10. 객체지향 쿼리 언어

...

Blur

객체지향 쿼리

- 식별자통한 조회: EntityManager.find() - 엔티티 조회
- 객체 그래프 탐색: a.getB().getC() -> 연관된 엔티티 조회

위의 방법은 메모리에 등재시켜 조회

ORM을 사용해서 검색도 엔티티 객체를 대상으로 하는 방법 필요

⇒ JPQL

객체지향 쿼리

1. **JPQL(Java Persistence Query Language)** - 객체지향 SQL
2. Criteria - builder API
3. QueryDSL - builder API
4. Native SQL

JPQL

- SQL과 비슷한 문법, 표준 SQL이 제공하는 기능 유사하게 지원
- SQL을 추상화시켜 특정 데이터베이스에 의존하지 않음

```
public static void userJPQL() {
    String jpql = "select m from Member as m where m.username = 'kim'";
    List<Member> resultList = em.createQuery(jpql, Member.class).getResultList();
}
```

- 테이블 대상이 아닌 엔티티 대상 쿼리

JPQL

- 기본 문법

  select문 :: = select절 from절 where절 groupby절 having절 orderby절

  update문 :: = update절 [where절]

  delete문 :: = delete절 [where절]
- 대소문자 구분, JPQL키워드만 대소문자 구분X
- class명이 아닌 Entity명 사용
- 별칭(as ~~)는 필수로 지정 - as는 생략가능(Membrer as m == Member m)

JPQL

- Query result type에 따라 객체 사용

```
public static void useTypeQuery() {
    TypedQuery<Member> query = em.createQuery( s: "SELECT m FROM Member m", Member.class);

    List<Member> resultList = query.getResultList();
    for(Member member : resultList) {
        System.out.println("member = " + member);
    }
}
public static void useQuery() {
    Query query = em.createQuery( s: "SELECT m.usename, m.age from Member m");
    List resultList = query.getResultList();

    for (Object o : resultList) {
        Object[] result = (Object[]) o;
        System.out.println("username = " + result[0]);
        System.out.println("age = " + result[1]);
    }
}
```

JPQL

- 결과 조회

  query.getResultList() = 결과를 예제로 반환, 없을시 빈 컬렉션

  query.getSingleResult() = 결과가 정확히 하나일때 사용(1개 아닐시 예외발생)

JPQL

- 파라미터 바인딩

```
public static void bindingParameter() {
    String usernameParam = "User1";

    TypedQuery<Member> query = em.createQuery( s: "SELECT m FROM Member m where m.username = :username", Member.class);

    query.setParameter( s: "username", usernameParam);
    List<Member> resultList = query.getResultList();

    //chaining
    List<Member> members = em.createQuery( s: "SELECT m FROM Member m where m.usename = :username", Member.class)
            .setParameter( s: "username", usernameParam)
            .getResultList();

    //position base
    List<Member> members1 = em.createQuery( s: "SELECT m FROM Member m where m.username = ?1", Member.class)
            .setParameter( i: 1, usernameParam)
            .getResultList();
}
```

Criteria 쿼리

- 문자가 아닌 프로그래밍 코드로 JPQL을 작성
  1. 컴파일 시점에 오류 발견
  2. IDE를 통해 코드 자동완성 지원
  3. 동적 쿼리 작성 편리

```
public static void useCriteria() {
    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<Member> query = cb.createQuery(Member.class);

    Root<Member> m = query.from(Member.class);

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

⇒ 메타 모델 사용해서 필드명도 코드로 작성 가능

Criteria 쿼리

```
public static void startCriteriaQuery() {
    //JPQL: select m from Member m

    CriteriaBuilder cb = em.getCriteriaBuilder();

    CriteriaQuery<Member> cq = cb.createQuery(Member.class);

    Root<Member> m = cq.from(Member.class);
    cq.select(m);

    TypedQuery<Member> query = em.createQuery(cq);
    List<Member> members = query.getResultList();
}
```

Criteria 코드가 한눈에 들어오지 않음

```
public static void searchCriteriaQuery() {
    // select m from Member m
    // where m.username='회원1'
    // order by m.age desc

    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<Member> cq = cb.createQuery(Member.class);
    Root<Member> m = cq.from(Member.class);

    Predicate usernameEqual = cb.equal(m.get("username"), o: "회원1");

    javax.persistence.criteria.Order ageDesc = cb.desc(m.get("age"));

    cq.select(m)
            .where(usernameEqual)
            .orderBy(ageDesc);

    List<Member> resultList = em.createQuery(cq).getResultList();
}
```

QueryDSL

- 오픈소스 프로젝트
- QueryDSL 쿼리 전용 클래스 생성해서 활용(plugin)

```
@Generated("com.mysema.query.codegen.EntitySerializer")
public class QMember extends EntityPathBase<Member> {

    private static final long serialVersionUID = -1627262689L;

    public static final QMember member = new QMember( variable: "member1");

    public final NumberPath<Integer> age = createNumber( property: "age", Integer.class);

    public final StringPath username = createString( property: "username");
```

QueryDSL

```java
public void queryDSL() {
    JPAQuery query = new JPAQuery(em);
    QMember qMember = new QMember( variable: "m");
    List<Member> members =
            query.from(qMember)
            .where(qMember.username.eq( right: "회원1"))
            .orderBy(qMember.username.desc())
            .list(qMember);
}
```

```
// select m from Member m
// where m.username='회원1'
// order by m.age desc
```

NativeSQL

- SQL을 직접 사용 할 수 있는 JPA에서 지원하는 기능
- 특정 데이터베이스 의존 기능 사용 할 때 사용

```
public void useNativeSQL() {
    String sql = "SELECT ID, AGE, TEAM_ID, NAME " +
                 "FROM MEMBER" +
                 "WHERE NAME = 'kim'";
    List<Member> resultList = em.createNativeQuery(sql, Member.class).getResultList();
}
```

벌크 연산

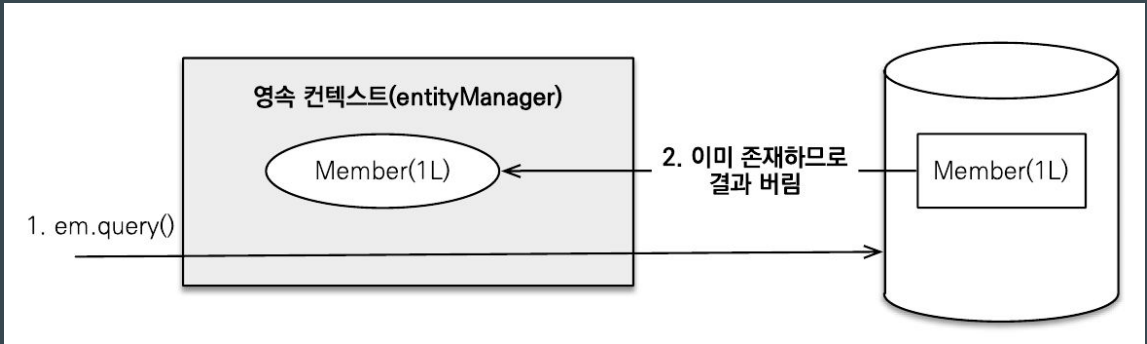
- 한번에 여러 데이터 수정하는 기법
- 영속성 컨텍스트를 무시하고 데이터베이스에 직접 쿼리(JPQL)

  ⇒ 영속성 컨텍스트와 DB간의 데이터 차이 발생

1. em.flush()
2. 벌크 연산 먼저 실행
3. 벌크 연산 후 영속성 컨텍스트 초기화

영속성 컨텍스트와 JPQL

- 영속성 컨텍스트는 조회한 엔티티만 관리 및 동일성 보장



JPQL과 플러시 모드

```
em.setFlushMode(FlushModeType.AUTO);    //커밋 또는 쿼리 실행 시 플러시(기본값)
em.setFlushMode(FlushModeType.COMMIT);//커밋시에만 플러시
```

- COMMIT - 수동으로 flush 시켜 무의미하게 발생하는 flush를 줄임

  ⇒ 성능 최적화

Thank you