JPA 쿼리

JPA는 다양한 쿼리 방법을 지원

- JPQL
- JPA Criteria
- QueryDSL
- Native SQL
- JDBC API 사용 (MyBatis, SpringJdbcTemplate) 함께 사용

- JPQL
  - JPA를 사용하면 엔티티 객체를 중심으로 개발
  - 검색을 할 때도 **테이블이 아닌 엔티티 객체를 대상으로 검색**
  - 모든 DB 데이터를 객체로 변환해서 검색하는 것은 불가능
  - 애플리케이션이 필요한 데이터만 DB에서 불러오려면 결국 검색 조건이 포함된 SQL이 필요
  - JPA는 SQL을 추상화한 JPQL이라는 객체 지향 쿼리 언어 제공
  - SQL과 문법 유사, SELECT, FROM, WHERE, GROUP BY, HAVING, JOIN 지원
  - JPQL은 엔티티 객체를 대상으로 쿼리
  - SQL은 데이터베이스 테이블을 대상으로 쿼리
  - SQL을 추상화해서 특정 데이터베이스 SQL에 의존X

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

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

  - 단점
    - 동적 쿼리 만들기가 어렵다.
    - 복잡한 쿼리 작성하기 어렵다.

```
//검색
String jpql = "select m from Member m where m.age > 18";

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

```
실행된 SQL
      select
          m.id as id,
          m.age as age,
          m.USERNAME as USERNAME,
          m.TEAM_ID as TEAM_ID
      from
          Member m
      where
          m.age>18
```

- Criteria

```
//Criteria 사용 준비
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Member> query = cb.createQuery(Member.class);

//루트 클래스 (조회를 시작할 클래스)
Root<Member> m = query.from(Member.class);

//쿼리 생성 CriteriaQuery<Member> cq =
query.select(m).where(cb.equal(m.get("username"), "kim"));
List<Member> resultList = em.createQuery(cq).getResultList();
```

  - 문자가 아닌 자바코드로 JPQL을 작성할 수 있음
  - JPQL 빌더 역할
  - JPA 공식 기능
  - 단점
    - 너무 복잡하고 실용성이 없다

- QueryDSL
  - 문자가 아닌 자바코드로 JPQL을 작성할 수 있음
  - JPQL 빌더 역할
  - 컴파일 시점에 문법 오류를 찾을 수 있음
  - 동적쿼리 작성 편리함
  - **단순하고 쉬움**

- **실무 사용 권장**

```
//JPQL
//select m from Member m where m.age > 18
JPAFactoryQuery query = new JPAQueryFactory(em);
QMember m = QMember.member;

List<Member> list =
    query.selectFrom(m)
         .where(m.age.gt(18))
         .orderBy(m.name.desc())
         .fetch();
```

- 네이티브 SQL
  - JPA가 제공하는 SQL을 직접 사용하는 기능
  - JPQL로 해결할 수 없는 특정 데이터베이스에 의존적인 기능

```
String sql =
    "SELECT ID, AGE, TEAM_ID, NAME FROM MEMBER WHERE NAME = 'kim'";

List<Member> resultList =
            em.createNativeQuery(sql, Member.class).getResultList();
```

- JDBC 직접 사용
  - JPA를 사용하면서 JDBC 커넥션을 직접 사용하거나,

    스프링 JdbcTemplate, 마이바티스등을 함께 사용 가능
  - 단 영속성 컨텍스트를 적절한 시점에 **강제로 플러시 필요**

    예) JPA를 우회해서 SQL을 실행하기 직전에 영속성 컨텍스트 수동 플러시
  - **수동 flush 필수!!!!!**

- JPQL 문법
  - select m from **Member** as m where **m.age** > 18
  - 엔티티와 속성은 대소문자 구분O (Member, age)
  - JPQL 키워드는 대소문자 구분X (SELECT, FROM, where)

    - 엔티티 이름 사용, 테이블 이름이 아님(Member)
    - **별칭은 필수****(m)** (as는 생략가능)
    - GROUP BY, HAVING
    - ORDER BY
- TypeQuery: 반환 타입이 명확할 때 사용
- Query: 반환 타입이 명확하지 않을 때 사용
- query.getResultList(): **결과가 하나 이상일 때**, 리스트 반환
    - 결과가 없으면 빈 리스트 반환
- query.getSingleResult(): **결과가 정확히 하나**, 단일 객체 반환
    - 결과가 없으면: javax.persistence.NoResultException
    - 둘 이상이면: javax.persistence.NonUniqueResultException

- Spring JPA 에서는 null 나 optional로 반환한다.

```
SELECT m FROM Member m where m.username=:username

query.setParameter("username", usernameParam);
```

```
SELECT m FROM Member m where m.username=?1

query.setParameter(1, usernameParam);
```

-

- Projection
    - 프로젝션 대상: 엔티티, 임베디드 타입, 스칼라 타입(숫자, 문자등 기본 데이터 타입)
    - join은 명시해야한다.

- Paging
    - JPA는 페이징을 다음 두 API로 추상화
    - **setFirstResult**(int startPosition) : 조회 시작 위치 (0부터 시작)
    - **setMaxResults**(int maxResult) : 조회할 데이터 수

```
//페이징 쿼리
String jpql = "select m from Member m order by m.name desc";
List<Member> resultList = em.createQuery(jpql, Member.class)
      .setFirstResult(10)
      .setMaxResults(20)
      .getResultList();
```

- **JOIN**
  - Inner join
  - Outer join
  - Theta join: 연관관계 없는 테이블 조인

- 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:
  select count(m) from Member m, Team t where m.username = t.name

  JPQL:
  SELECT m, t FROM Member m LEFT JOIN m.team t **on** t.name = 'A'

  SQL:
  SELECT m.*, t.* FROM
  Member m LEFT JOIN Team t **ON** m.TEAM_ID=t.id and t.name='A'

- **Subquery**
  - 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

- [NOT] EXISTS (subquery): 서브쿼리에 결과가 존재하면 참
  - {ALL | ANY | SOME} (subquery)
  - ALL 모두 만족하면 참
  - ANY, SOME: 같은 의미, 조건을 하나라도 만족하면 참
- [NOT] IN (subquery): 서브쿼리의 결과 중 하나라도 같은 것이 있으면 참
- 팀A 소속인 회원
  select m from Member m
  where **exists** (select t from m.team t where t.name = '팀A')

- 전체 상품 각각의 재고보다 주문량이 많은 주문들
  select o from Order o
  where o.orderAmount > **ALL** (select p.stockAmount from Product p)

- 어떤 팀이든 팀에 소속된 회원
  select m from Member m
  where m.team = **ANY** (select t from Team t)

  - JPA는 WHERE, HAVING 절에서만 서브 쿼리 사용 가능
  - SELECT 절도 가능(하이버네이트에서 지원)
  - FROM 절의 서브 쿼리는 현재 JPQL에서 불가능
    - 조인으로 풀 수 있으면 풀어서 해결

- **묵시적인 내부 조인을 지양하라**
  - 단일 검색은 연관 경로 . 으로 탐색가능하다
  - 컬렉션 값은 연관 경로 . 으로 탐색할 수 없다.
- 조인은 SQL튜닝에 중요

- **Fetch Join**
  - SQL 조인 종류X
  - JPQL에서 **성능 최적화**를 위해 제공하는 기능
  - 연관된 엔티티나 컬렉션을 **SQL 한 번에 함께 조회**하는 기능
  - join fetch 명령어 사용
  - 페치 조인 ::= [LEFT [OUTER] | INNER] JOIN FETCH 조인경로

- 회원을 조회하면서 연관된 팀도 함께 조회(SQL 한 번에)
- SQL을 보면 회원 뿐만 아니라 **팀***(*T.*)도 함께 **SELECT**

  - **[JPQL]**
    select m from Member m **join fetch** m.team

  - **[SQL]**
    SELECT M.*, **T.*** FROM MEMBER M
    INNER JOIN TEAM T ON M.TEAM_ID=T.ID

[MEMBER]

ID (PK)	NAME	TEAM_ID (FK)
1	회원1	1
2	회원2	1
3	회원3	2
4	회원4	null

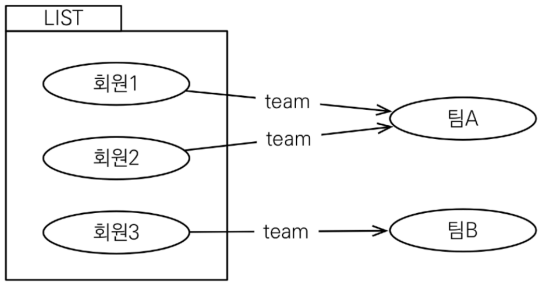
[TEAM]

ID (PK)	NAME
1	팀A
2	팀B
3	팀C

FK -> PK
(MEMBER.TEAM_ID -> TEAM.ID)

[MEMBER JOIN TEAM]

ID (PK)	NAME	TEAM_ID (FK)	ID (PK)	NAME
1	회원1	1	1	팀A
2	회원2	1	1	팀A
3	회원3	2	2	팀B



fetch join

-

페치 조인 사용 코드

```
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());
}
```

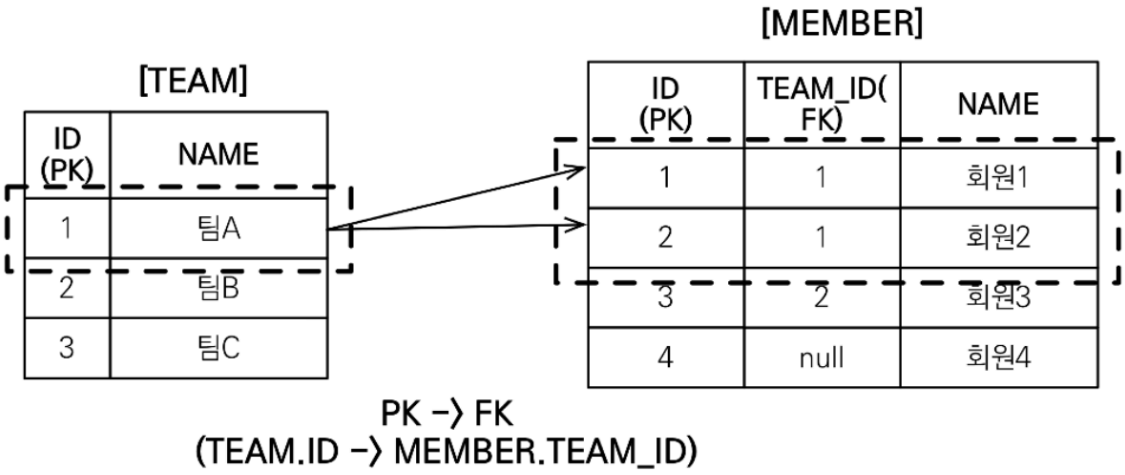
username = 회원1, teamname = 팀A
username = 회원2, teamname = 팀A
username = 회원3, teamname = 팀B

컬렉션 페치 조인

- 일대다 관계, 컬렉션 페치 조인

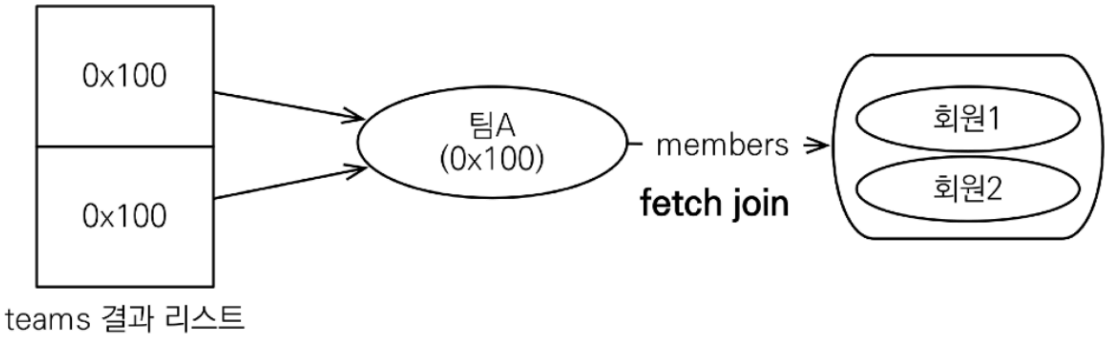
- **[JPQL]**
  select t
  from Team t **join fetch t.members**
  where t.name = ‘팀A'

- **[SQL]**
  SELECT T.*, **M.***
  FROM TEAM T
  INNER JOIN MEMBER M ON T.ID=M.TEAM_ID
  WHERE T.NAME = '팀A'



[TEAM JOIN MEBER]

ID (PK)	NAME	ID (PK)	TEAM_ID (FK)	NAME
1	팀A	1	1	회원1
1	팀A	2	1	회원2



컬렉션 페치 조인 사용 코드

```
String jpql = "select t from Team t join fetch t.members where t.name = '팀A'"
List<Team> teams = em.createQuery(jpql, Team.class).getResultList();

for(Team team : teams) {
    System.out.println("teamname = " + team.getName() + ", team = " + team);
    for (Member member : team.getMembers()) {
      //페치 조인으로 팀과 회원을 함께 조회해서 지연 로딩 발생 안함
       System.out.println("-> username = " + member.getUsername()+ ", member = " + member);
  }
}
```

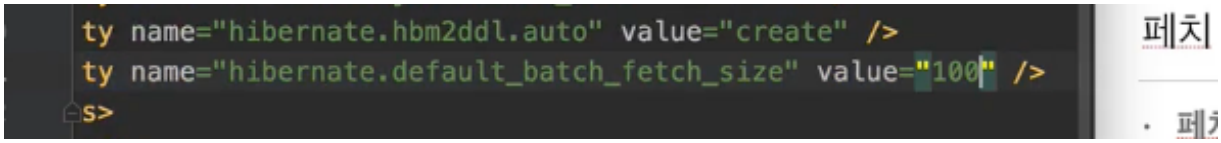
```
teamname = 팀A, team = Team@0x100
-> username = 회원1, member = Member@0x200
-> username = 회원2, member = Member@0x300
teamname = 팀A, team = Team@0x100
-> username = 회원1, member = Member@0x200
-> username = 회원2, member = Member@0x300
```

- SQL의 DISTINCT는 중복된 결과를 제거하는 명령
- JPQL의 DISTINCT 2가지 기능 제공
  - 1)SQL에 DISTINCT를 추가
  - 2)애플리케이션에서 엔티티 중복 제거

Fetch join vs join 차이

- 일반 조인 실행시 연관된 엔티티를 함께 조회하지 않음
- JPQL은 결과를 반환할 때 연관관계 고려X
- 단지 SELECT 절에 지정한 엔티티만 조회할 뿐
- 여기서는 팀 엔티티만 조회하고, 회원 엔티티는 조회X
- 페치 조인을 사용할 때만 연관된 엔티티도 함께 조회(****즉시 로딩)
- **페치 조인은 객체 그래프를 SQL 한번에 조회하는 개념**

Fetch Join 한계 및 특징

- **페치 조인 대상에는 별칭을 줄 수 없다.**
- **둘 이상의 컬렉션은 페치 조인 할 수 없다.**
- 컬렉션을 페치 조인하면 페이징 API(setFirstResult, setMaxResults)를 사용할 수 없다.
- 연관된 엔티티들을 SQL 한 번으로 조회 - 성능 최적화
  - N + 1 ➜ @BatchSize 를 사용하는 방법이 있다.
  - 

- 엔티티에 직접 적용하는 글로벌 로딩 전략보다 우선함
  - @OneToMany(fetch = FetchType.LAZY) //글로벌 로딩 전략
- 실무에서 글로벌 로딩 전략은 모두 지연 로딩
- 최적화가 필요한 곳은 페치 조인 적용 (N:1 문제)

- 모든 것을 페치 조인으로 해결할 수 는 없음
- 페치 조인은 객체 그래프를 유지할 때 사용하면 효과적
- 여러 테이블을 조인해서 엔티티가 가진 모양이 아닌 전혀 다른 결과를 내야 하면, 페치 조인 보다는 일반 조인을 사용하고 필요 한 데이터들만 조회해서 DTO로 반환하는 것이 효과적
  - Fetch join 으로 Entity 를 가져와 사용
  - Fetch join 의 Entity 를 Application DTO 변환 사용
  - 조회 결과를 DTO로 받아 사용

벌크 연산

- 쿼리 한 번으로 여러 테이블 로우 변경(엔티티)
- 벌크 연산은 영속성 컨텍스트를 무시하고 데이터베이스에 직접 쿼리
  - 벌크 연산을 먼저 실행
  - **벌크 연산 수행 후 영속성 컨텍스트 초기화**

#dev/jpa/tutorial