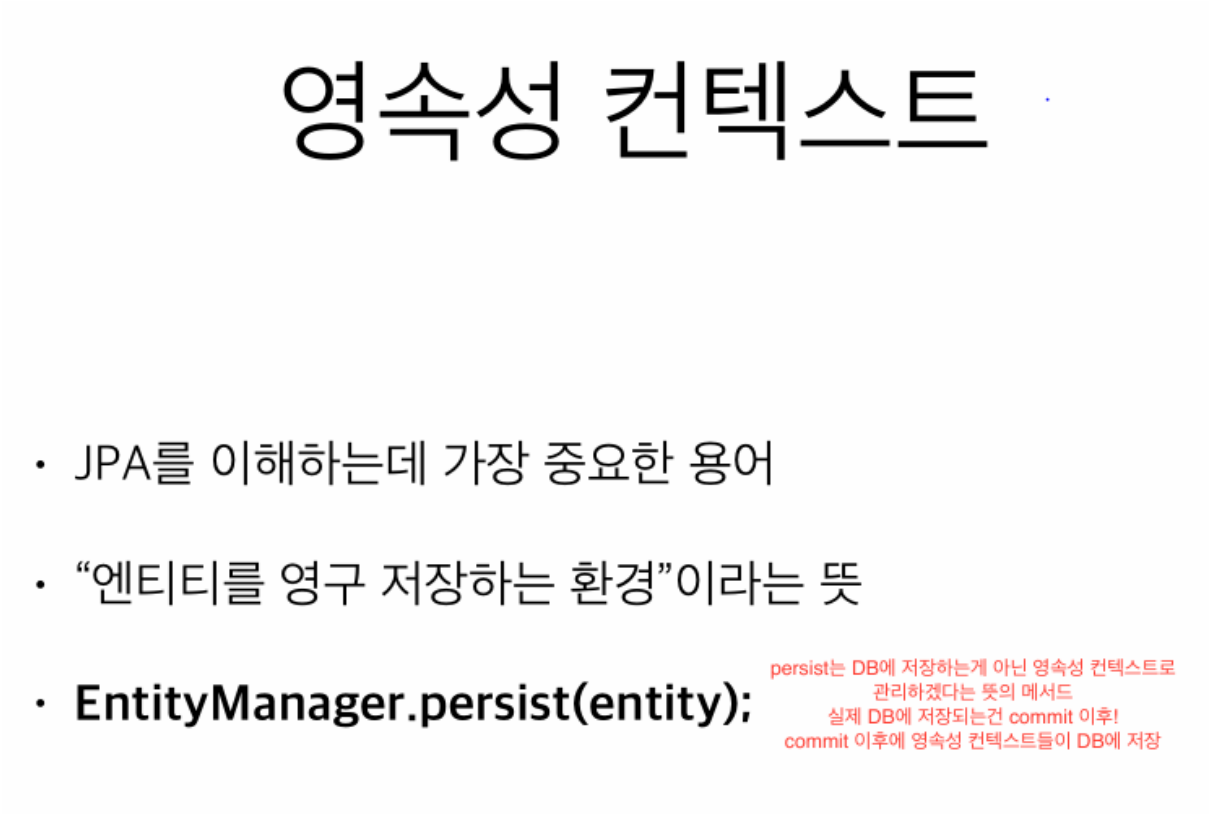
N+1

ORM

- ORM(Object-Relational Mapping) 정의
  - 객체와 관계형 데이터베이스의 데이터를 자동으로 매핑해주는 것을 말한다.
    객체 지향 프로그래밍은 클래스를 사용하고, 관계형 데이터베이스는 테이블을 사용하는데 이때 객체 모델과 관계형 모델 간에 불일치가 존재
  - ORM을 통해 객체 간의 관계를 바탕으로 SQL을 자동으로 생성하여 불일치를 해결해줍니다.
  - 데이터베이스 데이터 <- mapping → Object 필드
    객체를 통해 간접적으로 데이터베이스 데이터를 다룬다.
- ORM의 장점
  - 객체 지향적인 코드로 인해 더 직관적이고, SQL 쿼리가 아닌 직관적인 코드로 데이터를 조작할 수 있어 개발자가 객체 지향 프로그래밍하는데 집중할 수 있도록 도와줍니다. 이에 따라 개발자는 비즈니스 로직에 더 집중할 수 있다.
  - 선언문, 할당, 종료 같은 부수적인 코드가 없거나 급격히 줄어든다.
  - 각종 객체에 대한 코드를 별도록 작성하기 때문에 코드의 가독성을 올려준다
    SQL의 절차적이고 순차적인 접근이 아닌 객체 지향적인 접근으로 인해 생산성이 증가한다.
  - **DB에 종속적이지 않다!**
    - **SQL dialect로 데이터베이스 방언을 설정**할 수 있다.
    - 서로 다른 데이터베이스 문법, 타입 등을 알아서 처리해주므로 **개발자는 데이터베이스를 바꾸더라도 코드를 크게 바꾸지 않게 된다.**

JPA는 ORM을 위한 자바 표준 (API) , Hibernate는 JPA를 구현한 구현체!

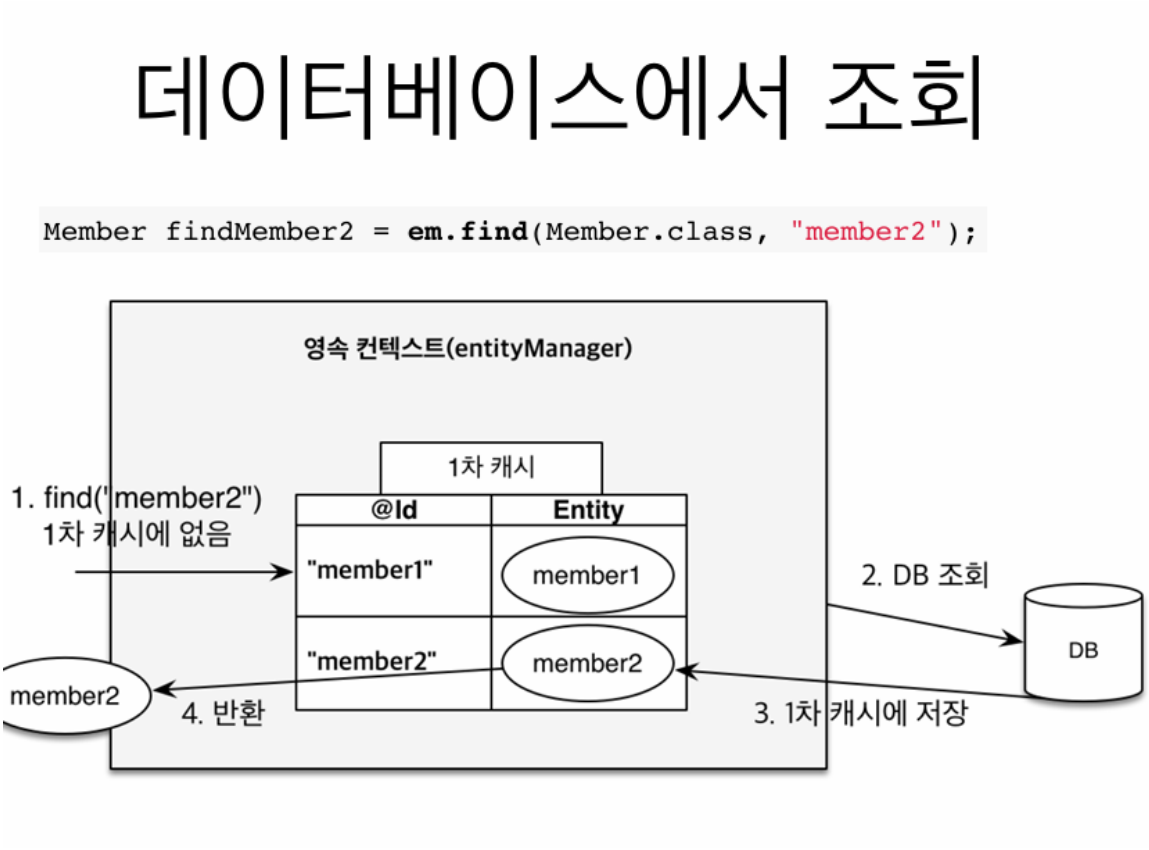
JPA & 영속성 컨텍스트



- 영속성 컨텍스트란 논리적인 개념
- 눈에 보이지 않으며 애플리케이션과 데이터베이스 사이의 버퍼 역할

영속성 컨텍스트의 주요 기능

- 1차 캐시: 영속성 컨텍스트 내에는 1차 캐시가 존재하여, 한 트랜잭션 내에서 엔티티를 반복해서 조회할 때 데이터베이스에 접근하는 것이 아니라, 처음 조회한 엔티티를 1차 캐시에서 가져온다.
- 쓰기 지연 (Transactional write-behind): 영속성 컨텍스트는 트랜잭션이 커밋되기 전까지 데이터베이스에 바로 쓰지 않고, 변경 내용을 내부 쿼리 저장소에 모아두었다가 트랜잭션이 커밋될 때 한꺼번에 데이터베이스에 반영한다.
- 변경 감지 (Dirty Checking): 영속성 컨텍스트는 엔티티의 변경 사항을 감지한다. 트랜잭션을 커밋할 때, 영속성 컨텍스트에 있는 엔티티 중 변경된 엔티티를 찾아 해당 변경 내용을 데이터베이스에 반영한다.
- 지연 로딩 (Lazy Loading): 엔티티의 관계가 설정된 필드를 실제 사용하는 시점까지 로딩을 지연시킨다. 즉, 필요한 시점에만 관련 데이터를 불러오므로 성능을 최적화할 수 있다.



엔티티 영구 저장의 의미

엔티티를 "영구 저장한다"는 것은, 엔티티를 생성하여 영속 상태로 만든 후, 이를 데이터베이스에 저장하여 데이터를 영구적으로 보관하는 과정을 말한다.

- 엔티티 생성 및 관리: 애플리케이션에서 엔티티를 생성하고, 이를 영속성 컨텍스트에 포함시켜 영속 상태로 만든다.
- 영속성 컨텍스트의 기능 이용: 영속 상태의 엔티티는 영속성 컨텍스트의 관리를 받으며, 변경 감지, 쓰기 지연 등의 기능을 통해 데이터베이스와의 상호 작용이 최적화된다.
- 데이터베이스에 저장: 최종적으로 트랜잭션이 커밋될 때, 변경된 엔티티의 상태가 데이터베이스에 반영된다. 이를 통해 엔티티의 상태가 데이터베이스에 영구적으로 저장된다. (JPA에서는 flush, jpql의 실행 등의 과정을 통해 영속성 컨텍스트의 엔티티들을 DB에 저장할 수 있다.)

엔티티의 생명주기

- 비영속 (New/Transient): 엔티티가 생성되었지만 아직 영속성 컨텍스트에 포함되지 않은 상태. (new로 객체를 생성만 한 상태)

- 영속 (Managed): 엔티티가 영속성 컨텍스트에 포함되어 관리되는 상태 이 상태에서 엔티티는 데이터베이스에 저장되기 전이거나 이미 저장된 상태일 수 있다. 영속 상태의 엔티티는 영속성 컨텍스트의 1차 캐시, 쓰기 지연, 변경 감지 등의 기능의 대상이 된다.
- 준영속 (Detached): 영속성 컨텍스트에서 분리된 상태 이 상태의 엔티티는 더 이상 영속성 컨텍스트에 의해 관리되지 않지만, 엔티티의 식별자 값은 유지된다.
- 삭제 (Removed): 엔티티가 영속성 컨텍스트와 데이터베이스에서 삭제된 상태

N+1문제

N+1 문제는 ORM에서 발생하는 문제로, 엔티티와 관련된 연관 관계의 데이터를 가져올 때 발생하는 문제이다. 이 문제는 데이터를 조회할 때, 처음에는 한 번의 쿼리로 엔티티를 조회하지만, 연관된 엔티티를 실제로 접근하는 시점에 각 연관 엔티티마다 **추가적인 쿼리**를 발생시키는 현상을 말한다.

→ 실제 조회되는 데이터는 동일하지만 쿼리가 많이 발생하여 DB 접근이 많아지는 것이 문제

EX)

예를 들어, 게시글(Post)과 댓글(Comment)이 1:N 관계에 있다고 가정합시다. 여기서 게시글 1개와 연관된 댓글을 모두 조회하고 싶을 때, JPA는 먼저 게시글에 대한 쿼리 1번을 실행합니다. 그리고 각 게시글에 연관된 댓글들을 조회하기 위해 추가로 N번의 쿼리가 더 실행됩니다. 만약 10개의 게시글이 각각 10개의 댓글을 가지고 있다면, 최초의 1번 쿼리와 추가적인 100번의 쿼리가 발생하여 총 101번의 쿼리가 실행되는 것입니다.

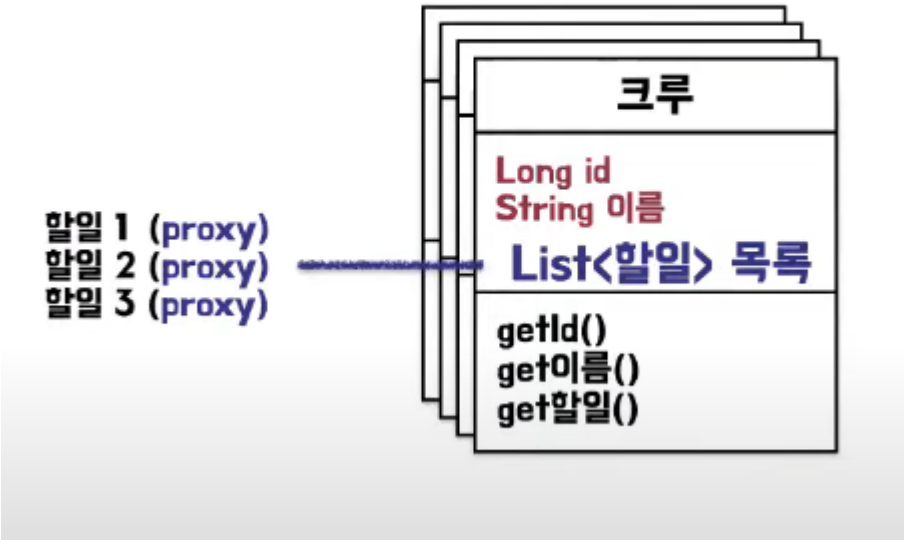
→ 일반적으로 직접 SQL을 작성한다면 게시글과 댓글을 join하여 하나의 쿼리로 데이터를 가져오지만, 위 상황에서는 ORM이 우리가 객체지향적으로 만든 코드를 쿼리로 변환하는 과정에서 101개의 쿼리가 발생

왜 이런 문제가 발생할까..?

JAP는 기본적으로 **지연 로딩**을 사용 → 최초 쿼리에서 게시글 목록을 조회하는 쿼리 1번만으로는 각 게시글에 연관된 댓글 데이터가 로드되지 않음. 실제로 게시글에 포함된 댓글 데이터에 접근하려고 시도할 때, JPA는 그때서야 각 게시글에 대한 댓글 데이터를 로드하기 위해 추가 쿼리를 실행. 따라서, 10개의 게시글이 각각 10개의 댓글을 가지고 있다면, 최초의 1번 쿼리(게시글 조회)와 추가적인 100번의 쿼리(각 게시글에 대한 댓글 조회)가 발생하여 총 101번의 쿼리가 실행된다.

해결방법 1

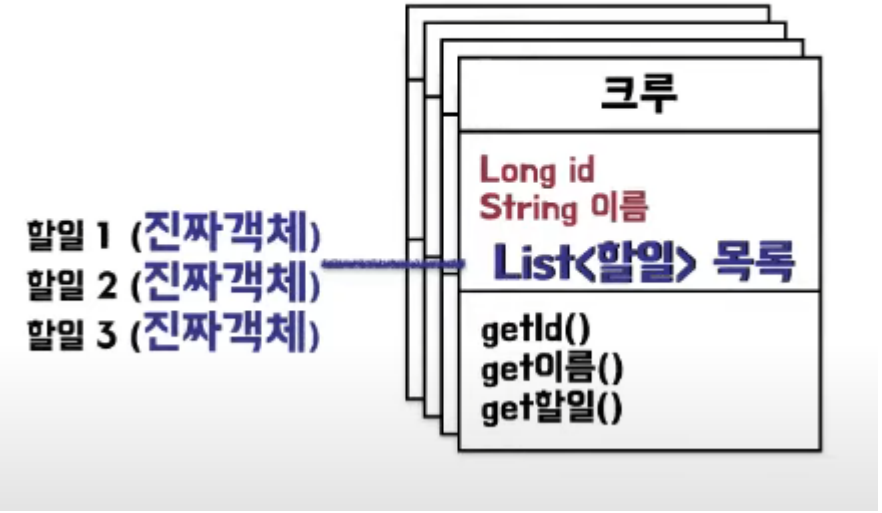
fetch join 사용



최초 조회시 List 객체를 직접 가져오지 않고 위 그림처럼 프록시 객체로 가지고 있다가 후에 리스트에 접근 시, 목록을 조회, 이 때 목록 개수만큼 N개의 쿼리 발생.

→ fetch join을 사용한다면 최초 조회시 프록시 객체가 아닌 실제 객체를 조회함!

이 객체는 1차 캐시에 저장되며 후에 목록 데이터 접근 시, DB에 접근하지 않고 영속성 컨텍스트 내부의 1차 캐시에 접근하여 조회가 가능하다!



그럼 즉시로딩을 사용하면 안될까..?

즉시로딩을 사용해도 N+1 문제가 발생할 수 있다!

```
// User.java
@OneToMany(mappedBy = "user", fetch = FetchType.EAGER)
private Set<Article> articles = emptySet();

// Article.java
@ManyToOne(fetch = FetchType.EAGER)
private User user;
```

즉시로딩을 사용하면 내부적으로 inner join문 하나가 날아가서 User가 조회됨과 동시에 Article까지 즉시로딩된다. `findById`, 즉 `EntityManager`에서 `entityManager.find()`; 같은 경우 jpa가 내부적으로 join문에 대한 쿼리를 만들어서 반환을 하기 때문에 즉시로딩으로는 문제가 없어보이기도 한다.

User findAll

```
== start ==
Hibernate:
    /* select
        generatedAlias0
    from
        User as generatedAlias0 */ select
            user0_.id as id1_1_,
            user0_.name as name2_1_
        from
            users user0_
```

Eager 감지하고 Article 추가 조회

```
Hibernate:
    select
        articles0_.user_id as user_id4_0_0_,
        articles0_.id as id1_0_0_,
        articles0_.id as id1_0_1_,
        articles0_.content as content2_0_1_,
        articles0_.title as title3_0_1_,
        articles0_.user_id as user_id4_0_1_
    from
        articles articles0_
    where
        articles0_.user_id=?
Hibernate:
    select
        articles0_.user_id as user_id4_0_0_,
        articles0_.id as id1_0_0_,
        articles0_.id as id1_0_1_,
        articles0_.content as content2_0_1_,
        articles0_.title as title3_0_1_,
        articles0_.user_id as user_id4_0_1_
    from
        articles articles0_
    where
        articles0_.user_id=?
Hibernate:
    select
        articles0_.user_id as user_id4_0_0_,
        articles0_.id as id1_0_0_,
```

다만 문제는 jpql에 있다. 우리는 findById만 사용하는 것이 아니라 직접 jpql문을 짜서 전달하기도하고, data jpa에서 findBy~의 쿼리메소드 같은 경우에도 data jpa 내부에서 jpql이 만들어져서 나간다.

아니 그럼 jpql은 뭐가 문제일까?🤔

일단 jpql이 sql로 그대로 번역이 된다. 만약 User의 `findAll()`을 요청하는 것이라면 `select u from User u ;`라는 쿼리가 발생하게 되는 것이다 User를 찾는거는 문제가 없었지만 여기서 우리는 "즉시로딩"을 Article column에 걸어두었던 것을 잊으면 안된다.

```
@OneToMany(mappedBy = "user", fetch = FetchType.EAGER)
private Set<Article> articles = emptySet();
```

User를 select 해왔지만 JPA는 Article에 대해서 EAGER가 걸려있는 것을 보고 select한 모든 User에 대해서 article이 있는지를 검색하게 된다.

즉, 모든 User에 대해서 검색하고 싶어서 select 쿼리를 하나 날렸지만**(1)**, 즉시로딩이 걸려있기 때문에 각각의 User가 가진 Article을 모두 검색한다**(N)**라는 N+1 문제가 발생한다.