BACKEND CS · STUDY GUIDE

Spring · JPA 완전 정복 8강

영속성 컨텍스트 · Fetch 전략 · @Transactional · AOP · 배치

8강 · 4부 구성

ttukttak-coding.dev

2026

이 책은

이 책은 Spring과 JPA를 "튜토리얼이 아닌 구조"로 풀어낸 학습서입니다. 영속성 컨텍스트라는 단일 구조에서 1차 캐시 · 변경 감지·flush·Fetch 전략 · 트랜잭션 경계 · 배치 쓰기까지 어떻게 파생되는지 — 그리고 Spring AOP 프록시가 @Transactional을 어떻게 가능하게 만드는지를 한 권으로 정리합니다.

각 강의는 "이 동작은 왜 이렇게 설계됐는가?" 를 먼저 묻습니다. save()·@Transactional·LAZY 같은 익숙한 도구가 실제로는 무엇을 대신해 주고, 어떤 경계에서 조용히 무력화되는지 — 그 구조를 이해하면 "왜 안 되지?" 에서 "여기서 프록시를 안 거쳤구나" 로 바로 디버깅이 옮겨갑니다.

1부는 영속성 컨텍스트의 동작 원리와 경계, 2부는 Fetch 전략과 N+1 해결 도구 비교, 3부는 Spring 트랜잭션 전파와 AOP 프록시의 함정, 4부는 대량 쓰기 성능과 엔티티 상태 다루기입니다. 한 번 통독한 뒤에는 JPA·Spring 관련 이슈를 만났을 때 옆에 두고 점검 도구로 쓰셔도 좋습니다.

> **구성**: 4부 8강 / **대상**: Spring/JPA를 실무에서 쓰지만 내부 동작이 불투명하게 느껴지는 백엔드 개발자 / **블로그**: 더 많은 글은 **ttukttak-coding.dev** 에서 보실 수 있습니다.

목차

1부. 영속성 컨텍스트 — 상태와 경계

01 JPA 영속성 컨텍스트 완전 정복 — 1차 캐시와 변경 감지는 어떻게 동작하나요? 9

02 JPA `flush`와 OSIV 완전 정복 — 영속성 컨텍스트의 경계는 어디까지인가요? 27

2부. Fetch 전략과 N+1 해결

03 JPA Fetch 전략 완전 정복 — `LAZY` vs `EAGER`와 N+1이 생기는 진짜 이유 45

04 N+1 해결 도구 완전 정복 — `fetch join` / `@EntityGraph` / `@BatchSize`는 언제 쓰나요? 59

3부. Spring 트랜잭션과 AOP

05 Spring `@Transactional` 완전 정복 — 전파 속성과 롤백 규칙은 어떻게 동작하나요? 77

06 Spring AOP 프록시와 `self-invocation` 함정 — `@Transactional`이 왜 안 먹나요? 93

4부. 성능과 엔티티 다루기

07 JPA 배치 쓰기 완전 정복 — `hibernate.jdbc.batch_size`와 `IDENTITY` 함정 111

08 JPA `merge` vs `persist` 완전 정복 — `detached` 엔티티를 어떻게 다뤄야 하나요? 125

PART 1

1부. 영속성 컨텍스트 — 상태와 경계

Chapter 1

JPA 영속성 컨텍스트 완전 정복 — 1차 캐시와 변경 감지는 어떻게 동작하나요?

#영속성 컨텍스트

JPA가 개발자를 대신해 엔티티 상태를 관리하고 SQL을 뒤늦게 내보내는 구조인 영속성 컨텍스트의 동작 원리를 엔티티 상태, 1차 캐시, 변경 감지, flush 시점 중심으로 정리합니다.

영속성 컨텍스트, 왜 알아야 하나요?

JPA를 쓰다 보면 분명히 배운 대로 썼는데 결과가 이상한 상황을 자주 만납니다.

- `save()`를 부르지 않았는데 `UPDATE`가 나갔습니다
- 같은 `findById()`를 두 번 호출했는데 SQL은 한 번만 나갔습니다
- 트랜잭션 밖에서 엔티티 필드를 바꿨더니 반영되지 않았습니다
- `persist()`를 불렀는데 `INSERT`가 트랜잭션 커밋 직전까지 미뤄졌습니다

이 모든 동작의 중심에 **영속성 컨텍스트(`Persistence Context`)**가 있습니다. 영속성 컨텍스트는 JPA가 엔티티를 바로 DB로 내보내지 않고 **트랜잭션 동안 엔티티를 관리하는 작업 공간**입니다. 개발자가 명시적으로 `U-`

`PDATE`를 쓰지 않아도 값을 바꾸기만 하면 DB에 반영되는 이유가 이 작업 공간의 동작 원리 때문입니다.

> **기준:** 이 글은 **Jakarta Persistence 3.1 (JPA 3.1) 명세**와 **Hibernate 6.x** 구현을 기준으로 작성합니다. 개념 정의는 Jakarta Persistence 3.1 Specification의 `§3. Entity Operations` 장을 참고하고, `flush`·변경 감지·배치 구현 세부는 Hibernate 6 User Guide를 인용합니다. 실제 SQL 동작은 **MySQL 8.4 + `InnoDB`** 기준이고, 트랜잭션 경계는 Spring의 `@Transactional`이 관리하는 상황을 전제합니다. 코드 예시는 Kotlin + Spring Data JPA로 작성합니다.

먼저 가장 짧은 답부터 보면

영속성 컨텍스트가 대신해 주는 일은 네 가지로 줄일 수 있습니다.

- **1차 캐시** — 같은 트랜잭션 안에서 같은 ID를 여러 번 조회해도 SQL은 한 번만 나갑니다
- **변경 감지** — 엔티티 필드를 바꾸기만 해도 `UPDATE`가 자동으로 만들어집니다
- **쓰기 지연** — `persist()`/`save()`는 바로 SQL을 내보내지 않고 모아서 내보냅니다
- **동일성 보장** — 같은 영속성 컨텍스트에서 같은 ID로 조회한 엔티티는 `===` 비교가 성립합니다

이 네 가지는 서로 독립된 기능이 아니라 **하나의 구조에서 파생**됩니다. 영속성 컨텍스트가 엔티티의 "현재 모습"과 "DB에서 처음 읽어온 모습

(스냅샷)"을 함께 들고 있기 때문에 가능한 일입니다.

Phase 1. 영속성 컨텍스트는 무엇을 대신해 주나요?

핵심: 영속성 컨텍스트는 엔티티의 논리 저장소입니다

Jakarta Persistence 3.1 명세 `§3.1`은 영속성 컨텍스트를 "**엔티티 인스턴스의 집합으로, 그 안에서 각 엔티티의 영속 식별자(persistent identity)에 대해 최대 하나의 인스턴스만 존재**"하는 공간으로 정의합니다.

한 줄로 줄이면 이렇습니다.

- 영속성 컨텍스트는 `EntityManager` **하나가 관리하는 엔티티들의 모음**입니다
- 한 영속성 컨텍스트 안에서 같은 `(엔티티 클래스, 식별자)` 쌍은 **정확히 한 개의 인스턴스**로만 존재합니다

이 규칙 하나가 뒤에 나올 1차 캐시, 변경 감지, 동일성 보장을 전부 떠받칩니다.

트랜잭션과의 관계

Spring 환경에서 `@Transactional`이 붙은 메서드에 들어가는 순간 **새 영속성 컨텍스트가 열리고**, 커밋되거나 롤백될 때 **영속성 컨텍스트도 닫힙니다**. 즉, 기본 설정에서 **영속성 컨텍스트의 수명 = 트랜잭션의 수명**입니다.

```
@Transactional 메서드 진입
  ├─ 영속성 컨텍스트 열림
  ├─ find/save/update 호출들이 이 안에서 일어남
  └─ 커밋 직전에 flush → 이후 영속성 컨텍스트 닫힘
```

트랜잭션이 끝나면 그 안에서 관리되던 엔티티들은 전부 `detached` 상태가 됩니다. 이 지점부터는 필드를 바꿔도 DB에 반영되지 않습니다.

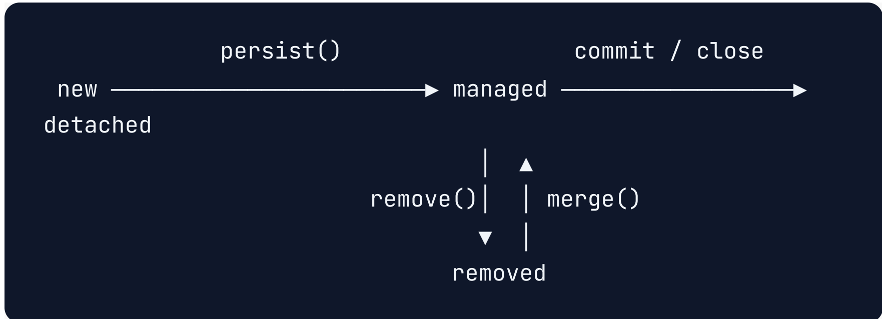
Phase 2. 엔티티의 네 가지 상태

JPA 명세는 엔티티가 가질 수 있는 상태를 네 가지로 정의합니다. 영속성 컨텍스트의 대부분의 동작은 이 네 상태 사이의 전이로 설명됩니다.

상태 정의

상태	설명	영속성 컨텍스트와의 관계
`new` / `transient`	`new User()` 로 막 만든 상태	관리 대상 아님
`managed` / `persistent`	영속성 컨텍스트가 추적 중인 상태	변경 시 자동으로 `UPDATE` 생성
`detached`	한때 관리됐지만 영속성 컨텍스트가 닫혀 추적이 끊긴 상태	변경해도 DB 반영 안 됨
`removed`	삭제 예약된 상태	커밋 시 `DELETE`

상태 전이 흐름



각 전이는 구체적으로 이렇게 일어납니다.

- `persist(entity)` — `new` 엔티티를 `managed`로 바꿉니다. 이 시점에 `INSERT` SQL이 바로 나가지는 않습니다 (단, `IDENTITY` 전략은 예외. Phase 5에서 다룹니다)
- `find()` / `JPQL` 조회 — DB에서 읽어온 엔티티를 `managed` 상태로 영속성 컨텍스트에 넣습니다
- `remove(entity)` — `managed` 엔티티를 `removed`로 바꿉니다. 커밋 시점에 `DELETE`가 나갑니다
- `detach(entity)` / 영속성 컨텍스트 종료 — `managed`가 `detached`로 바뀝니다
- `merge(detachedEntity)` — `detached` 엔티티의 **내용을 새 `managed` 인스턴스에 복사**합니다

가장 많이 틀리는 지점: `merge`는 원본을 영속화하지 않습니다

JPA 명세에서 `merge()`는 **인자로 받은 `detached` 엔티티를 반환값이 가리키는 새로운 `managed` 인스턴스에 복사**하는 연산입니다. 원본 `detached` 엔티티는 그대로 남아 있고, **반환된 새 인스턴스**만 `managed`입니다.

```
val detached = User(id = 1, name = "before")
val managed = entityManager.merge(detached)

detached.name = "after"    // 반영 안 됨 (still detached)
managed.name = "after"     // 반영됨 (managed)
```

`merge`가 필요한 경우는 제한적입니다. 대부분의 실무 코드는 **`find()`로 `managed` 상태의 엔티티를 먼저 가져와서 값을 바꾸는 패턴**이 더 안전합니다.

Phase 3. 1차 캐시 — 왜 같은 ID 조회가 한 번만 나가나요?

핵심: 식별자 기반 동일성 맵

영속성 컨텍스트는 내부적으로 **`(엔티티 클래스, 식별자) → 엔티티 인스턴스`** 로 매핑되는 Map을 들고 있습니다. 이걸 JPA에서는 **1차 캐시**, 또는 **identity map** 이라고 부릅니다.

`find()`가 호출되면 JPA는 이렇게 동작합니다.

1. 영속성 컨텍스트의 1차 캐시에 해당 `(클래스, id)` 키가 있는지 먼저 봅니다
2. 있으면 **DB에 쿼리를 보내지 않고** 이미 들고 있는 인스턴스를 반환합니다
3. 없으면 DB에서 읽어와 1차 캐시에 넣고 반환합니다

```
@Transactional
fun demo(id: Long) {
  val u1 = userRepository.findById(id).get()   // SELECT 실행
  val u2 = userRepository.findById(id).get()   // SQL 나가지 않음
  println(u1 === u2)   // true
}
```

동일성 보장의 의미

같은 영속성 컨텍스트 안에서 같은 ID로 조회한 엔티티는 `===` 가 성립합니다. 서로 다른 인스턴스가 아니라 **정확히 같은 객체**입니다. 이 보장이 있기 때문에 한 트랜잭션 안에서 한쪽에서 바꾼 값이 다른 쪽에서 그대로 보입니다.

1차 캐시의 범위가 좁다는 점도 같이 기억해야 합니다

- **트랜잭션 스코프**에서 끝납니다. 트랜잭션이 끝나면 캐시도 사라집니다
- **다른 트랜잭션과 공유되지 않습니다**. 1차 캐시는 요청 단위 캐시지, 애플리케이션 레벨 캐시가 아닙니다
- **JPQL로 조회하면 `SELECT` 가 반드시 나갑니다**. JPQL은 "먼저 DB에 물어보고, 그 결과를 1차 캐시에 있는 것과 맞춰" 반환합니다. JPQL

자체는 캐시를 읽지 않습니다

> **참고:** 애플리케이션 전체에서 공유되는 캐시가 필요하면 **2차 캐시(`SessionFactory` 레벨)** 나 **Redis 같은 외부 캐시**를 따로 구성해야 합니다. 2차 캐시는 별도 글에서 다룹니다. 외부 캐시 전략은 캐시 전략 글을 참고하세요.

Phase 4. 변경 감지(`Dirty Checking`) — 왜 `UPDATE` 없이 값이 반영되나요?

핵심: 스냅샷 비교로 바뀐 필드를 찾아냅니다

영속성 컨텍스트가 엔티티를 처음 로드할 때, JPA는 그 엔티티의 **스냅샷(초기 필드 값들)** 을 같이 저장해 둡니다. 그리고 커밋이나 `flush` 시점에 현재 엔티티의 값과 스냅샷을 비교해서 **바뀐 필드에 대해서만 `UPDATE` SQL을 생성**합니다.

```
@Transactional
fun rename(id: Long, newName: String) {
  val user = userRepository.findById(id).get()  // 스냅샷 저장
  user.name = newName                           // 필드만 변경
  // save() 호출 없이 트랜잭션 커밋
}
```

위 코드는 `save()` 를 부르지 않았지만 커밋 직전에 `UPDATE user SET ... WHERE id = ?` 이 나갑니다.

왜 이 설계가 안전한가요?

- **바뀐 엔티티만 SQL이 생성**되므로 불필요한 `UPDATE` 가 줄어듭니다
- **낙관적 잠금과 결합**하면 충돌을 정확히 판정할 수 있습니다 (`@Version` 과 함께)
- **트랜잭션 경계 안에서만 반영**되기 때문에 의도하지 않은 중간 상태가 남지 않습니다

주의: 기본적으로 모든 컬럼이 `UPDATE` 문에 포함됩니다

Hibernate는 기본 동작으로 **변경 감지로 바꿀 필드가 하나라도 생기면 엔티티의 모든 컬럼을 `UPDATE` SQL에 포함**시킵니다. 바뀐 컬럼만 포함시키려면 엔티티에 `@DynamicUpdate` 를 붙여야 합니다. 다만 이 옵션을 기본으로 켜는 것은 권장되지 않습니다. SQL이 동적으로 바뀌면 DB의 SQL 플랜 캐시 적중률이 떨어지기 때문에, 테이블의 컬럼 수가 아주 많거나 특정 컬럼의 `UPDATE` 비용이 현저히 큰 경우에 선택적으로 적용합니다.

```
@Entity
@DynamicUpdate
class Product(
  @Id val id: Long,
  var name: String,
  var description: String,
  // ... 수십 개 컬럼
)
```

흔한 오해: 변경 감지는 `setter` 호출 시점에 일어나지 않습니다

변경 감지는 `setter`**가 호출될 때 즉시 동작하지 않습니다.** `flush` 시점에 **모든** `managed` **엔티티의 현재 값과 스냅샷을 한꺼번에 비교**하는 방식입니다. 그래서 한 트랜잭션 안에서 같은 필드를 여러 번 바꿔도 최종적으로 나가는 `UPDATE`는 한 번입니다.

Phase 5. 쓰기 지연과 `flush` — SQL은 언제 실제로 나가나요?

핵심: 영속성 컨텍스트는 SQL을 모아뒀다가 한꺼번에 내보냅니다

`persist()`로 예약된 `INSERT`, 변경 감지로 생긴 `UPDATE`, `remove()`로 생긴 `DELETE`는 **바로 DB로 전송되지 않습니다.** 영속성 컨텍스트가 내부 큐(**action queue**)에 쌓아뒀다가 `flush` **시점에 한꺼번에 내보냅니다.**

`flush`가 일어나는 세 가지 시점

Hibernate 6 기준, `flush`는 다음 중 하나일 때 일어납니다.

1. **트랜잭션 커밋 직전** — 기본 동작. 가장 흔합니다
2. **JPQL / Criteria 쿼리 실행 직전** — `FlushMode`가 `AUTO`(기본값)일 때. 쿼리가 방금 변경한 내용을 보려면 DB에 먼저 반영돼야 하기 때문입니다
3. **`em.flush()` 를 명시적으로 호출** — ID가 필요하거나 특정 시점에 강제로 내보내야 할 때

```
@Transactional
fun createOrder(userId: Long): Long {
    val order = Order(userId = userId)
    val saved = orderRepository.save(order)

    // 여기서 JPQL을 실행하면 AUTO flush로 INSERT가 먼저 나감
    val count = orderRepository.countByUserId(userId)

    return saved.id
    // 메서드 종료 시 커밋 직전에 남은 작업들이 flush
}
```

`FlushMode`를 `COMMIT`으로 바꾸면?

JPA의 `FlushModeType`은 `AUTO`(기본)와 `COMMIT` 두 가지입니다. `COMMIT`으로 바꾸면 쿼리 실행 전에 `flush`가 일어나지 않고 **오직 커밋 직전에만** 일어납니다.

- **장점**: 불필요한 중간 `flush`가 줄어 성능에 유리할 수 있습니다
- **단점**: 쿼리 결과가 "방금 변경한 내용"을 반영하지 않을 수 있습니다. 정말 의도가 분명한 경우에만 선택하세요

`INSERT`가 앞당겨지는 이유 — `IDENTITY` 전략의 함정

ID 생성 전략이 `GenerationType.IDENTITY`일 때는 `persist()` 호출 순간에 바로 `INSERT`가 실행됩니다. ID를 DB가 생성하기 때문에, ID를 모르면 영속성 컨텍스트에 넣을 수조차 없습니다. 이 때문에 **`IDENTITY` 전략은 JDBC batch insert 최적화를 막습니다**. Hibernate가 엔티티를 영속화하는 매 순간 ID를 얻으려고 `INSERT`를 개별적으로 실행하기 때문에, `hibernate.jdbc.batch_size`를 설정해도 묶이지 않습니다.

대용량 `INSERT`가 많은 테이블이라면 다른 선택지를 검토합니다.

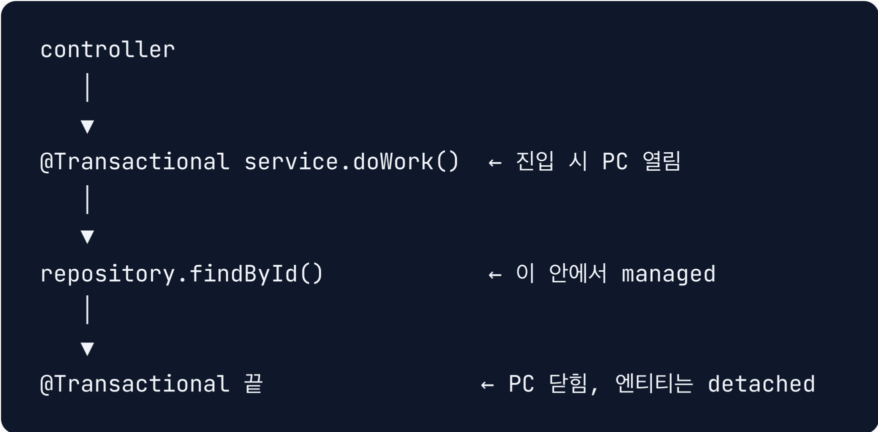
- **PostgreSQL/Oracle**: `SEQUENCE` 전략 (ID를 미리 받아와 배치 가능)
- **MySQL**: `AUTO_INCREMENT = IDENTITY`이므로 시퀀스가 없습니다. 애플리케이션이 ID를 만들어 `UUIDv7`이나 `Snowflake ID`를 쓰는 접근이 현실적입니다

배치 쓰기 최적화 자체는 별도 글에서 깊게 다룹니다.

Phase 6. 영속성 컨텍스트의 범위는 어디까지인가요?

기본값: 트랜잭션 스코프

Spring의 `@Transactional`과 함께 쓰는 일반적인 설정에서 영속성 컨텍스트는 **하나의 트랜잭션과 같은 수명**을 가집니다.



OSIV — 컨트롤러/뷰까지 확장되는 범위

Spring Boot는 기본적으로 `spring.jpa.open-in-view=true` 입니다. 이 옵션이 켜져 있으면 **영속성 컨텍스트의 수명이 HTTP 요청 단위**로 확장됩니다. 컨트롤러와 뷰 렌더링 단계에서도 `LAZY` 연관이 로딩됩니다.

OSIV의 실질적 의미는 다음과 같습니다.

- **장점**: 컨트롤러/뷰에서 `LAZY` 연관을 접근해도 `LazyInitializationException` 이 터지지 않습니다
- **단점**: **DB 커넥션이 뷰 렌더링이 끝날 때까지 반환되지 않습니다**. 요청 단위로 커넥션을 물고 있기 때문에 외부 API 호출이 섞이거나 뷰 렌더링이 느려지면 커넥션 풀이 금방 고갈됩니다

트래픽이 큰 서비스는 **`open-in-view=false`** 로 끄고, 서비스 레이어에서 필요한 연관을 전부 로딩해 DTO로 내려주는 패턴을 권장합니다. 커넥션 풀 동작과 고갈 원인은 커넥션 풀 글에서 다뤘습니다.

`Extended` 영속성 컨텍스트

JPA 명세는 `@PersistenceContext(type = PersistenceContextType.EXTENDED)` 로 영속성 컨텍스트의 수명을 여러 트랜잭션에 걸쳐 유지할 수 있게 허용합니다. 다만 Stateful Session Bean 같은 환경이 전제되기 때문에 일반적인 Spring Boot + `@Transactional` 구성에서는 거의 쓰지 않습니다.

Phase 7. 자주 만나는 함정

1. 트랜잭션 밖에서 엔티티 수정

```
fun update(id: Long, newName: String) {
  val user = userRepository.findById(id).get()
  user.name = newName
  // 반영 안 됨
}
```

`@Transactional`이 없어 `findById` 호출 직후 트랜잭션이 끝나고, 엔티티는 `detached` 상태가 됩니다. 변경은 메모리의 인스턴스에만 남고 DB로 흘러가지 않습니다. 이런 실수는 대부분 **서비스 메서드에 `@Transactional`을 빼먹어서** 생깁니다.

2. `@Transactional` 이 `self-invocation` 으로 우회됨

`@Transactional` 은 Spring AOP 프록시 기반이라 **같은 클래스 안에서 다른 메서드를 직접 호출하면 프록시를 거치지 않습니다**. 즉 `@Transactional` 이 아예 걸리지 않습니다. 이 주제는 다음 글에서 따로 다룹니다.

3. 영속성 컨텍스트가 너무 커짐

한 트랜잭션에서 수만 건의 엔티티를 `findAll()` 해서 루프를 돌리면 1차 캐시가 메모리를 크게 차지합니다. 벌크 작업에서는 다음 중 하나를 선택합니다.

- 주기적으로 `em.flush()` + `em.clear()` 로 영속성 컨텍스트를 비웁니다
- JPQL 벌크 `UPDATE` / `DELETE` 를 씁니다
- 아예 `JdbcTemplate` 이나 `JDBC` 배치로 우회합니다

4. 벌크 JPQL 이후 영속성 컨텍스트 오염

```
@Transactional
fun deactivateAll() {
  val users = userRepository.findAll()        // 1000건
managed

  em.createQuery("UPDATE User u SET u.active = false")
    .executeUpdate()                          // DB만 바뀜

  users.first().active   // true (1차 캐시는 그대로)
}
```

JPQL 벌크 연산은 **영속성 컨텍스트를 거치지 않고 DB에만 반영**합니다. 이 시점 이후의 1차 캐시는 DB와 달라진 상태가 됩니다. 벌크 연산 직후에는 `em.clear()` 를 부르거나, 그 이후에 해당 엔티티를 쓰지 않도록 흐름을 나눕니다.

정리

영속성 컨텍스트는 "자동으로 SQL을 만들어 주는 마법"이 아니라, **엔티티의 현재 모습과 DB에서 읽어온 스냅샷을 같이 들고 있으면서 커밋 직전에 그 둘을 비교해 필요한 SQL을 만드는 구조**입니다. 네 가지 효과는 이 구조에서 자연스럽게 따라 나옵니다.

효과	원리
1차 캐시	`(클래스, id) → 인스턴스` 매핑을 유지
변경 감지	현재 값과 스냅샷을 `flush` 시점에 비교
쓰기 지연	SQL을 큐에 쌓아 `flush` 시점에 배치 전송
동일성 보장	같은 식별자에 대해 항상 같은 인스턴스만 반환

실무에서 실수는 대부분 **영속성 컨텍스트의 범위를 오해**해서 생깁니다. 트랜잭션 밖, `self-invocation`으로 프록시를 거치지 않은 경우, 벌크 JPQL 이후처럼 **영속성 컨텍스트가 닫혔거나 우회된 상황**을 먼저 의심하면 원인을 빨리 찾을 수 있습니다.

이 글에서 잡은 기반 위에서 다음 글은 **`LAZY` / `EAGER` Fetch 전략과 N+1이 생기는 진짜 이유**로 이어집니다. 변경 감지와 1차 캐시가 머리에 들어오면, N+1이 왜 생기고 왜 `fetch join`이 그것을 해결하는지가 훨씬 명확해집니다.

Chapter 2

JPA `flush`와 OSIV 완전 정복 — 영속성 컨텍스트의 경계는 어디까지인가요?

#영속성 컨텍스트

flush가 언제 일어나는지, FlushMode별 동작 차이, OSIV가 커넥션을 언제까지 붙들고 있는지, 그리고 실무에서 open-in-view를 끄면 벌어지는 일들을 Hibernate/Spring Boot 레퍼런스 기준으로 정리합니다.

`flush`와 OSIV, 왜 따로 다뤄야 하나요?

영속성 컨텍스트의 동작 원리는 앞 글에서 다뤘습니다. 이 글은 그 다음 질문 — **"그래서 `flush`는 정확히 언제 일어나고, 영속성 컨텍스트는 언제까지 열려 있는가"**에 답합니다. 실무에서 자주 만나는 증상들이 전부 이 경계에서 발생합니다.

- **목록 API 응답이 튀는데 쿼리 로그에는 이상한 `SELECT`가 뷰 렌더링 중에 찍혀 있습니다**
- 벌크 `UPDATE` JPQL 이후 **이전에 로드한 엔티티 값이 DB와 다릅니다**
- 외부 API를 호출하는 서비스에서 **커넥션 풀이 쉽게 고갈됩니다**
- `@Transactional(readOnly = true)`로 바꿨는데 **기대한 만큼 빨라지지 않습니다**

이 증상들은 `flush`가 언제 일어나는지와 영속성 컨텍스트가 어느 범위에서 살아 있는지를 알지 못하면 디버깅이 오래 걸립니다. 이 글은 두 개념을 한 줄에 묶어 **"쓰기 타이밍"과 "스코프 경계"** 두 축으로 정리합니다.

> **기준:** 이 글은 **Jakarta Persistence 3.1 (JPA 3.1) 명세**와 **Hibernate 6.x** / **Spring Boot 3.x** 기준으로 작성합니다. `flush`는 Jakarta Persistence 3.1 — §3.2.4 Synchronization to the Database와 Hibernate 6 User Guide — §8. Flushing 기준으로 설명합니다. OSIV는 Spring Boot의 `spring.jpa.open-in-view` 속성 동작과 `OpenEntityManagerInViewInterceptor` JavaDoc을 참조합니다. 코드 예시는 Kotlin + Spring Boot입니다.

먼저 가장 짧은 답부터 보면

- `flush`는 `COMMIT`**이 아닙니다.** SQL을 DB에 "보내는 것"일 뿐이며, 커밋은 별개 단계에서 일어납니다
- `FlushMode`의 기본값은 `AUTO`로, **JPQL 실행 직전과 커밋 직전**에 자동으로 `flush`합니다
- OSIV(Open Session In View)는 영속성 컨텍스트를 **HTTP 요청이 끝날 때까지** 열어두는 옵션입니다. 기본 `true`입니다
- OSIV가 켜져 있으면 **DB 커넥션이 서비스 레이어를 벗어나서도 뷰 렌더링이 끝날 때까지 점유**됩니다
- 트래픽이 크면 OSIV를 끄고 **서비스 레이어에서 DTO로 내려주는 패턴**이 권장됩니다

Phase 1. `flush`와 `commit`은 같은 게 아닙니다

핵심: `flush`는 SQL 전송, `commit`은 트랜잭션 확정

이 둘은 아주 자주 혼동됩니다. 실제로 일어나는 일은 완전히 다릅니다.

동작	의미
`flush`	영속성 컨텍스트에 쌓여 있던 SQL을 DB로 **전송**
`commit`	그 시점까지 DB에 쓰인 변경 사항을 **영구화**

`flush` 후에 `rollback`이 오면, DB에 이미 전송된 SQL은 **롤백**됩니다. 즉 `flush`로 보낸 내용은 아직 **트랜잭션 내부의 중간 상태**일 뿐입니다. 이 차이 때문에 `em.flush()`를 호출해도 다른 트랜잭션에서는 그 변경이 보이지 않습니다 (기본 격리 수준 `REPEATABLE READ` 기준).

```
BEGIN
  INSERT ...      ← em.persist()
  UPDATE ...      ← dirty checking
  em.flush()      ← 여기서 위 SQL이 DB로 전송
  SELECT ...      ← 같은 트랜잭션에서는 보임
  em.flush()를 여러 번 해도 커밋 전까지는 다른 세션에 안 보임
COMMIT            ← 이 순간부터 영구 반영
```

왜 이 구분이 중요한가요?

`flush`를 명시적으로 호출해야 하는 경우는 드뭅니다. 다만 다음 상황에서는 `flush` 시점을 정확히 알아야 합니다.

- **벌크 JPQL 직전**: `AUTO` flush 모드는 JPQL 실행 전에 `flush`를 내보내 **쓰기 지연 상태를 DB에 반영**합니다
- **ID가 당장 필요할 때**: `IDENTITY` 전략이 아닌 경우, `persist()` 후에 ID를 써야 하면 명시적 `flush`가 필요할 수 있습니다
- **벌크 루프 처리**: 영속성 컨텍스트가 너무 커지지 않게 주기적으로 `em.flush() + em.clear()`를 수동으로 호출

Phase 2. `FlushMode` 세 가지의 차이

JPA 명세는 `FlushModeType.AUTO`와 `FlushModeType.COMMIT` 두 가지만 정의합니다. Hibernate는 여기에 `MANUAL`을 확장으로 추가합니다.

모드	`flush` 트리거	용도
`AUTO` (기본값)	커밋 직전 + JPQL/Criteria 실행 직전	대부분의 실무 코드
`COMMIT`	커밋 직전에만	중간 flush를 의도적으로 피하고 싶을 때
`MANUAL` (Hibernate 확장)	`em.flush()` 호출 시에만	**읽기 전용 전용** 최적화

`AUTO` — 왜 JPQL 전에 `flush`가 필요한가요?

영속성 컨텍스트에만 있는 변경 사항은 **DB에 아직 반영되지 않은 상태**입니다. 이 상태에서 JPQL `SELECT` 를 날리면, 방금 영속성 컨텍스트에서 바꾼 값은 조회 결과에 반영되지 않습니다. `AUTO` 모드는 이 상황을 막기 위해 **쿼리 직전에 자동으로 `flush`** 합니다.

```
@Transactional
fun demo(id: Long) {
  val order = em.find(Order::class.java, id)
  order.status = "CANCELED"   // 아직 DB에 안 감

  val canceled = em.createQuery(
    "SELECT o FROM Order o WHERE o.status = 'CANCELED'",
    Order::class.java
  ).resultList
  // 위 SELECT 직전에 AUTO flush → UPDATE가 먼저 나가고
  // 그 결과 canceled에 해당 Order가 포함됨
}
```

`COMMIT` — 중간 flush 비용을 줄이고 싶을 때

루프 안에서 수많은 JPQL을 섞어 쓰는 코드가 있다면 `AUTO` 는 매 쿼리 직전에 `flush`를 반복할 수 있습니다. `COMMIT`으로 바꾸면 **커밋 직전에만** `flush`가 일어나서 불필요한 중간 전송이 줄어듭니다.

단점은 **JPQL 결과가 방금 변경한 내용을 반영하지 않을 수 있다**는 것입니다. 팀 전체가 이 동작을 인지하지 못하면 **조용히 잘못된 결과**를 내는 버그가 생깁니다. 이 모드를 전역으로 켜는 것은 권장하지 않습니다.

Hibernate `MANUAL` — 읽기 전용 전용

Hibernate 확장 `MANUAL` 모드는 **`em.flush()`를 명시적으로 호출하지 않는 한 절대 `flush`하지 않습니다.** `@Transactional(readOnly = true)` 를 켜면 Spring의 JPA 통합이 이 모드를 자동으로 적용합니다.

`readOnly = true` 의 실제 효과가 바로 이것입니다.

- 변경 감지 스냅샷 비교를 **건너뜁니다**
- `flush` 자체가 일어나지 않아 **쓰기 지연 큐도 처리하지 않습니다**
- 결과적으로 읽기 전용 트랜잭션의 성능이 쓰기 트랜잭션보다 가볍습니다

이 효과 때문에 **목록 조회/리포트 API의 서비스 메서드에는 거의 기본값처럼** `@Transactional(readOnly = true)` 를 붙입니다.

Phase 3. `flush` 시점에 Hibernate가 실제로 하는 일

순서

Hibernate는 `flush` 시점에 다음 작업을 **이 순서대로** 수행합니다.

```
1. 영속성 컨텍스트의 모든 managed 엔티티를 훑음
2. 변경 감지 — 스냅샷과 비교해 달라진 엔티티 목록을 추림
3. action queue 정렬 — 고정된 실행 순서로 SQL 생성
   (1) OrphanRemoval      ← 고아 객체 제거
   (2) EntityInsert        ← INSERT
   (3) EntityUpdate        ← UPDATE
   (4) CollectionRemove / CollectionUpdate /
CollectionRecreate
   (5) EntityDelete        ← DELETE
4. JDBC 배치로 SQL 전송
5. 영속성 컨텍스트 상태를 "flushed"로 마킹
```

순서가 고정되어 있다는 점의 의미

같은 엔티티에 대해 `UPDATE` 와 `DELETE` 를 섞어 부르면 Hibernate가 재정렬합니다. 그래서 **메서드에서 호출한 순서와 실제 SQL이 나가는 순서는 다를 수 있습니다**. 특히 `orphanRemoval` 은 `INSERT` **보다 먼저 실행**되기 때문에, 같은 트랜잭션에서 "자식 하나를 제거하고 새 자식을 추가"하면 기대와 달리 유니크 제약이 잡히는 등 예상치 못한 동작이 생기기 쉽습니다.

정렬 순서를 보장하면서 순차적으로 내보내야 한다면 **중간에** `em.flush()` **를 수동으로** 호출해 경계를 명확히 나누는 편이 안전합니다.

벌크 JPQL 이후 영속성 컨텍스트는 이미 "오염"되어 있습니다

```
@Transactional
fun deactivateAll() {
  val users = userRepository.findAll()  // 영속성 컨텍스트에 로딩

  em.createQuery("UPDATE User u SET u.active = false")
    .executeUpdate()  // DB는 바뀌지만 1차 캐시는 그대로

  users.first().active  // 여전히 true
}
```

JPQL 벌크 연산은 **영속성 컨텍스트를 거치지 않고 DB에 직접 반영**됩니다. `flush`와 무관합니다. 이 시점 이후 1차 캐시의 엔티티는 **DB와 다른 상태**가 됩니다. 벌크 연산 직후에는 `em.clear()`를 호출해 영속성 컨텍스트를 비우고, 필요하면 다시 로딩하는 편이 안전합니다.

Phase 4. OSIV — 영속성 컨텍스트를 HTTP 요청 전체로 확장

OSIV가 하는 일

OSIV(Open Session In View)는 Spring Boot가 기본으로 `true`로 켜두는 설정입니다. 이 설정이 켜져 있으면 **영속성 컨텍스트의 수명이 서비스 레이어가 아니라 HTTP 요청 전체**로 늘어납니다.

```
OSIV OFF
  [Filter] → [Controller] → [Service @Transactional] →
[Repository]
                                   └ PC 열림 ─ PC 닫힘 ┘
                            (이후 컨트롤러/뷰에서 LAZY 접근 시 예
외)

OSIV ON
  [Filter] → [Controller] → [Service @Transactional] →
[Repository]
      └ PC 열림 ───────────────────────────────────── PC 닫힘 ┘
                                                      (View
렌더링 이후)
```

OSIV가 커넥션을 언제 점유하는가

흔한 오해가 있습니다. "OSIV가 켜져 있으면 요청 전체가 같은 트랜잭션이다"라는 생각입니다. 사실은 다릅니다.

- **트랜잭션**은 여전히 `@Transactional` 경계에서 시작하고 끝납니다. OSIV와 무관합니다
- OSIV가 유지하는 것은 **영속성 컨텍스트(`EntityManager`)와 DB 커넥션** 입니다

즉, `@Transactional`이 끝나 커밋되었어도 **커넥션 자체는 반환되지 않고** 영속성 컨텍스트에 매달려 뷰 렌더링이 끝날 때까지 대기합니다. 이 때문에 **뷰 렌더링이 느리면 커넥션이 그만큼 오래 점유**됩니다.

왜 이 설계가 만들어졌나요?

OSIV는 한때 **뷰에서 `LazyInitializationException`을 피하기 위한 실용적 해결책**이었습니다. 서버 사이드 템플릿(JSP, Thymeleaf)이 엔티티를 직접 받아 렌더링하는 시절에는, 뷰에서 `user.orders`처럼 LAZY 연관에 접근해도 예외가 터지지 않도록 영속성 컨텍스트를 계속 열어두는 것이 유용했습니다. 그래서 Spring Boot는 **호환성과 편의를 위해 기본값을 `true`** 로 둡니다.

트래픽이 커지면 무엇이 문제인가요?

API 서버 시대에는 다음 이유로 **OSIV가 오히려 병목**이 됩니다.

1. **커넥션 점유 시간이 뷰/직렬화 단계까지 늘어납니다**. HTTP 응답이 큰 JSON 직렬화에 시간이 걸리면 그 동안 커넥션이 반환되지 않습니다
2. **외부 API 호출이 컨트롤러 레이어에 섞이면** 그 동안 커넥션이 물린 채 네트워크 대기를 합니다
3. **동시 요청 수에 비례해 커넥션 수요**가 커져 풀이 쉽게 고갈됩니다. 이 동작은 커넥션 풀 글에서 다룬 고갈 패턴과 직접 연결됩니다

OSIV를 끌 때 벌어지는 일

```yaml
spring:
  jpa:
    open-in-view: false
```

이 설정을 끄면 **서비스 레이어를 벗어나는 순간 영속성 컨텍스트가 닫힙니다**. 커넥션도 그 시점에 반환됩니다.

대신 이런 증상들이 드러납니다.

- 컨트롤러/뷰에서 LAZY 연관을 접근하면 `LazyInitializationException` 이 발생합니다
- 엔티티를 그대로 JSON 직렬화하던 코드가 연관 필드에서 예외를 냅니다

이 예외들은 **원래부터 있던 구조적 위험이 수면 위로 올라온 것**입니다. OSIV는 이걸 "뷰까지 LAZY가 되도록" 미뤄뒀을 뿐, 위험 자체를 없앤 적이 없습니다.

Phase 5. OSIV를 끄는 실무 패턴

원칙: 서비스 레이어에서 필요한 모든 것을 조합해 DTO로 내려준다

OSIV를 끄는 순간 구조적 질문 하나가 강제됩니다. **"응답에 필요한 데이터를 서비스 레이어에서 확정 지을 수 있는가?"** 답이 예여야 합니다.

```
@Service
class OrderQueryService(
  private val orderRepository: OrderRepository
) {

  @Transactional(readOnly = true)
  fun getOrderSummary(userId: Long): List<OrderSummaryDto>
{
    val orders =
orderRepository.findWithItemsByUserId(userId)
    // 이 시점에 items도 이미 로딩됨
    return orders.map { OrderSummaryDto.from(it) }
    // 트랜잭션이 끝나기 전에 DTO 변환 완료
  }
}
```

핵심은 두 가지입니다.

- 서비스 메서드 안에서 **필요한 연관을 미리 로딩** (fetch join, `@EntityGraph`, `@BatchSize`. 앞 글 참고)
- 서비스 메서드 안에서 **DTO로 변환해 내려줌**

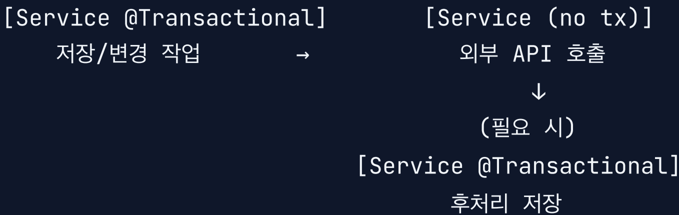
컨트롤러에는 순수 DTO만 전달

컨트롤러 레이어까지 엔티티를 내려보내면 OSIV가 꺼진 상태에서 LAZY 예외가 언제든 튈 수 있습니다. 컨트롤러에서 받는 타입은 **DTO**로 제한합니다. 이 규칙을 유지하면 OSIV가 켜졌든 꺼졌든 동작이 바뀌지 않습니다.

외부 API 호출은 트랜잭션 밖에서

외부 HTTP 호출은 **트랜잭션 안에 들어오면 안 됩니다.** 응답이 느리거나 멈추면 커넥션이 그 시간만큼 점유되고, 요청이 몰리면 풀이 금방 비어 버립니다.

일반적인 실무 구조는 이렇습니다.



외부 호출을 기준으로 트랜잭션을 쪼개면, 커넥션 점유 시간이 크게 줄어듭니다.

`readOnly = true`를 기본값처럼 쓰기

조회 API의 서비스 메서드에는 **대부분** `@Transactional(readOnly = true)` 를 기본으로 붙입니다. Phase 2에서 설명한 대로 Hibernate의 flush가 비활성화되고, 리드 레플리카로 라우팅하는 구성을 조합하기 쉬워집니다.

정리

`flush`와 OSIV는 한 가지 공통 질문을 던집니다. "**영속성 컨텍스트의 쓰기와 스코프가 정확히 어디서 일어나는가?**"

`flush` 쪽의 기억할 점은 이렇습니다.

- `flush` 는 **커밋이 아닙니다**. 커밋 전까지는 트랜잭션 내부 상태일 뿐입니다
- `AUTO` 모드의 기본 동작은 **JPQL 실행 직전과 커밋 직전** 두 시점에 자동으로 일어납니다
- `@Transactional(readOnly = true)` 는 Hibernate `MANUAL` 모드를 켜서 flush 자체를 건너뛰게 합니다
- 벌크 JPQL은 **영속성 컨텍스트를 우회**하므로 직후에는 `em.clear()` 를 고려합니다

OSIV 쪽의 기억할 점은 이렇습니다.

- 기본값은 `true` 이고, 켜져 있으면 **커넥션이 뷰 렌더링이 끝날 때까지 점유**됩니다
- 트래픽이 큰 API 서버에서는 끄는 편이 거의 항상 안전합니다
- 끄면 **LAZY 예외가 서비스 레이어 위에서 드러나므로**, 필요한 연관은 서비스 메서드에서 미리 로딩해 **DTO로 확정**시켜 내려줍니다
- 외부 API 호출은 트랜잭션 밖으로 빼 커넥션을 일찍 반환시킵니다

이 두 축을 맞추고 나면, 서비스 레이어와 컨트롤러 사이에서 벌어지는 대부분의 "이상한 쿼리"와 "커넥션 고갈"을 줄일 수 있습니다. 다음 글은 JPA의 쓰기 성능을 밀어붙이는 실제 방법 — **배치** `INSERT` **와** `hibernate.jdbc.batch_size` 를 다룹니다.

PART 2

2부. Fetch 전략과 N+1 해결

Chapter 3

JPA Fetch 전략 완전 정복 — `LAZY` vs `EAGER`와 N+1이 생기는 진짜 이유

#Fetch 전략 #N+1

연관 관계 기본 Fetch 전략, 프록시로 구현되는 LAZY 동작, EAGER에서도 N+1이 생기는 이유, 컬렉션과의 조합에서 발생하는 카르테시안 폭발까지 JPA Fetch 전략의 내부 동작을 정리합니다.

Fetch 전략, 왜 알아야 하나요?

JPA에서 N+1이 생기는 원인은 대부분 "뭐가 `LAZY`고 뭐가 `EAGER`인지 정확히 모른 채 연관을 걸었기 때문"입니다.

- `@ManyToOne`만 걸어뒀는데 목록 조회마다 연관 테이블로 20번씩 쿼리가 나갑니다
- `fetch = FetchType.EAGER`로 바꿨더니 오히려 쿼리 수가 더 늘었습니다
- 두 개의 `@OneToMany`를 `JOIN FETCH`했더니 결과 행이 수백 배로 불어났습니다
- 로그에는 `SELECT` 한 번만 찍혔는데 응답 시간은 여전히 튑니다

이 문제들의 공통점은 **"어떤 쿼리가 언제 나가는가"**를 모르는 데 있습니다. `LAZY`/`EAGER`는 단순한 튜닝 옵션이 아니라 **JPA가 프록시와 영속성**

컨텍스트로 어떻게 연관을 해석하는지를 결정하는 구조적 선택입니다. 이 글은 그 내부 동작을 풀어봅니다.

> **기준:** 이 글은 **Jakarta Persistence 3.1 (JPA 3.1) 명세**와 **Hibernate 6.x** 구현을 기준으로 작성합니다. 연관 관계 Fetch 디폴트 정의는 Jakarta Persistence 3.1 Specification `§11. Metadata Annotations`, 프록시/`Bytecode Enhancement` 구현 세부는 Hibernate 6 User Guide — § 5. Fetching을 참조합니다. 1차 캐시와 변경 감지의 동작은 앞 글을 전제로 합니다. 코드 예시는 Kotlin + Spring Data JPA입니다.

먼저 가장 짧은 답부터 보면

- `@ManyToOne`과 `@OneToOne`은 기본값이 **`EAGER`** 입니다
- `@OneToMany`와 `@ManyToMany`는 기본값이 **`LAZY`** 입니다
- `LAZY`는 실제 데이터를 쓸 때 추가 SQL을 내보냅니다 → **루프 안에서 접근하면 N+1**
- `EAGER`도 **연관마다 개별 쿼리**를 내보낼 수 있습니다 → **목록 조회 시 N+1**
- 두 개 이상의 `@OneToMany`를 `JOIN FETCH`하면 **카르테시안 곱**으로 행이 폭증합니다
- 실무 원칙은 **모든 연관을 `LAZY`로 시작하고, 쿼리별로 필요한 연관을 명시적으로 로딩**합니다

이 글은 왜 이 원칙이 나오는지 순서대로 짚습니다.

Phase 1. 연관 관계의 기본 Fetch 전략

명세가 정한 디폴트

JPA 3.1 명세는 연관 관계 애너테이션마다 기본 Fetch 전략을 정해두고 있습니다.

애너테이션	기본 Fetch	이유
`@ManyToOne`	`EAGER`	역사적 이유로 단일 연관은 즉시 로딩이 기본값
`@OneToOne`	`EAGER`	단일 연관이라 같은 이유
`@OneToMany`	`LAZY`	컬렉션을 무조건 읽으면 대부분 오버페치가 됩니다
`@ManyToMany`	`LAZY`	같은 이유

이 디폴트가 왜 위험한가요?

대부분의 엔티티에는 `@ManyToOne` 연관이 하나 이상 있습니다. 그리고 그것들은 **기본적으로 `EAGER`** 입니다. 그래서 아무 생각 없이 엔티티를 설계하면, **목록 조회 때마다 연관 엔티티가 매 행마다 개별 `SELECT` 로 따라 나오는 구조**가 됩니다.

실무에서 권장되는 첫 번째 규칙은 거의 하나입니다.

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "user_id")
val user: User

@OneToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "profile_id")
val profile: Profile
```

모든 연관을 `LAZY` 로 명시한 다음, 필요한 곳에서 `fetch join` 또는 `@EntityGraph` 로 읽는 것입니다. 이 도구들의 비교는 다음 글에서 다룹니다.

Phase 2. `LAZY` 는 어떻게 구현돼 있나요?

핵심: `LAZY` 연관은 엔티티가 아니라 "프록시 객체"로 채워집니다

`fetch = LAZY` 가 걸린 연관은, 부모 엔티티를 로드한 시점에 **실제 엔티티 대신 프록시 객체**가 필드에 들어갑니다. 이 프록시는 `Hibernate` 가 런타임에 생성한 엔티티의 서브클래스(`ByteBuddy` 또는 `CGLIB`)로, 내부에는 **ID만 채워져 있고** 다른 필드는 비어 있습니다.

```
Order (managed)
 ├─ id = 1
 └─ user  → User$HibernateProxy (id = 10, 나머지 필드 = 초기화 X)
```

프록시가 실제 데이터를 채우는 순간

프록시의 `id`**를 제외한 다른 필드**를 읽으려 하면, 그 순간 프록시가 `SELECT`를 내보내 영속성 컨텍스트를 통해 실제 데이터를 채웁니다. 이걸 **프록시 초기화(Proxy Initialization)** 라고 부릅니다.

```
val order = orderRepository.findById(1L).get()
// 이 시점까지는 User 프록시만 있음, SQL 안 나감

println(order.user.id)     // SQL 안 나감 — 프록시가 ID는 알고 있음
println(order.user.name)   // 이 순간 SELECT user WHERE id = 10 실행
```

영속성 컨텍스트가 닫힌 뒤에 접근하면?

프록시는 **영속성 컨텍스트가 열려 있을 때만** 초기화할 수 있습니다. 트랜잭션이 끝난 뒤 `order.user.name`을 접근하면 `LazyInitializationException`이 발생합니다. OSIV가 켜진 상태에서는 이 예외가 뷰 렌더링까지는 늦춰집니다 — 그래서 OSIV를 끄면 이 예외가 자주 튀어나오는 것처럼 보이지만, **사실은 이전에 숨어 있던 문제가 드러난 것**입니다.

Bytecode Enhancement를 쓰면?

Hibernate는 프록시 대신 **바이트코드 enhancement**로 `LAZY`를 구현하는 옵션을 제공합니다. 이 방식은 엔티티 클래스 자체에 훅을 심어 필드 접근 시점에 로딩합니다. 프록시 기반 구현의 한계(`final` 클래스/

메서드 사용 불가, equals/hashCode 함정 등)를 우회할 때 선택합니다. 다만 빌드 도구 설정이 늘어나기 때문에 대부분 프로젝트는 프록시 방식을 그대로 씁니다.

Phase 3. 왜 `LAZY`가 N+1을 만드나요?

`LAZY`는 **한 번의 접근 = 한 번의 `SELECT`** 이기 때문에, 루프 안에서 접근하면 반복 횟수만큼 쿼리가 나갑니다.

```
@Transactional(readOnly = true)
fun listOrderSummaries(): List<OrderView> {
  val orders = orderRepository.findAll()    // SELECT order
× 1
  return orders.map { o →
    OrderView(o.id, o.user.name)            // SELECT user
× N
  }
}
```

주문이 20건이면 쿼리가 21번 나갑니다. 이 구조적 원인은 N+1 글에서 이미 다뤘으므로, 여기서는 **왜 이 구조가 JPA 구현의 자연스러운 귀결인지**만 짚습니다.

- `LAZY` 프록시는 **자기 자신의 필드 접근에 대해서만 초기화**를 트리거합니다
- Hibernate는 루프의 다음 이터레이션에서 어떤 프록시가 접근될지 **예측하지 않습니다**
- 그래서 각 프록시는 **접근되는 순간에 하나씩** 쿼리를 내보냅니다

다시 말해, N+1은 `LAZY`의 결함이 아니라 **`LAZY`가 일부러 "요청받을 때만 로딩"하도록 설계된 결과**입니다. 이 결과를 회피하려면 **읽기 전에 필요한 연관을 한 번에 같이 로딩하도록 쿼리를 다시 써야** 합니다. 그 도구가 `fetch join`, `@EntityGraph`, `@BatchSize` 등이고, 다음 글의 주제입니다.

Phase 4. `EAGER`도 N+1을 만듭니다

많은 문서가 "`EAGER`는 N+1을 피한다"고 단순화하지만, 정확하지 않습니다. **`EAGER`는 N+1을** 더 조용히 **만드는 경우가 많습니다.**

`EAGER`가 쿼리를 내보내는 방식

`fetch = EAGER` 연관은 부모를 로딩한 뒤, **로딩된 부모마다 개별 `SELECT`**를 추가로 내보낼 수 있습니다. JPQL로 엔티티 목록을 읽었을 때 특히 이 동작이 잘 보입니다.

```
@Entity
class Order(
  @Id val id: Long,

  @ManyToOne(fetch = FetchType.EAGER)
  @JoinColumn(name = "user_id")
  val user: User,
)

@Transactional(readOnly = true)
fun list() {
  em.createQuery("SELECT o FROM Order o",
Order::class.java)
    .resultList
}
```

이 상황의 SQL은 다음처럼 나갑니다.

```
SELECT * FROM `order`;                     -- 주문 목록 1번
SELECT * FROM user WHERE id = ?;           -- 주문 20건 × 1
SELECT * FROM user WHERE id = ?;
...
```

N+1이 `EAGER`에서도 그대로 재현됩니다. `@ManyToOne(fetch = EAGER)` 디폴트가 위험한 진짜 이유가 이것입니다.

왜 JOIN으로 묶어서 가져오지 않나요?

JPA 구현체가 `EAGER` 연관을 처리하는 방식은 두 가지가 있습니다.

- `SELECT` — **개별 쿼리로 하나씩** (Hibernate의 기본 동작)
- `JOIN` — **부모 쿼리에 LEFT OUTER JOIN으로 붙여 한 번에**

JPQL로 엔티티 목록을 조회할 때는 Hibernate가 `EAGER` 연관을 **`JOIN` 으로 묶어 주지 않고**, 부모를 먼저 로드한 뒤 **개별 `SELECT` 로 따라잡는 경우가 많습니다.** 연관을 `JOIN` 으로 묶고 싶다면 `@Fetch(FetchMode.JOIN)` 같은 벤더 확장을 쓰거나, JPQL에 직접 `fetch join` 을 씁니다. 그래서 `EAGER` 는 "알아서 효율적으로 묶어주겠지"가 아니라 **실제로는 개별 쿼리가 따라 나오는 경우가 흔합니다.**

`EAGER` 가 더 나쁜 이유

`LAZY` 는 적어도 "안 쓰는 연관은 안 불러옵니다". `EAGER` 는 **쓰지 않더라도 무조건 로딩**합니다. 엔티티가 조금만 많아지면 다음이 전부 동시에 벌어집니다.

- 안 쓰는 데이터까지 메모리에 올라갑니다
- `LAZY` 에서 나던 N+1이 그대로 재현됩니다
- 나중에 쿼리를 튜닝하려 해도 `EAGER` 는 **전역 설정**이라 특정 쿼리만 다르게 동작하게 만들기 어렵습니다

그래서 실무 권장은 **모든 연관을 `LAZY` 로 시작**하고, 쿼리 단위로 필요한 연관을 명시적으로 로딩하는 쪽입니다.

Phase 5. `JOIN FETCH` 와 컬렉션 — 카르테시안 폭발

단일 연관 `JOIN FETCH`는 안전합니다

`@ManyToOne`을 `JOIN FETCH`하면 부모와 단일 연관을 한 번의 `JOIN` 쿼리로 가져옵니다.

```
SELECT o, u FROM Order o JOIN FETCH o.user u
```

이 쿼리는 주문이 20건이면 결과도 20행입니다.

컬렉션을 `JOIN FETCH`하면 행이 곱해집니다

`@OneToMany`나 `@ManyToMany`를 `JOIN FETCH`하면 상황이 다릅니다. DB가 돌려주는 결과 집합은 **부모 × 자식 조합**만큼 불어납니다.

```
SELECT o, i FROM Order o JOIN FETCH o.items i
```

주문 20건 × 각 주문의 아이템 10개 = **200행**이 돌아옵니다. 주문 엔티티는 개념적으로 20개인데, 영속성 컨텍스트는 이 200행을 받아서 **중복 제거와 컬렉션 조립**을 합니다. 건수가 적으면 문제가 안 되지만, 페이지당 수백 건 이상이 되면 결과 집합이 쉽게 수만 행이 됩니다.

여러 컬렉션을 동시에 `JOIN FETCH`하면?

Hibernate가 `MultipleBagFetchException`을 던집니다. 이것은 JPA 명세 차원의 금지가 아니라 **Hibernate 구현 제약**입니다. 두 개 이상의 **bag(순서 없는 `List`) 컬렉션을 동시에 fetch**하면 결과 행이 `부모 × 자식A`

`× 자식B` 로 폭증하면서 중복을 안정적으로 제거할 방법이 없기 때문에 Hibernate가 미리 막습니다. 컬렉션 타입을 `Set` 으로 바꾸면 예외 자체는 피할 수 있지만, **카르테시안 곱 문제는 그대로 남습니다**.

해결책은 나눠서 로딩하는 것입니다.

- `fetch join` 은 연관 하나만 씁니다
- 나머지 연관은 **`@BatchSize`** 로 묶어 로딩하거나, 별도 쿼리로 분리합니다

이 도구들의 비교는 다음 글에서 상세히 다룹니다.

페이징과 컬렉션 `JOIN FETCH` 의 함정

컬렉션을 `JOIN FETCH` 한 쿼리에 `setFirstResult()` / `setMaxResults()` 로 페이징을 걸면, Hibernate는 **DB 수준 페이징을 포기하고 전체 결과를 메모리로 가져와** 잘라 냅니다. 로그에는 다음 경고가 찍힙니다.

```
HHH90003004: firstResult/maxResults specified with
collection fetch;
             applying in memory
```

주문 10만 건을 `items` 와 함께 읽고 20건만 페이징하면, DB에서는 **10만 × N개** 행이 다 돌아옵니다. 컬렉션 페이징이 필요하면 다음 중 하나를 선택합니다.

- **ID만 페이징**하고, 그 ID들로 실제 엔티티를 한 번 더 조회합니다 (`@BatchSize` 와 조합하기 좋습니다)

- 컬렉션을 분리해 **별도의 `IN` 쿼리**로 가져옵니다
- 목록 API 레벨에서는 **컬렉션을 DTO로 따로 집계**해 내려줍니다

Phase 6. 실무 원칙 요약

원칙	이유
모든 연관을 `LAZY` 로 시작	안 쓰는 연관이 목록 조회마다 개별 쿼리를 만드는 것을 방지
쿼리별로 필요한 연관을 명시	공통 설정으로 한 번에 "좋은 답"을 내는 것은 불가능
단일 연관은 `JOIN FETCH`	`@ManyToOne` / `@OneToOne` 은 행이 안 늘어나 안전
컬렉션은 `JOIN FETCH` 를 아껴서	행이 곱해지고, 두 개 이상은 아예 불가
페이징 + 컬렉션 `JOIN FETCH` 는 피함	DB 페이징을 잃고 전체 결과를 메모리에 적재
필드 기반 `open-in-view=false` 유지	`LazyInitializationException` 이 서비스 레이어에서 드러나게 만들기

정리

`LAZY` 와 `EAGER` 는 **"어떤 쿼리가 언제 나갈지"** 를 결정합니다. `LAZY` 는 "접근할 때 그 프록시만 초기화"하기 때문에 구조상 루프 안에서 N+1을 만듭니다. `EAGER` 는 그 N+1을 **부모 로딩 시점으로 앞당길 뿐** 없애지

않습니다. 안전한 답은 전략 자체를 바꾸는 것이 아니라, **쿼리 단위로 필요한 연관을 명시적으로 로딩**하는 쪽입니다.

다음 글은 그 도구들 — `fetch join`, `@EntityGraph`, `@BatchSize` — 을 비교하면서 **언제 어떤 것을 고르면 되는지** 를 정리합니다. 세 가지 모두 N+1을 줄이지만, 각각이 **해결하는 축이 달라서** 상황별 선택이 필요합니다.

Chapter 4

N+1 해결 도구 완전 정복 — `fetch join` / `@EntityGraph` / `@BatchSize`는 언제 쓰나요?

#N+1 #Fetch 전략

JPA에서 N+1을 줄이는 세 가지 도구인 fetch join, @EntityGraph, @BatchSize는 해결하는 축이 각각 다릅니다. 단일 연관과 컬렉션, 페이징 여부, 재사용성 관점에서 상황별 선택 기준을 정리합니다.

세 도구, 왜 비교해서 알아야 하나요?

N+1을 줄일 수 있는 도구는 JPA에 여러 개 있는데, 많은 문서가 "`fetch join`을 쓰자"로 끝납니다. 실제로는 상황에 따라 `fetch join`으로 해결이 안 되거나 오히려 더 나쁜 선택이 됩니다.

- `fetch join`으로 묶고 싶은 연관이 **두 개 이상의 컬렉션**입니다
- 한 레포지토리 메서드를 **두 가지 컨텍스트**에서 쓰는데 한쪽만 연관이 필요합니다
- **페이징**이 필요한 목록 조회에서 컬렉션까지 로딩해야 합니다
- 이미 여러 경로에서 같은 엔티티를 조회하는데, 조회마다 `fetch join` JPQL을 새로 짜기 번거롭습니다

이 상황마다 적합한 도구가 다릅니다. `fetch join`, `@EntityGraph`, `@BatchSize` 는 **N+1을 줄인다는 목표는 같지만, 해결하는 축이 서로 다릅니다.** 이 글은 세 도구의 동작을 각각 풀고, 마지막에 **상황별 선택 기준**을 표로 정리합니다.

> **기준:** 이 글은 **Jakarta Persistence 3.1 (JPA 3.1)** 명세와 **Hibernate 6.x** 구현을 기준으로 작성합니다. `fetch join`은 Jakarta Persistence 3.1 — §4.4.5.3 Fetch Joins, `@EntityGraph`는 같은 명세의 `§3.7 Entity Graphs`, `@BatchSize`는 Hibernate 6 User Guide — §5.1.6 Batch fetching을 참조합니다. 이 글은 JPA Fetch 전략 글과 영속성 컨텍스트 글을 전제로 하고, 기본 N+1 개념은 N+1 글에서 이미 다뤘습니다. 코드 예시는 Kotlin + Spring Data JPA입니다.

먼저 가장 짧은 답부터 보면

세 도구는 **"쿼리 수를 어떻게 줄이는가"** 가 다릅니다.

- `fetch join` — **한 번의 `JOIN` 쿼리로** 연관을 함께 조회합니다
- `@EntityGraph` — **같은 아이디어**지만 JPQL을 바꾸지 않고 메서드에 선언합니다
- `@BatchSize` — 쿼리를 **0으로 줄이는 게 아니라 `N` 개를 `IN` 쿼리 하나로 묶습니다**

관점	`fetch join`	`@EntityGraph`	`@BatchSize`
표준	JPQL 문법	JPA 표준	Hibernate 확장
접근 방식	쿼리 재작성	메서드 애너테이션	로딩 시점 개선
쿼리 모양	`LEFT JOIN`	`LEFT JOIN`	원래 쿼리 + `IN` 쿼리 추가
컬렉션 2개	불가	불가	가능
페이징 + 컬렉션	위험 (메모리 페이징)	위험 (같은 이유)	안전
재사용성	쿼리마다 새로 작성	메서드 단위 재사용	엔티티 전역 설정

즉, "무조건 `fetch join`"이 아니라 **상황에 맞게 조합**하는 것이 정답입니다.

Phase 1. `fetch join` — 한 번의 `JOIN`으로 끌어오기

핵심: JPQL에서 연관을 `JOIN FETCH`로 선언합니다

```
@Query("""
  SELECT o FROM Order o
  JOIN FETCH o.user
  WHERE o.status = :status
""")
fun findPaidOrdersWithUser(status: String): List<Order>
```

이 쿼리는 다음 하나의 SQL을 만듭니다.

```
SELECT o.*, u.*
FROM `order` o
LEFT JOIN user u ON o.user_id = u.id
WHERE o.status = ?;
```

부모와 연관이 한 번에 로딩되므로 루프 안에서 `order.user.name` 을 접근해도 추가 쿼리가 나가지 않습니다.

장점

- **표준 JPQL 문법**이라 어떤 JPA 구현체에서도 동작합니다
- **해당 쿼리에만** 적용되기 때문에, 같은 엔티티를 다른 쿼리에서는 `LAZY` 그대로 둘 수 있습니다

한계

- **JPQL을 직접 써야** 합니다. Spring Data 파생 쿼리(`findByStatus`)에는 적용할 수 없습니다
- **컬렉션 `JOIN FETCH`는 페이징과 궁합이 나쁩니다** (앞 글에서 다룬 카르테시안 곱과 메모리 페이징 경고)
- **두 개 이상의 컬렉션을 `JOIN FETCH`** 하면 `MultipleBagFetchException`이 납니다

`JOIN FETCH` + `DISTINCT`

컬렉션을 `JOIN FETCH` 하면 부모가 자식 수만큼 중복됩니다. JPA에서는 `SELECT DISTINCT`를 붙이는 관습이 있습니다.

```
@Query("""
  SELECT DISTINCT o FROM Order o
  JOIN FETCH o.items
  WHERE o.status = :status
""")
fun findPaidOrdersWithItems(status: String): List<Order>
```

Hibernate 6부터는 **엔티티 참조 중복 제거가 Java 측에서 자동**으로 처리됩니다. 그래서 JPQL의 `DISTINCT`는 이제 **SQL에 그대로 전달되어 DB 레벨 `DISTINCT`를 강제**하는 역할만 남았습니다 (이전 버전의 `hibernate.query.passDistinctThrough=false` 같은 힌트는 제거됐습니다). 불필요하게 `DISTINCT`를 붙이면 DB의 `DISTINCT` 비용만 더해지므로,

Hibernate 6에서는 **`fetch join`에 `DISTINCT`를 붙이지 않는 편이 오히려 깔끔**합니다.

Phase 2. `@EntityGraph` — JPQL 없이 쿼리 단위로 연관 지정

핵심: 어떤 연관을 함께 로딩할지 메서드에 선언합니다

`@EntityGraph`는 JPA 2.1부터 표준입니다. Spring Data JPA에서는 리포지토리 메서드에 애너테이션으로 붙일 수 있습니다.

```
interface OrderRepository : JpaRepository<Order, Long> {

  @EntityGraph(attributePaths = ["user",
"shippingAddress"])
  fun findByStatus(status: String): List<Order>
}
```

실행 시 Hibernate는 `user`와 `shippingAddress`를 **`LEFT JOIN`으로 묶은 쿼리**를 만듭니다. 동작 결과는 `fetch join`과 거의 같습니다.

`fetch join`과 무엇이 다른가요?

기능상 겹치지만, **쿼리를 직접 쓰지 않고 연관만 선언**한다는 점이 중요합니다.

- Spring Data의 **파생 쿼리**(`findByStatus`, `findByUserId...`)에 그대로 붙일 수 있습니다

- **같은 메서드를 재사용**하면서 연관만 바꾸고 싶을 때 편리합니다
- 페치 그래프를 **재사용 가능한 이름**(`@NamedEntityGraph`)으로 정의해 둘 수도 있습니다

한계

- 내부적으로 `LEFT JOIN` 을 쓰기 때문에 **컬렉션을 두 개 이상** 넣으면 `fetch join` 과 같은 카르테시안 곱 문제가 생깁니다
- **페이징 + 컬렉션** 조합도 `fetch join` 과 똑같이 메모리 페이징 경고가 뜹니다
- 연관마다 로딩 전략을 세밀하게 바꾸려면 **`EntityGraphType.FETCH`** / **`LOAD`** 의 차이를 이해해야 합니다 (대부분 기본값으로 충분합니다)

> **참고:** `@EntityGraph(type = EntityGraphType.FETCH)` 는 그래프에 포함된 연관만 `EAGER` 로 당기고, 나머지는 `LAZY` 로 둡니다. `LOAD` 타입은 그래프에 없는 연관을 **엔티티에 선언된 기본 Fetch 전략**대로 둡니다. 기본값은 `FETCH` 입니다.

Phase 3. `@BatchSize` — `IN` 쿼리로 묶어서 줄이기

핵심: 쿼리 수를 0으로 만들지는 않지만, N을 상수로 줄입니다

`@BatchSize`는 Hibernate 확장입니다. 프록시 또는 컬렉션이 초기화될 때 **한 번에 여러 부모의 연관을 묶어서 읽습니다.**

```
@Entity
class Order(
  @Id val id: Long,

  @OneToMany(mappedBy = "order", fetch = FetchType.LAZY)
  @BatchSize(size = 100)
  val items: List<OrderItem>,
)
```

이 엔티티에서 주문 50건을 읽고 각각 `items`를 접근하면, SQL은 다음처럼 나갑니다.

```
SELECT * FROM `order` WHERE status = ?;                  -- 1번
SELECT * FROM order_item WHERE order_id IN (?, ?, ..., ?);
-- 1번 (50개 IN)
```

쿼리는 **1 + 1 = 2번** 으로 끝납니다. 페이지 크기가 100을 넘으면 `IN` 묶음이 2번, 3번으로 늘어나지만 **여전히 상수입니다**.

장점

- **카르테시안 곱이 없습니다** — 각 테이블을 개별 쿼리로 읽기 때문에 행이 곱해지지 않습니다
- **두 개 이상의 컬렉션을 같이 로딩**해도 문제가 없습니다. 각각 별개의 `IN` 쿼리로 나갑니다
- **페이징과 궁합이 좋습니다** — 부모 페이징은 DB에서 그대로 일어나고, 자식은 `IN`으로 끌어옵니다

한계

- **표준이 아닙니다** (Hibernate 확장). JPA 명세를 100% 지키는 코드베이스면 쓸 수 없습니다
- 쿼리가 **완전히 한 번**은 아닙니다. 부모 쿼리 + 연관 쿼리가 합쳐 최소 2번은 나갑니다
- **엔티티 레벨 애너테이션**이기 때문에, 이 엔티티를 읽는 **모든 쿼리에 영향**을 줍니다. 특정 쿼리에서만 끄기 어렵습니다

전역 설정 — `default_batch_fetch_size`

`@BatchSize`를 연관마다 붙이지 않고 한 번에 적용하려면 `application.yml`에서 전역 설정을 켭니다.

```
spring:
  jpa:
    properties:
      hibernate:
        default_batch_fetch_size: 100
```

프로젝트 대부분에서는 이 설정만 켜두고, 특별히 더 큰 값이 필요한 연관에만 `@BatchSize`를 덧붙이는 패턴을 많이 씁니다.

`IN` 절 크기에 대한 주의

DB마다 `IN` 절에 들어갈 수 있는 최대 파라미터 수에 제한이 있습니다. MySQL은 명시적 상한이 없지만 파서 메모리 한계에 의존합니다.

PostgreSQL은 과거 `$32767` 파라미터 제한이 있었지만 최신은 완화됐습니다. Oracle은 `1000` 제한이 있습니다. 대부분의 실무 `@BatchSize` 값은 **100~500 사이**에서 잡습니다. 너무 크게 잡으면 쿼리 캐시 적중률이 떨어지고, 너무 작게 잡으면 쿼리 수가 늘어납니다.

Phase 4. 실전 선택 기준

세 도구의 선택은 **세 가지 축**만 보면 빠르게 결정됩니다.

축 1 — 연관이 단일이냐 컬렉션이냐

- **단일 연관 (`@ManyToOne` / `@OneToOne`)**: `fetch join` 또는 `@EntityGraph`. 행이 안 늘어나므로 한 번의 `JOIN`으로 끝납니다
- **컬렉션**: 건수가 적으면 `fetch join` 가능. 많거나 두 개 이상이면 **`@BatchSize`**

축 2 — 페이징이 필요한가

- 컬렉션이 포함되고 페이징도 필요하면 `fetch join`/`@EntityGraph`는 **메모리 페이징 경고**가 뜹니다
- 이 조합에서는 거의 항상 **`@BatchSize`** 가 정답입니다

축 3 — 한 번만 쓰는 쿼리냐, 여러 곳에서 재사용되는 연관이냐

- **한 번만 쓰는 목록 쿼리**: JPQL + `fetch join`이 읽기 쉬움
- **여러 API에서 같은 연관을 자주 로딩**: `@EntityGraph`로 메서드에 선언하거나 `@NamedEntityGraph`로 이름 붙여 재사용

- **엔티티를 읽는 대부분의 경로에서 필요:** `@BatchSize` 또는 `default_batch_fetch_size` 를 설정해 전역으로 커버

상황별 추천

상황	추천	이유
단일 연관만 함께 조회	`JOIN FETCH` 또는 `@EntityGraph`	단일 `JOIN` 으로 끝남
컬렉션 하나, 작은 건수, 페이징 없음	`JOIN FETCH`	Hibernate 6은 Java 측에서 중복 제거 자동 처리
컬렉션 하나, 페이징 있음	`@BatchSize`	메모리 페이징 회피
컬렉션 둘 이상	`@BatchSize`	`fetch join` 은 불가능
파생 쿼리(`findByXxx`)에 연관 추가	`@EntityGraph`	JPQL을 쓰지 않고 선언만
여러 API에서 반복되는 페치	`@NamedEntityGraph`	이름 붙여 재사용
전역 기본값으로 덜어내기	`default_batch_fetch_size`	프로젝트 전반의 N+1을 저렴하게 완화

조합해서 쓰기

현실에서는 **하나의 조회**가 세 도구를 조합하기도 합니다. 예를 들어 주문 목록에 `user` 는 `fetch join`으로 묶고, `items` 컬렉션과 `coupons` 컬렉션은 `@BatchSize`로 가져오는 식입니다.

```
@Entity
class Order(
  @Id val id: Long,

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "user_id")
  val user: User,

  @OneToMany(mappedBy = "order", fetch = FetchType.LAZY)
  @BatchSize(size = 200)
  val items: List<OrderItem>,

  @OneToMany(mappedBy = "order", fetch = FetchType.LAZY)
  @BatchSize(size = 200)
  val coupons: List<Coupon>,
)

@Query("""
  SELECT o FROM Order o
  JOIN FETCH o.user
  WHERE o.status = :status
""")
fun findPaidOrdersWithUser(status: String, pageable:
Pageable): Page<Order>
```

위 조합은 다음처럼 동작합니다.

- `user`는 `JOIN FETCH`로 한 쿼리 안에 포함
- 페이징은 `JOIN FETCH`가 단일 연관만 포함하기 때문에 DB에서 안전하게 수행

- `items`와 `coupons`는 접근 시 **`IN` 쿼리 두 번**으로 묶여서 끌려옴

즉, `fetch join`은 **안전한 단일 연관에만** 사용하고, **컬렉션은 `@BatchSize`에 위임**하는 패턴이 가장 자주 쓰이는 실전 형태입니다.

Phase 5. 함께 주의해야 할 지점

1. `fetch join`과 `GROUP BY`/집계

`JOIN FETCH`는 결과를 엔티티로 돌려주기 때문에 **집계 쿼리와 잘 맞지 않습니다**. 집계가 필요하면 `JOIN FETCH` 대신 일반 `JOIN` + `GROUP BY`를 쓰고, DTO 프로젝션으로 결과를 받습니다.

2. `@EntityGraph` + `native query`

`@EntityGraph`는 JPQL/Criteria에만 적용됩니다. 네이티브 쿼리에는 동작하지 않으므로, 네이티브 쿼리가 필요하면 `JOIN`으로 직접 써서 DTO에 매핑합니다.

3. `@BatchSize`의 로그 확인

`@BatchSize`가 실제로 `IN` 쿼리로 묶이는지 확인하려면 `hibernate.show_sql` 또는 `p6spy` 같은 SQL 로거로 쿼리를 봐야 합니다. "N+1이 해결된 줄 알았는데 여전히 수십 번 쿼리가 나가는" 원인의 상당수가 **`@BatchSize`가 예상한 연관에 안 붙어 있거나, 해당 트랜잭션이 닫혀 다른 트랜잭션에서 로딩되는 경우**입니다.

4. DTO 프로젝션이 더 깔끔한 순간

세 도구를 조합해도 엔티티 전체를 끌고 올 필요가 없는 목록 API는 많습니다. 이럴 때는 **DTO 프로젝션**이 더 단순한 답입니다.

```
@Query("""
  SELECT new com.example.OrderView(o.id, u.name,
o.totalPrice)
  FROM Order o JOIN o.user u
  WHERE o.status = :status
""")
fun listOrderViews(status: String): List<OrderView>
```

이 쿼리는 영속성 컨텍스트도, 프록시도, N+1도 신경 쓸 필요가 없습니다. **목록 API에서 엔티티 자체가 필요 없다면 DTO가 거의 항상 더 단순합니다**.

정리

N+1을 줄이는 도구는 하나가 아닙니다. 세 도구의 역할은 명확히 다릅니다.

- `fetch join` — JPQL로 쿼리 모양을 다시 씁니다. 단일 연관에 안전합니다
- `@EntityGraph` — 같은 효과를 파생 쿼리에도 붙일 수 있게 합니다
- `@BatchSize` — 쿼리 수를 0으로 만들지 않지만, 컬렉션과 페이징이 있을 때 가장 안전한 선택입니다

상황을 가르는 기준은 **"연관이 단일인가 컬렉션인가"**, **"페이징이 필요한가"**, **"이 쿼리만 쓸 것인가 아니면 여러 곳에서 쓸 것인가"** 세 가지입니다. 현실에서는 조합해 쓰는 경우가 많으며, 실전 패턴은 **단일 연관은 `fetch join`, 컬렉션은 `@BatchSize`** 입니다.

다음 글에서는 JPA 영역을 잠시 벗어나 **Spring `@Transactional`의 전파 속성과 롤백 규칙**을 다룹니다. Fetch 전략까지 다져놓았다면, 그 다음은 트랜잭션 경계를 설계하는 방식이 제일 큰 설계 결정입니다.

P A R T 3

3부. Spring 트랜잭션과 AOP

Chapter 5

Spring `@Transactional` 완전 정복 — 전파 속성과 롤백 규칙은 어떻게 동작하나요?

#트랜잭션

Spring의 @Transactional이 제공하는 7가지 전파 속성, 기본 롤백 규칙, readOnly와 timeout 힌트, 그리고 실무에서 자주 오해되는 REQUIRES_NEW와 NESTED의 동작을 Spring 레퍼런스 기준으로 정리합니다.

`@Transactional`, 왜 전파 속성까지 알아야 하나요?

대부분의 코드는 `@Transactional`을 기본 설정 그대로 씁니다. 그러다 보니 다음 상황에서 결과가 의아해집니다.

- `@Transactional`이 걸린 메서드가 다른 `@Transactional` 메서드를 호출했는데 **트랜잭션이 하나로 합쳐졌습니다**
- 안쪽 트랜잭션에서 `RuntimeException`이 터졌는데 **바깥 트랜잭션까지 통째로 롤백**됐습니다
- `try-catch`로 예외를 잡았는데도 **커밋 시점에** **`UnexpectedRollbackException`** 이 나옵니다
- `REQUIRES_NEW`로 바꿨더니 **커넥션 풀이 고갈**되기 시작했습니다

이 모든 동작의 중심에 **전파 속성(`Propagation`)**이 있습니다. 전파 속성은 "호출 시점에 이미 실행 중인 트랜잭션이 있을 때 어떻게 동작할지"를 정의하는 규칙입니다. 기본값인 `REQUIRED` 하나로도 대부분의 코드가 돌아가지만, 조금만 복잡한 비즈니스 로직이 되면 **전파와 롤백 규칙을 모르고는 구조적 버그를 피하기 어렵습니다.**

> **기준:** 이 글은 **Spring Framework 6 / Spring Boot 3.x** 기준으로 작성합니다. 전파 속성과 롤백 규칙 정의는 Spring Framework Reference — Transaction Management와 `org.springframework.transaction.annotation.Propagation` JavaDoc을 참조합니다. 격리 수준은 트랜잭션 격리 수준 글, ACID는 ACID 글, 영속성 컨텍스트와의 관계는 영속성 컨텍스트 글에서 이미 다뤘습니다. 코드 예시는 Kotlin + Spring Boot입니다. AOP 프록시 동작과 `self-invocation` 문제는 다음 글에서 따로 다룹니다.

먼저 가장 짧은 답부터 보면

- 전파 속성은 **"이미 트랜잭션이 있을 때"의 동작**을 정합니다. 없으면 대부분 새로 시작합니다
- 기본값 `REQUIRED` — 있으면 참여, 없으면 새로 시작
- `REQUIRES_NEW` — 항상 새 물리 트랜잭션. 바깥 트랜잭션은 일시 정지
- `NESTED` — 같은 트랜잭션 안의 **세이브포인트**. 부분 롤백만
- 기본 롤백은 `RuntimeException`과 `Error`에서만 일어납니다. **`Checked Exception`은 기본적으로 롤백하지 않습니다**

- `readOnly = true`는 **힌트**이지 강제가 아닙니다. Hibernate는 이걸 flush 모드 최적화에 사용합니다

Phase 1. 전파 속성이 결정하는 것

핵심: "이미 트랜잭션이 진행 중인가?"에 따라 분기합니다

`@Transactional`이 붙은 메서드가 호출되면, Spring의 트랜잭션 인터셉터는 **먼저 현재 스레드에 이미 열려 있는 트랜잭션이 있는지** 확인합니다. 이 판단 후의 동작을 결정하는 것이 **전파 속성**입니다.

```
호출 시점
   │
   ▼
현재 트랜잭션이 있는가?
   ├─ 없음 → Propagation 설정에 따라 "새로 시작" or "에러" or "비트
랜잭션 실행"
   └─ 있음 → Propagation 설정에 따라 "참여" or "새 물리 트랜잭션" or
"세이브포인트" or ...
```

Spring의 `Propagation` enum은 **7가지 값**을 제공합니다.

Phase 2. 7가지 전파 속성 한 번에 보기

속성	현재 트랜잭션이 있을 때	현재 트랜잭션이 없을 때
`REQUIRED` (기본값)	참여	새로 시작
`REQUIRES_NEW`	바깥을 **일시 정지**하고 새로 시작	새로 시작
`SUPPORTS`	참여	비트랜잭션으로 실행
`NOT_SUPPORTED`	바깥을 **일시 정지**하고 비트랜잭션으로 실행	비트랜잭션으로 실행
`MANDATORY`	참여	예외 (`IllegalTransactionStateException`)
`NEVER`	예외	비트랜잭션으로 실행
`NESTED`	**세이브포인트** 생성 후 실행	새로 시작 (`REQUIRED` 처럼)

이 중 실무에서 자주 보는 것은 **`REQUIRED`**, **`REQUIRES_NEW`**, **`NESTED`** 세 가지입니다. 나머지는 개념 확인용으로 한 번 훑어두면 됩니다.

Phase 3. `REQUIRED` — 논리 트랜잭션과 `UnexpectedRollbackException`

동작

`REQUIRED`는 "트랜잭션이 있으면 참여하고, 없으면 새로 시작한다"입니다. **가장 흔한 기본값**이며, 대부분의 서비스 메서드가 이 설정으로 동작합니다.

```
@Service
class OrderService(
  private val paymentService: PaymentService
) {
  @Transactional  // REQUIRED
  fun placeOrder() {
    paymentService.charge()  // 같은 트랜잭션에 참여
  }
}

@Service
class PaymentService {
  @Transactional  // REQUIRED
  fun charge() {
    // 바깥 트랜잭션에 그대로 참여
  }
}
```

물리 트랜잭션과 논리 트랜잭션

`REQUIRED`로 참여한 내부 메서드는 **자기만의 트랜잭션을 갖는 것이 아닙니다**. Spring은 이런 상황을 **물리 트랜잭션(physical transaction) = 1개, 논리 트랜잭션(logical transaction) = 2개**로 설명합니다.

- **물리 트랜잭션**: 실제 DB 커넥션 수준의 트랜잭션. 커밋/롤백의 단위

- **논리 트랜잭션**: Spring이 참여 여부를 추적하기 위한 논리적 경계

여기서 생기는 중요한 규칙이 하나 있습니다. **어떤 논리 트랜잭션이 한 번이라도 "롤백 전용(rollback-only)"으로 마킹되면, 전체 물리 트랜잭션이 롤백됩니다.**

왜 `try-catch`로 잡았는데도 롤백되나요?

바깥 서비스에서 내부 서비스의 예외를 잡으면 로직은 이어지지만, **내부 메서드가 빠져나오는 순간 Spring은 현재 트랜잭션에 "rollback-only" 플래그를 찍어둡니다.** 그래서 바깥이 아무 문제 없이 끝나도, 커밋 시점에 `UnexpectedRollbackException`이 발생합니다.

```
@Transactional
fun placeOrder() {
  try {
    paymentService.charge()  // 내부에서 예외, rollback-only 마킹
  } catch (e: Exception) {
    // 예외는 잡았지만 이미 rollback-only 상태
  }
  // 커밋 시점: UnexpectedRollbackException
}
```

이 동작을 피하려면 내부 메서드를 **`REQUIRES_NEW`로 분리**해 독립된 물리 트랜잭션으로 만들거나, 예외가 나지 않게 구조를 바꿉니다.

Phase 4. `REQUIRES_NEW` — 독립된 물리 트랜잭션

동작

`REQUIRES_NEW`는 **항상 새로운 물리 트랜잭션**을 시작합니다. 바깥에 트랜잭션이 있으면 **일시 정지(`suspend`)** 시킨 다음, 내부 메서드가 끝나면 다시 이어갑니다.

```
@Service
class OrderService(
  private val auditService: AuditService
) {
  @Transactional
  fun placeOrder() {
    // 주문 처리 ...
    try {
      auditService.logOrderEvent()
    } catch (e: Exception) {
      // 감사 로그 실패는 주문 실패로 이어지면 안 됨
    }
    // 주문 트랜잭션은 그대로 이어짐
  }
}

@Service
class AuditService {
  @Transactional(propagation = REQUIRES_NEW)
  fun logOrderEvent() { ... }
}
```

위 구조는 **바깥(주문)과 안쪽(감사 로그)이 독립된 커밋 단위**로 동작합니다. 감사 로그가 실패해도 주문 커밋에 영향이 없습니다.

함정 1 — 커넥션 두 개를 동시에 씁니다

바깥 트랜잭션을 **일시 정지**한다고 해서 커넥션을 반납하는 것은 아닙니다. Spring은 바깥의 커넥션을 **그대로 물고 있는 상태**에서 내부 메서드에 **새 커넥션**을 추가로 할당합니다. 즉 **`REQUIRES_NEW` 호출 한 번당 커넥션 두 개**를 동시에 사용합니다.

이 점이 커넥션 풀 글에서 다룬 커넥션 고갈과 연결됩니다. 목록 API 안에서 루프를 돌며 `REQUIRES_NEW`를 부르면, 동시 요청 수에 비례해 커넥션 수요가 **2배 이상**으로 불어납니다.

함정 2 — 바깥에서 공유했던 1차 캐시가 사라집니다

영속성 컨텍스트는 물리 트랜잭션과 수명이 같습니다. `REQUIRES_NEW`로 새 트랜잭션이 열리면 **그 안에서는 전혀 다른 영속성 컨텍스트**가 동작합니다. 바깥에서 로딩했던 엔티티는 내부 메서드에서 보면 **서로 다른 인스턴스**이거나 아예 안 보입니다. 영속성 컨텍스트의 동일성 보장이 `REQUIRES_NEW` 경계에서 끊어진다는 것을 염두에 둬야 합니다.

언제 쓰기 좋은가요?

- **감사 로그, 통계 기록처럼 실패해도 본 업무가 계속돼야 하는 경우**
- **외부 시스템 호출이 끼어 있어 바깥과 커밋 시점을 분리하고 싶을 때**
- **일부 작업이 성공해야 하는 batch 처리에서 실패 건만 스킵하고 싶을 때**

Phase 5. `NESTED` — 세이브포인트로 부분 롤백

동작

`NESTED` 는 **같은 물리 트랜잭션 안에서 세이브포인트(`SAVEPOINT`)** 를 만들고 실행합니다. 내부에서 예외가 나면 그 세이브포인트까지만 롤백되고, 바깥 트랜잭션은 계속 진행합니다.

```
BEGIN
  ├─ SAVEPOINT sp1
  │     └─ inner method 작업
  │          예외 시: ROLLBACK TO SAVEPOINT sp1
  ├─ 바깥 메서드 작업 계속
COMMIT
```

`REQUIRES_NEW` 와 무엇이 다른가요?

관점	`REQUIRES_NEW`	`NESTED`
물리 트랜잭션	새로 생성	**같은 트랜잭션**
커넥션	두 개 동시 사용	하나 공유
내부 커밋/롤백	독립	**세이브포인트만** 롤백
바깥이 롤백되면	내부는 영향 없음	내부도 같이 롤백

전제 조건

`NESTED`는 **JDBC 드라이버가 세이브포인트를 지원**해야 동작합니다. 대부분의 관계형 DB는 지원합니다.

`JpaTransactionManager`도 `NESTED` 자체는 지원하지만, 기본값이 **`nestedTransactionAllowed=false`**로 꺼져 있습니다. 켜더라도 **세이브포인트는 JDBC 커넥션 수준에서만 동작**하기 때문에, 롤백 시점에 **`EntityManager`의 1차 캐시 상태는 되돌아가지 않습니다**. 영속성 컨텍스트 관점에서는 부분 롤백이 깔끔하게 이루어지지 않으므로, JPA 환경에서는 `NESTED`보다 **`REQUIRES_NEW`로 트랜잭션을 분리**하는 편이 안전합니다. 실무에서 `NESTED`는 **JDBC 기반 반복 처리 중 일부만 스킵**하는 batch job에서 가끔 쓰입니다.

Phase 6. 롤백 규칙 — 기본은 `RuntimeException`만

기본 동작

Spring은 `@Transactional`의 기본 롤백 규칙을 다음처럼 정합니다.

- **`RuntimeException`과 `Error`** — 롤백
- **`Checked Exception` (Java의 `IOException` 등) — 롤백하지 않음**

이 규칙은 EJB 시대부터 이어진 관례로, Spring도 그대로 유지합니다.

왜 Checked Exception은 커밋되나요?

Spring 레퍼런스의 설명을 요약하면, Checked Exception은 **비즈니스 로직이 예측한 예외**로 간주하기 때문입니다. "환불 가능 금액 초과"처럼 의도된 예외까지 롤백하지 않도록 기본값을 잡은 것입니다. 실무에서는 이 기본값이 낯설게 느껴질 때가 많아 주의가 필요합니다.

`rollbackFor` / `noRollbackFor`로 규칙 확장

기본 규칙을 바꾸려면 애너테이션에 명시합니다.

```
@Transactional(rollbackFor = [Exception::class])
fun transfer() { ... }

@Transactional(noRollbackFor =
[NotificationException::class])
fun placeOrder() { ... }
```

Kotlin에서는 **예외를 명시적으로 `throw` 해도 컴파일러가 Checked/Unchecked를 구분하지 않습니다**. 그래서 Kotlin 코드가 Java 라이브러리의 Checked Exception을 재던지면 런타임에 롤백이 안 되는 경우가 생깁니다. Kotlin 프로젝트에서 **`rollbackFor = Exception::class` 를 기본값처럼 쓰는 관습**이 있는 이유가 이 때문입니다.

예외를 잡고 삼키면?

예외가 `@Transactional` 메서드 **밖으로 빠져나오지 않으면** Spring은 예외 자체를 인식하지 못합니다. 이 경우 롤백되지 않고 그대로 커밋됩니다.

그러나 Phase 3에서 본 것처럼 **내부 `REQUIRED` 메서드가 rollback-only를 이미 찍어뒀다면**, 바깥에서 예외를 삼켜도 커밋 시점에 `UnexpectedRollbackException`이 납니다.

Phase 7. `readOnly`와 `timeout` — 힌트와 실제 동작

`readOnly = true`

`@Transactional(readOnly = true)`는 "이 트랜잭션은 데이터를 바꾸지 않습니다"라는 **힌트**입니다. Spring이 강제로 막지는 않지만 여러 계층에서 최적화에 사용됩니다.

- **Hibernate**: `FlushMode`를 `MANUAL`로 바꿔 flush를 건너뜁니다. 변경 감지를 생략해 성능에 유리합니다
- **JDBC 드라이버**: 커넥션에 `setReadOnly(true)` 를 호출합니다. DB별로 내부 최적화가 다릅니다
- **DB 라우팅**: `AbstractRoutingDataSource`로 읽기 전용 트랜잭션을 **리드 레플리카로 분기**하는 패턴의 훅으로 자주 쓰입니다

`timeout`

```
@Transactional(timeout = 5)  // 5초
```

트랜잭션이 이 시간을 넘으면 `TransactionTimedOutException`이 발생하며 롤백됩니다. 이 값은 **트랜잭션 매니저의 모니터링 쪽에서 검사**되는 것이라, **DB의 `lock wait timeout`** 과는 다른 레이어입니다. 둘 다

타임아웃이 될 수 있으니, 장시간 대기가 의심되면 두 설정을 함께 봐야 합니다.

`isolation`

Spring의 `@Transactional(isolation = ...)`은 격리 수준을 트랜잭션 시작 시점에 설정합니다. DB에 따라 지원 여부가 다릅니다. 격리 수준 자체의 동작은 격리 수준 글에서 자세히 다뤘습니다.

Phase 8. 실무에서 자주 만나는 함정

1. 전파 속성은 프록시를 거칠 때만 동작합니다

`@Transactional`은 Spring AOP 프록시 기반이라 **같은 클래스 안에서 메서드를 직접 호출하면 프록시를 거치지 않습니다.** 즉 `REQUIRES_NEW`로 바꿨어도 같은 클래스 내 `self-invocation`이면 **그냥 `REQUIRED` 처럼 동작**합니다. 이 주제는 다음 글에서 깊게 다룹니다.

2. `REQUIRES_NEW` 를 루프 안에서 쓰기

```
@Transactional
fun processBatch(orders: List<Order>) {
  orders.forEach { o ->
    try {
      innerService.handle(o)  //
@Transactional(propagation = REQUIRES_NEW)
    } catch (e: Exception) { ... }
  }
}
```

이 구조는 주문 N건마다 **새 트랜잭션과 새 커넥션**을 쓰고, 바깥 커넥션까지 동시에 물고 있습니다. 동시 요청 수에 따라 커넥션 풀이 금방 고갈됩니다. batch 처리는 **트랜잭션 경계를 batch 단위**로 잡거나, 실패 허용 건만 별도 DB 연결로 분리하는 편이 안전합니다.

3. 외부 API 호출을 트랜잭션 안에서 하기

외부 HTTP 호출은 응답이 느리거나 멈출 수 있습니다. 그 동안 **DB 커넥션은 그대로 점유**됩니다. 외부 호출은 **가능하면 트랜잭션 밖**에서 하고, 내부 상태 변경만 트랜잭션 안에서 처리합니다. OSIV가 기본값이라면 이 문제가 컨트롤러 레이어까지 번진다는 점도 기억해야 합니다.

4. `noRollbackFor` 남용

"로그성 예외는 롤백 안 하게 해두자"라는 의도로 `noRollbackFor` 를 넓게 잡으면, **데이터 일관성 문제가 생겨도 커밋**됩니다. 롤백을 막고 싶은 예외

타입은 **명확한 비즈니스 예외로 한정**하고, 시스템 예외까지 삼키지 않도록 주의합니다.

정리

`@Transactional` 은 애너테이션 하나처럼 보이지만, **전파 속성 + 롤백 규칙 + 격리 수준 + readOnly/timeout** 이 함께 맞물려 동작합니다. 각 축의 핵심은 이렇게 줄일 수 있습니다.

축	기본값	기억할 점
전파	`REQUIRED`	참여 시 **rollback-only** 전파 문제 주의
새 트랜잭션	`REQUIRES_NEW`	커넥션 두 개, 영속성 컨텍스트 분리
세이브포인트	`NESTED`	드라이버/매니저 지원 필요, JPA 환경에서는 제약 많음
롤백 규칙	`RuntimeException/Error` 만	Kotlin은 `rollbackFor = Exception::class` 관습
읽기 힌트	`readOnly = false`	Hibernate flush 생략·리드 레플리카 라우팅 훅
타임아웃	`-1` (무제한)	DB `lock wait timeout` 과 층이 다름

중요한 것은 기본값 `REQUIRED` + `RuntimeException 롤백` 조합이 **대부분의 코드에서 그대로 맞다**는 것입니다. 조합을 일부러 바꿔야 하는

상황을 만났을 때, 각 옵션이 **어떤 구조적 비용을 지불하는지** 를 알면 기본값을 벗어나는 결정이 훨씬 명확해집니다.

다음 글은 이 전파 속성이 **AOP 프록시를 거치지 않으면 전부 무의미해지는 이유** — `self-invocation` 함정과 Spring의 프록시 생성 방식을 파헤칩니다.

Chapter 6

Spring AOP 프록시와 `self-invocation` 함정 — `@Transactional`이 왜 안 먹나요?

#AOP

Spring AOP가 프록시 기반으로 동작한다는 사실이 @Transactional, @Async, @Cacheable이 같은 클래스 내부 호출에서 무력화되는 원인입니다. JDK 동적 프록시와 CGLIB의 차이, self-invocation이 프록시를 우회하는 이유, 네 가지 해결책을 정리합니다.

Spring AOP 프록시, 왜 알아야 하나요?

`@Transactional`을 분명히 달았는데 트랜잭션이 안 걸리는 상황을 한 번쯤 만나 봤을 겁니다.

- 같은 클래스의 `public` 메서드를 불렀는데 `@Transactional`이 **무시**됐습니다
- `@Async`를 붙인 메서드를 불렀는데 **비동기로 실행되지 않습니다**
- `@Cacheable`을 붙였는데 캐시가 **적용되지 않습니다**
- `private` 메서드에 `@Transactional`을 붙였는데 **경고도 없이 무시**됐습니다

이 네 가지 증상의 원인은 전부 하나입니다. **Spring AOP가 프록시 기반이기 때문**입니다. 프록시는 호출이 "**밖에서 들어올 때만**" 가로챌 수 있는 구조라서, 같은 객체 내부에서 메서드를 직접 호출하면 프록시를 거치지 않고 AOP가 전부 우회됩니다. 이걸 **`self-invocation` 문제**라고 부르고, `@Transactional`을 비롯한 모든 Spring AOP 기반 애너테이션이 공통으로 겪는 함정입니다.

> **기준:** 이 글은 **Spring Framework 6 / Spring Boot 3.x** 기준으로 작성합니다. AOP 프록시 동작과 self-invocation 한계는 Spring Framework Reference — Understanding AOP Proxies와 Spring AOP를 참조합니다. 트랜잭션 전파 속성은 앞 글에서 다뤘고, 이 글은 그 전파 속성이 프록시를 거치지 않으면 왜 전부 무력화되는지에 초점을 맞춥니다. 코드 예시는 Kotlin + Spring Boot입니다.

먼저 가장 짧은 답부터 보면

- Spring AOP는 **프록시 객체**를 만들어 빈을 감쌉니다. 외부 호출은 프록시를 거치지만, **같은 객체 내부 호출은 프록시를 거치지 않습니다**
- 프록시가 없으면 `@Transactional`, `@Async`, `@Cacheable`, `@Retryable`, `@Validated` 등 **모든 AOP 기반 애너테이션이 동작하지 않습니다**
- **가장 단순한 해결책은 메서드를 다른 빈으로 분리**하는 것입니다. 한 빈 안에서 풀고 싶다면 `self` 주입, `ApplicationContext`, AspectJ 세 가지 선택지가 있지만 대부분 분리가 먼저입니다

- `private` / `final` 메서드, `static` 메서드에는 프록시가 적용되지 않습니다

Phase 1. Spring AOP는 실제로 무엇을 만들까요?

핵심: 빈을 직접 쓰지 않고, 빈을 감싼 "대리 객체"를 씁니다

Spring 컨테이너가 `@Transactional`이 붙은 빈을 발견하면, **그 빈을 감싸는 프록시 객체를 대신 생성**해 컨테이너에 등록합니다. 주입 시점에 다른 빈이 받는 것은 원본이 아니라 **프록시**입니다.

```
컨테이너 등록
  ├─ OrderService (원본 빈) — 내부 구현만 보유
  └─ OrderService$$SpringCGLIB$$0 (프록시) ← 다른 빈에 주입되는
것

프록시.placeOrder() 호출
  ├─ 트랜잭션 시작 (TransactionInterceptor)
  ├─ 원본.placeOrder() 호출
  └─ 예외 여부 확인 후 커밋/롤백
```

이 프록시가 하는 일은 두 가지입니다.

- 메서드 호출 전에 **AOP 어드바이스(advice)** 를 먼저 실행
- 그다음에 **원본 메서드**로 위임

`@Transactional`의 "트랜잭션 시작 → 원본 실행 → 커밋/롤백" 흐름은 이 어드바이스가 하는 일입니다. 프록시를 안 거치면 이 어드바이스가 **아예**

실행되지 않습니다.

컨테이너 관점에서 벌어지는 일

```
@Service
class OrderService {
  @Transactional
  fun placeOrder() { ... }
}

@Service
class CheckoutService(
  private val orderService: OrderService   // ← 실제로는 프록시
) {
  fun checkout() {
    orderService.placeOrder()   // 프록시 호출 → 트랜잭션 동작
  }
}
```

`CheckoutService`의 필드 타입은 `OrderService`이지만, 주입되는 인스턴스는 `OrderService$$SpringCGLIB$$0` 같은 프록시 서브클래스입니다. 그래서 `orderService.placeOrder()`는 프록시를 거쳐 트랜잭션 어드바이스를 발동시킵니다.

Phase 2. 프록시 두 종류 — `JDK Dynamic Proxy` vs `CGLIB`

JDK Dynamic Proxy

- **인터페이스 기반**입니다
- 빈이 인터페이스를 구현하고 있으면 기본적으로 `java.lang.reflect.Proxy` 로 프록시를 만듭니다
- 프록시는 **인터페이스 타입**으로만 캐스팅 가능합니다. 구체 클래스로 캐스팅하면 `ClassCastException`이 납니다

CGLIB (Spring이 repackage한 fork)

- **서브클래스 기반**입니다. 원본 클래스를 상속한 서브클래스를 런타임에 생성합니다
- 인터페이스가 없어도 프록시를 만들 수 있습니다
- 서브클래싱이라 **`final` 클래스와 `final` 메서드는 프록시를 만들 수 없습니다**
- Spring Framework 6은 **CGLIB을 직접 fork하여 `spring-core` 안에 repackage**해서 씁니다 (`org.springframework.cglib.*`)

Spring Boot의 기본값

Spring Boot 2.0부터 기본값은 **CGLIB(서브클래스 방식)** 입니다. `@EnableAspectJAutoProxy(proxyTargetClass = true)` 와 같은 의미가 기본값으로 설정되어 있습니다. 인터페이스 없이 구현 클래스만 있어도 프록시가 만들어지기 때문에 대부분의 실무 환경에서 이 설정이 편합니다.

```yaml
spring:
  aop:
    proxy-target-class: true  # 기본값
```

Kotlin에서 특히 중요한 주의점

Kotlin의 클래스는 **기본이 `final`** 입니다. 이 상태로는 CGLIB이 서브클래스를 만들 수 없어 프록시 생성이 실패합니다. Spring Boot는 이를 돕기 위해 `kotlin-spring` 플러그인을 제공합니다.

```kotlin
// build.gradle.kts
plugins {
  kotlin("plugin.spring") version "..."
}
```

이 플러그인은 `@Component`, `@Service`, `@Repository`, `@Transactional` 등이 붙은 클래스를 자동으로 `open` 으로 바꿔 줍니다. 플러그인을 빠뜨리면 트랜잭션이 "조용히" 동작하지 않는 상황을 만나게 됩니다.

Phase 3. `self-invocation`이 왜 AOP를 건너뛸까요?

핵심: 프록시는 "외부에서 들어오는 호출"만 가로챌 수 있습니다

같은 객체 안에서 다른 메서드를 직접 호출하면, 그 호출은 **프록시의 메서드 테이블을 통하지 않고 JVM 레벨에서 바로 원본 객체의 메서드 포인터로 점프**합니다. 프록시가 감쌌어도 이 점프는 감쌀 방법이 없습니다.

```
@Service
class OrderService {

  fun placeOrder() {
    updateStock()  // ← self-invocation: 프록시 거치지 않음
  }

  @Transactional
  fun updateStock() {
    // 프록시를 거치지 않았기 때문에 @Transactional이 동작하지 않음
  }
}
```

`placeOrder()` 안의 `updateStock()` 호출은 **`this.updateStock()`** 과 동일합니다. 이 `this` 는 프록시가 아니라 **원본 객체**입니다. 그래서 `@Transactional` 이 무시됩니다.

그림으로 본 호출 경로

```
외부 호출 (정상)
[외부] → [프록시] → 어드바이스 발동 → [원본.메서드]

같은 객체 내부 호출 (self-invocation)
[외부] → [프록시] → 어드바이스 발동 → [원본.placeOrder]
                                        └→ this.updateStock
← 프록시 우회
```

두 번째 그림에서 `updateStock` 에 붙은 AOP 어드바이스(`@Transactional` , `@Async` 등)는 **발동되지 않습니다**.

이 구조는 왜 바뀌지 않나요?

JDK 동적 프록시도 CGLIB도 **원본 객체의 메서드 포인터를 바꾸지는 않습니다**. 메서드 호출을 가로채는 장치는 **프록시 객체 위에서만** 존재합니다. 원본 객체에서 `this.메서드()`를 호출하는 것은 JVM이 프록시와 무관하게 처리하기 때문에, Spring이 개입할 틈이 없습니다.

이 한계를 원천적으로 없애려면 **AspectJ** 처럼 바이트코드를 직접 조작하는 방식이 필요합니다. 이건 Phase 4의 네 번째 해결책에서 다룹니다.

Phase 4. 해결책 네 가지

1. 메서드를 다른 빈으로 분리 (거의 항상 1순위)

가장 간단하고 안전한 방법입니다. `self-invocation`이 일어나는 메서드를 **다른 빈으로 이동**시키면, 호출이 자연스럽게 프록시를 거치게 됩니다.

```
@Service
class OrderService(
  private val stockService: StockService
) {
  fun placeOrder() {
    stockService.updateStock()  // 다른 빈 호출 → 프록시 거침
  }
}

@Service
class StockService {
  @Transactional
  fun updateStock() { ... }
}
```

이 방법의 장점은 **추가 설정이 없다**는 것입니다. 단점은 "이 정도로 빈을 나눠야 하나" 하는 저항감이 생길 수 있다는 점입니다. 다만 **트랜잭션 경계가 다른 메서드를 같은 빈에 두는 것 자체가 책임 분리가 애매해진 신호**일 때가 많습니다. 분리가 구조적으로 나쁜 선택이 아닐 가능성이 큽니다.

2. `self` 주입 — 자기 자신을 프록시로 주입

자기 자신을 주입받아 프록시를 통해 호출합니다.

```
@Service
class OrderService(
  private val self: OrderService
) {
  fun placeOrder() {
    self.updateStock()  // self는 프록시 → 어드바이스 발동
  }

  @Transactional
  fun updateStock() { ... }
}
```

Spring 4.3부터는 이 순환 참조를 컨테이너가 자연스럽게 처리합니다. 다만 **읽는 사람을 혼란스럽게 만드는 패턴**입니다. "이 서비스가 자기 자신을 주입받는 이유가 뭐지?" 하는 질문이 반복되면, 구조를 나누는 편이 낫다는 신호입니다.

3. `ApplicationContext`로 프록시 빈 꺼내기

```
@Service
class OrderService(
  private val context: ApplicationContext
) {
  fun placeOrder() {
    val proxy = context.getBean(OrderService::class.java)
    proxy.updateStock()
  }

  @Transactional
  fun updateStock() { ... }
}
```

테스트 용도라면 몰라도, 애플리케이션 코드에 넣기에는 컨테이너 API가 도메인 레이어에 섞여 좋지 않습니다. 실무에서 이걸 쓰는 일은 거의 없습니다.

4. `AspectJ` 위빙 — 프록시 없이 바이트코드에 직접 심기

AspectJ는 **컴파일 타임** 또는 **클래스 로딩 타임**에 바이트코드를 직접 수정해 어드바이스를 심습니다. 그래서 `this.method()` 같은 내부 호출에도 어드바이스가 동작합니다.

```
@EnableTransactionManagement(mode = AdviceMode.ASPECTJ)
@EnableLoadTimeWeaving
@Configuration
class AppConfig
```

위빙 방식의 장점은 self-invocation을 비롯한 모든 제약이 사라진다는 것입니다. 단점은 **설정이 복잡하고, 빌드 시스템(`maven`/`gradle`)과 JVM 옵션에 변경**이 필요하다는 것입니다. 대부분의 프로젝트는 1번(빈 분리)으로 충분하기 때문에, AspectJ까지 도입하는 일은 드뭅니다.

Phase 5. 같은 함정을 공유하는 다른 애너테이션

`self-invocation` 문제는 `@Transactional`만의 문제가 아닙니다. **Spring AOP 기반의 모든 애너테이션**이 같은 구조를 공유합니다.

애너테이션	역할	self-invocation 시 증상
`@Transactional`	트랜잭션 경계	트랜잭션이 아예 시작되지 않음
`@Async`	비동기 실행	호출 스레드에서 **동기 실행**
`@Cacheable` / `@CacheEvict`	캐시	캐시가 적용되지 않음
`@Retryable`	재시도	재시도 없이 예외가 그대로 전파
`@Validated`	메서드 인자 검증	검증이 실행되지 않음
`@PreAuthorize` / `@Secured`	메서드 수준 보안	권한 체크가 걸리지 않음

애너테이션이 달려 있는데 **아무 동작도 하지 않는 것처럼 보일 때**는 가장 먼저 self-invocation을 의심합니다. 로그에 에러도 경고도 찍히지 않고 "조용히" 무력화되는 것이 이 문제의 특징입니다.

Phase 6. 프록시가 아예 적용되지 않는 경우

`private` 메서드

JDK 동적 프록시도 CGLIB도 `private` **메서드를 가로채지 못합니다.** 서브클래스에서 접근할 수 없거나(CGLIB), 인터페이스에 존재하지 않기

때문(JDK)입니다. `private` 메서드에 `@Transactional`을 붙이면 **경고 없이 무시**됩니다.

`final` 메서드 / `final` 클래스

CGLIB은 서브클래싱으로 동작하기 때문에 `final` 메서드와 `final` 클래스는 오버라이드할 수 없습니다. Spring Boot는 프록시 생성 실패 시 런타임 에러를 냅니다.

```
Cannot subclass final class ...
```

Kotlin에서는 앞서 언급한 `kotlin-spring` 플러그인이 해결해 줍니다. Java에서는 클래스와 메서드에 직접 `final`을 빼야 합니다.

`static` 메서드

Spring AOP는 **인스턴스 메서드**만 가로챕니다. `static` 메서드는 애너테이션이 붙어 있어도 프록시가 동작하지 않습니다. `static` 컨텍스트에서 트랜잭션/캐시가 필요하면 **인스턴스 메서드로 옮기거나,** 필요하면 AspectJ 위빙을 검토합니다.

프록시 내부에서 `this`로 호출

이 글의 핵심 주제인 self-invocation입니다. 반복하면 — **프록시가 감싸는 것은 외부에서 들어오는 호출뿐**입니다. 원본 객체 안에서 `this`로 호출하면 전부 우회합니다.

정리

Spring AOP 기반의 애너테이션들은 **프록시가 호출을 가로채는 구조 위에서 동작**합니다. 이 구조는 외부 호출에는 자연스럽게 들어맞지만, 같은 객체 안에서의 내부 호출에는 개입할 수 없습니다. 결과적으로 `@Transactional`, `@Async`, `@Cacheable`, `@Retryable` 같은 애너테이션이 self-invocation에서는 전부 조용히 무력화됩니다.

실무에서 가장 먼저 꺼낼 답은 **메서드를 다른 빈으로 분리**하는 것입니다. `self` 주입, `ApplicationContext`, AspectJ 위빙도 가능하지만 대부분 빈 분리보다 복잡도가 높습니다. 그리고 Kotlin 프로젝트라면 **`kotlin-spring` 플러그인**을 빠뜨리지 않도록 빌드 설정부터 확인하는 것이 안전합니다.

"분명히 애너테이션을 달았는데 아무 일도 일어나지 않는다"라는 증상을 만났을 때, 점검 순서는 거의 정해져 있습니다.

1. 프록시가 생성되어 외부에서 호출되고 있는가 (self-invocation 여부)
2. 메서드가 `public`인가, 클래스/메서드가 `final`이 아닌가
3. Kotlin이면 `kotlin-spring` 플러그인이 켜져 있는가
4. 주입되는 필드 타입이 실제 프록시를 받고 있는가 (디버거로 확인 가능)

이 네 가지만 순서대로 봐도 대부분의 "조용한 무력화"를 빠르게 찾을 수 있습니다. 다음 글은 시리즈를 JPA 쪽으로 다시 돌려서, **영속성 컨텍스트의 `flush`와 `OSIV`가 실제로 어디까지 열려 있는지**를 상세히 다룹니다.

PART 4

4부. 성능과 엔티티 다루기

Chapter 7

JPA 배치 쓰기 완전 정복 — `hibernate.jdbc.batch_size`와 `IDENTITY` 함정

#배치 쓰기

JPA에서 대량 INSERT/UPDATE가 느린 진짜 원인은 네트워크 왕복입니다. JDBC 배치, hibernate.jdbc.batch_size, order_inserts, MySQL의 rewriteBatchedStatements, 그리고 IDENTITY 전략이 배치를 막는 이유와 해결책을 정리합니다.

배치 쓰기, 왜 따로 알아야 하나요?

대량 `INSERT`나 `UPDATE`가 느릴 때 대부분의 첫 반응은 "DB가 느려서"입니다. 그러나 실제 원인은 DB가 아니라 **네트워크 왕복(round-trip)** 인 경우가 압도적으로 많습니다. 한 행씩 `INSERT`를 반복하면 **행 수 × 왕복 비용**이 그대로 응답 시간에 더해집니다.

- 사용자 데이터 1만 건을 한 번에 저장하는데 **응답이 수 분** 걸립니다
- SQL 로그를 보니 `INSERT`가 **한 건씩 수천 번** 반복됩니다
- `hibernate.jdbc.batch_size`를 설정했는데 **여전히 개별 쿼리가 나갑니다**
- 배치 옵션을 다 켰는데도 **MySQL에서는** 속도 차이가 작습니다

이 증상들의 원인은 전부 **JPA의 배치 쓰기가 어떤 조건에서 켜지고, 어떤 조건에서 꺼지는지**를 몰라서 생깁니다. 특히 `IDENTITY` ID 생성 전략은 배치를 **구조적으로 막는 가장 큰 이유**인데, MySQL 환경에서는 이걸 우회하기가 까다롭습니다. 이 글은 배치가 동작하는 구조, 막히는 이유, 그리고 실무에서 우회하는 방법을 순서대로 정리합니다.

> **기준:** 이 글은 **Jakarta Persistence 3.1** 명세와 **Hibernate 6.x, MySQL 8.4 Connector/J** 기준으로 작성합니다. Hibernate 배치 옵션은 Hibernate 6 User Guide — § 13. Batching, MySQL JDBC 드라이버의 `rewriteBatchedStatements` 파라미터는 MySQL Connector/J Reference를 참조합니다. 영속성 컨텍스트와 flush 동작은 앞 글을 전제로 합니다. 코드 예시는 Kotlin + Spring Data JPA입니다.

먼저 가장 짧은 답부터 보면

- JDBC 드라이버의 **배치(`addBatch` / `executeBatch`)**는 여러 SQL을 **한 번의 네트워크 왕복**으로 전송하는 메커니즘입니다
- Hibernate는 `hibernate.jdbc.batch_size`가 설정되어 있어야 배치를 **시도**합니다. 기본값은 **비활성화**입니다
- **`IDENTITY` ID 전략은 배치를 구조적으로 막습니다**. ID를 DB가 만들어 주기 때문에 `persist()` 순간 개별 `INSERT`가 필요합니다
- MySQL은 `IDENTITY = AUTO_INCREMENT`이므로 이 함정을 정면으로 맞습니다. **애플리케이션에서 ID를 만드는 방식**(UUIDv7, Snowflake)이 실질적 대안입니다

- MySQL 드라이버의 **`rewriteBatchedStatements=true`** 를 켜면 다중 `INSERT`가 단일 `multi-VALUES` 문으로 재작성되어 실제 효과가 큽니다

Phase 1. JDBC 배치가 실제로 해결하는 것

핵심: 네트워크 왕복 수를 줄입니다

`INSERT` 한 건의 비용은 대부분 **DB의 쓰기 자체**가 아니라 **네트워크 왕복 + SQL 파싱**입니다. 한 건을 1ms에 처리할 수 있는 DB라도, 왕복이 5ms면 10,000건을 차례로 보내는 데 수십 초가 걸립니다.

```
개별 INSERT 10,000건
  app ──INSERT──▶ db      왕복 1
   ◀──ack────
  app ──INSERT──▶ db      왕복 2
   ◀──ack────
  ... × 10,000
```

JDBC 배치는 **여러 SQL을 한 번의 왕복으로 전송**합니다.

```
배치 INSERT 10,000건 (batch_size = 500)
  app ──INSERT × 500──▶ db   왕복 1
   ◀───acks──────
  ... × 20 (왕복 수 1/500로 축소)
```

JDBC 수준에서의 동작

JDBC 드라이버는 `Statement#addBatch()`로 쌓인 SQL을 `executeBatch()`로 한 번에 전송합니다. Hibernate의 `hibernate.jdbc.batch_size`는 이 `executeBatch()`가 언제 호출될지를 조정합니다.

Phase 2. Hibernate 배치 설정

기본 설정

```
spring:
  jpa:
    properties:
      hibernate:
        jdbc:
          batch_size: 100
        order_inserts: true
        order_updates: true
```

각 설정의 역할은 이렇습니다.

속성	역할
`hibernate.jdbc.batch_size`	한 배치에 묶을 최대 SQL 수. 기본값 `0` (비활성화)
`hibernate.order_inserts`	`INSERT` 를 **테이블별로 정렬**해서 같은 테이블 것끼리 연속 배치 가능하게 함
`hibernate.order_updates`	`UPDATE` 도 같은 방식으로 정렬

왜 `order_inserts` 가 필요한가요?

JDBC 배치는 **같은 SQL 문자열**만 묶을 수 있습니다. Hibernate는 엔티티를 영속화한 순서대로 `INSERT` 큐를 만들기 때문에, 여러 종류의 엔티티가 섞여 있으면 다음처럼 됩니다.

```
INSERT user ...
INSERT order ...
INSERT user ...
INSERT order ...
```

이 상태에서는 배치 크기가 아무리 커도 **같은 테이블 `INSERT` 가 연달아 있지 않아** 묶이지 않습니다. `order_inserts=true` 는 이 큐를 테이블별로 정렬해 다음처럼 바꿉니다.

```
INSERT user ...
INSERT user ...
INSERT order ...
INSERT order ...
```

이 상태가 되어야 배치가 실제로 작동합니다.

배치가 실행되는 시점

Hibernate는 다음 중 하나의 시점에 배치를 내보냅니다.

1. 배치 큐 크기가 **`batch_size` 에 도달**할 때
2. **다른 테이블에 대한 SQL**이 끼어들 때 (같은 문자열이 아니므로 먼저 flush)
3. **커밋 직전**의 최종 `flush`

이 동작은 `order_inserts` 와 결합되어야 효과가 납니다. 둘 다 켜지지 않은 상태에서는 설정 값만 바꿔도 실제 배치로 묶이지 않을 수 있습니다.

Phase 3. `IDENTITY` 전략이 배치를 막는 진짜 이유

핵심: ID를 DB가 만들어야 해서 `INSERT` 를 하나씩 보낼 수밖에 없습니다

`GenerationType.IDENTITY` 는 `AUTO_INCREMENT` 같은 DB 기능으로 ID를 만듭니다. 이 전략의 문제는 "**`INSERT` 를 실행해야 ID를 알 수 있다**"는 점입니다.

영속성 컨텍스트는 엔티티를 1차 캐시에 넣을 때 ID를 키로 씁니다. 그래서 `persist()` 순간 ID를 알아야 하고, 결과적으로 `INSERT`를 **즉시** 실행합니다. 이 순간 배치 큐는 사용할 수 없습니다.

```
@Transactional
fun insertMany(users: List<User>) {
  users.forEach { em.persist(it) }  // 각 persist()마다 INSERT 실행
}
```

`batch_size` 설정이 아무리 커도 위 코드는 건별 `INSERT`를 냅니다. MySQL 환경에서 배치가 "안 먹는" 듯 보이는 가장 큰 원인이 이것입니다.

Hibernate는 이 상황에서 조용히 배치를 끕니다

Hibernate는 `IDENTITY` 전략의 엔티티에 대해 **경고 없이 배치를 비활성화**합니다. 별도의 로그 메시지가 뜨지 않기 때문에 "`batch_size`를 줬는데 왜 안 빨라지지?" 하고 의아한 상황을 만들기 쉽습니다. SQL 로그에서 **개별 `INSERT`가 줄줄이 찍히는지**를 직접 확인하는 것이 가장 빠른 진단입니다.

Phase 4. `IDENTITY`를 우회하는 네 가지 방법

1. `SEQUENCE` (PostgreSQL / Oracle 환경)

`SEQUENCE` 전략은 DB에 "다음 ID"를 미리 물어볼 수 있습니다. Hibernate는 시퀀스를 **pooled 방식**으로 여러 ID를 한꺼번에 가져와

캐싱할 수 있어, `persist()` 시점에 `INSERT` 없이도 ID를 확보할 수 있습니다.

```
@Entity
class User(
  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE,
generator = "user_seq")
  @SequenceGenerator(name = "user_seq", sequenceName =
"user_seq", allocationSize = 50)
  val id: Long = 0,
  val name: String
)
```

이 전략에서는 배치가 자연스럽게 동작합니다. 다만 **MySQL에는 SEQUENCE가 없습니다**.

2. `TABLE` 전략 (모든 DB 사용 가능, 그러나 느림)

`TABLE` 전략은 별도의 ID 관리 테이블을 만들어 ID를 할당합니다. 이식성은 좋지만 **매 할당마다 ID 테이블을 `UPDATE`** 해야 하기 때문에 성능이 좋지 않습니다. MySQL 환경에서도 대안으로 거론되지만 대규모 환경에서는 거의 사용되지 않습니다.

3. 애플리케이션이 ID를 생성 — `UUIDv7`, `Snowflake`

가장 실무적인 선택은 **ID를 애플리케이션이 만드는 것**입니다. ID를 영속화 전에 이미 알고 있기 때문에 `persist()` 순간 `INSERT`를 내보낼 필요가 없습니다. 배치 큐에 그대로 쌓일 수 있습니다.

```
@Entity
class User(
  @Id
  val id: UUID,   // UUIDv7 등 애플리케이션에서 생성
  val name: String
)
```

UUIDv7은 RFC 9562로 표준화되어 있고, 시간 단조 증가 속성이 있어 **`BIGINT AUTO_INCREMENT`에 가까운 인덱스 성능**을 보입니다. 완전 랜덤 UUIDv4는 `InnoDB` 클러스터드 인덱스에 삽입할 때 페이지 분할 비용이 커서, 대량 `INSERT`에는 적합하지 않습니다. 애플리케이션 ID를 선택할 때는 **시간 단조 증가 여부**를 반드시 확인합니다.

4. `JdbcTemplate` / 네이티브 배치 — 최후의 수단

정말 큰 `INSERT`(수십만 건 이상)는 JPA를 우회해 **`JdbcTemplate.batchUpdate()`** 를 쓰는 편이 현실적입니다. 영속성 컨텍스트를 거치지 않고 직접 SQL을 배치로 보내기 때문에, ID 전략과 무관하게 최고 성능이 나옵니다.

```
jdbcTemplate.batchUpdate(
  "INSERT INTO user (id, name) VALUES (?, ?)",
  users.chunked(1000).flatMap { chunk ->
    chunk.map { arrayOf<Any>(it.id, it.name) }
  }
)
```

이 방법의 트레이드오프는 명확합니다. **영속성 컨텍스트의 장점(변경 감지, 연관 관리 등)이 사라집니다**. 대량 초기 로딩이나 로그 적재처럼 **영속성 컨텍스트가 오히려 짐**이 되는 경우에만 선택합니다.

Phase 5. MySQL에서 반드시 확인해야 할 드라이버 설정

`rewriteBatchedStatements=true`

MySQL Connector/J는 기본적으로 **배치 INSERT를 여러 개의 개별 `INSERT` 문으로 전송**합니다. 즉, 네트워크 왕복 수는 줄어들어도 SQL 파싱 비용은 그대로입니다. 이 문제를 풀기 위한 드라이버 옵션이 `rewriteBatchedStatements=true`입니다.

```
spring:
  datasource:
    url: jdbc:mysql://.../mydb?rewriteBatchedStatements=true
```

이 옵션을 켜면 드라이버가 다음처럼 SQL을 재작성합니다.

```
-- 옵션 꺼짐
INSERT INTO user (id, name) VALUES (?, ?);
INSERT INTO user (id, name) VALUES (?, ?);
...

-- 옵션 켜짐
INSERT INTO user (id, name) VALUES (?, ?), (?, ?), (?, ?),
...;
```

이 한 줄의 설정만으로 **대량 `INSERT` 성능이 5~10배 이상** 빨라지는 경우가 많습니다. MySQL 환경에서 **배치를 켜고도 효과가 작게 느껴진다면, 가장 먼저 이 옵션을 확인**합니다.

`max_allowed_packet`

`rewriteBatchedStatements`를 켜면 한 `INSERT` 문의 크기가 커집니다. MySQL 서버의 `max_allowed_packet` 이 너무 작으면 "packet too large" 에러가 납니다. MySQL 8.x의 기본값은 `64MB` 로 충분하지만, 운영 환경에서는 명시적으로 확인합니다.

Phase 6. `UPDATE` 배치와 `@Version`

낙관적 잠금이 켜진 엔티티도 배치로 묶입니다

`@Version`으로 낙관적 잠금이 걸린 엔티티의 `UPDATE`를 배치로 묶는 것은 **Hibernate 5 이후부터 기본값으로 활성화**되어 있습니다. 관련 설정은 `hib-`

`ernate.jdbc.batch_versioned_data` 이고, **기본값이 `true`** 입니다(pre-12c Oracle dialect처럼 일부 환경에서만 꺼져 있습니다).

```yaml
spring:
  jpa:
    properties:
      hibernate:
        jdbc:
          batch_versioned_data: true   # 기본값, 명시적 확인용
```

이때 Hibernate는 `executeBatch()` 반환 값에서 **각 행의 업데이트 수**를 확인해 버전 충돌을 감지합니다. **드라이버가 배치 결과의 개별 row count를 지원**해야 동작하고, MySQL Connector/J는 이 기능을 지원합니다. 즉 현대 MySQL 환경이라면 이 설정은 이미 켜진 상태로 동작합니다.

Phase 7. 실전 점검 순서

"배치를 켰는데 왜 안 빨라지지?"라는 의심이 들 때, 다음 순서로 점검하면 대부분의 원인이 드러납니다.

1. **ID 생성 전략이 `IDENTITY`가 아닌가?** — MySQL이라면 애플리케이션 ID(UUIDv7 등)로 바꾸는 것이 가장 큰 효과
2. **`hibernate.jdbc.batch_size`가 설정되어 있고 0이 아닌가?** — 기본값이 0이라 많은 프로젝트가 놓칩니다
3. **`order_inserts` / `order_updates`가 켜져 있는가?** — 엔티티가 섞여 있을 때 반드시 필요

4. **MySQL 드라이버의 `rewriteBatchedStatements=true` 가 켜져 있는가?** — MySQL 환경에서는 이 한 줄이 결정적
5. **`@Version` 이 걸린 엔티티의 `UPDATE` 를 쓰고 있다면** `batch_versioned_data` 가 꺼져 있지 않은지 확인 (기본값 `true` 이므로 보통 문제없음)
6. **대량 처리 중 `em.flush() + em.clear()` 를 주기적으로 호출하고 있는가?** — 영속성 컨텍스트가 커지면 변경 감지 비용만으로도 병목이 됩니다

대량 처리 루프의 일반적인 틀

```
@Transactional
fun bulkInsert(items: List<Item>) {
  items.chunked(1000).forEach { chunk →
    chunk.forEach { em.persist(it) }
    em.flush()
    em.clear()   // 영속성 컨텍스트를 비워 메모리와 dirty-check 비용
을 통제
  }
}
```

이 구조에 **애플리케이션 ID + 배치 설정 + `rewriteBatchedStatements`** 가 맞물리면 JPA로도 수만 건/초 수준의 처리량이 나옵니다. 그 이상의 처리량이 필요하면 `JdbcTemplate` 로 우회합니다.

정리

JPA 배치 쓰기의 효과는 **"네트워크 왕복 수를 줄인다"** 이 한 줄에 집약됩니다. 그러나 이 효과를 실제로 보려면 여러 조각이 동시에 맞아야 합니다.

조각	확인할 것
Hibernate	`batch_size`, `order_inserts`, `order_updates`
`UPDATE` 배치	`batch_versioned_data` (Hibernate 5+ 기본 `true`) + 드라이버의 개별 row count 지원
ID 전략	`IDENTITY`면 배치 불가, 애플리케이션 ID로 대체
MySQL 드라이버	`rewriteBatchedStatements=true` + `max_allowed_packet` 확인
메모리 관리	주기적 `em.flush() + em.clear()`

MySQL 환경에서 이 조합 중 **`IDENTITY` 회피**와 **`rewriteBatchedStatements`**가 가장 큰 레버입니다. 둘 중 하나라도 빠지면 배치 효과의 상당 부분을 잃습니다. 설계 시점부터 ID 생성 전략을 고민하는 것이, 뒤늦게 튜닝으로 회복하는 것보다 훨씬 저렴합니다.

다음 글은 JPA 시리즈의 마지막 — **`merge` vs `persist`와 `detached` 엔티티 다루기**를 정리합니다. 지금까지 쌓인 영속성 컨텍스트·fetch 전략·트랜잭션·배치의 개념이 모여, 실무에서 엔티티 상태를 언제 어떻게 올려야 하는지가 정돈됩니다.

Chapter 8

JPA `merge` vs `persist` 완전 정복 — `detached` 엔티티를 어떻게 다뤄야 하나요?

#영속성 컨텍스트

persist와 merge가 실제로 하는 일, Spring Data JPA의 save가 왜 둘을 섞어 쓰는지, detached 엔티티를 save로 저장할 때 생기는 구조적 위험, 그리고 find + modify 패턴이 왜 권장되는지를 정리합니다.

`merge`와 `persist`, 왜 정확히 알아야 하나요?

Spring Data JPA의 `save()` 한 줄로 저장을 끝내는 코드는 많습니다. 그런데 조금만 상황이 복잡해지면 이런 증상이 나타납니다.

- 수정하려고 `save()`를 불렀는데 `UPDATE`가 아니라 **`INSERT`가 튀어나옵니다**
- 일부 필드만 바꿨는데 `UPDATE`에 **모든 컬럼**이 포함됩니다
- `save()`가 건별로 `SELECT`를 먼저 내보내 **쿼리 수가 예상의 두 배**가 됩니다
- DTO로 받은 요청을 `User` 엔티티로 바꿔 `save()`했더니 **기존 컬럼 값이 `null`로 덮여버렸습니다**

이 모든 증상의 뿌리는 하나입니다. **`persist` 와 `merge` 가 무엇을 하는지 다르고, `save()` 가 내부에서 이 두 가지를 상황에 따라 섞어 쓴다**는 것입니다. 그리고 그 안에서 `detached` 엔티티를 부주의하게 다루면 **데이터가 덮이거나 사라집니다.**

이 글은 시리즈의 마지막으로, 지금까지 쌓아온 영속성 컨텍스트·변경 감지·flush 개념을 **엔티티 상태 다루기** 관점으로 정리합니다.

> **기준:** 이 글은 **Jakarta Persistence 3.1 (JPA 3.1) 명세**와 **Hibernate 6.x, Spring Data JPA 3.x** 기준으로 작성합니다. `persist` / `merge` 정의는 Jakarta Persistence 3.1 — § 3.2 Entity Instance Life Cycle, Spring Data의 `save()` 구현은 `SimpleJpaRepository.save` JavaDoc과 소스를 참조합니다. 영속성 컨텍스트와 엔티티 상태 전이는 앞 글을 전제로 합니다. 코드 예시는 Kotlin + Spring Data JPA입니다.

먼저 가장 짧은 답부터 보면

- **`persist(entity)`** — `new` 엔티티를 `managed` 로 바꿉니다. `detached` 에는 쓸 수 없습니다
- **`merge(entity)`** — `detached` 엔티티의 **내용을 새 `managed` 인스턴스에 복사**합니다. 반환값이 managed고 원본은 그대로 detached
- **Spring Data JPA의 `save()`** — 엔티티가 **새로 만든 것**이면 `persist`, 아니면 `merge` 로 **분기**합니다
- `save()` 로 기존 엔티티를 수정하려고 DTO → 엔티티 변환을 쓰면 **의도하지 않은 필드가 `null` 로 덮일 수 있습니다**

- 실무 권장은 **`find` 로 `managed` 엔티티를 가져와 필드를 바꾸는 패턴**입니다. `merge` 는 대부분 필요 없습니다

Phase 1. `persist` 는 무엇을 하나요?

핵심: `new` 상태의 엔티티를 `managed` 로 올립니다

```
val user = User(name = "Alice")   // new 상태, id 아직 없음
em.persist(user)                    // managed로 전이
// user.id가 채워짐 (ID 전략에 따라 INSERT 타이밍은 다름)
```

JPA 명세는 `persist` 의 역할을 이렇게 정의합니다.

- 인자 엔티티를 **managed** 상태로 만듭니다
- 영속성 컨텍스트에 **같은 식별자를 가진 엔티티가 이미 있으면 `EntityExistsException`**
- **detached 엔티티에 `persist` 를 호출하면** `EntityExistsException` 또는 `PersistenceException`

핵심 제약: `persist` 는 `new` 만 받는 연산

`persist` 는 **아직 식별자를 가지지 않은 새 인스턴스**를 위한 API입니다. 이미 DB에 존재하는 엔티티를 "다시 persist"한다는 개념은 JPA에 없습니다. 이 제약이 뒤에서 `merge` 와 역할이 갈리는 원인이 됩니다.

Phase 2. `merge` 는 무엇을 하나요?

핵심: `detached` 엔티티의 내용을 새 `managed` 인스턴스에 복사합니다

```
val detached = User(id = 1, name = "after")  // detached
(ID만 가짐)

val managed = em.merge(detached)

detached.name = "changed"  // 반영 안 됨 (여전히 detached)
managed.name = "changed"   // 반영됨 (managed)
```

`merge` 의 동작은 정확히 이렇습니다.

1. 같은 식별자의 `managed` 엔티티가 영속성 컨텍스트에 있는지 확인
2. 있으면 그 엔티티의 **필드 값들을 인자 엔티티의 값으로 덮어씀**
3. 없으면 **DB에서** `SELECT` 해서 `managed` 상태로 로딩한 뒤, 같은 방식으로 덮어씀
4. 작업이 끝난 `managed` 인스턴스를 **반환**

`merge` 가 만드는 숨은 `SELECT`

영속성 컨텍스트에 없는 엔티티를 `merge` 하면 Hibernate는 **DB에서 먼저** `SELECT` 해서 현재 상태를 가져옵니다. 그 다음에 `UPDATE` 를 판단합니다.

그래서 `merge` 한 번은 **`SELECT` + (필요 시) `UPDATE`** 두 쿼리를 만들 수 있습니다.

대량 업데이트 경로에서 `merge`를 반복하면, 의도하지 않은 `SELECT` 가 건별로 따라 붙어 성능이 선형으로 나빠집니다.

반환값을 꼭 써야 합니다

`merge`가 처리한 **`managed` 인스턴스는 반환값**입니다. 인자로 넘긴 엔티티는 여전히 `detached` 상태입니다. 인자를 그대로 쓰면 이후 변경이 반영되지 않습니다.

```
// ❌ 자주 하는 실수
em.merge(detached)
detached.name = "x"     // 반영 안 됨

// ✅ 올바른 사용
val managed = em.merge(detached)
managed.name = "x"
```

Phase 3. Spring Data JPA의 `save()`는 무엇을 할까요?

`SimpleJpaRepository.save`의 실제 로직

Spring Data JPA의 `save()`는 한 줄로 보이지만, 내부에서 **`persist`와 `merge`를 상황에 따라 분기**합니다.

```
@Transactional
override fun <S : T> save(entity: S): S {
  return if (entityInformation.isNew(entity)) {
    entityManager.persist(entity)
    entity
  } else {
    entityManager.merge(entity)   // 반환값이 managed
  }
}
```

즉 `save()` 는 엔티티가 "새것"이냐 아니냐에 따라 완전히 다른 연산입니다.

"새것"을 어떻게 판별하나요?

Spring Data JPA는 `isNew()` 판단에 다음 우선순위를 씁니다.

1. 엔티티가 **`Persistable` 인터페이스**를 구현하면 그 `isNew()` 를 사용
2. 그렇지 않으면 **`@Id` 필드의 값이 `null` 또는 원시 타입의 기본값(0)** 인지 확인

이 기본 판별이 맞는 경우가 대부분입니다. 그러나 다음 상황에서는 **판단이 잘못되기 쉽습니다.**

- **`@Id` 가 `Long` 원시 타입**으로 선언된 엔티티: 0이 기본값이라 **ID를 직접 할당한 엔티티가 "새것"으로 잘못 판단**될 수 있습니다
- **애플리케이션에서 생성한 UUID/ID**로 저장하는 경우: 이미 ID가 있는데 `save()` 가 `merge` 로 동작해 불필요한 `SELECT` 를 내보냅니다

ID를 애플리케이션이 관리하는 패턴(이전 글의 `UUIDv7`/`Snowflake`)을 쓸 때는 `Persistable` 인터페이스를 구현해 `isNew()`를 명시적으로 지정하는 편이 안전합니다.

`save()` 한 번에 `SELECT`가 붙는 이유

`save()`가 `merge`로 분기되면 앞서 본 대로 **`SELECT` + (변경 시) `UPDATE`**를 수행합니다. DB에서 현재 상태를 먼저 읽는 것이 필연적입니다.

이 때문에 **"저장만 하면 된다"고 생각한 코드가 예상의 두 배 쿼리를 냅니다**. 대량 업데이트 루프에서 `save()`를 반복하면 이 비용이 선형으로 쌓입니다. 변경 감지로 충분한 경로에서는 `save()`를 부르지 않는 편이 오히려 저렴합니다.

Phase 4. `detached` 엔티티 + `save()`의 구조적 위험

요청을 엔티티로 바꿔 `save()` 하면?

가장 자주 하는 실수는 이 패턴입니다.

```
// 요청 DTO
data class UpdateUserRequest(val id: Long, val name:
String)

// ❌ 위험한 코드
@PutMapping("/users/{id}")
fun update(@PathVariable id: Long, @RequestBody req:
UpdateUserRequest) {
  val user = User(id = req.id, name = req.name)  //
detached, 일부 필드만 세팅
  userRepository.save(user)                       // merge
→ 모든 필드를 이 값으로 덮음
}
```

`merge`는 인자 엔티티의 **모든 필드를 그대로 복사**합니다. `req.name`만 세팅하고 다른 필드는 세팅하지 않은 `detached` 엔티티를 넘기면, `merge`는 다음과 같이 동작합니다.

- `name = req.name`
- **그 외 필드 = `null` 또는 기본값**

결과적으로 **이메일, 가입일, 프로필 등 세팅하지 않은 필드가 전부 `null`로 덮입니다**. 이 버그는 테스트에서 보이지 않다가 운영에서 데이터를 망치는 전형적인 패턴입니다.

Hibernate의 `@DynamicUpdate`가 해결해 주지 않습니다

`@DynamicUpdate`는 **변경 감지 결과에서 바뀐 컬럼만 `UPDATE`에 포함**시키는 옵션입니다. 그러나 `merge`는 **변경 감지와 무관하게 인자**

엔티티의 필드 값을 전부 복사합니다. `@DynamicUpdate` 가 켜져 있어도 `merge` 경로에서는 **모든 필드가 복사된 이후의 상태**가 "현재 값"이 됩니다.

즉 `@DynamicUpdate` 가 `null` 덮어씀을 막아주지 않습니다.

Phase 5. 권장 패턴 — `find` + modify

구조

```
@PutMapping("/users/{id}")
@Transactional
fun update(@PathVariable id: Long, @RequestBody req:
UpdateUserRequest) {
  val user = userRepository.findById(id).orElseThrow {
NotFoundException() }
  user.name = req.name        // 필요한 필드만 변경
  // save() 호출 불필요: 변경 감지가 커밋 시점에 UPDATE 생성
}
```

이 패턴의 장점은 명확합니다.

- `find` 로 **`managed` 엔티티**를 가져왔기 때문에 이후 변경은 **변경 감지**로 자동 반영
- **바뀐 필드만** `UPDATE` 에 포함 (`@DynamicUpdate` 와 결합 시)
- `merge` 가 내부에서 **덮어쓰는 위험이 원천적으로 없음**
- `SELECT` 1번 + 커밋 시점 `UPDATE` 1번으로 끝남 (`save()` 의 `SELECT` + `UPDATE` 와 동일하거나 적음)

엔티티 내부에 update 메서드

도메인 관점에서는 필드를 직접 바꾸기보다 엔티티에 update 메서드를 두는 편이 더 안전합니다.

```
@Entity
class User(
  @Id val id: Long,
  var name: String,
  var email: String
) {
  fun changeName(newName: String) {
    require(newName.isNotBlank())
    this.name = newName
  }
}

@Transactional
fun update(id: Long, req: UpdateUserRequest) {
  val user = userRepository.findById(id).orElseThrow()
  user.changeName(req.name)
}
```

이 패턴은 **어떤 필드가 어떤 조건에서만 바뀌어야 하는지**를 엔티티가 책임지게 합니다. `merge`로 외부에서 필드를 통째로 덮는 방식보다 버그 위험이 훨씬 낮습니다.

Phase 6. 언제 `merge`가 필요한가요?

`merge`가 꼭 필요한 경우는 실무에서 제한적입니다.

1. 외부에서 받은 `detached` 엔티티를 그대로 써야 하는 경우

`HttpSession`, 캐시, 또는 외부 프로세스에서 온 엔티티를 다시 영속성 컨텍스트에 붙여야 하는 경우입니다. 대부분의 현대 API 서버는 DTO-Entity 경계를 유지하기 때문에 이 상황은 드뭅니다.

2. 파일 / 배치에서 읽어온 엔티티 스냅샷을 한 번에 반영

외부 파일이나 다른 DB의 스냅샷을 그대로 한 번에 반영할 때, "모든 필드를 통째로 덮는 것"이 의도일 수 있습니다. 이 경우 `merge`가 의미를 가집니다. 다만 **전체 필드를 스냅샷에 맞게 완전하게 채웠는지** 확인해야 합니다.

3. 그 외에는 거의 `find + modify`

앞서 본 대로 대부분의 수정 API는 **`find + modify`** 가 더 안전하고 단순합니다. "일단 save로 해보자"가 일으키는 버그의 많은 수가 이 지점에서 생깁니다.

Phase 7. 엔티티 수명과 `equals` / `hashCode`

`detached` 엔티티와 `managed` 엔티티를 섞어 쓰다 보면 `Set`, `Map`, `HashSet` 에서 **같은 엔티티인데 `contains()`가 false**를 반환하는 상황을 만납니다. 원인은 `equals`/`hashCode` 구현 때문입니다.

기본값의 문제

- **`equals` 기본 구현** — 참조 동일성(`===`). `detached` 후 같은 ID로 다시 조회된 인스턴스와 같다고 보지 않습니다
- **`hashCode` 기본 구현** — 객체 헤더 기반이라 엔티티의 `identity` 와 무관

많이 권장되는 패턴

- `equals` 와 `hashCode` 를 **식별자 기반으로 구현**하되, **`id` 가 아직 없을 때(`new` 상태)** 도 일관되도록 유의
- 일반적인 실무 조언은 **`id` 가 있으면 id 기반으로, 없으면 `System.identityHashCode`** 로 fallback
- Lombok의 `@EqualsAndHashCode(onlyExplicitlyIncluded = true)` 로 id만 포함시키는 방법이 자주 쓰임

Kotlin `data class` 는 모든 필드를 `equals` / `hashCode` 에 포함시키므로 **엔티티 전용으로는 권장되지 않습니다**. 엔티티는 일반 클래스로 선언하고 `equals` / `hashCode` 를 식별자 기준으로 명시적으로 정의하는 편이 안전합니다.

정리

`persist` 와 `merge` 는 겹쳐 보이지만 역할이 다릅니다.

연산	받는 상태	하는 일
`persist`	`new`	`managed` 로 전이. `detached` 에는 쓸 수 없음
`merge`	`detached` / `new`	새 `managed` 인스턴스에 **필드 전체 복사**. 기존 엔티티는 그대로 detached
`save()` (Spring Data)	자동 판별	`isNew()` 면 `persist` , 아니면 `merge`

`merge` 가 갖는 본질적 위험은 **"모든 필드를 덮는다"** 는 데 있습니다. 요청 DTO를 엔티티로 변환해 `save()` 하는 패턴이 이 위험의 정면에 서 있습니다. 그래서 실무 원칙은 단순합니다.

- **수정은 `find + modify`**
- **생성은 새 인스턴스 + `persist` 또는 `save`**
- **`merge` 는 구조가 그것을 요구할 때만**

이 시리즈에서 정리한 여덟 편이 맞물리면, JPA가 "마법"처럼 느껴지던 부분들이 **영속성 컨텍스트라는 단일 구조에서 파생된 일관된 동작**으로 읽힙니다. 이후 실무에서 만나는 대부분의 증상은 다음 네 질문 안에서 답이 나옵니다.

1. **엔티티가 지금 어떤 상태인가?** (`new` / `managed` / `detached` / `removed`)
2. **영속성 컨텍스트가 열려 있는가?** (트랜잭션 경계, OSIV, self-invocation)

3. **Fetch와 배치 전략이 이 쿼리에 맞는가?**
4. **필드를 `find + modify` 로 바꾸고 있는가, 아니면 `merge` 로 덮고 있는가?**

이 네 질문을 쓸 수 있게 된 지점이 Spring/JPA를 **튜토리얼이 아닌 구조로** 이해하는 출발점입니다.