code", referencedColumnName = "code")
    private Product product; // Product.code는 unique이지만 PK가 아님

    // 실제 동작: EAGER처럼 즉시 로딩됨
    // 예상 쿼리: SELECT o.*, p.* FROM order_item o LEFT JOIN product p ON o.product_code
}

// 권장 해결책 1: 연관관계 제거 후 비즈니스 로직에서 처리
@Entity
public class OrderItem {
    @Id
    private Long id;

    @Column(name = "product_code")
    private String productCode; // 단순 컬럼으로 저장

    // 필요시 서비스 계층에서 Product 조회
    @Transient
    private Product product;
}

// 권장 해결책 2: @BatchSize로 N+1 문제 완화
@Entity
public class OrderItem {
    @Id
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "product_code", referencedColumnName = "code")
    @BatchSize(size = 10) // 10개씩 배치로 로딩
    private Product product;
}
```



## 요약

PK가 아닌 unique 컬럼과 @ManyToOne(fetch = FetchType.LAZY) 조합의 예상 동작:

1. **LAZY 설정 무시**: 프록시 객체 생성 불가로 인해 EAGER처럼 즉시 로딩
2. **추가 쿼리 발생**: 연관 엔티티 조회를 위한 별도 쿼리 실행
3. **성능 저하**: 의도치 않은 쿼리로 인한 성능 문제
4. **이식성 부족**: JPA 구현체에 따라 동작이 달라질 수 있음

영어 원문: "The issue here in trying to support a JoinColumn to a non-primary key is in 2 main areas: Loading check against the persistence context (which is @id / primary key based) - Support for lazy loading (which is @id / primary key based)"

한글 해석: "PK가 아닌 컬럼에 대한 JoinColumn 지원의 주요 문제는 두 가지입니다: PK 기반으로 동작하는 영속성 컨텍스트의 로딩 확인과 PK 기반으로 설계된 지연 로딩 지원"

따라서 실무에서는 이러한 매핑을 피하고, 대신 명시적인 쿼리나 비즈니스 로직으로 처리하는 것이 안전하고 예측 가능한 방법입니다.



## 올바른 정보 정리: PK가 아닌 unique 컬럼과 @ManyToOne LAZY 설정의 동작

사용자의 지적이 맞습니다. 이전 답변에서 일부 잘못된 정보가 있었으므로, 정확한 정보를 바탕으로 다시 설명드리겠습니다.

### 정확한 동작 원리

DB에 외래키 제약이 없고, PK가 아닌 unique 컬럼을

@JoinColumn(referencedColumnName="unique\_col")로 조인하며, @ManyToOne(fetch = FetchType.LAZY)로 설정했을 때의 실제 동작:

1. **Order.findAll() 실행 시**: Order 테이블만 조회하는 쿼리 1개 실행
2. **Customer 접근 시**: 각 Order별로 Customer 조회하는 개별 추가 쿼리 N개 실행 (N+1 문제)
3. **JOIN 쿼리가 아닌 개별 SELECT 쿼리들이 발생**

### 왜 JOIN이 아닌 개별 쿼리가 발생하는가?

#### 1. JPA/Hibernate의 PK가 아닌 컬럼 조인에 대한 제한사항

- JPA 명세상 PK가 아닌 컬럼에 대한 참조는 선택적 기능(optional feature)입니다<sup>[40]</sup>.
- Hibernate는 PK가 아닌 컬럼을 참조할 때 프록시 객체 생성과 지연 로딩에 제약이 있습니다<sup>[40]</sup>.

영어 원문: "The issue here in trying to support a JoinColumn to a non-primary key is in 2 main areas: Loading check against the persistence context (which is @id / primary key based) - Support for lazy loading (which is @id / primary key based)"

한글 해석: "PK가 아닌 컬럼에 대한 JoinColumn 지원의 주요 문제는 두 가지입니다: PK 기반으로 동작하는 영속성 컨텍스트의 로딩 확인과 PK 기반으로 설계된 지연 로딩 지원"

## 2. 실제 동작 메커니즘

- **LAZY 설정이 유지되지만:** 프록시 객체는 생성됩니다
- **연관 엔티티 접근 시:** PK가 아닌 컬럼으로는 효율적인 배치 로딩이나 JOIN이 어려워, **각각 개별 SELECT 쿼리 실행** <sup>[41]</sup> <sup>[42]</sup> <sup>[43]</sup>
- **결과:** N+1 문제 발생 (JOIN이 아닌 추가 쿼리들)

## 공식 자료 및 실무 경험 증거

### Hibernate 공식 포럼의 설명 <sup>[43]</sup>:

"@ManyToOne (with a reciprocal OneToMany) and Lazy fetching works perfectly if the join column is the primary key. If it's not, lazy loading breaks and Hibernate eagerly fetches all @ManyToOne's every time an object is instantiated"

한글 해석: "조인 컬럼이 기본 키인 경우 @ManyToOne과 지연 로딩이 완벽하게 작동합니다. 그렇지 않으면 지연 로딩이 깨지고 Hibernate가 객체가 인스턴스화될 때마다 모든 @ManyToOne을 즉시 가져옵니다"

### 실무 개발자 경험 <sup>[41]</sup>:

"referencedColumnName을 사용하여 엔티티를 연결한 경우 정체모를 쿼리가 하나 더 나간다"

"이 현상이 왜 발생하는지, 그리고 이를 어떻게 해결하는지에 대한 포스팅이다. (해결방법은 찾지 못했다)"

## 예시 코드 및 예상 쿼리

```
@Entity
public class Order {
    @Id
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY) // LAZY로 설정해도 개별 쿼리 발생
    @JoinColumn(name = "customer_name", referencedColumnName = "name")
    private Customer customer; // Customer.name은 unique이지만 PK가 아님
}

@Entity
public class Customer {
    @Id
    private Long id;

    @Column(unique = true)
    private String name; // PK가 아닌 unique 컬럼
}
```

## 실제 발생하는 쿼리 패턴:

```
-- 1. findAll() 시
SELECT o.* FROM order o;

-- 2. 각 Order의 customer 접근 시 (N+1 문제)
SELECT c.* FROM customer c WHERE c.name = ?; -- Order 1개당 1번씩
SELECT c.* FROM customer c WHERE c.name = ?; -- Order 2개당 1번씩
SELECT c.* FROM customer c WHERE c.name = ?; -- Order 3개당 1번씩
...
```

## 해결 방안

### 1. fetch join 사용 (권장):

```
@Query("SELECT o FROM Order o JOIN FETCH o.customer")
List<Order> findAllWithCustomer();
```

### 2. @BatchSize 사용:

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "customer_name", referencedColumnName = "name")
@BatchSize(size = 10) // 10개씩 배치로 조회
private Customer customer;
```

### 3. 연관관계 제거하고 비즈니스 로직에서 처리:

```
@Column(name = "customer_name")
private String customerName; // 단순 컬럼으로 저장

// 필요시 서비스 계층에서 Customer 조회
```

## 요약

### PK가 아닌 unique 컬럼과 @ManyToOne(fetch = FetchType.LAZY) 조합의 실제 동작:

1. **LAZY 설정은 유지:** 프록시 객체 생성됨
2. **JOIN이 아닌 개별 쿼리:** 연관 엔티티 접근 시 각각 SELECT 쿼리 실행
3. **N+1 문제 발생:** Order가 N개면 Customer 조회 쿼리도 N개 추가 실행
4. **성능 저하:** 의도치 않은 다수의 개별 쿼리로 인한 성능 문제

영어 원문: "What started out as a single query for 1000 records can balloon into 5000 queries to eagerly fetch a variety of @ManyToOne's using individual selects."

한글 해석: "1000개 레코드에 대한 단일 쿼리로 시작된 것이 개별 SELECT를 사용하여 다양한 @ManyToOne을 즉시 가져오는 5000개의 쿼리로 급증할 수 있습니다."

따라서 실무에서는 PK가 아닌 컬럼을 조인할 때 이런 성능 문제를 인지하고, fetch join이나 다른 최적화 방법을 적용하는 것이 중요합니다.

# JPA 연관관계 제거 후 비즈니스 로직에서 처리하는 방법들

사용자가 질문하신 코드에서 "비즈니스 로직에서 처리한다"는 것은 **JPA의 연관관계 매핑 (@ManyToOne 등)**을 사용하지 않고, 서비스 계층에서 필요할 때마다 직접 조회하고 설정하는 방식을 의미합니다.

## 주요 처리 방법들

### 1. 서비스 계층에서 Repository를 통한 개별 조회

```
@Service
@Transactional
@RequiredArgsConstructor
public class OrderItemService {

    private final OrderItemRepository orderItemRepository;
    private final ProductRepository productRepository; // Product 조회용 Repository

    public OrderItem createOrderItem(OrderItemCreateRequest request) {
        // 1. OrderItem 생성 (product 연관관계 없이)
        OrderItem orderItem = OrderItem.builder()
            .productCode(request.getProductCode()) // 단순 컬럼으로 저장
            .quantity(request.getQuantity())
            .build();

        OrderItem savedOrderItem = orderItemRepository.save(orderItem);

        // 2. 필요시 Product 정보를 별도 조회하여 @Transient 필드에 설정
        Product product = productRepository.findByCode(request.getProductCode())
            .orElseThrow(() -> new EntityNotFoundException("Product not found"));
        savedOrderItem.setProduct(product); // @Transient 필드에 설정

        return savedOrderItem;
    }

    public OrderItemDetailResponse getOrderItemDetail(Long orderItemId) {
        // 1. OrderItem 조회
        OrderItem orderItem = orderItemRepository.findById(orderItemId)
            .orElseThrow(() -> new EntityNotFoundException("OrderItem not found"));

        // 2. Product 정보를 별도 조회
        Product product = productRepository.findByCode(orderItem.getProductCode())
            .orElse(null); // Product가 없을 수도 있음을 고려

        // 3. @Transient 필드에 설정
        orderItem.setProduct(product);

        // 4. DTO로 변환하여 반환
        return OrderItemDetailResponse.from(orderItem);
    }
}
```

```

    }
}

```

## 2. 배치 조회를 통한 성능 최적화

```

@Service
@RequiredArgsConstructor
public class OrderItemService {

    private final OrderItemRepository orderItemRepository;
    private final ProductRepository productRepository;

    public List<OrderItemDetailResponse> getOrderItemsWithProducts(List<Long> orderItemIds) {
        // 1. OrderItem들을 한 번에 조회
        List<OrderItem> orderItems = orderItemRepository.findAllById(orderItemIds);

        // 2. 모든 productCode를 수집
        Set<String> productCodes = orderItems.stream()
            .map(OrderItem::getProductCode)
            .collect(Collectors.toSet());

        // 3. Product들을 한 번에 배치 조회 (N+1 문제 방지)
        List<Product> products = productRepository.findAllByCodeIn(productCodes);

        // 4. productCode를 key로 하는 Map 생성
        Map<String, Product> productMap = products.stream()
            .collect(Collectors.toMap(Product::getCode, Function.identity()));

        // 5. OrderItem에 해당하는 Product 설정
        orderItems.forEach(orderItem -> {
            Product product = productMap.get(orderItem.getProductCode());
            orderItem.setProduct(product); // @Transient 필드에 설정
        });

        // 6. DTO로 변환하여 반환
        return orderItems.stream()
            .map(OrderItemDetailResponse::from)
            .collect(Collectors.toList());
    }
}

```

## 3. Custom Repository 메서드를 활용한 JOIN 쿼리

```

// Custom Repository 인터페이스
public interface OrderItemRepositoryCustom {
    List<OrderItemWithProductDto> findOrderItemsWithProductByIds(List<Long> orderItemIds)
}

// Custom Repository 구현체
@RequiredArgsConstructor
public class OrderItemRepositoryImpl implements OrderItemRepositoryCustom {

    private final JPAQueryFactory queryFactory;
}

```

```

@Override
public List<OrderItemWithProductDto> findOrderItemsWithProductByIds(List<Long> orderItemIds) {
    return queryFactory
        .select(Projections.constructor(OrderItemWithProductDto.class,
            orderItem.id,
            orderItem.productCode,
            orderItem.quantity,
            product.name,
            product.price
        ))
        .from(orderItem)
        .leftJoin(product).on(orderItem.productCode.eq(product.code)) // 연관관계 없는 엔티티에 대해
        .where(orderItem.id.in(orderItemIds))
        .fetch();
}

// 메인 Repository
public interface OrderItemRepository extends JpaRepository<OrderItem, Long>, OrderItemRepositoryCustom {
    // 기본 JPA 메서드들
}

// 서비스에서 사용
@Service
@RequiredArgsConstructor
public class OrderItemService {

    private final OrderItemRepository orderItemRepository;

    public List<OrderItemWithProductDto> getOrderItemsWithProduct(List<Long> orderItemIds) {
        // Custom Repository의 JOIN 쿼리 사용
        return orderItemRepository.findOrderItemsWithProductByIds(orderItemIds);
    }
}

```

#### 4. @PostLoad를 활용한 자동 설정

```

@Entity
public class OrderItem {
    @Id
    private Long id;

    @Column(name = "product_code")
    private String productCode;

    @Transient
    private Product product;

    // ApplicationContext에서 ProductRepository를 가져오기 위한 static method
    private static ProductRepository getProductRepository() {
        return ApplicationContextProvider.getBean(ProductRepository.class);
    }

    @PostLoad // 엔티티가 DB에서 로드된 후 자동 실행
}

```

```

private void loadProduct() {
    if (this.productCode != null) {
        try {
            this.product = getProductRepository()
                .findByCode(this.productCode)
                .orElse(null);
        } catch (Exception e) {
            // 예외 처리 (Repository를 가져올 수 없는 경우 등)
            this.product = null;
        }
    }
}
}
}

```

## 5. 지연 로딩 패턴 구현

```

@Entity
public class OrderItem {
    @Id
    private Long id;

    @Column(name = "product_code")
    private String productCode;

    @Transient
    private Product product;

    @Transient
    private boolean productLoaded = false; // 로딩 상태 체크용

    // Getter에서 지연 로딩 구현
    public Product getProduct() {
        if (!productLoaded && productCode != null) {
            loadProduct(); // 실제 사용 시점에 로딩
        }
        return product;
    }

    private void loadProduct() {
        try {
            ProductRepository productRepository =
                ApplicationContextProvider.getBean(ProductRepository.class);
            this.product = productRepository.findByCode(this.productCode)
                .orElse(null);
        } catch (Exception e) {
            this.product = null;
        } finally {
            this.productLoaded = true; // 로딩 시도 완료 표시
        }
    }

    // Setter
    public void setProduct(Product product) {
        this.product = product;
        this.productLoaded = true; // 직접 설정된 경우도 로딩 완료로 표시
    }
}

```

```
}  
}
```

## 장단점 비교

### 장점

- **데이터 정합성 제어**: DB 제약이 없어도 비즈니스 로직에서 검증 가능 <sup>[44]</sup> <sup>[45]</sup>
- **유연성**: 필요한 시점에만 연관 데이터 조회 가능
- **성능 최적화**: 배치 조회 등으로 N+1 문제 방지 가능 <sup>[46]</sup> <sup>[47]</sup>
- **트랜잭션 분리**: 각 엔티티별로 독립적인 트랜잭션 관리 가능 <sup>[48]</sup> <sup>[49]</sup>

### 단점

- **개발 복잡도 증가**: 수동으로 연관 데이터를 관리해야 함 <sup>[44]</sup> <sup>[45]</sup>
- **코드 중복**: 여러 서비스에서 비슷한 조회 로직 반복 가능성
- **런타임 에러 위험**: 연관 데이터가 없을 경우 적절한 예외 처리 필요 <sup>[50]</sup> <sup>[51]</sup>

### 결론

비즈니스 로직에서 처리한다는 것은 JPA의 자동 연관관계 매핑 대신, 서비스 계층에서 Repository를 통해 필요한 연관 엔티티를 직접 조회하고 @Transient 필드에 설정하는 방식입니다.<sup>[52]</sup> <sup>[45]</sup> <sup>[53]</sup>. 이 방법은 외래키 제약이 없거나 복잡한 조인 조건이 있을 때 유용하며, 성능 최적화와 데이터 정합성을 개발자가 직접 제어할 수 있는 장점이 있습니다.<sup>[46]</sup> <sup>[47]</sup> <sup>[49]</sup>.



## jpql ,querydsl로 직접조인 하는 방안은 어때

JPQL과 Querydsl로 직접 조인하는 방안은 연관관계가 없는 테이블 간 조인이나 복잡한 조인 조건이 필요할때 매우 효과적인 해결책입니다.

### JPQL을 활용한 직접 조인

#### 1. 세타 조인(Theta Join) 사용

```
// 연관관계 없는 테이블 간 조인  
@Query("SELECT o FROM Order o, Customer c WHERE o.customerName = c.name")  
List<Order> findOrdersWithCustomerByName();  
  
// 또는 카운트 조회  
@Query("SELECT count(o) FROM Order o, Customer c WHERE o.customerName = c.name")  
long countOrdersWithCustomer();
```



## 2. ON절을 활용한 외부 조인

```
// JPA 2.1 이상에서 지원하는 ON절 조인
@Query("SELECT o FROM Order o LEFT JOIN Customer c ON o.customerName = c.name")
List<Order> findOrdersWithCustomerLeftJoin();

// 조건을 추가한 조인
@Query("SELECT o FROM Order o LEFT JOIN Customer c ON o.customerName = c.name AND c.statu
List<Order> findOrdersWithActiveCustomer();
```

## Querydsl을 활용한 직접 조인

### 1. 연관관계 없는 조인 (권장)

```
@Repository
@RequiredArgsConstructor
public class OrderRepositoryImpl implements OrderRepositoryCustom {

    private final JPAQueryFactory queryFactory;

    public List<OrderWithCustomerDto> findOrdersWithCustomer() {
        return queryFactory
            .select(Projections.constructor(OrderWithCustomerDto.class,
                order.id,
                order.customerName,
                customer.name,
                customer.email
            ))
            .from(order)
            .join(customer).on(order.customerName.eq(customer.name)) // 연관관계 없는 조인
            .fetch();
    }

    // 외부 조인도 가능
    public List<OrderWithCustomerDto> findOrdersWithCustomerLeftJoin() {
        return queryFactory
            .select(Projections.constructor(OrderWithCustomerDto.class,
                order.id,
                order.customerName,
                customer.name,
                customer.email
            ))
            .from(order)
            .leftJoin(customer).on(order.customerName.eq(customer.name)) // LEFT JOIN
            .fetch();
    }
}
```

## 2. 복잡한 조인 조건 처리

```
public List<OrderDetailDto> findOrdersWithComplexJoin() {
    return queryFactory
        .select(Projections.constructor(OrderDetailDto.class,
            order.id,
            order.customerName,
            customer.name,
            product.name,
            product.price
        ))
        .from(order)
        .leftJoin(customer).on(order.customerName.eq(customer.name)
            .and(customer.status.eq("ACTIVE"))) // 복잡한 조인 조건
        .leftJoin(product).on(order.productCode.eq(product.code)
            .and(product.isDeleted.eq(false)))
        .where(order.orderDate.goe(LocalDate.now().minusDays(30)))
        .fetch();
}
```

## 3. 동적 조인 조건

```
public List<OrderWithCustomerDto> findOrdersWithDynamicJoin(String customerStatus, String
    BooleanBuilder customerJoinCondition = new BooleanBuilder();
    customerJoinCondition.and(order.customerName.eq(customer.name));

    if (StringUtils.hasText(customerStatus)) {
        customerJoinCondition.and(customer.status.eq(customerStatus));
    }

    return queryFactory
        .select(Projections.constructor(OrderWithCustomerDto.class,
            order.id,
            customer.name,
            product.name
        ))
        .from(order)
        .leftJoin(customer).on(customerJoinCondition) // 동적 조인 조건
        .leftJoin(product).on(order.productCode.eq(product.code)
            .and(productCategory != null ? product.category.eq(productCategory) : null))
        .fetch();
}
```

## 장점과 단점 비교

### 장점

- **유연성**: 연관관계가 없어도 자유롭게 조인 가능 <sup>[54]</sup>
- **성능 최적화**: 한 번의 쿼리로 필요한 모든 데이터 조회 가능
- **N+1 문제 해결**: 추가 쿼리 없이 모든 데이터를 한 번에 가져옴

- **복잡한 조건 처리:** ON절을 통해 복잡한 조인 조건 설정 가능<sup>[55]</sup>
- **타입 안정성:** Querydsl은 컴파일 타임에 오류 검출 가능

## 단점

- **개발 복잡도:** 수동으로 조인 조건을 작성해야 함
- **유지보수:** 테이블 구조 변경 시 쿼리도 함께 수정 필요
- **가독성:** 복잡한 조인일 경우 쿼리가 길어질 수 있음

## 실무 권장사항

### 1. Querydsl 우선 사용

```
// JPQL보다는 Querydsl 권장 (타입 안정성, 가독성)
public List<OrderSummaryDto> getOrderSummary() {
    return queryFactory
        .select(Projections.constructor(OrderSummaryDto.class,
            customer.name.as("customerName"),
            order.id.count().as("orderCount"),
            order.totalAmount.sum().as("totalAmount")
        ))
        .from(order)
        .join(customer).on(order.customerName.eq(customer.name))
        .groupBy(customer.name)
        .orderBy(order.totalAmount.sum().desc())
        .fetch();
}
```

### 2. DTO 프로젝트 활용

```
// 엔티티가 아닌 DTO로 결과 반환 (메모리 효율성)
@Getter
@AllArgsConstructor
public class OrderWithCustomerDto {
    private Long orderId;
    private String customerName;
    private String customerEmail;
    private LocalDateTime orderDate;
    private BigDecimal totalAmount;
}
```

### 3. 배치 조회와 조합

```
// 대량 데이터 처리 시 배치 조회와 조합
public List<OrderDetailDto> findOrdersWithBatchProcessing(List<String> customerNames) {
    return queryFactory
        .select(Projections.constructor(OrderDetailDto.class,
            order.id,
```

```

        customer.name,
        customer.email
    ))
    .from(order)
    .join(customer).on(order.customerName.eq(customer.name))
    .where(customer.name.in(customerNames)) // IN 조건으로 배치 처리
    .fetch();
}

```

## 결론

JPQL과 Querydsl을 활용한 직접 조인은 **연관관계가 없는 테이블 간 조인이나 복잡한 비즈니스 로직이 필요할 때 매우 효과적**입니다<sup>[56] [52] [54]</sup>. 특히 Querydsl의 `join(엔티티).on(조건)` 방식은 SQL과 유사하면서도 타입 안정성을 제공하므로, 실무에서 연관관계 없는 조인을 처리할 때 가장 권장되는 방법입니다.

영어 원문: "QueryDSL supports joining entities without relationships using `join(entity).on(condition)`, similar to SQL."<sup>[57]</sup>

한글 해석: "QueryDSL은 SQL과 유사하게 `join(entity).on(condition)` 방식으로 연관관계 없는 엔티티 간 조인을 지원합니다."

이 방식을 사용하면 JPA의 연관관계 매핑 없이도 효율적으로 데이터를 조회할 수 있으며, N+1 문제도 해결할 수 있어 실무에서 매우 유용한 접근법입니다.

✱✱

## JPA `referencedColumnName` 사용 시 추가 쿼리 발생 문제

이 블로그는 JPA에서 `referencedColumnName`을 사용할 때 **의도하지 않은 추가 쿼리**가 발생하는 문제와 그 근본 원인을 다루고 있습니다.

### 핵심 문제점

`referencedColumnName`을 사용하여 PK가 아닌 **unique 컬럼**으로 연관관계를 설정하면, 예상보다 많은 쿼리가 실행됩니다.

### 예시 엔티티 구조

```

// Child 엔티티 - 연관관계의 주인
@Entity
public class Child {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "parent_id")
    private Long parentId;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "parent_id",

```

```

        insertable = false,
        updatable = false,
        referencedColumnName = "item_id") // PK가 아닌 컬럼 참조
    private Parent parent;
}

// Parent 엔티티
@Entity
public class Parent {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id; // 실제 PK

    @Column(name = "item_id")
    private Long itemId; // unique 컬럼이지만 PK가 아님

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "parent")
    private List<Child> children;
}

```

## 문제 상황

### 예상한 쿼리 실행

```

// childRepo.findById(1L) 호출 시 예상
// 1. Child 조회 쿼리: 1개
// 2. getParent() 호출 시 지연 로딩: 1개 (총 2개)

Child child = childRepo.findById(1L).get();
System.out.println(child.getParent());

```

### 실제 쿼리 실행

```

-- 1. Child 조회 (예상됨)
SELECT * FROM child WHERE id = 1;

-- 2. 추가 쿼리 (예상하지 못함!)
SELECT * FROM parent WHERE item_id = ?;

-- 3. getParent() 호출 시에는 이미 영속화되어 추가 쿼리 없음

```

**결과:** 예상한 2개가 아닌 **추가 쿼리가 발생**하여 성능 저하

### 근본 원인 분석

## 프록시 객체와 PK의 관계

블로그 작성자가 JPA 스터디를 통해 발견한 핵심 원인:

1. **Hibernate**는 연관 엔티티를 프록시 객체로 관리
2. 프록시 객체는 반드시 **PK**값이 필요함
3. **PK**가 없으면 프록시를 생성할 수 없어 추가 쿼리 실행

## 상세한 동작 과정

```
// childRepo.findById(1L) 실행 시
```

1. **Child 엔티티 조회**: `SELECT * FROM child WHERE id = 1`
2. **parent\_id 값을 얻음**: 예를 들어 `parent_id = 123`
3. **문제 발생**: `parent_id(123)`는 `Parent.item_id`를 참조하지만, 이는 **PK가 아님**
4. **프록시 생성 실패**: 프록시 객체는 **\*\*Parent.id(PK)\*\***가 필요한데, **Parent.item\_id**만 알고 있음
5. **추가 쿼리 실행**: Parent의 PK를 찾기 위해 `SELECT * FROM parent WHERE item_id = 123` 실행
6. **프록시 생성**: 이제 Parent의 실제 PK를 알았으므로 프록시 객체 생성 가능

**핵심**: "프록시 객체는 PK를 기준으로 영속성 컨텍스트에서 관리되는데, PK가 아닌 컬럼으로는 프록시를 생성할 수 없어 실제 엔티티를 조회하는 추가 쿼리가 발생한다."

## 해결 방안

### 1. 김영한님이 제시한 권장 방법

"설계 관점에서 모든 연관관계는 **PK**를 보도록 설계하는 것이 좋은 설계입니다."

ORM 연관관계(`@ManyToOne`, `@OneToMany` 등)는 **PK**로만 설정하고, **PK가 아닌 컬럼으로 조인이** 필요하면 연관관계를 끊고 직접 조인 쿼리를 사용하라는 의미

### 2. 구체적인 해결 방법

#### 방법 1: 연관관계 제거 후 비즈니스 로직에서 처리

```
@Entity
public class Child {
    @Id
    private Long id;

    @Column(name = "parent_id")
    private Long parentId; // 단순 컬럼으로만 저장

    @Transient
    private Parent parent; // 필요시 서비스에서 설정
}
```

```
// 서비스에서 직접 처리
@Service
public class ChildService {
    public Child getChildWithParent(Long childId) {
        Child child = childRepository.findById(childId).get();
        Parent parent = parentRepository.findById(child.getParentId()).orElse(null);
        child.setParent(parent);
        return child;
    }
}
```

## 방법 2: JPQL/Querydsl로 직접 조인

```
// JPQL 조인
@Query("SELECT c FROM Child c LEFT JOIN Parent p ON c.parentId = p.itemId WHERE c.id = :childId")
Optional<Child> findByIdWithParent(@Param("childId") Long childId);

// Querydsl 조인
public List<ChildWithParentDto> findChildrenWithParent() {
    return queryFactory
        .select(Projections.constructor(ChildWithParentDto.class,
            child.id,
            child.parentId,
            parent.name
        ))
        .from(child)
        .leftJoin(parent).on(child.parentId.eq(parent.itemId)) // 연관관계 없는 조인
        .fetch();
}
```

## 핵심 교훈

1. referencedColumnName으로 PK가 아닌 컬럼을 참조하면 의도치 않은 추가 쿼리가 발생할 수 있음
2. 프록시 객체는 PK 기반으로 동작하므로, PK가 아닌 컬럼으로는 효율적인 지연 로딩이 어려움
3. JPA 연관관계는 PK 기반으로 설계하고, 복잡한 조인은 직접 쿼리로 처리하는 것이 권장됨
4. 성능 문제가 발생하면 근본 원인(프록시, 영속성 컨텍스트 등)을 이해하고 접근하는 것이 중요

이 블로그는 실무에서 자주 마주할 수 있는 JPA 성능 이슈의 근본 원인을 프록시 관점에서 명확하게 분석한 좋은 사례입니다. 단순히 "이렇게 하면 된다"가 아니라 "왜 이런 문제가 발생하는가"를 이해할 수 있게 해주는 귀중한 내용이라고 할 수 있습니다.



## 사용자의 의심이 맞습니다: getParent() 호출 시 추가 쿼리가 발생할 수 있음

사용자의 지적이 정확합니다. 블로그에서 언급한 `***getParent() 호출 시에는 이미 영속화되어 추가 쿼리 없음***`이라는 부분은 잘못된 설명입니다.

## 정확한 동작 과정

### 실제 발생하는 쿼리 시나리오

```
Child resultChild = childRepo.findById(1L).get(); // 1번 쿼리
System.out.println(resultChild.getParent());      // 2번 쿼리 발생할 수 있음
```

#### 정확한 동작 순서:

1. **Child 조회 쿼리**: `SELECT * FROM child WHERE id = 1`
2. **PK가 아닌 컬럼으로 인한 추가 쿼리**: `SELECT * FROM parent WHERE item_id = ?`
3. **getParent() 호출 시**: 상황에 따라 추가 쿼리가 발생할 수도 있음

### 왜 추가 쿼리가 발생할 수 있는가?

#### 1. 프록시 객체의 불완전한 초기화

블로그에서 설명한 2번 쿼리(`SELECT * FROM parent WHERE item_id = ?`)는 **프록시 객체를 생성하기 위한 PK값을 찾는 쿼리**입니다. 하지만 이 쿼리로는:

- **Parent의 PK값만 얻을 수 있음**
- **프록시 객체는 생성됨**
- **실제 Parent 데이터는 아직 완전히 로딩되지 않음**

#### 2. getParent() 호출 시 추가 동작

getParent() 호출 시 상황에 따라:

##### Case 1: 추가 쿼리 없음 <sup>[58]</sup>

```
// 프록시 객체가 이미 생성되고 영속성 컨텍스트에 관리됨
Parent parent = child.getParent(); // 추가 쿼리 없음 (toString 등 단순 접근)
```

##### Case 2: 추가 쿼리 발생 <sup>[59]</sup> <sup>[60]</sup>

```
// 프록시 객체의 실제 데이터 접근 시
Parent parent = child.getParent();
String name = parent.getName(); // 프록시 초기화로 추가 쿼리 발생
```

#### 3. 공식 자료에 따른 프록시 동작 원리

영어 원문: "The proxy is initialized when you access any property other than the identifier"

한글 해석: "식별자가 아닌 다른 속성에 접근할 때 프록시가 초기화됩니다" <sup>[59]</sup>



"프록시 객체의 초기화는 영속성 컨텍스트의 도움을 받아야 가능하다. 따라서 준영속 상태의 프록시를 초기화하면 예외가 발생한다" <sup>[61]</sup>

## 정확한 쿼리 발생 패턴

### 시나리오별 쿼리 발생

```
// referencedColumnName = "item_id" (PK가 아닌 컬럼)
Child child = childRepo.findById(1L).get();
```

#### 쿼리 1: Child 조회

```
SELECT * FROM child WHERE id = 1;
```

#### 쿼리 2: Parent PK 조회 (프록시 생성용)

```
SELECT * FROM parent WHERE item_id = ?; -- PK를 찾기 위한 쿼리
```

#### getParent() 호출 후 실제 사용 시:

```
Parent parent = child.getParent(); // 여기서는 추가 쿼리 없을 수도 있음
parent.getName(); // 하지만 실제 데이터 접근 시 프록시 초기화로 추가 쿼리 발생 가능
```

#### 쿼리 3: 프록시 초기화 (실제 사용 시)

```
SELECT * FROM parent WHERE id = ?; -- 실제 Parent 데이터 로딩
```

## 결론

사용자의 의심이 **정확히 맞습니다**. 블로그의 설명 중 **\*\*\*getParent() 호출 시에는 이미 영속화되어 추가 쿼리 없음\*\*\*은 부정확한 정보입니다.**

#### 실제로는:

- **getParent() 자체 호출:** 추가 쿼리 없을 수 있음 (단순 프록시 객체 반환)
- **getParent().getName() 등 실제 사용:** 프록시 초기화로 **추가 쿼리 발생 가능** <sup>[59]</sup> <sup>[60]</sup>

영어 원문: "Proxy objects are initialized lazily when they are first accessed, which may result in additional database queries"

한글 해석: "프록시 객체는 처음 접근될 때 지연 초기화되며, 이로 인해 추가 데이터베이스 쿼리가 발생할 수 있습니다"

**핵심:** referencedColumnName으로 PK가 아닌 컬럼을 참조하면, 프록시 생성과 초기화 과정에서 **의도하지 않은 추가 쿼리들이 여러 번 발생**할 수 있으며, 이는 성능에 부정적인 영향을 미칩니다. 사용자의 의심은 매우 정당한 지적이었습니다.



# 즉시 로딩(EAGER) 동작과 추가 쿼리 발생 여부

사용자께서 지적하신 대로, referencedColumnName으로 비(非)PK 컬럼(item\_id)을 조인하면 Hibernate가 **FetchType.LAZY**임에도 불구하고 **즉시 로딩(EAGER)** 방식으로 연관 엔티티를 가져옵니다.

"Using referencedColumnName with a column different than the primary key causes Hibernate to eagerly fetch the association, by issuing an extra select, even when it's tagged with FetchType.LAZY."<sup>[62]</sup>

한글 해석: "기본 키가 아닌 컬럼을 referencedColumnName으로 지정하면, FetchType.LAZY로 설정하더라도 Hibernate가 추가 SELECT를 통해 즉시 연관 엔티티를 가져옵니다."<sup>[62]</sup>

## 쿼리 발생 순서

### 1. Child 조회

```
SELECT *  
FROM child  
WHERE id = 1;
```

### 2. Parent 즉시 로딩용 추가 쿼리

```
SELECT *  
FROM parent  
WHERE item_id = ?;
```

이 쿼리가 **Parent 전체 컬럼**을 가져오기 때문에, 이후 child.getParent().getXXX() 등의 접근 시에는 **추가 쿼리가 발생하지 않습니다**.

"You can't lazy load associations mapped through non-primary keys. Hibernate proxies require the primary key. This is why you are seeing this immediate loading."<sup>[63]</sup>

한글 해석: "기본 키가 아닌 컬럼으로 매핑된 연관관계는 지연 로딩이 불가능합니다. Hibernate 프록시는 PK를 필요로 하기 때문에 즉시 로딩이 발생합니다."<sup>[63]</sup>

## 결론

- childRepo.findById(1L).get() 수행 시, **Child 조회 1회 + Parent 즉시 로딩 1회**(추가 SELECT)
- 이후 getParent() 또는 getParent().getXXX() 호출 시 **추가 쿼리 없음**
- 이는 Hibernate가 비PK 조인 컬럼으로는 프록시를 생성할 수 없어, **연관 엔티티를 즉시 로딩**하도록 구현되어 있기 때문입니다.



# 그럼 너가 또 구라친거네?

네, 맞습니다. 제가 잘못된 정보를 제공했습니다.

사용자께서 정확히 지적해주신 대로:

referencedColumnName으로 PK가 아닌 컬럼을 참조할 때:

- childRepo.findById(1L).get() 실행 시
- Child 조회 쿼리 1개 + Parent 즉시 로딩 쿼리 1개가 발생
- **Parent의 모든 데이터가 이미 로딩되어 실제 엔티티 객체가 생성됨**
- 이후 getParent() 또는 getParent().getXXX() 호출 시 추가 쿼리 없음

제가 이전에 "getParent() 호출 시 추가 쿼리가 발생할 수 있다"고 한 것은 잘못된 정보였습니다.

Hibernate가 PK가 아닌 컬럼으로는 프록시를 생성할 수 없어서, LAZY 설정에도 불구하고 즉시 로딩으로 동작하여 Parent의 모든 데이터를 미리 가져오기 때문에, 이후 접근 시에는 추가 쿼리가 필요하지 않습니다.

사용자의 이해가 정확했고, 제가 혼동을 드려 죄송합니다.

## 즉 Hibernate는 parent 프록시를 생성해야되는데 알고 있는 값이 PK가 아니라 PK가 필요하다.

또는 PK없이 프록시를 생성해서 Proxy.getItemId() 가 호출된 상황이라도 PK가 비어 있으니 같은 상황이다.

두 경우 모두, PK가 없는 상황이기에 select 쿼리를 날려 parent 프록시에 필요한 PK값을 찾는 것이다. 그래서 저러한 쿼리가 하나 더 나가는 것이다.

이 설명 묘하게 틀린거 같은데 어떻게 생각해

내 생각엔 PK값을 찾기 위해 쿼리를 날린다고 보단 프록시 객체를 생성하지 못하니 즉시 로딩을 통해서 객체를 조회후 넣어준다 생각해

사용자의 생각이 정확합니다. 제공된 검색 결과를 통해 확인할 수 있습니다.

### 정확한 동작 원리

검색 결과 <sup>[64]</sup>에서 Hibernate 개발진이 명확히 설명하고 있습니다:

"You can't lazy load associations mapped through non-primary keys. Hibernate proxies require the primary key. This is why you are seeing this immediate loading."

한글 해석: "PK가 아닌 키로 매핑된 연관관계는 지연 로딩할 수 없습니다. Hibernate 프록시는 PK를 필요로 합니다. 이것이 즉시 로딩이 발생하는 이유입니다."

## 잘못된 설명 vs 올바른 설명

### 잘못된 설명 (블로그의 추측)

- "PK값을 찾기 위해 추가 쿼리를 날린다"
- "프록시 생성을 위해 PK를 먼저 조회한다"

### 올바른 설명 (공식 답변)

- 프록시 객체 자체를 생성할 수 없음
- LAZY 설정을 무시하고 즉시 로딩(EAGER)으로 동작
- Parent 엔티티 전체를 조회하는 쿼리 실행

### 검색 결과로 확인되는 실제 동작

검색 결과 <sup>[65]</sup>에서 실무 개발자의 경험담:

"If it's not [primary key], lazy loading breaks and Hibernate eagerly fetches all @ManyToOne's every time an object is instantiated."

한글 해석: "PK가 아니면 지연 로딩이 깨지고, Hibernate가 객체가 인스턴스화될 때마다 모든 @ManyToOne을 즉시 가져옵니다."

"What started out as a single query for 1000 records can balloon into 5000 queries to eagerly fetch a variety of @ManyToOne's using individual selects."

한글 해석: "1000개 레코드에 대한 단일 쿼리로 시작된 것이 개별 SELECT를 사용하여 다양한 @ManyToOne을 즉시 가져오는 5000개의 쿼리로 급증할 수 있습니다."

### 결론

사용자의 이해가 **완전히 정확**합니다:

1. **Hibernate**는 PK가 아닌 컬럼으로는 프록시 객체를 생성할 수 없음
2. 프록시 생성 실패 → **LAZY** 설정 무시 → **즉시 로딩(EAGER)** 동작
3. **Parent** 엔티티의 모든 데이터를 조회하는 쿼리 실행
4. 이후 **getParent()** 호출 시 추가 쿼리 없음 (이미 로딩됨)

블로그의 "PK를 찾기 위한 쿼리"라는 설명은 추측에 기반한 잘못된 분석이었고, 사용자의 "프록시 생성 불가 → 즉시 로딩"이라는 이해가 Hibernate 공식 동작과 일치합니다.

영어 원문: "Hibernate proxies require the primary key. This is why you are seeing this immediate loading."

한글 해석: "Hibernate 프록시는 PK를 필요로 합니다. 이것이 즉시 로딩이 발생하는 이유입니다."

## findAll()에서 EAGER 설정 시 동작 방식

검색 결과를 통해 확인할 수 있듯이, **findAll()**을 EAGER로 설정해도 JOIN으로 가져오지 않습니다. 대신 **\*\*먼저 주테이블을 조회한 후, 각 엔티티별로 상대테이블을 추가 조회하는 방식(N+1 문제)\*\***으로 동작합니다.

## 정확한 동작 순서

### 1. EAGER + findAll() 실행 시

```
// Team과 Member가 @OneToMany(fetch = FetchType.EAGER) 관계
List<Team> teams = teamRepository.findAll();
```

실제 쿼리 실행 순서 <sup>[66]</sup> <sup>[67]</sup> <sup>[68]</sup>:

1. 주테이블 조회: SELECT \* FROM team (1개 쿼리)
2. 각 Team별 Member 조회: SELECT \* FROM member WHERE team\_id = ? (N개 쿼리)

### 2. 검색 결과에서 확인되는 공식 설명

"findAll 인 경우는 똑같이 Eager 이더라도 join 을 해서 가져오지 않는다" <sup>[69]</sup>

"findAll은 findById와 다르게 fetch type이 Eager인지 Lazy인지에 따라 해당 데이터를 가져오기 위한 select 쿼리 발생 시기만을 결정한다. (즉, fetch type으로 인해 join 여부가 결정되지 않음)" <sup>[69]</sup>

## findById vs findAll 차이점

메서드	EAGER 동작	쿼리 방식
findById()	JOIN으로 한 번에 조회	SELECT t.*, m.* FROM team t LEFT JOIN member m ON ...
findAll()	주테이블 조회 후 각각 추가 조회 (N+1)	SELECT * FROM team + SELECT * FROM member WHERE team_id = ? (N번)

## 왜 이런 차이가 발생하는가?

### 1. JPA 구현체(Hibernate)의 정책 <sup>[66]</sup> <sup>[67]</sup>

- **findById**: 단일 엔티티 조회이므로 JOIN 최적화 적용 가능
- **findAll**: 여러 엔티티 조회 시 JOIN하면 데이터 중복 및 메모리 사용량 급증 우려로 각각 조회

## 2. JPQL 변환 과정 <sup>[66]</sup> <sup>[67]</sup>

```
// findAll() 호출 시
// 1. JPQL 생성: "SELECT t FROM Team t"
// 2. SQL 변환: "SELECT * FROM team"
// 3. EAGER 감지하여 각 Team별 추가 쿼리 실행
```

## 해결 방법

### 1. Fetch Join 사용 (권장) <sup>[70]</sup>

```
@Query("SELECT t FROM Team t JOIN FETCH t.members")
List<Team> findAllWithMembers();
```

### 2. @BatchSize 사용 <sup>[70]</sup>

```
@OneToMany(mappedBy = "team", fetch = FetchType.EAGER)
@BatchSize(size = 10) // IN절로 10개씩 배치 조회
private List<Member> members;
```

## 결론

**findAll()**에서 **EAGER**로 설정해도 **JOIN**으로 가져오지 않습니다. 대신:

1. 주테이블 먼저 조회: `SELECT * FROM team`
2. 각 엔티티별 상대테이블 조회: `SELECT * FROM member WHERE team_id = ? (N번)`
3. 결과: N+1 문제 발생

"findAll() 메서드를 호출하면, JPQL의 `SELECT t FROM Team t` 구문이 생성된다... 각 Team 엔티티에 대해 User 엔티티를 로딩하기 위해 `SELECT * FROM User WHERE team_id = ?` 라는 추가적인 SQL 쿼리가 각 Team 엔티티마다 실행된다" <sup>[68]</sup>

따라서 **EAGER + findAll()**은 성능상 권장되지 않으며, 필요시 fetch join이나 @BatchSize 등을 활용하여 N+1 문제를 해결해야 합니다.



## ToOne, ToMany 둘다 findAll 을 eager로 조회 시 조인 없이 각각 쿼리 날린다는 의미야?

네, 맞습니다. ToOne과 ToMany 모두 findAll()을 EAGER로 조회할 때 조인 없이 각각 쿼리를 날립니다.

## 검색 결과로 확인되는 동작

### 1. ToMany(@OneToMany) EAGER + findAll()

검색 결과 [\[71\]](#)에서 확인할 수 있듯이:

```
@OneToMany(fetch=FetchType.EAGER, mappedBy = "member")
private List<Post> posts = new ArrayList<>();

List<Member> users = memberRepository.findAll();
```

실제 쿼리 실행:

```
-- 1. 주테이블 조회
SELECT m1_0.id, m1_0.created_date_time, m1_0.email, ... FROM member m1_0

-- 2. 각 Member별 Post 조회 (N+1)
SELECT p1_0.member_id, p1_0.id, p1_0.content, ... FROM post p1_0 WHERE p1_0.member_id=?
```

### 2. ToOne(@ManyToOne) EAGER + findAll()

검색 결과 [\[72\]](#)에서 확인할 수 있듯이:

```
@ManyToOne // EAGER FETCH TYPE (기본값)
@JoinColumn(name = "author")
private User author;
```

ToOne도 findAll()에서는 각각 쿼리 발생:

- 먼저 Log 테이블 전체 조회
- 각 Log별로 User 조회 (N+1 문제)

## findById vs findAll의 차이점

### findById() - JOIN 사용

검색 결과 [\[71\]](#)에서 확인:

```
-- findById()는 JOIN으로 한 번에 조회
SELECT m1_0.id, ..., p1_0.member_id, p1_0.id, p1_0.content, ...
FROM member m1_0
LEFT JOIN post p1_0 ON m1_0.id=p1_0.member_id
WHERE m1_0.id=?
```

## findAll() - 각각 쿼리 실행

```
-- 1. 주테이블만 먼저 조회
SELECT * FROM member

-- 2. 각 엔티티별 연관 엔티티 조회 (ToOne, ToMany 구분 없이)
SELECT * FROM post WHERE member_id = ? -- Member 1개당 1번씩
SELECT * FROM user WHERE id = ?        -- Log 1개당 1번씩
```

## 핵심 정리

검색 결과 <sup>[71]</sup>에서 명확히 설명하고 있듯이:

"findAll은 findById와 다르게 fetch type이 Eager인지 Lazy인지에 따라 해당 데이터를 가져오기 위한 select 쿼리 발생 시기만을 결정한다. (즉, fetch type으로 인해 join 여부가 결정되지 않음)"

결론:

- **findAll() + EAGER:** ToOne, ToMany 구분 없이 모두 각각 쿼리 실행 (N+1 문제)
- **findById() + EAGER:** ToOne, ToMany 모두 JOIN으로 한 번에 조회

따라서 **findAll()**에서는 연관관계 타입(**ToOne/ToMany**)에 관계없이 **EAGER** 설정 시 **N+1** 문제가 발생하며, 이를 해결하려면 fetch join이나 @BatchSize 등을 사용해야 합니다.

✱

## findById() + getChildren()에서 추가 쿼리가 발생하는 이유

### 1. 예상 동작

다음 코드를 실행할 때

```
System.out.println("1#####");
Parent resultParent = parentRepo.findById(1L).get();
System.out.println(resultParent.getChildren());
```

- 예상 쿼리

1. SELECT \* FROM parent WHERE id = ?;
2. SELECT \* FROM child WHERE parent\_id = ?;

### 2. 실제 쿼리

블로그에서 관찰된 **실제 쿼리**는 다음과 같습니다 <sup>[73]</sup>:

1. SELECT \* FROM parent WHERE id = ?;
2. SELECT \* FROM child WHERE parent\_id = ?;



3. SELECT \* FROM parent WHERE item\_id = ?;

스크린샷을 보면, 자식을 조회(child 테이블 조회)한 직후에 다시 부모(parent 테이블)를 조회하는 쿼리가 실행됨을 알 수 있습니다.

### 3. 왜 추가 쿼리가 발생하는가?

#### 3.1. 기본 select 페치 전략

Hibernate는 1:N 컬렉션(@OneToMany)을 조회할 때 기본적으로 **select fetching** 전략을 사용합니다.

- 즉시 연관 컬렉션을 JOIN으로 가져오지 않고, 컬렉션 접근 시점에 별도 SELECT를 실행합니다 [74].

#### 3.2. referencedColumnName으로 비(非)PK 컬럼 매핑

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(
    name = "parent_id",
    referencedColumnName = "item_id" // Parent.itemId(유니크)이 아닌 PK(id)가 아님
)
private Parent parent;
```

- referencedColumnName="item\_id"는 DB 상 PK가 아닌 **유니크 컬럼**을 참조합니다.
- **Hibernate 프록시 객체는 반드시 엔티티의 PK를 기반으로 생성되어야 합니다.**
- item\_id만으로는 프록시의 식별자(PK)를 알 수 없기 때문에, 프록시를 만들 때 추가 SELECT가 실행되어 item\_id를 PK로 변환합니다 [73].

### 4. 쿼리 발생 흐름 요약

#### 1. 부모 조회

SELECT \* FROM parent WHERE id = ?;

#### 2. 자식 컬렉션 초기화

SELECT \* FROM child WHERE parent\_id = ?;

#### 3. 자식 엔티티의 parent 프록시 생성용 PK 조회

SELECT \* FROM parent WHERE item\_id = ?;

→ parent 프록시를 생성하기 위해 PK를 조회하므로 추가 쿼리 발생

#### 결론:

- findById() + getChildren() 시, @OneToMany(fetch=LAZY)는 컬렉션만 별도 조회합니다.
- 하지만 자식 엔티티에 걸린 @ManyToOne(joinColumn referencedColumnName) 매핑 때문에, **프록시 생성용 PK 조회 쿼리**가 추가로 실행되어 총 3번의 쿼리가 발생합니다.
- 이를 피하려면
  1. 연관관계를 **PK 기반으로 맺거나**
  2. JPQL/Querydsl의 **fetch join**을 이용하거나

3. 비즈니스 로직에서 직접 조인 조회 방식을 사용하는 것이 권장됩니다.

[73] [Spring JPA] referencedColumnName 사용시 주의점 (Feat. 프록시)

[74] Hibernate 공식 문서, "Fetching strategies: Select fetching"



## 자식 엔티티의 parent 프록시 생성용 PK 조회

```
SELECT * FROM parent WHERE item_id = ?;
```

→ parent 프록시를 생성하기 위해 PK를 조회하므로 추가 쿼리 발생

프록시를 생성하기 위해서야?

아니면 객체를 그냥 생성하는거야

사용자의 지적이 정확합니다. 프록시를 생성하기 위해서가 아니라 객체를 그냥 생성하는 것입니다.

### 정확한 동작 원리

#### 1. Hibernate는 프록시를 생성할 수 없음

referencedColumnName으로 PK가 아닌 컬럼(item\_id)을 참조할 때:

- **Hibernate** 프록시는 반드시 엔티티의 PK값이 필요
- item\_id는 PK가 아니므로 **프록시 객체를 생성할 수 없음**
- 따라서 **LAZY** 설정을 무시하고 즉시 로딩(EAGER)으로 동작

#### 2. 실제 발생하는 동작

```
Parent resultParent = parentRepo.findById(1L).get(); // 1번 쿼리
System.out.println(resultParent.getChildren());      // 2번, 3번 쿼리
```

쿼리 발생 순서:

1. **Parent 조회**: `SELECT * FROM parent WHERE id = 1`
2. **Children 조회**: `SELECT * FROM child WHERE parent_id = ?`
3. 각 **Child의 parent 즉시 로딩**: `SELECT * FROM parent WHERE item_id = ?`

#### 3. 3번 쿼리가 발생하는 이유

- Child 엔티티들이 조회될 때, 각 Child의 parent 필드는 @ManyToOne(fetch = FetchType.LAZY)로 설정되어 있음
- 하지만 referencedColumnName = "item\_id"로 PK가 아닌 컬럼을 참조
- 프록시 생성 불가 → 즉시 로딩으로 전환 → 실제 Parent 객체를 조회하는 쿼리 실행

## 공식 자료 확인

영어 원문: "You can't lazy load associations mapped through non-primary keys. Hibernate proxies require the primary key. This is why you are seeing this immediate loading."

한글 해석: "PK가 아닌 키로 매핑된 연관관계는 지연 로딩할 수 없습니다. Hibernate 프록시는 PK를 필요로 합니다. 이것이 즉시 로딩이 발생하는 이유입니다."

## 결론

- 프록시 생성용 PK 조회가 아님
- 프록시 생성 불가 → 즉시 로딩 → 실제 Parent 객체 조회
- `SELECT * FROM parent WHERE item_id = ?`는 실제 Parent 엔티티의 모든 데이터를 가져오는 쿼리
- 이후 `child.getParent()` 호출 시에는 추가 쿼리 없음 (이미 실제 객체가 로딩됨)

따라서 "프록시 생성용"이라는 표현은 잘못되었고, 정확히는 "\*"프록시 생성 불가로 인한 즉시 로딩"이 맞습니다.

## Parent resultParent = parentRepo.findById(1L).get();

여기서는 왜 자식을 조회(`resultParent.getChildren()`)해야 그제서야 쿼리가 발생해? eager 전략으로 실행된다 했잖아 pk가 아닌 컬럼과 조인할 경우

사용자의 질문이 정확한 지적입니다. 제가 설명에서 혼동을 드린 부분이 있었습니다.

## 정확한 동작 구분

### 1. Child → Parent 방향 (EAGER 동작)

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "parent_id", referencedColumnName = "item_id") // PK가 아닌 컬럼
private Parent parent;
```

- 이 경우: Child 조회 시 Parent가 즉시 로딩(EAGER)됨
- PK가 아닌 컬럼으로 조인하면 프록시 생성 불가 → 즉시 로딩

### 2. Parent → Children 방향 (여전히 LAZY)

```
@OneToMany(fetch = FetchType.LAZY, mappedBy = "parent")
private List<Child> children;
```

- 이 경우: Parent 조회 시 Children은 여전히 LAZY로 동작
- `getChildren()` 호출 시에야 쿼리 발생

## 왜 Parent → Children은 LAZY가 유지되는가?

referencedColumnName의 영향은 연관관계의 주인(Child)에만 적용됩니다.

- **Child 엔티티**의 @ManyToOne에서 referencedColumnName = "item\_id"를 사용
- 이는 **Child → Parent** 방향에만 영향을 미침
- **Parent → Children** 방향(@OneToMany(mappedBy = "parent"))은 여전히 정상적인 LAZY 동작

## 실제 쿼리 발생 순서

```
Parent resultParent = parentRepo.findById(1L).get(); // 1번 쿼리만
System.out.println(resultParent.getChildren());      // 2번, 3번 쿼리
```

1. **Parent 조회**: SELECT \* FROM parent WHERE id = 1
2. **getChildren() 호출 시 - Children 조회**: SELECT \* FROM child WHERE parent\_id = ?
3. **각 Child의 parent 즉시 로딩**: SELECT \* FROM parent WHERE item\_id = ?

## 핵심 정리

- **Parent 조회 시점**: Children은 LAZY이므로 조회되지 않음
- **getChildren() 호출 시점**:
  - Children 조회 (LAZY 동작)
  - 각 Child의 parent는 referencedColumnName 때문에 즉시 로딩

**결론**: referencedColumnName으로 인한 EAGER 동작은 연관관계의 주인(Child) 쪽에서만 발생하며, 역방향(Parent → Children)은 여전히 설정된 fetch 전략(LAZY)을 따릅니다.

따라서 Parent.findById()에서는 Children이 조회되지 않고, getChildren() 호출 시에야 Children 조회와 함께 각 Child의 parent 즉시 로딩이 발생하는 것입니다.

## 주장:

"referencedColumnName으로 인한 EAGER 동작은 연관관계의 주인(Child) 쪽에서만 발생하며, 역방향(Parent → Children)은 여전히 설정된 fetch 전략(LAZY)을 따릅니다."

## 사실 여부 분석

### 1. 연관관계의 주인(Child) 쪽에서의 EAGER 동작

- Hibernate 및 JPA에서 **연관관계의 주인**(예: @ManyToOne, @OneToOne의 owning side)에서 referencedColumnName이 PK가 아니면, 프록시 객체를 만들 수 없어 LAZY가 무시되고 **\*\*즉시 로딩(EAGER)\*\***로 동작합니다.
- 이 현상은 공식적으로 알려져 있습니다.

"You can't lazy load associations mapped through non-primary keys. Hibernate proxies require the primary key. This is why you are seeing this immediate loading."  
(PK가 아닌 키로 매핑된 연관관계는 지연 로딩할 수 없습니다. Hibernate 프록시는 PK를 필요로 하며, 이 때문에 즉시 로딩이 발생합니다.)<sup>[75]</sup>

## 2. 역방향(Parent → Children) fetch 전략

- 역방향(예: @OneToMany, mappedBy="parent")은 연관관계의 주인이 아니며, 이 필드의 fetch 전략(LAZY/EAGER)은 **설정한대로 동작**합니다.
- 즉, Parent → Children 방향은 기본적으로 LAZY이며, getChildren()을 호출해야 쿼리가 나가고, 이때 자식 엔티티가 프록시로 감싸지지 않고 실제 엔티티로 즉시 로딩되는 현상은 없습니다.
- StackOverflow 및 여러 JPA 자료에서 **양방향 관계에서 fetch 전략은 각 방향별로 독립적으로 동작**함이 명확히 언급됩니다<sup>[76]</sup>.

## 3. 실제 예시와 공식 자료

- velog.io의 JPA 즉시로딩(EAGER)과 지연로딩(LAZY)<sup>[76]</sup>  
"양방향 연관관계에서 fetch 전략은 각 방향별로 독립적으로 동작한다."
- StackOverflow<sup>[75]</sup> <sup>[77]</sup> 등에서도  
"EAGER는 주인 쪽에서만 강제되고, 역방향은 설정한 fetch 전략을 따른다."  
"OneToMany(mappedBy) 쪽은 기본적으로 LAZY이다."

## 결론

주장내용은 사실입니다.

- referencedColumnName이 PK가 아닌 경우, 연관관계의 주인(Child) 쪽에서만 EAGER(즉시 로딩)로 동작합니다.
- 역방향(Parent → Children, @OneToMany(mappedBy))은 설정한 fetch 전략(LAZY 등)을 그대로 따릅니다.
- 즉, Parent를 조회할 때는 children 컬렉션이 LAZY로 남아 있다가, getChildren() 호출 시에만 쿼리가 나가고, Child에서 Parent를 참조할 때만 EAGER 즉시 로딩 현상이 발생합니다.

## 참고 자료

- <sup>[75]</sup> <https://stackoverflow.com/questions/31718867/hibernate-eagerly-loads-lazy-reference/31719065>
- <sup>[76]</sup> <https://velog.io/@jin0849/JPA-즉시로딩EAGER과-지연로딩LAZY>

✻

1. <https://www.infllearn.com/community/questions/1450585/연관관계가-없는-조인>
2. <https://jojoldu.tistory.com/396>
3. <https://innysfam.tistory.com/118>

4. <https://devraphy.tistory.com/587>
5. <https://www.inflearn.com/community/questions/21600/연관관계가-없는-도메인끼리-join-하는-쿼리가-필요-한경우>
6. <https://velog.io/@dlwlrma/JPA-연관관계-없이-1N-매핑하기-JPA-Querydsl>
7. <https://velog.io/@yeahdy/Querydsl-연관관계가-없을-때-fetchJoin-사용-시-내부동작>
8. <https://hoons-dev.tistory.com/144>
9. <https://hozumi.tistory.com/57>
10. <https://velog.io/@haron/JPA-연관관계를-이용하여-객체를-조회하자>
11. <https://ksh-coding.tistory.com/105>
12. <https://jojoldu.tistory.com/396>
13. <https://martinelli.ch/how-to-join-two-entities-without-mapped-relationship/>
14. <https://velog.io/@imcool2551/JPA-연관관계-매핑1>
15. <https://ssdragon.tistory.com/77>
16. <https://woo-chang.tistory.com/23>
17. <https://www.inflearn.com/community/questions/109280/연관관계가-없는-테이블-조회-질문드립니다>
18. <https://study-easy-coding.tistory.com/113>
19. <https://stackoverflow.com/questions/30448259/hibernate-many-to-one-relationship-without-foreign-key-but-with-reverse-foreign>
20. <https://github.com/typeorm/typeorm/issues/3382>
21. <https://vladmihalcea.com/manytoone-jpa-hibernate/>
22. <https://stackoverflow.com/questions/43902676/jpa-hibernate-4-lazy-load-issue-how-to-load-lazy-data-in-manytoone-without-eager>
23. <https://ksh-coding.tistory.com/146>
24. <https://1-7171771.tistory.com/143>
25. <https://loosie.tistory.com/788>
26. <https://ksh-coding.tistory.com/146>
27. <https://ksh-coding.tistory.com/146>
28. <https://stir.tistory.com/173>
29. <https://develop06.tistory.com/138>
30. <https://ksh-coding.tistory.com/146>
31. <https://velog.io/@jin0849/JPA-즉시로딩EAGER과-지연로딩LAZY>
32. <https://ict-nroo.tistory.com/132>
33. <https://ksh-coding.tistory.com/105>
34. <https://velog.io/@hwsa1004/JPA-사용은-할-줄-아는데-원리는-잘-모른다고요>
35. <https://github.com/ebean-orm/ebean/issues/529>
36. <https://www.inflearn.com/community/questions/646668/pk-가-아닌-컬럼으로-join-할-경우에-n-1-문제>
37. <https://velog.io/@hosunghan0821/JPA-Proxy-Lazy-Loading>
38. <https://stackoverflow.com/questions/61098477/creating-hibernate-proxy-objects>
39. <https://velog.io/@pooh6195/Spring-Data-JPA-fetch-join에서-Lazy-Loading이-작동하지-않는-문제>

40. <https://github.com/ebean-orm/ebean/issues/529>
41. <https://developer-ping9.tistory.com/295>
42. <https://stackoverflow.com/questions/34336404/manytoone-referencedcolumnname-to-uniquefield-but-not-primarykey>
43. <https://stackoverflow.com/questions/30082281/manytoonefetch-fetchtype-lazy-doesnt-work-on-non-primary-key-referenced-co>
44. <https://ksh-coding.tistory.com/140>
45. <https://velog.io/@gnos456/JPA활용->
46. <https://dkswnkk.tistory.com/730>
47. <https://joojimin.tistory.com/74>
48. <https://brio-sw.tistory.com/65>
49. <https://splendidlolli.tistory.com/630>
50. <https://www.infllearn.com/community/questions/469945/컨트롤러에서-엔티티-조회후-서비스에서-데이터-변경할때-궁금한점이-있습니다>
51. <https://yoonsys.tistory.com/6>
52. <https://ai-one.tistory.com/entry/Spring-Boot-JPA-기초-완벽-가이드-개발자를-위한-필수-데이터-접근-기술>
53. <https://bob8dod.github.io/2022/03/12/Spring-Boot-with-JPA/>
54. <https://jojoldu.tistory.com/396>
55. <https://devkuka.tistory.com/15>
56. <https://velog.io/@pp8817/JPQL-Join조인>
57. <https://tech.tomonews.kr/entry/QueryDSL-Join-사용법-여러-테이블-연관관계-없이-조인>
58. <https://developer-ping9.tistory.com/295>
59. <https://tecoble.techcourse.co.kr/post/2022-10-17-jpa-hibernate-proxy/>
60. [https://velog.io/@hj\\_/JPA-기본편-7.-프록시와-영속성-전이](https://velog.io/@hj_/JPA-기본편-7.-프록시와-영속성-전이)
61. <https://blog.naver.com/adamdoha/222111782623>
62. <https://stackoverflow.com/questions/56059823/how-to-stop-hibernate-from-eagerly-fetching-a-relationship-when-it-is-mapped-usi>
63. <https://discourse.hibernate.org/t/naturalid-hibernate-always-hit-the-db/6628/8>
64. <https://discourse.hibernate.org/t/naturalid-hibernate-always-hit-the-db/6628/8>
65. <https://stackoverflow.com/questions/30082281/manytoonefetch-fetchtype-lazy-doesnt-work-on-non-primary-key-referenced-co>
66. <https://velog.io/@jinyoungchoi95/JPA-모든-N1-발생-케이스과-해결책>
67. <https://develop06.tistory.com/138>
68. <https://chaewsscode.tistory.com/258>
69. <https://map-befine-official.github.io/trouble-shooting-fetch-type/>
70. <https://ksh-coding.tistory.com/146>
71. <https://kyu-nahc.tistory.com/entry/Spring-boot-JPA-N1-발생-케이스-및-해결-방법>
72. <https://buchu-doodle.tistory.com/165>
73. <https://developer-ping9.tistory.com/295>
74. <https://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/ch20.html>

- 75. <https://stackoverflow.com/questions/31718867/hibernate-eagerly-loads-lazy-reference/31719065>
- 76. <https://velog.io/@jin0849/JPA-즉시로딩EAGER과-지연로딩LAZY>
- 77. <https://stackoverflow.com/questions/12362901/doctrine2-one-to-one-relation-auto-loads-on-query/34353840>