Querydsl

JPQL – How to Define Queries in JPA and Hibernate

http://querydsl.com/static/querydsl/5.0.0/reference/html_single/#d0e147

소개

- **Querydsl**
  - 동적쿼리 작성을 용이하게 함
  - 자바 메소드를 사용하므로 문자열 덧셈 연산에서 발생하기 쉬운 잘못된 SQL문 작성 문제 완화
  - SQL문의 가독성이 높아짐
  - 문자열로 표현된 SQL은 type 체크가 불가능하지만 코드로 작성된 SQL문은 type 체크가 가능
    - 컴파일 단계에서 에러가 발생하므로 시스템 운영 시 발생하는 큰 문제를 미연에 방지
  - 일관성: 동일한 인터페이스와 동일한 구현을 기반으로 쿼리가 작성되므로 일관성 있는 쿼리 작성(실행) 가능

Maven integration

- **pom.xml에 라이브러리 추가**

```
<!-- https://mvnrepository.com/artifact/com.querydsl/querydsl-jpa -->
    <dependency>
        <groupId>com.querydsl</groupId>
        <artifactId>querydsl-jpa</artifactId>
        <version>4.2.1</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/com.querydsl/querydsl-apt -->
    <dependency>
        <groupId>com.querydsl</groupId>
        <artifactId>querydsl-apt</artifactId>
        <version>4.1.4</version>
    </dependency>
```

  - querydsl-jpa: QueryDSL JPA 라이브러리
  - querydsl-apt: 쿼리 타입(Q)을 생성할 때 필요한 라이브러리

Maven integration

- **pom.xml에 환경설정**

```
<build>
    <plugins>
      <plugin>
        <groupId>com.mysema.maven</groupId>
        <artifactId>apt-maven-plugin</artifactId>
        <version>1.1.3</version>
        <executions>
          <execution>
            <goals>
              <goal>process</goal>
            </goals>
            <configuration>
              <outputDirectory>target/generated-sources/java</outputDirectory>
              <processor>com.querydsl.apt.jpa.JPAAnnotationProcessor</processor>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
```

  - target/generated-sources/java: Q파일 생성 경로

Maven integration

- **Q파일 생성**
  - 메이븐을 설치하고 메이븐 명령어를 통해 컴파일해도 되지만 IDE의 GUI환경을 사용

    Failed to execute goal com.mysema.maven:apt-maven-plugin:1.1.3

  - The querydsl-apt dependency is an annotation processing tool (APT)
  - pom.xml에 필요라이브러리 추가

```
<!-- https://mvnrepository.com/artifact/jakarta.annotation/jakarta.annotation-api -->
        <dependency>
            <groupId>jakarta.annotation</groupId>
            <artifactId>jakarta.annotation-api</artifactId>
            <version>1.3.5</version>
        </dependency>
```

- **Q파일 생성**

Querying

- **Querydsl**
  - 동적쿼리 작성을 용이하게 함
  - 자바 메소드를 사용하므로 문자열 덧셈 연산에서 발생하기 쉬운 잘못된 SQL문 작성 문제 완화
  - SQL문의 가독성이 높아짐
  - 문자열로 표현된 SQL은 type 체크가 불가능하지만 코드로 작성된 SQL문은 type 체크가 가능
    - 컴파일 단계에서 에러가 발생하므로 시스템 운영 시 발생하는 큰 문제를 미연에 방지
  - 일관성: 동일한 인터페이스와 동일한 구현을 기반으로 쿼리가 작성되므로 일관성 있는 쿼리 작성(실행) 가능

데이터 준비

- **테스트 데이터를 넣는 코드가 길어지는 것을 막기 위해**
  - 생성자 추가
  - 기본 생성자 추가
  - Helper메소드 추가

데이터 준비

```
Address address1 = new Address("street1","city1", "zipcode1");
Address address2 = new Address("street2","city2", "zipcode2");
Address address3 = new Address("street3","city3", "zipcode3");

Person kim = new Person("kim",20);
Person lee = new Person("lee",30);
Person park = new Person("park",25);
Person hong = new Person("hong",15);

kim.addAddress(address1);
kim.addAddress(address2);
lee.addAddress(address3);

em.persist(kim);
em.persist(lee);
em.persist(park);
em.persist(hong);
```

Querying

- 간단 예제

```
JPAQueryFactory query = new JPAQueryFactory(em);
QPerson person = new QPerson("p");
List<Person> persons = query.select(person)
        .from(person)
.fetch();
```

- select와 from을 묶어 selectFrom 하나로 작성 가능

```
JPAQueryFactory query = new JPAQueryFactory(em);
QPerson person = new QPerson("p");
List<Person> persons = query.selectFrom(person).fetch();
```

기본 동작

- **의존성 설정**

SLF4J: Failed to load class "org.slf4j.impl.StaticMDCBinder".

```
<dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.7.31</version>
    </dependency>
<dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-simple</artifactId>
        <version>1.7.31</version>
</dependency>
```

기본 동작

- **JPAQueryFactory**
  - JPAQueryFactory를 통해 QueryDSL에서 제공하는 쿼리 생성
- **QueryDSL 구분**
  - JPAQuery, JPAQueryFactory, JPASQLQuery, SQLQuery, SQLQueryFactory
  - 위 방식은 모두 쿼리문을 작성하기 위해 Q 타입 클래스를 사용
  - JPAQuery, JPAQueryFactory는 EntityManager를 통해서 질의가 처리되고 이 때 사용하는 쿼리문은 JPQL
  - SQLQuery, SQLQueryFactory는 JDBC를 이용하여 질의가 처리되고 이 때 사용하는 쿼리문은 SQL
  - JPASQLQuery는 EntityManager를 통해서 질의가 처리되고 이 때 사용하는 쿼리문은 SQL
  - JPAQueryFactory를 통해 JPAQuery를 만들어 낼 수 있음
  - JPASQLQuery, SQLQuery, SQLQueryFactory는 엔티티 클래스를 만들지 않고 사용 가능. 이미 테이블이 존재하는 경우 이를 바탕으로 쿼리타입을 생성

Select문

- **Where절에 다양한 조건 걸기**
  - 조건 활용

```
JPAQueryFactory query = new JPAQueryFactory(em);
QPerson person = new QPerson("p");
List<Person> persons = query.select(person)
        .from(person)
        .where(person.name.eq("kim"))
.fetch();
```

  - 조건 여러 개

```
JPAQueryFactory query = new JPAQueryFactory(em);
QPerson person = new QPerson("p");
List<Person> persons = query.select(person)
        .from(qPerson)
        .where(person.name.eq("kim"), person.age.gt(15))
.fetch();
```

Select문

- **Where절에 다양한 조건 걸기**
  - chaining 방식으로 and 혹은 or 연결
    - where안에 조건을 콤마로 구분하여 나열하면 and 관계

```
JPAQueryFactory query = new JPAQueryFactory(em);
QPerson person = new QPerson("p");
List<Person> persons = query.select(person)
        .from(person)
        .where(person.name.eq("kim").and(person.name.gt(30))
.fetch();
```

```
JPAQueryFactory query = new JPAQueryFactory(em);
QPerson person = new QPerson("p");
List<Person> persons = query.select(person)
        .from(person)
        .where(person.name.eq("kim").or(person.name.gt(15)))
.fetch();
```

Join문

- 조인
  - 다양한 형태의 조인 사용 가능
  - 조인 대상이 되는 엔티티에 대한 Q클래스를 함께 참조해야 함

```
JPAQueryFactory query = new JPAQueryFactory(em);
QPerson person = new QPerson("p");
QAddress address = new QAddress("a");
List<Person> persons = query.selectFrom(person)
        .innerjoin(person.addresses, address)
.fetch();
```

```
JPAQueryFactory query = new JPAQueryFactory(em);
QPerson person = new QPerson("p");
QAddress address = new QAddress("a");
List<Person> persons = query.selectFrom(person)
        .innerjoin(person.addresses, address).fetchJoin()
.fetch();
```

결과 반환

- **fetch 메소드**
  - fetch : 여러 개의 조회 결과를 컬렉션으로 반환
  - fetchOne : 한 건의 조회 결과를 지정한 타입으로 반환
  - fetchFirst : 여러 개의 조회 결과 중 가장 처음 결과만 반환
  - fetchCount : 조회 결과가 아닌 조회에 포함된 레코드 개수를 long 타입 반환
  - fetchResults : 조회한 리스트와 함께 전체 개수를 QueryResults로 반환

페이징

- **페이징**
  - offset은 처음 위치(0부터 시작), limit은 몇 개 가져올 것인가를 지정

```
JPAQueryFactory query = new JPAQueryFactory(em);
QPerson person = new QPerson("p");
List<Person> persons = query.selectFrom(person)
        .orderBy(person.age.asc())
        .offset(0).limit(2)
.fetch();
```

```
JPAQueryFactory query = new JPAQueryFactory(em);
QPerson person = new QPerson("p");
QueryResults<Person> results = query.selectFrom(person)
        .orderBy(person.age.asc())
        .offset(0).limit(2)
.fetchResults();
List<Person> persons = results.getResults();
```

count쿼리를 실행하여
전체 레코드 수도 함께 조회