- **QueryDSL**
  - **QueryDSL 설정하기**
    - - 필요 라이브러리
    - - 환경 설정
  - **시작하기**
    - - 기본 Q 생성
  - **검색 조건 쿼리**
  - **결과 조회**
  - **페이징과 정렬**
  - **그룹**
  - **조인(Join)**
  - **서브 쿼리**
  - **프로젝션과 결과 반환**
    - - 프로젝션 대상이 하나
    - - 여러 컬럼 반환과 튜플
    - - 빈 생성(Bean population)
    - - DISTINCT
  - **수정, 삭제 배치 쿼리**
  - **동적 쿼리**
  - **메서드 위임(Delegate methods)**

QueryDSL

JPA Criteria는 문자가 아닌 코드로 JPQL을 작성하므로 문법 오류를 컴파일 단계에서 잡을 수 있고 IDE 자동완성 기능의 도움을 받을 수 있는 등 여러가지 장점이 있다. 하지만 Criteria의 가장 큰 단점은 너무 복잡하고 어렵다는 것이다. 작성된 코드를 보면 그 복잡성으로 인해 어떤 JPQL이 생성될지 파악하기가 쉽지 않다.

쿼리를 문자가 아닌 코드로 작성해도, 쉽고 간결하며 그 모양도 쿼리와 비슷하게 개발할 수 있는 프로젝트가 바로 QueryDSL이다. QueryDSL도 Criteria처럼 JPQL 빌더 역할을 하는데 JPA 크리테라아를 대체할 수 있다.

QueryDSL은 오픈소스 프로젝트다. 처음에는 HQL(하이버네이트 쿼리언어)을 코드로 작성할 수 있도록 해주는 프로젝트로 시작해서 지금은 JPA, JDO, JDBC, Lucene, Hibernate Search, MongoDB, 자바 컬렉션등을 다양하게 지원한다. 참고로 QueryDSL은 이름 그대로 쿼리 즉 데이터를 조회하는데 기능이 특화되어 있다.

QueryDSL 설정하기

- 필요 라이브러리

참고로 예제에 사용한 버전은 3.6.2이다.

====== pom.xml 추가 ======

```
<dependency>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-jpa</artifactId>
  <version>3.6.2</version>
</dependency>

<dependency>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-apt</artifactId>
  <version>3.6.2</version>
  <scope>provided</scope>
</dependency>
```

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

- 환경 설정

QueryDSL을 사용하려면 Criteria의 메타 모델처럼 엔티티를 기반으로 쿼리 타입이라는 쿼리용 클래스를 생성해야 한다.

====== pom.xml 추가 ======

```
<build>
    <plugins>
        <plugin>
            <groupId>com.mysema.maven</groupId>
            <artifactId>apt-maven-plugin</artifactId>
            <version>1.0.9</version>
            <executions>
                <execution>
                    <goals>
                        <goal>process</goal>
                    </goals>
                    <configuration>
                        <outputDirectory>target/generated-sources/java</outputDi
                        <processor>com.mysema.query.apt.jpa.JPAAnnotationProcess
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

콘솔에서 `mvn compile`을 입력하면 `outputDirectory`에 지정한 `target/generated-sources` 위치에 `QMember.java`처럼 Q로 시작하는 쿼리 타입들이 생성된다.

이제 `target/generated-sources`를 소스 경로에 추가하면 된다.

> **참고:** 이클립스 LUNA 버전을 사용하면 메이븐과 연동되어서 쿼리 타입이 자동으로 생성되고 소스 경로도 자동으로 추가된다.

시작하기

QueryDSL을 어떻게 사용하는지 간단한 예제로 알아보자.

```
public void queryDSL() {
```

```
    EntityManager em = emf.createEntityManager();

    JPAQuery query = new JPAQuery(em);
    QMember qMember = new QMember("m"); //생성되는 JPQL의 별칭이 m
    List<Member> members =
            query.from(qMember)
                    .where(qMember.name.eq("회원1"))
                    .orderBy(qMember.name.desc())
                    .list(qMember);
}
```

QueryDSL을 사용하려면 우선 `com.mysema.query.jpa.impl.JPAQuery` 객체를 생성해야 하는데 이때 엔티티 매니저를 생성자에 넘겨준다. 다음으로 사용할 쿼리 타입(Q)을 생성하는데 생성자에는 별칭이름을 주면 된다. 이 별칭을 JPQL에서 별칭으로 사용한다.

그 다음에 나오는 `from`, `where`, `orderBy`, `list`는 코드만 보아도 쉽게 이해가 될 것이다. 다음 실행된 JPQL을 보면 둘이 얼마나 비슷한지 알 수 있다.

```
select m from Member m
where m.name = ?1
order by m.name desc
```

- 기본 Q 생성

쿼리 타입(Q)은 사용하기 편리하도록 기본 인스턴스를 보관하고 있다. 하지만 같은 엔티티를 조인하거나 같은 엔티티를 서브쿼리에 사용하면 같은 별칭이 사용되므로 이때는 별칭을 직접 지정해서 사용해야 한다.

==== Member 쿼리 타입 ====

```
public class QMember extends EntityPathBase<Member> {

    public static final QMember member = new QMember("member1");
    ...
}
```

==== 쿼리 타입 사용 ====

```
QMember qMember = new QMember("m"); //직접 지정
```

```
QMember qMember = QMember.member;    //기본 인스턴스 사용
```

기본 인스턴스를 사용하면 `import static`을 활용할 수 있다.

```
import static jpabook.jpashop.domain.QMember.member; //기본 인스턴스

public void basic() {

    EntityManager em = emf.createEntityManager();

    JPAQuery query = new JPAQuery(em);
    List<Member> members =
            query.from(member)
                    .where(member.name.eq("회원1"))
                    .orderBy(member.name.desc())
                    .list(member);
}
```

검색 조건 쿼리

QueryDSL의 기본 쿼리 기능을 알아보자.

```
JPAQuery query = new JPAQuery(em);
QItem item = QItem.item;
List<Item> list = query.from(item)
                .where(item.name.eq("좋은상품").and(item.price.gt(20000)))
                .list(item); //조회할 프로젝션 지정
```

====== 실행된 JPQL ======

```
select item
from Item item
where item.name = ?1 and item.price > ?2
```

where 절에는 `and` 나 `or`을 사용할 수 있다. 또한 다음처럼 여러 검색 조건을 사용해도 된다. 이때는 `and` 연산이 된다.

```
.where(item.name.eq("좋은상품"), item.price.gt(20000))
```

쿼리 타입의 필드는 필요한 대부분의 메서드를 명시적으로 제공한다. 몇가지만 예를 들어 보자. 다음은 `where()`에서 사용되는 메서드다.

```
item.price.between(10000,20000);//가격이 10000원 ~ 20000원 상품
item.name.contains("상품1");      //상품1 이라는 이름을 포함한 상품
item.name.startsWith("고급");     //이름이 고급으로 시작하는 상품
```

코드로 작성되어 있으므로 IDE가 제공하는 코드 자동 완성 기능의 도움을 받으면 필요한 메서드를 손쉽게 찾을 수 있다.

결과 조회

쿼리 작성이 끝나고 결과 조회 메서드를 호출하면 실제 데이터베이스를 조회한다. 보통 `uniqueResult()`나 `list()`를 사용하고 파라미터로 프로젝션 대상을 넘겨준다. 결과 조회 API는 `com.mysema.query.Projectable`에 정의되어 있다.

대표적인 결과 조회 메서드는 다음과 같다.

- `uniqueResult()` : 조회 결과가 단건일 때 사용한다. 조회 결과가 없으면 `null`을 반환하고 결과가 하나 이상이면 `com.mysema.query.NonUniqueResultException` 예외가 발생한다.
- `singleResult()` : `uniqueResult()`와 같지만 결과가 하나 이상이면 처음 데이터를 반환한다.
- `list()` : 결과가 하나 이상일 때 사용한다. 결과가 없으면 빈 컬렉션을 반환한다.

페이징과 정렬

```
QItem item = QItem.item;

query.from(item)
     .where(item.price.gt(20000))
     .orderBy(item.price.desc(), item.stockQuantity.asc())
     .offset(10).limit(20)
     .list(item);
```

정렬은 `orderBy`를 사용하는데 쿼리타입(Q)이 제공하는 `asc()`, `desc()`를 사용한다. 페이징은 `offset`과 `limit`을 적절히 조합해서 사용하면 된다.

페이징은 다음처럼 `restrict()` 메서드에 `com.mysema.query.QueryModifiers`를 파라미터로 사용해도 된다.

```
QueryModifiers queryModifiers = new QueryModifiers(20L, 10L); //limit, offset
List<Item> list =
        query.from(item)
        .restrict(queryModifiers)
        .list(item);
```

실제 페이징 처리를 하려면 검색된 전체 데이터 수를 알아야 한다. 이때는 `list()` 대신에 `listResults()`를 사용한다.

```
SearchResults<Item> result =
        query.from(item)
                .where(item.price.gt(10000))
                .offset(10).limit(20)
                .listResults(item);

long total = result.getTotal(); //검색된 전체 데이터 수
long limit = result.getLimit();
long offset = result.getOffset();
List<Item> results = result.getResults(); //조회된 데이터
```

`listResults()`를 사용하면 전체 데이터 조회를 위한 count 쿼리를 한 번 더 실행한다. 그리고 `SearchResults`를 반환하는데 이 객체에서 전체 데이터 수를 조회할 수 있다.

그룹

그룹은 `groupBy`를 사용하고 그룹화된 결과를 제한하려면 `having`을 사용하면 된다.

```
query.from(item)
        .groupBy(item.price)
        .having(item.price.gt(1000))
        .list(item);
```

조인(Join)

조인은 `innerJoin(join)`, `leftJoin`, `rightJoin`, `fullJoin`을 사용할 수 있고 추가로 JPQL의 `on`과 성능 최적화를 위한 `fetch` 조인도 사용할 수 있다.

조인의 기본 문법은 첫 번째 파라미터에 조인 대상을 지정하고, 두 번째 파라미터에 별칭(alias)으로 사용할 쿼리 타입을 지정하면 된다.

```
join(조인 대상, 별칭으로 사용할 쿼리 타입)
```

====== 기본 조인 ======

```
QOrder order = QOrder.order;
QMember member = QMember.member;
QOrderItem orderItem = QOrderItem.orderItem;

query.from(order)
        .join(order.member, member)
        .leftJoin(order.orderItems, orderItem)
        .list(order);
```

====== on 사용 ======

```
query.from(order)
        .leftJoin(order.orderItems, orderItem)
            .on(orderItem.count.gt(2))
        .list(order);
```

====== 페치 조인 사용 ======

```
query.from(order)
        .innerJoin(order.member, member).fetch()
        .leftJoin(order.orderItems, orderItem).fetch()
        .list(order);
```

====== from 절에 여러 조건 사용 ======

```
QOrder order = QOrder.order;
QMember member = QMember.member;
```

```
query.from(order, member)
        .where(order.member.eq(member))
        .list(order);
```

서브 쿼리

서브 쿼리는 `com.mysema.query.jpa.JPASubQuery`를 생성해서 사용한다. 서브 쿼리의 결과가 하나면 `unique()`, 여러 건이면 `list()`를 사용할 수 있다.

====== 서브 쿼리 예제 - 하나 ======

```
QItem item = QItem.item;
QItem itemSub = new QItem("itemSub");

query.from(item)
    .where(item.price.eq(
        new JPASubQuery().from(itemSub).unique(itemSub.price.max())
    ))
    .list(item);
```

====== 서브 쿼리 예제 - 여러 건 ======

```
QItem item = QItem.item;
QItem itemSub = new QItem("itemSub");

query.from(item)
    .where(item.in(
        new JPASubQuery().from(itemSub)
                .where(item.name.eq(itemSub.name))
                .list(itemSub)
    ))
    .list(item);
```

프로젝션과 결과 반환

`select` 절에 조회 대상을 지정하는 것을 프로젝션이라 한다.

- 프로젝션 대상이 하나

프로젝션 대상이 하나면 해당 타입으로 반환한다.

```
QItem item = QItem.item;
List<String> result = query.from(item).list(item.name);

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

- 여러 컬럼 반환과 튜플

프로젝션 대상으로 여러 필드를 선택하면 QueryDSL은 기본으로 `com.mysema.query.Tuple` 이라는 `Map` 과 비슷한 내부 타입을 사용한다. 조회 결과는 `tuple.get()` 메서드에 조회한 쿼리 타입을 지정하면 된다.

====== 튜블 사용 예제 ======

```
QItem item = QItem.item;

List<Tuple> result = query.from(item).list(item.name, item.price);
//List<Tuple> result = query.from(item).list(new QTuple(item.name, item.price));

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

- 빈 생성(Bean population)

쿼리 결과를 엔티티가 아닌 특정 객체로 받고 싶으면 빈 생성 기능을 사용한다. QueryDSL은 객체를 생성하는 다양한 방법을 제공한다.

다양한 빈 생성 기능

- 프로퍼티 접근
- 필드 직접 접근
- 생성자 사용

원하는 방법을 지정하기 위해 `com.mysema.query.types.Projections` 를 사용하면 된다. 다양한 방법으로 다음 `ItemDTO` 에 값을 채워보자.

====== 예제 `ItemDTO` ======

```
public class ItemDTO {

    private String username;
    private int price;

    public ItemDTO() {}

    public ItemDTO(String username, int price) {
        this.username = username;
        this.price = price;
    }

    //Getter, Setter
    public String getUsername() {...}
    public void setUsername(String username) {...}
    public int getPrice() {...}
    public void setPrice(int price) {...}
}
```

====== 프로퍼티 접근(Setter) ======

```
QItem item = QItem.item;
List<ItemDTO> result = query.from(item).list(
        Projections.bean(ItemDTO.class, item.name.as("username"), item.price));
```

`Projections.bean()` 메서드는 수정자(`Setter`)를 사용해서 값을 채운다. 예제를 보면 쿼리 결과는 `name` 인데 `ItemDTO` 는 `username` 프로퍼티를 가지고 있다. 이처럼 쿼리 결과와 매핑할 프로퍼티 이름이 다르면 `as` 를 사용해서 별칭을 주면 된다.

====== 필드 직접 접근 ======

```
QItem item = QItem.item;
List<ItemDTO> result = query.from(item).list(
        Projections.fields(ItemDTO.class, item.name.as("username"), item.price))
```

`Projections.fields()` 메서드를 사용하면 필드에 직접 접근해서 값을 채워준다. 필드를 `private` 으로 설정해도 값을 채운다.

====== 생성자 사용 ======

```
QItem item = QItem.item;
List<ItemDTO> result = query.from(item).list(
        Projections.constructor(ItemDTO.class, item.name, item.price)
);
```

`Projections.constructor()` 메서드는 생성자를 사용할 수 있다. 물론 지정한 프로젝션과 파라미터 순서가 같은 생성자가 필요하다.

- DISTINCT

`distinct`는 다음과 같이 사용한다.

```
query.distinct().from(item)...
```

수정, 삭제 배치 쿼리

QueryDSL도 수정, 삭제 같은 배치 쿼리를 지원한다. JPQL 배치 쿼리와 같이 영속성 컨텍스트를 무시하고 데이터베이스를 직접 쿼리한다는 점에 유의하자.

====== 수정 배치 쿼리 ======

```
QItem item = QItem.item;
JPAUpdateClause updateClause = new JPAUpdateClause(em, item);
long count = updateClause.where(item.name.eq("시골개발자의 JPA 책"))
        .set(item.price, item.price.add(100))
        .execute();
```

수정 배치 쿼리는 `com.mysema.query.jpa.impl.JPAUpdateClause`를 사용한다. 예제는 상품의 가격을 100원 증가시킨다.

====== 삭제 배치 쿼리 ======

```
QItem item = QItem.item;
JPADeleteClause deleteClause = new JPADeleteClause(em, item);
long count = deleteClause.where(item.name.eq("시골개발자의 JPA 책"))
        .execute();
```

삭제 배치 쿼리는 `com.mysema.query.jpa.impl.JPADeleteClause`를 사용한다. 예제는 이름이 같은 상품을 삭제한다.

동적 쿼리

`com.mysema.query.BooleanBuilder`를 사용하면 특정 조건에 따른 동적 쿼리를 편리하게 생성할 수 있다.

====== 동적 쿼리 예제 ======

```
SearchParam param = new SearchParam();
param.setName("시골개발자");
param.setPrice(10000);

QItem item = QItem.item;

BooleanBuilder builder = new BooleanBuilder();
if (StringUtils.hasText(param.getName())) {
    builder.and(item.name.contains(param.getName()));
}
if (param.getPrice() != null) {
    builder.and(item.price.gt(param.getPrice()));
}
List<Item> result = query.from(item)
        .where(builder)
        .list(item);
```

예제는 상품 이름과 가격 유무에 따라 동적으로 쿼리를 생성한다.

메서드 위임(Delegate methods)

메서드 위임 기능을 사용하면 쿼리 타입에 검색 조건을 직접 정의할 수 있다. 예제를 보자.

====== 검색 조건 정의 ======

```
public class ItemExpression {

    @QueryDelegate(Item.class)
    public static BooleanExpression isExpensive(QItem item, Integer price) {
        return item.price.gt(price);
```

```
    }
}
```

메서드 위임 기능을 사용하려면 우선 정적(static) 메서드를 만들고 `@com.mysema.query.annotations.QueryDelegate` 어노테이션에 속성으로 이 기능을 적용할 엔티티를 지정한다. 정적 메서드의 첫번째 파라미터에는 대상 엔티티의 쿼리 타입(Q)을 지정하고 나머지는 필요한 파라미터를 정의한다.

====== 쿼리 타입에 생성된 결과 ======

```
public class QItem extends EntityPathBase<Item> {
    ...
    public com.mysema.query.types.expr.BooleanExpression isExpensive(Integer pri
        return ItemExpression.isExpensive(this, price);
    }
}
```

쿼리 타입을 보면 기능이 추가된 것을 확인할 수 있다. 이제 메서드 위임 기능을 사용해보자.

====== 메서드 위임 기능 사용 ======

```
query.from(item).where(item.isExpensive(30000)).list(item);
```

필요하다면 `String`, `Date` 같은 자바 기본 내장 타입에도 메서드 위임 기능을 사용할 수 있다.

```
@QueryDelegate(String.class)
public static BooleanExpression isHelloStart(StringPath stringPath) {
    return stringPath.startsWith("Hello");
}
```

정리

나는 JPA를 사용하면서 두 가지 고민이 있었는데 문자가 아닌 코드로 안전하게 쿼리를 작성하고 싶다는 것과 복잡한 동적 쿼리를 어떻게 해결해야 하는가였다. JPA Criteria가 이런 고민을 해결해 주기는 했지만, 막상 사용해보면 너무 복잡해서 JPQL을 직접 사용하고 싶어진다. 반면에 QueryDSL은 두 가지를 모두 만족하면서 쉽고 단순하다. 앞으로 JPA3 명세가 나온다면 JPA Criteria가 QueryDSL 같이 사용하기 쉽고 깔끔하게 다시 설계되었으면 하는 바람이다.