4. 중급 문법

#1.인강/JPA활용편/querydsl/강의

- /프로젝션과 결과 반환 - 기본
- /프로젝션과 결과 반환 - DTO 조회
- /프로젝션과 결과 반환 - @QueryProjection
- /동적 쿼리 - BooleanBuilder 사용
- /동적 쿼리 - Where 다중 파라미터 사용
- /수정, 삭제 벌크 연산
- /SQL function 호출하기

프로젝션과 결과 반환 - 기본

프로젝션: select 대상 지정

프로젝션 대상이 하나

```
List<String> result = queryFactory
        .select(member.username)
        .from(member)
        .fetch();
```

- 프로젝션 대상이 하나면 타입을 명확하게 지정할 수 있음
- 프로젝션 대상이 둘 이상이면 튜플이나 DTO로 조회

튜플 조회

프로젝션 대상이 둘 이상일 때 사용

`com.querydsl.core.Tuple`

```
List<Tuple> result = queryFactory
        .select(member.username, member.age)
        .from(member)
        .fetch();

for (Tuple tuple : result) {
    String username = tuple.get(member.username);
    Integer age = tuple.get(member.age);
    System.out.println("username=" + username);
```

```
        System.out.println("age=" + age);
    }
```

프로젝션과 결과 반환 - DTO 조회

순수 JPA에서 DTO 조회

MemberDto

```
package study.querydsl.dto;

import lombok.Data;

@Data
public class MemberDto {
    private String username;
    private int age;

    public MemberDto() {
    }

    public MemberDto(String username, int age) {
        this.username = username;
        this.age = age;
    }
}
```

순수 JPA에서 DTO 조회 코드

```
List<MemberDto> result = em.createQuery(
        "select new study.querydsl.dto.MemberDto(m.username, m.age) " +
                "from Member m", MemberDto.class)
        .getResultList();
```

- 순수 JPA에서 DTO를 조회할 때는 new 명령어를 사용해야함
- DTO의 package이름을 다 적어줘야해서 지저분함
- 생성자 방식만 지원함

Querydsl 빈 생성(Bean population)

결과를 DTO 반환할 때 사용

다음 3가지 방법 지원

- 프로퍼티 접근
- 필드 직접 접근
- 생성자 사용

프로퍼티 접근 - Setter

```
List<MemberDto> result = queryFactory
        .select(Projections.bean(MemberDto.class,
                member.username,
                member.age))
        .from(member)
        .fetch();
```

필드 직접 접근

```
List<MemberDto> result = queryFactory
        .select(Projections.fields(MemberDto.class,
                member.username,
                member.age))
        .from(member)
        .fetch();
```

별칭이 다를 때

```
package study.querydsl.dto;

import lombok.Data;

@Data
public class UserDto {
    private String name;
    private int age;
}
```

```
List<UserDto> fetch = queryFactory
        .select(Projections.fields(UserDto.class,
```

```
                member.username.as("name"),
                ExpressionUtils.as(
                        JPAExpressions
                                .select(memberSub.age.max())
                                .from(memberSub), "age")
                )
        ).from(member)
        .fetch();
```

- 프로퍼티나, 필드 접근 생성 방식에서 이름이 다를 때 해결 방안
- `ExpressionUtils.as(source,alias)` : 필드나, 서브 쿼리에 별칭 적용
- `username.as("memberName")` : 필드에 별칭 적용

생성자 사용

```
List<MemberDto> result = queryFactory
        .select(Projections.constructor(MemberDto.class,
                member.username,
                member.age))
        .from(member)
        .fetch();
}
```

프로젝션과 결과 반환 - @QueryProjection

생성자 + @QueryProjection

```
package study.querydsl.dto;

import com.querydsl.core.annotations.QueryProjection;
import lombok.Data;

@Data
public class MemberDto {
    private String username;
    private int age;

    public MemberDto() {
```

```
    }

    @QueryProjection
    public MemberDto(String username, int age) {
        this.username = username;
        this.age = age;
    }
}
```

- `./gradlew compileQuerydsl`
- `QMemberDto` 생성 확인

@QueryProjection 활용

```
List<MemberDto> result = queryFactory
        .select(new QMemberDto(member.username, member.age))
        .from(member)
        .fetch();
```

이 방법은 컴파일러로 타입을 체크할 수 있으므로 가장 안전한 방법이다. 다만 DTO에 QueryDSL 어노테이션을 유지해야 하는 점과 DTO까지 Q 파일을 생성해야 하는 단점이 있다.

distinct

```
List<String> result = queryFactory
        .select(member.username).distinct()
        .from(member)
        .fetch();
```

> 참고: distinct는 JPQL의 distinct와 같다.

동적 쿼리 - BooleanBuilder 사용

동적 쿼리를 해결하는 두가지 방식

- BooleanBuilder
- Where 다중 파라미터 사용

```
@Test
public void 동적쿼리_BooleanBuilder() throws Exception {
    String usernameParam = "member1";
```

```
    Integer ageParam = 10;

    List<Member> result = searchMember1(usernameParam, ageParam);
    Assertions.assertThat(result.size()).isEqualTo(1);
}

private List<Member> searchMember1(String usernameCond, Integer ageCond) {
    BooleanBuilder builder = new BooleanBuilder();
    if (usernameCond != null) {
        builder.and(member.username.eq(usernameCond));
    }
    if (ageCond != null) {
        builder.and(member.age.eq(ageCond));
    }
    return queryFactory
            .selectFrom(member)
            .where(builder)
            .fetch();
}
```

동적 쿼리 - Where 다중 파라미터 사용

```
@Test
public void 동적쿼리_WhereParam() throws Exception {
    String usernameParam = "member1";
    Integer ageParam = 10;

    List<Member> result = searchMember2(usernameParam, ageParam);
    Assertions.assertThat(result.size()).isEqualTo(1);
}


private List<Member> searchMember2(String usernameCond, Integer ageCond) {
    return queryFactory
            .selectFrom(member)
            .where(usernameEq(usernameCond), ageEq(ageCond))
            .fetch();
}

private BooleanExpression usernameEq(String usernameCond) {
    return usernameCond != null ? member.username.eq(usernameCond) : null;
```

```
}

private BooleanExpression ageEq(Integer ageCond) {
    return ageCond != null ? member.age.eq(ageCond) : null;
}
```

- `where` 조건에 `null` 값은 무시된다.
- 메서드를 다른 쿼리에서도 재활용 할 수 있다.
- 쿼리 자체의 가독성이 높아진다.

조합 가능

```
private BooleanExpression allEq(String usernameCond, Integer ageCond) {
    return usernameEq(usernameCond).and(ageEq(ageCond));
}
```

- `null` 체크는 주의해서 처리해야함

수정, 삭제 벌크 연산

쿼리 한번으로 대량 데이터 수정

```
long count = queryFactory
        .update(member)
        .set(member.username, "비회원")
        .where(member.age.lt(28))
        .execute();
```

기존 숫자에 1 더하기

```
long count = queryFactory
        .update(member)
        .set(member.age, member.age.add(1))
        .execute();
```

곱하기: `multiply(x)`

```
update member
```

```
    set age = age + 1
```

쿼리 한번으로 대량 데이터 삭제

```
long count = queryFactory
        .delete(member)
        .where(member.age.gt(18))
        .execute();
```

> 주의: JPQL 배치와 마찬가지로, 영속성 컨텍스트에 있는 엔티티를 무시하고 실행되기 때문에 배치 쿼리를 실행하고 나면 영속성 컨텍스트를 초기화 하는 것이 안전하다.

SQL function 호출하기

SQL function은 JPA와 같이 Dialect에 등록된 내용만 호출할 수 있다.

member➡M으로 변경하는 replace 함수 사용

```
String result = queryFactory
        .select(Expressions.stringTemplate("function('replace', {0}, {1}, {2})",
member.username, "member", "M"))
        .from(member)
        .fetchFirst();
```

소문자로 변경해서 비교해라.

```
.select(member.username)
.from(member)
.where(member.username.eq(Expressions.stringTemplate("function('lower', {0})",
member.username)))
```

lower 같은 ansi 표준 함수들은 querydsl이 상당부분 내장하고 있다. 따라서 다음과 같이 처리해도 결과는 같다.

```
.where(member.username.eq(member.username.lower()))
```