5-1 : 과제 피드백 및 4주차 내용 토론

들어가기 전에

이번 시간에는...

과제 피드백 & QnA

N+1 문제
페이징처리
객체지향쿼리

과제 피드백

Spring Data JPA를 활용한 코드 리팩토링

토론 키워드

JPA N+1 문제

프록시 객체

프록시 객체

DB 조회를 미루기 위해 조회된 Entity 객체를 상속한 가짜 엔티티
프록시 객체는 실제 객체의 참조(target)을 보관
프록시 객체는 원본 엔티티를 상속 받는다.
(따라서 타입 비교시엔 등호(==)가 아닌 instance of를 사용)

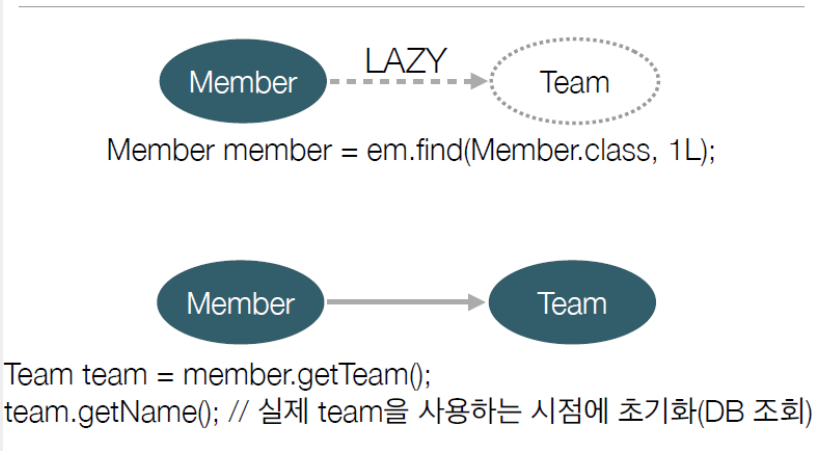
JPA는 곧바로 실제 객체를 조회하는 것이 아닌,
초기화 요청 -> 영속성 컨텍스트 -> DB 조회 -> 실제 Entity 생성 순으로 객체 생성

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

  private String name;

  @OneToMany(mappedBy = "team")
  List<Member> members = new ArrayList<Member>();
```

즉시로딩, 지연로딩

즉시로딩과 지연로딩

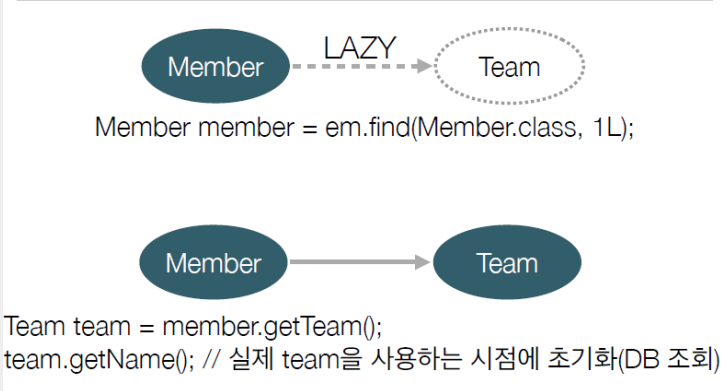
즉시로딩: DB에서 곧바로 실제 데이터를 조회
지연로딩: DB의 실제 데이터를 조회하기 전 프록시 객체로 대체하여 조회



N + 1 문제

N + 1 문제

즉시로딩을 사용할 경우, 관계를 맺고 있는 객체까지도 쿼리 조회가 발생
전체 조회 쿼리(1개) + 각 객체에 포함된 관계 객체 조회 쿼리(N개) = 총 N + 1개 쿼리 발생



ex) 전체 회원(Members) 조회

전체 회원 조회 쿼리(1개)
+
각 회원 당 Team 쿼리 조회(N개)
=
N + 1개 쿼리 발생

N + 1 문제

N + 1 문제 해결 방법

지연로딩을 활용

각 객체에 포함된 관계 객체는 실제 데이터가 아닌 프록시 객체로 대체하여 조회하는 지연 로딩을 활용
지연로딩 설정은 관계 매핑 어노테이션의 fetch 속성에 LAZY로 설정

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

  @Column(name = "USERNAME")
  private String name;

  @ManyToOne(fetch = FetchType.LAZY) //**
  @JoinColumn(name = "TEAM_ID")
  private Team team;
  ..
}
```

N + 1 문제

N + 1 문제 해결 방법

페치 조인을 활용

각 객체의 실제 데이터가 필요한 경우
객체 테이블 관계에 맞게 하나의 SQL로 조회가 가능한 페치 조인을 활용
해당 문법은 차후 한번 더 깊게 설명

```
String jpql = "select m from Member m join fetch m.team";
List<Member> members = em.createQuery(jpql, Member.class)
                         .getResultList();

for (Member member : members) {
  //페치 조인으로 회원과 팀을 함께 조회해서 지연 로딩X
  System.out.println("username = " + member.getUsername() + ", " +
                     "teamName = " + member.getTeam().name());
}
```

N + 1 문제

N + 1 문제 해결 방법

BatchSize 설정 활용

각 객체의 실제 데이터가 필요한 경우 + 여러 관계를 페치 조인 해야 하는 경우
페치 조인은 하나의 컬렉션만 가능하다
따라서 나머지 컬렉션을 조회할 때는 BatchSize 설정을 활용(단일 설정, 글로벌 설정 모두 가능)

BatchSize 설정 예시

```java
@Entity
class Member{
    @Id @GeneratedValue
    private Long id;

    @org.hibernate.annotations.BatchSize(size = 5)
    @OneToMany(mappedBy = "member", fetch = FetchType.EAGER)
    private List<Order> orders = new ArrayList<>();
}
```

SQL문 예시

```
// BatchSize 미설정 시
SELECT * FROM
ORDER WHERE MEMBER_ID = ?
```

한 번에 조건 1개 수행

X 5회

```
// BatchSize 설정 시
SELECT * FROM
ORDER WHERE MEMBER_ID IN (?, ?, ?, ?, ?)
```

한 번에 여러 개의 조건을 수행

X 1회

영속성 전이

영속성 전이

특정 엔티티를 영속 상태로 만들 때 연관된 엔티티도 함께 영속 상태로 만들고 싶을 때 사용
즉, **함께 저장하고 관리**하고 싶을 때 사용.
관계 매핑 어노테이션의 **CASCADE 속성**을 통해 설정

종류: **ALL(모두 적용), PERSIST(영속)**, REMOVE(삭제), MERGE(병합), REFRESH, DETACH

부모가 저장&수정될 때, 관련된 자식들도 함께 관리되길 원한다면,

```java
@Entity
public class Parent {

  @OneToMany(mappedBy = "parent", cascade = CascadeType.ALL)
  private List<Child> childList = new ArrayList<>();

}
```

```java
@Entity
public class Child {

  @ManyToOne
  @JoinColumn(name = "parent_id")
  private Parent parent;

}
```

토론 키워드

페이징 처리

페이징 처리

페이징 처리

페이지에 맞게 일정한 데이터 개수만큼 데이터를 내려주는 방식
일반적으로 페이징 처리는 SQL의 OFFSET과 LIMIT 조건을 활용하여 구현할 수 있다.

```
SELECT * FROM Review
LIMIT 5
OFFSET 0
```

- LIMIT: 몇 개의 데이터를 끊어서 가져올 것인지
- OFFSET: 어느 지점부터 가져올 것인지
OFFSET과 LIMIT의 순서가 바뀌면 안된다.

페이징 처리

페이징 처리

ex) 한 페이지에 5개의 리뷰가 보여져야 하는 경우

```
SELECT * FROM Review
LIMIT 5
OFFSET (pageNum - 1) * 5
```

위의 예시에서 (pageNum – 1) * 5 부분을 쿼리 파라미터로 하여 계산한 변수를 넣어주면 된다.

페이징 처리

Spring Data JPA 페이징 처리

Spring Data JPA에서는 인터페이스를 통해 간단하게 페이징 처리를 할 수 있도록 지원한다.

페이징과 정렬 파라미터 관련

org.springframework.data.domain.Sort: 정렬 기능
org.springframework.data.domain.Pageable: 페이징 기능(내부에 Sort 포함)

특별한 반환 타입

org.springframework.data.domain.Page: 추가 count 쿼리 결과 포함 페이징
org.springframework.data.domain.Slice: 추가 count 쿼리 없이 다음 페이지만 확인 가능한 페이징

페이징 처리

Spring Data JPA 페이징 처리

JPA Interface에서 메소드를 정의할 때,
다음 2가지만 추가로 해주면 간단하게 페이징 처리를 제공받을 수 있다.

1. 페이징 처리 반환 타입으로 반환
2. 인자에 Pageable 값 추가

```
Page<Member> findByUsername(String name, Pageable pageable); //count 쿼리 사용
Slice<Member> findByUsername(String name, Pageable pageable);
```

Page: 전체 데이터 개수를 알고 싶을 때 (count 쿼리가 추가로 발생)
Slice: 특정 단위 개수 + 1개로 끊어서 받고 싶을 때 (count 쿼리가 추가로 발생하지 않음)

1,2,3,4 … 페이지 목록을 누르며 페이징을 해야하는 화면은 Page
그 외 무한 스크롤 등의 화면은 Slice를 활용

페이징 처리

Spring Data JPA 페이징 처리

Page 사용 시 count 쿼리가 추가로 나가면서 성능 문제가 발생할 수 있다.
이를 보완하고자 count 쿼리를 따로 정의하여 활용할 수도 있다.

```
// @Query 활용
// count 쿼리를 따로 정의하여 활용할 수 있다.
@Query(value = “select m from Member m”,
        countQuery = “select count(m.username) from Member m”)
Page<Member> findMemberAllCountBy(Pageable pageable);
```

복잡한 쿼리를 JOIN 등을 통해 조회할 때
count 쿼리는 JOIN 할 필요가 없는 경우가 많다.
이러한 경우 count 쿼리를 분리하여 성능 이슈를 줄일 수 있다.

페이징 처리

Spring Data JPA 페이징 처리

페이징 처리 사용 예제

```
// 페이징 처리
int age = 10;
PageRequest pageRequest = PageRequest.of(시작 지점, 페이지 번호, Sort.by(Sort.Direction.DESC, "username"));
Page<Member> page = memberRepository.findByAge(age, pageRequest);
Page<MemberDto> dtoPage = page.map(m -> new MemberDto(생성자 매개변수));

// Page method 예제
List<Member> content = page.getContent(); //조회된 데이터
content.size(); //조회된 데이터 수
page.getTotalElements(); //전체 데이터 수
page.getNumber(); //페이지 번호
page.getTotalPages(); //전체 페이지 번호
page.isFirst(); //첫번째 항목인가?
page.hasNext(); //다음 페이지가 있는가?
```

PageRequest 객체를 만들어 (시작 위치, 페이지 번호, 정렬 조건)을 넣어준다.
시작 위치는 0부터 시작하며, 정렬 조건은 생략할 수 있다.
Page 처리 시 반드시 Entity를 DTO로 변경해서 처리해주자

페이징 처리

Page 인터페이스 제공 메소드

Page 인터페이스가 제공하는 메소드는 다음과 같다
해당 메소드에 Slice의 메소드를 포함한다.

Page 인터페이스

```
public interface Page<T> extends Slice<T> {
  int getTotalPages(); //전체 페이지 수
  long getTotalElements(); //전체 데이터 수
  <U> Page<U> map(Function<? super T, ? extends U> converter); //변환기
}
```

페이징 처리

Slice 인터페이스 제공 메소드

Slice 인터페이스가 제공하는 메소드는 다음과 같다

Slice 인터페이스

```java
public interface Slice<T> extends Streamable<T> {
  int getNumber(); //현재 페이지
  int getSize(); //페이지 크기
  int getNumberOfElements(); //현재 페이지에 나올 데이터 수
  List<T> getContent(); //조회된 데이터
  boolean hasContent(); //조회된 데이터 존재 여부
  Sort getSort(); //정렬 정보
  boolean isFirst(); //현재 페이지가 첫 페이지 인지 여부
  boolean isLast(); //현재 페이지가 마지막 페이지 인지 여부
  boolean hasNext(); //다음 페이지 여부
  boolean hasPrevious(); //이전 페이지 여부
  Pageable getPageable(); //페이지 요청 정보
  Pageable nextPageable(); //다음 페이지 객체
  Pageable previousPageable();//이전 페이지 객체
  <U> Slice<U> map(Function<? super T, ? extends U> converter); //변환기
}
```

페이징 처리

추가 내용 – 커서 기반 페이징 처리

기존에 사용하는 페이징 처리는 오프셋 기반 페이징 처리로 다음과 같은 단점이 있다.

오프셋 기반 페이징 처리 단점

1. 페이지를 조회 중, 새로운 데이터가 추가 되면 다음 페이지에서 중복 데이터가 조회
2. 대부분 RDBMS에서 OFFSET 쿼리의 성능 이슈

커서 기반 페이징 처리를 활용하면 위의 단점을 해결할 수 있다.

오프셋 기반 페이징 처리는 n 개의 row를 skip 후에 데이터를 내려주는 방식

커서 기반 페이징 처리는 특정 커서 지점부터 m개를 요청하는 방식

커서 기반 페이징 처리는 인덱스가 맞춰 있어야 하며, 복잡한 상황에서는 사용하기 어렵다.

따라서, 기본적으로 오프셋을 활용하고 꼭 필요한 곳에 사용

토론 키워드

객체지향쿼리

객체지향쿼리

JPQL

모든 DB 데이터를 객체로 변환해서 검색하는 것은 불가능
세부적인 조회까지 원한다면 결국 SQL과 유사한 문법이 필요

JPA는 SQL을 추상화한 JPQL이라는 객체 지향 쿼리 언어 제공
SQL과 문법 유사, SELECT, FROM, WHERE, GROUP BY, HAVING, JOIN 지원
SQL은 DB 테이블을 대상으로한 쿼리이지만, JPQL은 엔티티 객체를 대상으로 쿼리
JPQL을 한마디로 정의하면 **객체 지향 SQL**

```
//검색
String jpql = "select m From Member m where m.name like ‘%hello%'";

List<Member> result = em.createQuery(jpql, Member.class)
        .getResultList();
```

객체지향쿼리

QueryDSL

String 문자 형식이 아닌 **자바 코드로 JPQL을 작성**할 수 있게 해주는 라이브러리
자바 코드로 작성하기 때문에 컴파일 시점에 문법 오류를 찾을 수 있다

단순하고 사용하기 쉽다
주로 동적 쿼리를 작성할 때 사용한다.

동적 쿼리

특정 상황이나 조건들에 따라 쿼리가 변경되는 쿼리

여행지, 체크인&체크아웃, 여행자 등은 입력할 수도 있고, 안할 수도 있다.
입력 여부에 따라 쿼리가 동적으로 달라지게 된다.

객체지향쿼리

JDBC API 직접 사용

JPA를 사용하면서 JDBC 커넥션 직접 이용,
Spring JDBC Template, MyBatis 등도 함께 사용 가능
단, 영속성 컨텍스트를 적절한 시점에 강제로 플러시 하는게 필요

JPQL 문법

JPQL 기본 문법

사용 예제

```
em.createQuery("select m from Member as m where m.age > 18", Member.class);
```

특징

- 엔티티와 속성은 대소문자를 구분하고 그 외 JPQL 키워드는 구분하지 않음
  - 객체는 엔티티 이름을 사용(테이블 이름X)
  - 별칭은 필수(as는 생략 가능)
- 집합(COUNT, SUM 등), 정렬(ORDER BY 등) 함수 모두 사용 가능

JPQL 문법

JPQL 기본 문법 – 반환 타입

반환 타입은 TypeQuery, Query 데이터 타입으로 제공

TypeQuery: 반환 타입이 명확할 때 사용

```
TypedQuery<Member> query =
            em.createQuery("SELECT m FROM Member m", Member.class);
```

Query: 반환 타입이 명확하지 않을 때 사용

```
Query query =
            em.createQuery("SELECT m.username, m.age from Member m");
```

JPQL 문법

JPQL 기본 문법- 결과 조회 메소드

결과 조회 메소드에는 getResultList(), getSingleResult()가 있다.

getResultList: 결과가 하나 이상일 때, 리스트 반환

결과가 없으면, 빈 리스트를 반환

```
List<Member> query = em.createQuery("SELECT m FROM Member m", Member.class)
    .getResultList();
```

getSingleResult: 결과가 정확히 하나일 때, 단일 객체 반환

결과가 없으면 에러를 반환

```
Member query = em.createQuery("SELECT m FROM Member m", Member.class)
    .getSingleResult();
```

JPQL 문법

JPQL 기본 문법 – 파라미터 바인딩

파라미터 바인딩은 이름 기준과 위치 기준으로 작성할 수 있다.
위치 기준은 수정 시 값이 변동될 수 있기 때문에 이름 기준을 사용할 것

이름 기준

```
Member query = em.createQuery("SELECT m FROM Member m where m.username=:username", Member.class)
    .setParameter("username", usernameParam);
    .getSingleResult();
```

위치 기준

```
Member query = em.createQuery("SELECT m FROM Member m where m.username=?1", Member.class)
    .setParameter(1, usernameParam);
    .getSingleResult();
```

JPQL 문법

JPQL 기본 문법 – 조인

JPQL은 조인 기능을 제공한다.

```
// 내부 조인
SELECT m FROM Member m [INNER] JOIN m.team t
// 외부 조인:
SELECT m FROM Member m LEFT [OUTER] JOIN m.team t
// 세타 조인(전혀 관련없는 테이블끼리 조인)
select count(m) from Member m, Team t where m.username = t.name
```

JPQL 문법

JPQL 기본 문법 – 페치 조인

JPQL은 페치 조인 기능을 제공한다.
연관된 엔티티나 컬렉션을 SQL 한 번에 함께 조회하는 기능
JPQL에서 성능 최적화를 위해 제공하는 기능으로 **N + 1 문제 해결 방법**으로 활용
객체 그래프에 맞게 하나의 SQL로 조회하는 개념

```
String jpql = "select m from Member m join fetch m.team";
List<Member> members = em.createQuery(jpql, Member.class)
                         .getResultList();

for (Member member : members) {
  //페치 조인으로 회원과 팀을 함께 조회해서 지연 로딩X
  System.out.println("username = " + member.getUsername() + ", " +
                     "teamName = " + member.getTeam().name());
}
```

JPQL 문법

JPQL 기본 문법 – 서브쿼리

JPQL은 서브쿼리 기능을 제공한다.

```
// 나이가 평균보다 많은 회원
select m from Member m
where m.age > (select avg(m2.age) from Member m2)

// 한 건이라도 주문한 고객
select m from Member m
where (select count(o) from Order o where m = o.member) > 0
```

Hello World!