기본 문법

Created By	㉿ 영교
Class	실전! Querydsl
Tags	JPA SpringData
# index	3

JPQL vs Querydsl

Before:: 원천 데이터 셋업

▼ Code

```
@BeforeEach
public void before() throws Exception {
    //given
    Team teamA = new Team("teamA");
    Team teamB = new Team("teamB");
    em.persist(teamA);
    em.persist(teamB);

    Member member1 = new Member("member1", 10, teamA);
    Member member2 = new Member("member2", 20, teamA);

    Member member3 = new Member("member3", 30, teamB);
    Member member4 = new Member("member4", 40, teamB);
    em.persist(member1);
    em.persist(member2);
    em.persist(member3);
    em.persist(member4);

    em.flush();
    em.clear();

    List<Member> members = em.createQuery("select m from Member m", Member.class).getResultList();
    for (Member member : members) {
        System.out.println("member = " + member);
        System.out.println("member.getTeam() = " + member.getTeam());
    }

}
```

- 앞으로 할 코드에서 사용할 기본 원천 데이터 세팅 `@BeforeEach`를 이용해 DB에 셋업

JPQL vs Querydsl

▼ Code

```
@Autowired EntityManager em;

@Test
public void startJPQL() throws Exception {
    //member1 find
    String qlString = "select m from Member m where m.username = :username";
```

```
    Member findMember = em.createQuery(qlString, Member.class)
            .setParameter("username", "member1").getSingleResult();

    assertThat(findMember.getUsername()).isEqualTo("member1");

}

@Test
public void startQuerydsl() throws Exception {
    //member1을 찾아라.
    JPAQueryFactory queryFactory = new JPAQueryFactory(em);
    QMember m = new QMember("m");
    Member findMember = queryFactory .select(m)
            .from(m)
            .where(m.username.eq("member1"))//파라미터 바인딩 처리
            .fetchOne();
    assertThat(findMember.getUsername()).isEqualTo("member1");
}
```

- `EntityManager` 를 주입해서 `JPAQueryFactory` 를 생성한다.
- `Querydsl`은 JPQL빌더
- JPQL: 문자(실행 시점 오류) vs Querydsl: 코드(컴파일 시점 오류)
- JPQL: 파라미터 바인딩 직접 vs Querydsl: 파라미터 바인딩 자동 처리

JPAQueryFactory 를 필드로 뺄 수도 있다.

▼ Code

```
@Autowired
EntityManager em;

JPAQueryFactory queryFactory;



@BeforeEach
public void before() throws Exception {
    queryFactory = new JPAQueryFactory(em);
    ...
}

@Test
public void startJPQL() throws Exception {
    //member1 find
    String qlString = "select m from Member m where m.username = :username";
    Member findMember = em.createQuery(qlString, Member.class)
            .setParameter("username", "member1").getSingleResult();

    assertThat(findMember.getUsername()).isEqualTo("member1");

}

@Test
public void startQuerydsl() throws Exception {
    //given
    Member findMember = queryFactory
            .selectFrom(member)
            .where(member.username.eq("member1")
                    .and(member.age.eq(10))) //파라미터 바인딩 처
            .fetchOne();
    assertThat(findMember.getUsername()).isEqualTo("member1");
```

```
}
```

- `queryFactory` 를 필드로 빼고 before setup부분에서 EntityManger을 주입해준다.

> 참고: JPAQueryFactory를 필드로 제공하면 동시성 문제(Multi Threading)은 어떻게 될까?
> → 스프링 프레임워크는 여러 쓰레드에서 동시에 같은 EntityManager에 접근해도, 트랜잭션 마다 별도의 영속성 컨텍스트를 제공하기 때문에,
> 동시성 문제는 일어나지 않는다.

기본 Q-Type 활용

위의 Querydsl 예제에서는 Q클래스의 인스턴스를 사용할 때 `new QMember("m")` 와 같이 `new` 를 통해 별칭을 직접 지정해줬지만,
`Querydsl` 에서 제공하는 기본 인스턴스를 사용하는게 더 간편하다.

```
QMember qMember = new QMember("M"); // 별칭 직접 지정
QMember qMember = QMember.member; //기본 인스턴스 사용
```

- Q클래스를 static-import 해주면 `member` 도 바로 사용할 수 있다.

```
import static study.querydsl.entity.QMember.*;

@Test
public void startQuerydsl3(){
    Member findMember = queryFactory.selectFrom(member).where(member.id.eq(1L).fetchOne();
    assertThat(findMember.getId)).isEqualTo(1L);
}
```

검색 조건 쿼리

기본 검색 쿼리

▼ Code

```
@Test
public void searchAndParam() throws Exception {
    //given
    Member findMember = queryFactory
            .selectFrom(member)
            .where(
                    member.username.eq("member1")
                    .and(member.age.eq(10))
            ) //파라미터 바인딩 처
```

```
            .fetchOne();
    assertThat(findMember.getUsername()).isEqualTo("member1");
}
```

- 검색 조건은 `.and()` , `.or()` 를 메서드 체인으로 연결할 수 있습니다.
- `select` 와 `from` 은 `selectFrom`으로 합칠 수 있습니다.

AND 조건을 메서드 체인 말고 파라미터로 처리할 수도 있습니다.

▼ Code

```
@Test
public void searchAndParam() throws Exception {
    //given
    Member findMember = queryFactory
            .selectFrom(member)
            .where(
                    member.username.eq("member1"),
                    member.age.eq(10)
            ) //파라미터 바인딩 처
            .fetchOne();
    assertThat(findMember.getUsername()).isEqualTo("member1");
}
```

- null 값은 무시한다. → 동적 쿼리 생성에 용이하다.

JPQL이 제공하는 모든 검색 조건을 제공합니다.

- `member.username.eq("a")` : username = 'a'
- `member.username.ne("a")` : username ≠ 'a'
- `member.username.eq("a").not()` : username ≠ 'a'
- `member.username.isNotNull()` : username is not null
- `member.age.in(10,20)` : age in (10,20)
- `member.age.notIn(10,20)` : age not in(10,20)
- `member.age.between(10,30)` : age between 10, 30
- `member.age.goe(30)` : age ≥ 30
- `member.age.gt(30)` : age > 30
- `member.age.loe(30)` : age ≤ 30
- `member.age.lt(30)` : age < 30
- `member.username.like("member%")` : username like 'member%'
- `member.username.contains("member')` : username like '%member%'
- `member.username.startsWith("member")` : like 'member%'
- .... 등등

결과 조회

→ queryFactory 를 통해 생성한 쿼리들
(`queryFactory.select(member).from(member).where(member.id.eq(1L))...`)의 결과를 반환하는 함수들에 대해 소개한다.

- `fetch()` : 리스트 조회, 데이터 없으면 빈 리스트 반환
- `fetchOne()` : 단 건 조회
  - 결과가 없으면: null
  - 결과가 둘 이상이면: `com.querydsl.core.NonUniqueResultException`
- `fetchFirst()` : `limit(1).fetchOne()` 과 같다.
- `fetchResults()` : 페이징 정보 포함, total count 쿼리 추가 실행
- `fetchCount()` :count 쿼리로 변경해서 count 수 조회

테스트 코드 작성 및 검증

▼ Code

```
@Test
public void resultFetchTest() throws Exception {
    //given
    List<Member> fetch = queryFactory
            .selectFrom(member)
            .fetch();

    Member fetchOne = queryFactory
            .selectFrom(QMember.member)
            .fetchOne();

    Member fetchFirst = queryFactory
            .selectFrom(QMember.member)
            .fetchFirst();

    QueryResults<Member> results = queryFactory
            .selectFrom(member)
            .fetchResults();

    results.getTotal();
    List<Member> content = results.getResults();

    long total = queryFactory
            .selectFrom(member)
            .fetchCount();

}
```

정렬

QuerydsI 에서는 정렬(Sort)용 메서드 역시 제공된다.
→ .orderBy(인스턴스명.기준필드.정렬기준.(nullsLast()|nullsFirst()))

정렬 코드 작성 및 검수

▼ Code

```
/**
 * 회원 정렬 순서
 * 1. 회원 나이 내림차순(desc)
 * 2. 회원 이름 올림차순(asc)
 * 단 2에서 회원 이름이 없으면 마지막에 출력(nulls last)
 *
 * @throws Exception
 */
@Test
public void sort() throws Exception {
    //given
    em.persist(new Member(null, 100));
    em.persist(new Member("member5", 100));
    em.persist(new Member("member6", 100));

    List<Member> result = queryFactory
            .selectFrom(member)
            .where(member.age.eq(100))
            .orderBy(member.age.desc(), member.username.asc().nullsLast())
            .fetch();

    Member member5 = result.get(0);
    Member member6 = result.get(1);
    Member memberNull = result.get(2);
    assertThat(member5.getUsername()).isEqualTo("member5");
    assertThat(member6.getUsername()).isEqualTo("member6");
    assertThat(memberNull.getUsername()).isNull();
}
```

- `desc()`, `asc()` :일반 정렬
- `nullsLast()`, `nullsFirst()` :null 데이터 순서 부여

페이징

조회 건수 제한

▼ Code

```
@Test
public void paging1() throws Exception {
    List<Member> result = queryFactory
            .selectFrom(member)
            .orderBy(member.username.desc())
            .offset(1)
            .limit(2)
            .fetch();

    assertThat(result.size()).isEqualTo(2);
}
```

- 페이징할 때는 orderBy를 넣어서 정렬을 해줘야 잘 동작한다. offset(1).limit(2)는 index(0)을 생략하고 두개를 선택한다는 뜻

→ **[0][1][2][3]**

전체 조회 수가 필요한 경우

▼ Code

```
@Test
public void paging2() throws Exception {
    QueryResults<Member> result = queryFactory
            .selectFrom(member)
            .orderBy(member.username.desc())
            .offset(1)
            .limit(2)
            .fetchResults();

    assertThat(result.getTotal()).isEqualTo(4);
    assertThat(result.getLimit()).isEqualTo(2);
    assertThat(result.getOffset()).isEqualTo(1);
    assertThat(result.getResults().size()).isEqualTo(2);
}
```

- Count쿼리까지 총 두 번 실행된다.

집합

JPQL이 제공하는 모든 집합 함수를 Querydsl 에서 제공한다.

집함 함수 코드

▼ Code

```
@Test
public void aggregation() throws Exception {
    List<Tuple> result = queryFactory
            .select(
                    member.count(), //회원수
                    member.age.sum(),//나이 합
                    member.age.avg(),//나이 평균
                    member.age.max(),//최대 나이
                    member.age.min()//최소 나이
            )
            .from(member)
            .fetch();

    Tuple tuple = result.get(0);
    assertThat(tuple.get(member.count())).isEqualTo(4);
    assertThat(tuple.get(member.age.sum())).isEqualTo(100);
    assertThat(tuple.get(member.age.avg())).isEqualTo(25);
    assertThat(tuple.get(member.age.max())).isEqualTo(40);
    assertThat(tuple.get(member.age.min())).isEqualTo(10);
}
```

- `Tuple`에 대해서는 아래에서 따로 설명하도록 하겠습니다.

Group By 사용 → 팀의 이름과 각 팀의 평균 연령을 구해라.

▼ Code

```
/**
 * 팀의 이름과 각 팀의 평균 연령을 구해라.
 *
 * @throws Exception
 */
@Test
public void group() throws Exception {
    List<Tuple> result = queryFactory
            .select(team.name, member.age.avg())
            .from(member)
            .join(member.team, team)
            .groupBy(team.name) // GroupBy 함수 사용 -> Team.name기준으로 그룹핑을 해준다.
            .fetch();
    Tuple teamA = result.get(0);
    Tuple teamB = result.get(1);

    assertThat(teamA.get(team.name)).isEqualTo("teamA");
    assertThat(teamA.get(member.age.avg())).isEqualTo(15);

    assertThat(teamB.get(team.name)).isEqualTo("teamB");
    assertThat(teamB.get(member.age.avg())).isEqualTo(35);
}
```

- `having` 함수 역시 같이 사용 가능하다.
  → member의 age를 기준으로 그룹핑을 하되 25살 이상만 그룹핑 한다.

```
...
.groupBy(member.age)
.having(member.age.gt(25))
...
```

조인 - 기본 조인

→ Querydsl 에서는 JOIN 함수도 역시 제공된다.

- `join()` , `innerJoin()` : 내부 조인(inner join)
- `leftJoin()` : left 외부 조인(left outer join)
- `rightJoin()` : right 외부 조인(right outer join)
- JPQL의 `on` 과 성능 최적화를 위한 `fetch` 조인 제공

❓그렇다면 연관관계가 없는 엔티티간의 조인은 어떻게 하는가? → 세타 조인

▼ Code

```
/**
 * 세타 조인
 * 회원의 이름이 팀 이름과 같은 회원 조회
 *
 * @throws Exception
 */
```

```
@Test
public void theta_join() throws Exception {
    //given
    em.persist(new Member("teamA"));
    em.persist(new Member("teamB"));
    em.persist(new Member("teamC"));

    List<Member> result = queryFactory
            .select(member)
            .from(member, team)
            .where(member.username.eq(team.name))
            .fetch();

    assertThat(result)
            .extracting("username")
            .containsExactly("teamA", "teamB");
}
```

- `from` 절에서 여러 엔티티를 선택해서 세타 조인이 가능하다.
- 전혀 연관관계가 없는 필드로도 조인이 가능하다.
- 외부 조인(`leftJoin`, `rightJoin`)이 불가능한데 `on` 절을 사용해서 외부 조인이 가능하다.

조인 - on절

`ON` 절을 활용한 조인(JPA 2.1부터 지원)

- 조인 대상 필터링
- 연관관계 없는 엔티티 외부 조인

1. 조인 대상 필터링

회원과 팀을 조인하면서, 팀 이름이 teamA인 팀만 조인, 회원은 모두 조회

▼ Code

```
/**
 * 예) 회원과 팀을 조인하면서, 팀 이름이 teamA인 팀만 조인, 회원은 모두 조회
 * JPQL: select m, t from Member m left join m.team t on t.name = 'teamA';
 *
 * @throws Exception
 */
@Test
public void join_on_filtering() throws Exception {
    List<Tuple> result = queryFactory
            .select(member, team)
            .from(member)
            .leftjoin(member.team, team).on(team.name.eq("teamA"))
            .fetch();
    for (Tuple tuple : result) {
        System.out.println("tuple = " + tuple);
    }
}
/* 실행 결과 */
/*
tuple = [Member(id=3, username=member1, age=10), Team(id=1, name=teamA)]
tuple = [Member(id=4, username=member2, age=20), Team(id=1, name=teamA)]
tuple = [Member(id=5, username=member3, age=30), null]
```

```
tuple = [Member(id=6, username=member4, age=40), null]
*/
```

- `on` 절을 사용하여 조인 대상을 필터링 할 때 내부조인(inner Join)을 사용하면 where절에서 필터링 하는 것과 기능이 동일하다.

  ```
  .leftjoin(member.team, team).on(team.name.eq("teamA")) -> .join(member.team, team).on(team.name.eq("teamA"))
  ```

2. 연관관계 없는 엔티티 외부 조인

회원의 이름과 팀의 이름이 같은 대상 외부 조인

▼ Code

```
/**
 * 연관관계가 없는 엔티티 외부 조인
 * 회원의 이름이 팀 이름과 같은 대상을 외부 조인
 *
 * @throws Exception
 */
@Test
public void join_on_no_relation() throws Exception {
    //given
    em.persist(new Member("teamA"));
    em.persist(new Member("teamB"));
    em.persist(new Member("teamC"));

    List<Tuple> result = queryFactory
            .select(member, team)
            .from(member)
            .leftjoin(team).on(member.username.eq(team.name))
            .fetch();

    for (Tuple tuple : result) {
        System.out.println("tuple = " + tuple);
}
```

- 하이버네이트 5.1 부터 `on` 절을 사용해서 서로 관계가 없는 필드로 외부 조인하는 기능이 추가되었다. 물론 내부조인도 가능하다.
- **(Important) 일반 조인과 다르게 leftJoin() 인자로 엔티티 하나만 들어간다.**
  - 일반 조인 → `leftJoin(member.team, team)`
  - on 조인 → `from(member).leftJoin(team).on(xxx)`

조인 - 페치 조인

페치 조인은 SQL 자체적으로 제공하는 기능은 아니고 JPA에서 SQL조인을 활용해서 연관된 엔티티를 SQL 한 번에 조회해 가져오는 기능입니다.
성능 최적화를 위해 엔티티 연관관계에서 모든 로딩전략을 지연로딩(fetch = fetchType.LAZA)으로 설정하는데, 페치조인을 사용하면,
getter 호출시마다 쿼리를 수행하는 것을 막을 수 있습니다.

Before:: 페치 조인 미적용

→ 지연로딩으로 Member, Team SQL 쿼리 각각 실행

▼ Code

```
@PersistenceUnit
EntityManagerFactory emf;

@Test
public void fetchJoinNo() throws Exception {
    em.flush();
    em.clear();

    Member member1 = queryFactory
            .selectFrom(member)
            .where(member.username.eq("member1"))
            .fetchOne();

    boolean loaded = emf.getPersistenceUnitUtil().isLoaded(member1.getTeam());
    assertThat(loaded).as("페치 조인 미적용").isFalse();
}

/* 수행 쿼리 */
/*
    select member1
    from Member member1
    where member1.username = 'member1'
*/
```

- Member Entity만 조회되었을 뿐 연관관계에 있는 Team Entity는 조회되지 않았다.

  → 그렇기 때문에 `assertThat(loaded).as("페치 조인 미적용").isFalse();` 은 통과 된다.

After::페치 조인 적용

→ 즉시로딩으로 Member, Team SQL 쿼리 조인으로 한 번에 조회

▼ Code

```
@Test
public void fetchJoinUse() throws Exception {
    em.flush();
    em.clear();

    Member member1 = queryFactory
            .selectFrom(member)
            .join(member.team, team).fetchJoin()
            .where(member.username.eq("member1"))
            .fetchOne();

    boolean loaded = emf.getPersistenceUnitUtil().isLoaded(member1.getTeam());
    assertThat(loaded).as("페치 조인 적용").isTrue();
}
```

- `join(member.team, team)` 옆을보면 `.fetchJoin()`을 호출하는 것을 볼 수 있다.
  이렇게 `fetchJoin()`을 호출해주면 연관관계에 있는 Team Entity도 함께 조회하기 때문에 `loaded` 가 true가 된 것을 확인할 수 있다.

서브 쿼리

→ `com.querydsl.jpa.JPAExpressions` 를 사용하여 서브쿼리 사용이 가능하다.

서브쿼리 eq 사용

▼ Code

```
/**
 * 나이가 가장 많은 회원 조회
 *
 * @throws Exception
 */
@Test
public void subQuery() throws Exception {
    QMember memberSub = new QMember("memberSub");

    List<Member> result = queryFactory
            .selectFrom(member)
            .where(member.age.eq(JPAExpressions
                    .select(memberSub.age.max())
                    .from(memberSub)
            ))
            .fetch();

    assertThat(result)
            .extracting("age")
            .containsExactly(40);
}
```

서브쿼리 goe 사용

▼ Code

```
import static com.querydsl.jpa.JPAExpressions.select;
/**
 * 나이가 평균 이상 회원 조회
 *
 * @throws Exception
 */
@Test
public void subQueryGoe() throws Exception {
    QMember mSub = new QMember("memberSub");

    List<Member> result = queryFactory
            .selectFrom(member)
            .where(member.age.goe(
                    select(mSub.age.avg())
                            .from(mSub)
            ))
            .fetch();

    assertThat(result)
            .extracting("age")
            .containsExactly(30, 40);
}
```

서브쿼리 여러 건 처리 in절 사용

▼ Code

```
import static com.querydsl.jpa.JPAExpressions.select;
/**
 * 나이가 가장 많은 회원 조회
 *
 * @throws Exception
 */
@Test
public void subQueryIn() throws Exception {
    QMember mSub = new QMember("memberSub");

    List<Member> result = queryFactory
            .selectFrom(member)
            .where(member.age.in(
                    select(mSub.age)
                            .from(mSub)
                            .where(mSub.age.gt(10))
            ))
            .fetch();

    assertThat(result)
            .extracting("age")
            .containsExactly(20, 30, 40);
}
```

서브쿼리 select 절에서 사용sele

▼ Code

```
import static com.querydsl.jpa.JPAExpressions.select;

@Test
public void selectSubQuery() throws Exception {
    QMember mSub = new QMember("memberSub");

    List<Tuple> fetch = queryFactory
            .select(member.username,
                    select(mSub.age.avg())
                            .from(mSub)
            )
            .from(member)
            .fetch();
    for (Tuple tuple : fetch) {
        System.out.println("tuple = " + tuple);
    }
}
```

> from절의 서브쿼리 한계
> → JPA JPQL 서브쿼리의 한계점으로 from절의 서브쿼리(인라인 뷰)는 지원하지 않습니다.
> JPQL 쿼리 빌더인 Querydsl 역시 같은 이유로 지원하지 않습니다.

> 하이버네이트 구현체를 사용하면 select절의 서브쿼리는 지원합니다. Querydsl도 하이버네이트 구현체를 사용하면 select 절 서브쿼리를 지원합니다.

> from절의 서브쿼리 한계돌파 방안

1. 서브쿼리 → `JOIN` 으로 변경한다. (가능한 상황도 있고, 불가능한 상황도 있다.)
2. 애플리케이션에서 쿼리를 2번 분리해서 실행한다.
3. `nativeSQL` 을 사용한다.

Case 문

QuerydsI 에서 select, 조건절(where)에서 사용 가능합니다.

단순한 조건

▼ Code

```
@Test
public void basicCase() throws Exception {
    List<String> result = queryFactory
            .select(member.age
                    .when(10).then("열살")
                    .when(20).then("스무살")
                    .otherwise("기타")
            )
            .from(member)
            .fetch();
    for (String s : result) {
        System.out.println("s = " + s);
    }
}
/*실행결과*/
/*
열살
스무살
기타
기타
*/
```

- age가 10살이면 '열살', 20이면 '스무살' 그밖에는 '기타'로 출력된다.

복잡한 조건

▼ Code

```
@Test
public void complexCase() throws Exception {
    List<String> result = queryFactory
            .select(new CaseBuilder()
                    .when(member.age.between(0, 20)).then("0~20살")
```

```
                    .when(member.age.between(21, 30)).then("21살~30살")
                    .otherwise("기타")
            )
            .from(member)
            .fetch();
    for (String s : result) {
        System.out.println("s = " + s);
    }
}
```

- CaseBuilder()를 통해 동작합니다.
- basic한 case와는 다르게 `when` 절안에 조건이 들어갑니다.

상수, 문자 더하기

상수가 필요하다면 `Expressions.constant(xxx)` 사용

▼ Code

```
@Test
public void constant() throws Exception {
    List<Tuple> result = queryFactory
            .select(member.username, Expressions.constant("A"))
            .from(member)
            .fetch();

    for (Tuple tuple : result) {
        System.out.println("tuple = " + tuple);
    }
}
```

> 참고: 위와 같이 최적화가 가능하면 SQL에 constant 값을 넘기지 않는다. 상수를 더하는 것처럼 최적화가 어려우면 SQL에 constant값을 넘긴다

문자 더하기 concat

▼ Code

```
@Test
public void concat() throws Exception {
    List<String> result = queryFactory
            .select(member.username.concat("_").concat(member.age.stringValue()))
            .from(member)
            .fetch();

    for (String s : result) {
        System.out.println("s = " + s);
    }
}
```

> 참고: `member.age.stringValue()` : 문자가 아닌 다른타입(ex: int, float..)들을 문자로 변환해준다. 이 방법은 ENUM을 처리할 때도 자주 사용한다.
