QueryDSL 사용 매뉴얼

1. QueryDSL 설치

1) build.gradle파일에 의존성 추가

[적용전]

```
plugins {
	id 'java'
	id 'org.springframework.boot' version '3.2.3'
	id 'io.spring.dependency-management' version '1.1.4'
}

group = 'com.example'
version = '0.0.1-SNAPSHOT'

java {
	sourceCompatibility = '17'
}

configurations {
	compileOnly {
		extendsFrom annotationProcessor
	}
}

repositories {
	mavenCentral()
}

dependencies {

	implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
	implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
	implementation 'org.springframework.boot:spring-boot-starter-web'

	compileOnly 'org.projectlombok:lombok'
	developmentOnly 'org.springframework.boot:spring-boot-devtools'
	runtimeOnly 'com.mysql:mysql-connector-j'
	annotationProcessor 'org.projectlombok:lombok'

	testImplementation 'org.springframework.boot:spring-boot-starter-test'

}

tasks.named('test') {
	useJUnitPlatform()
}
```

[적용후]

```
// (추가)
buildscript {
	ext {
		queryDslVersion = "5.0.0"
	}
}

plugins {
	id 'java'
	id 'org.springframework.boot' version '3.2.3'
	id 'io.spring.dependency-management' version '1.1.4'
	// (추가)
	id "com.ewerk.gradle.plugins.querydsl" version "1.0.10"
}

group = 'com.example'
version = '0.0.1-SNAPSHOT'

java {
	sourceCompatibility = '17'
}

configurations {
	compileOnly {
		extendsFrom annotationProcessor
	}
}

repositories {
	mavenCentral()
}

dependencies {

	implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
	implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
	implementation 'org.springframework.boot:spring-boot-starter-web'

	//(추가)
	implementation "com.querydsl:querydsl-core:5.0.0"
	implementation "com.querydsl:querydsl-collections"
	implementation "com.querydsl:querydsl-jpa:5.0.0:jakarta"


	compileOnly 'org.projectlombok:lombok'
	developmentOnly 'org.springframework.boot:spring-boot-devtools'
	runtimeOnly 'com.mysql:mysql-connector-j'
	annotationProcessor 'org.projectlombok:lombok'
	// (추가)
	annotationProcessor "com.querydsl:querydsl-apt:5.0.0:jakarta"
	annotationProcessor "jakarta.annotation:jakarta.annotation-api"
	annotationProcessor "jakarta.persistence:jakarta.persistence-api:3.1.0"

	testImplementation 'org.springframework.boot:spring-boot-starter-test'
	// (추가)
	testImplementation 'jakarta.persistence:jakarta.persistence-api'
	testImplementation 'com.querydsl:querydsl-jpa:5.0.0:jakarta'


}

tasks.named('test') {
	useJUnitPlatform()
}
```

```
// (추가)
def querydslDir = "src/main/generated/querydsl"

querydsl {
        jpa = true
        querydslSourcesDir = querydslDir
}

sourceSets {
        main.java.srcDir querydslDir
}

compileQuerydsl{
        options.annotationProcessorPath = configurations.querydsl
}

configurations {
        compileOnly {
                extendsFrom annotationProcessor
        }
        querydsl.extendsFrom compileClasspath
}
```

- refresh build project를 실행하여
  src/main/genreated/querydsl경로에 Q클래스들이 생성되어있는지 확인한다.

- 위 경로에 Q클래스들이 생성되어있지 않다면
  > 윈도우 cmd를 관리자 권한으로 실행한다.
  > cd 명령어를 사용하여 프로젝트경로로 이동한다
    (예시 cd C:\users\user\git\20_spring_jpa)
  > gradlew build 명령어를 실행한다.

- 다시 이클립스로 돌아와 프로젝트이름에서 refresh를 실행하여 Q클래스들이 생성되어 있는지를 확인한다.

2) QueryDSL Config클래스생성

```
@Configuration
public class QueryDslConfig {

    @PersistenceContext
    private EntityManager em

    @Bean
    public JPAQueryFactory jpaQueryFactory() {
        return new JPAQueryFactory(em);
    }

}
```

3) ~~~RepositoryCustom인터페이스와 인터페이스를 구현한 ~~~RepositoryCustomImpl 클래스생성

```
public interface ~~~RepositoryCustom {

        public List<Product> customMethod()

}
```

```
@RequiredArgsConstructor
@Repository
public class ~~~RepositoryCustomImpl implements ~~~RepositoryCustom {

        private final JPAQueryFactory queryFactory

        @Override
        public List<Product> basicEx() {
                return queryFactory.selectFrom(product).fetch();
        }

}
```

4) ~~~Repository인터페이스에 다중상속후 querydsl 사용

```
@Repository
public interface ~~~Repository extends JpaRepository<Product, Long> ,
~~~RepositoryCustom {

}
```

2. QueryDSL 제공 메서드

- QueryDSL은 복잡한쿼리를 타입세이프하게 작성할 수 있게 해주는 강력한프레임워크이다.
- 여기에는 다양한 메서드들이 있어 쿼리를 다양한 방식으로 구성할 수 있게 해준다.

1) 선택(Selection)

select(...): 지정된 경로 또는 식을 기반으로 결과를 선택한다.
selectFrom(...): FROM 절에서 엔티티를 선택하고, 선택된 엔티티를 반환한다.

2) 조인(Join)

join(...), leftJoin(...), rightJoin(...): 지정된 엔티티 또는 식에대한 조인을 수행한다.
on(...): 조인 조건을 지정한다.

3) 필터링(Filtering)

where(...): 쿼리에 조건을 추가한다. 여러조건을 체이닝해서 복잡한 조건을 구성 할 수 있다.

4) 집계(Aggregation)

groupBy(...): 지정된 경로를 기준으로 결과를 그룹화한다.
having(...): 그룹화된 결과에 대한 조건을 지정한다.
집계함수: sum(), avg(), min(), max(), count() 등은 집계를 수행한다.

5) 정렬(Ordering)

orderBy(...): 결과를 지정된경로 또는 식을기반으로 정렬한다.
asc(), desc(): 정렬순서를 지정한다.

6) 결과처리(Result Fetching)

fetch(): 쿼리결과를 리스트로 반환한다.
fetchOne(): 쿼리결과의 첫 번째 항목을 반환한다. 결과가 없거나 둘 이상이면 예외를 발생시킨다.
fetchFirst(): limit(1).fetchOne()의축약형으로, 결과의 첫 번째 항목만 반환한다.
fetchResults(): 쿼리결과와 전체행수를 포함하는 QueryResults 객체를 반환한다. (페이징처리에유용)
fetchCount(): 쿼리에 해당하는 행의수를 반환한다.

7) 페이징및제한(Paging and Limiting)
offset(...): 결과세트의 시작점을 지정한다.
limit(...): 반환할 최대 결과수를 지정한다.

8) 서브쿼리(Subquery)
JPAExpressions.select(...): 서브쿼리를 생성하기위해 사용한다.

9) 비교연산자
eq(value): 값이 같은 경우(equals)
ne(value): 값이 다른 경우(not equals)
lt(value): 값보다 작은 경우(less than)
loe(value): 값보다 작거나 같은 경우(less than or equal to)
gt(value): 값보다 큰경우(greater than)
goe(value): 값보다크거나 같은경우(greater than or equal to)
between(start, end): 값이두값 사이에있는경우

10) 논리연산자
and(expression): 두조건이모두 참인경우
or(expression): 둘중하나의 조건이참인경우
not(expression): 조건이거짓인경우
BooleanExpression 객체를사용하여체인으로연결하여복잡한논리구조를만들수도있다. 예: .where(condition1.and(condition2).or(condition3))

11) 문자열연산

startsWith(value): 지정된 값으로 시작하는 경우
endsWith(value): 지정된 값으로 끝나는 경우
contains(value): 지정된 값을 포함하는 경우
like(pattern): 지정된 패턴과 일치하는 경우(SQL의 LIKE와 유사)
matches(regex): 정규표현식과 일치하는 경우

12) 컬렉션연산

isEmpty(): 컬렉션이 비어있는 경우
isNotEmpty(): 컬렉션이 비어있지 않은 경우
size(): 컬렉션의크기(길이)를 비교하는데 사용 될 수 있다.

13) 기타

isNull(): 값이 null인 경우
isNotNull(): 값이 null이아닌 경우
in(values): 값이 주어진 목록에 포함되어 있는 경우
notIn(values): 값이 주어진 목록에 포함되어 있지않은 경우

[참고] Tuple

Tuple은 여러 타입의 객체를 포함할 수 있는 컨테이너이다. QueryDSL에서는 주로 선택된 컬럼들의 값이나, 다른 타입의 결과들을 하나의 객체로 묶어서 반환할 때 사용된다.
Tuple을 사용하면, 각 컬럼 또는 표현식에 대한 결과를 인덱스나 타입 안전한 방법으로 접근할 수 있다.

QueryDSL에서 Tuple을 반환하는 이유는 쿼리의 결과로 여러 타입의 값을 포함하는 복합 결과를 효율적으로 다루기 위함이다. Tuple은 QueryDSL이 제공하는 타입으로, 다른 엔티티나 값을 그룹화해서 반환할 때 유용하다. 각각의 Tuple 인스턴스는 쿼리 결과의 단일 행을 나타내며, 이 행에는 여러 컬럼의 값이 포함될 수 있다.

Tuple은 QueryDSL 특유의 타입이므로, 다른 계층으로 결과를 전달할 때는 DTO(Data Transfer Object)와 같은 애플리케이션의 표준 객체로 변환하는 것이 좋다. 이렇게 함으로써 서비스 계층이나 컨트롤러 계층이 QueryDSL에 의존하지 않도록 할 수 있다.
QueryDSL의 Tuple은 복합 쿼리 결과를 다룰 때 강력하고 유연한 도구를 제공한다. 단, Tuple을 사용할 때는 애플리케이션의 다른 부분과의 결합도를 낮추기 위한 추가적인 고려가 필요하다.

[예시]

```
QPerson person = QPerson.person;
List<Tuple> results = queryFactory
    .select(person.name, person.age)
    .from(person)
    .fetch();

for (Tuple tuple : results) {
    String name = tuple.get(person.name);
    Integer age = tuple.get(person.age);
    System.out.println("Name: " + name + ", Age: " + age);
}
```