jpa-querydsl

▼ Reason to use repository

To hide implementation technology (like querydsl) underneath.

when implementation technology is changed or removed, only repository needs to be modified.

▼ Concept and advantage

jpql builder

▼ Advantage

easy to write complicate query and dynamic query

writing query in java code

finding expression error at compile time

sql-like grammar

automatic prepareStatement creation and parameter binding. (It’s important due to sql injection)

▼ configuration

▼ build.gradle

▼ Spring Boot 2.6 and later

1. Add buildscript

```
// querydsl
buildscript {
  ext {
    queryDslVersion = "5.0.0"
  }
}
```

2. Add plugin

```
plugins {
  ...
  id "com.ewerk.gradle.plugins.querydsl" version "1.0.10" // querydsl
  ...
}
```

3. Add library

```
dependencies {
  ...
  // querydsl
  implementation "com.querydsl:querydsl-jpa:${queryDslVersion}"
  implementation "com.querydsl:querydsl-apt:${queryDslVersion}"
  ...
}
```

4. Add definition

   Detail ref → 라이브러리 살펴보기 4:45

```
//querydsl begin -----
def querydslDir = "$buildDir/generated/querydsl"

querydsl {
    jpa = true
    querydslSourcesDir = querydslDir // Q file directory
}
sourceSets {
    main.java.srcDir querydslDir
}
configurations {
    // generate class path on compile
    querydsl.extendsFrom compileClasspath
}
compileQuerydsl {
    // annotation processsor generate Q files based on annotations.
    options.annotationProcessorPath = configurations.querydsl
}
//querydsl end -----
```

▼ Earlier

1. Add plugin

```
plugins {
  ...
  id "com.ewerk.gradle.plugins.querydsl" version "1.0.10" // querydsl
  ...
}
```

2. Add library

```
dependencies {
  ...
  implementation 'com.querydsl:querydsl-jpa' //querydsl
  ...
}
```

3. Add definition

```
//querydsl begin -----
def querydslDir = "$buildDir/generated/querydsl"

querydsl {
    jpa = true
    querydslSourcesDir = querydslDir
}
sourceSets {
    main.java.srcDir querydslDir
}
configurations {
    querydsl.extendsFrom compileClasspath
}
compileQuerydsl {
```

```
    options.annotationProcessorPath = configurations.querydsl
}
//querydsl end -----
```

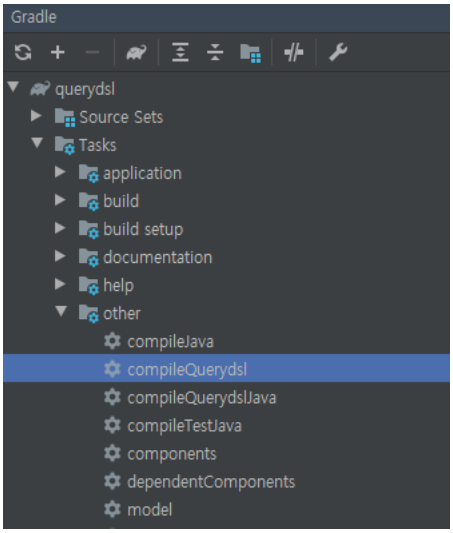
▼ ~~Question title : compileQuerydsl error~~

▼ Solution

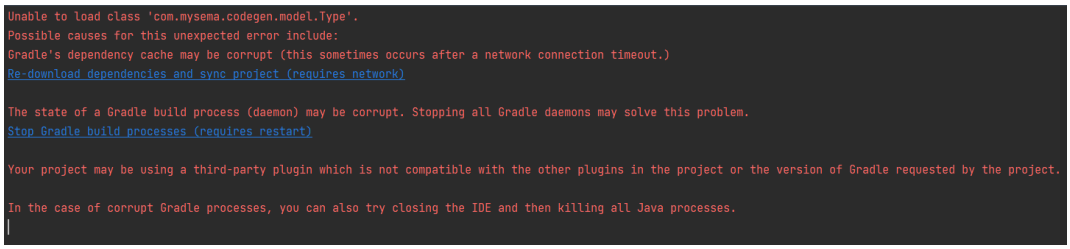
1. Regenerate project
2. follow the steps above

▼ Error

compileQuerydsl을 더블 클릭하여 컴파일 진행할 경우



아래와 같은 에러가 발생합니다.




```
Unable to load class 'com.mysema.codegen.model.Type'.
Possible causes for this unexpected error include:
Gradle's dependency cache may be corrupt (this sometimes occurs after a network connection timeout.)
Re-download dependencies and sync project (requires network)

The state of a Gradle build process (daemon) may be corrupt. Stopping all Gradle daemons may solve this problem.
Stop Gradle build processes (requires restart)

Your project may be using a third-party plugin which is not compatible with the other plugins in the project or the ve

In the case of corrupt Gradle processes, you can also try closing the IDE and then killing all Java processes.
```

하단 링크는 이미 확인하여 적용해 보았으니 해결되지 않았기에 문의드립니다.

https://www.inflearn.com/questions/149157

추가적으로 현시점 생성한 프로젝트 스펙을 참고용으로 첨부 드립니다.

▼ version

▼ build.gradle

```
plugins {
  id 'org.springframework.boot' version '2.6.7'
  id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id "com.ewerk.gradle.plugins.querydsl" version "1.0.10" // querydsl
  id 'java'
}

group = 'study'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

configurations {
  compileOnly {
    extendsFrom annotationProcessor
  }
}

repositories {
  mavenCentral()
}

dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
  implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'com.querydsl:querydsl-jpa' //querydsl
  compileOnly 'org.projectlombok:lombok'
  runtimeOnly 'com.h2database:h2'
  annotationProcessor 'org.projectlombok:lombok'
  testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

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

//querydsl begin -----
def querydslDir = "$buildDir/generated/querydsl"

querydsl {
    jpa = true
    querydslSourcesDir = querydslDir
}
sourceSets {
    main.java.srcDir querydslDir
}
configurations {
    querydsl.extendsFrom compileClasspath
}
compileQuerydsl {
    options.annotationProcessorPath = configurations.querydsl
}
//querydsl end -----
```

▼ gradle-wrapper.properties

```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
```

```
#distributionUrl=https\://services.gradle.org/distributions/gradle-7.4-all.zip # 오류 발생
distributionUrl=https\://services.gradle.org/distributions/gradle-6.8.3-bin.zip

zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
```

spring boot 2.6.7 을 초기 생성할 경우 gradle-7.4-all 로 설정되지만

확인한 바로는 제가 사용중인 IntelliJ 19.2 Ultimate과 호환이 되지 않는 것 같아 6.8.3-bin으로 수정하여 프로젝트를 실행하였습니다.

수정 전 오류는 아래와 같으며 해당 오류 관련하여 디테일한 내용일 필요할 경우 https://raejin.tistory.com/172 확인 부탁드립니다.

```
Unable to find method 'org.codehaus.groovy.runtime.StringGroovyMethods.capitalize(Ljava/lang/String;)Ljava/lang/String
Possible causes for this unexpected error include:
Gradle's dependency cache may be corrupt (this sometimes occurs after a network connection timeout.)
Re-download dependencies and sync project (requires network)

The state of a Gradle build process (daemon) may be corrupt. Stopping all Gradle daemons may solve this problem.
Stop Gradle build processes (requires restart)

Your project may be using a third-party plugin which is not compatible with the other plugins in the project or the ve

In the case of corrupt Gradle processes, you can also try closing the IDE and then killing all Java processes.
```

▼ yml

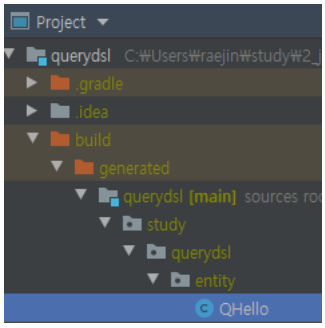
```
spring:
  datasource:
    url: jdbc:h2:tcp://localhost/~/querydsl
    username: sa
    password:
    driver-class-name: org.h2.Driver

  jpa:
    hibernate:
      ddl-auto: create            # drop and create table when application loaded. only in developing stage
    properties:
      hibernate:
        #show_sql: true           # print in console (System.out.println) -> use hibernate.SQL
        format_sql: true
        use_sql_comments: true  # print jpql

logging.level:
  org.hibernate.SQL: debug        # leave as log
  #org.hibernate.type: trace      # To check parameter, it's beeter to use p6spy library.
```

- use_sql_comment: true

```
2022-04-26 17:32:40.068 DEBUG 19476 --- [           main] org.hibernate.SQL
    /* select
        member1
    from
        Member member1
    where
        member1.username = ?1 */ select
            member0_.id as id1_1_,
            member0_.age as age2_1_,
            member0_.team_id as team_id4_1_,
            member0_.username as username3_1_
        from
            member member0_
        where
            member0_.username=?
```

▼ Qtype

qeurydsl generate and use Qtype corresponding to entity



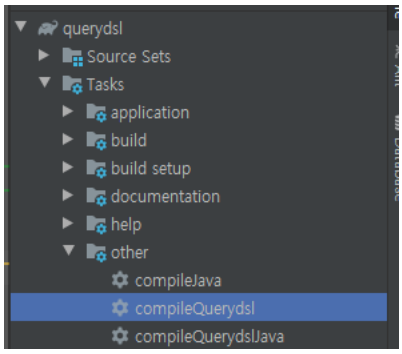
Path is defined in build.gradle

```
def querydslDir = "$buildDir/generated/querydsl"
```

▼ Compile

1. on build
2. manual compile

   double click ‘compileQuerydsl



▼ Library

- Core

  Spring MVC

  JPA & Hibernate

  Spring Data JPA

  Querydsl

- Others

  H2 database client

  connection pool → HikariCP

  Logging → SLF4J & LogBack

  Test

- p6spy (ref.)

  In operation, performance test is needed as it may affected

```
implementation 'com.github.gavlyukovskiy:p6spy-spring-boot-starter:1.5.7'
```

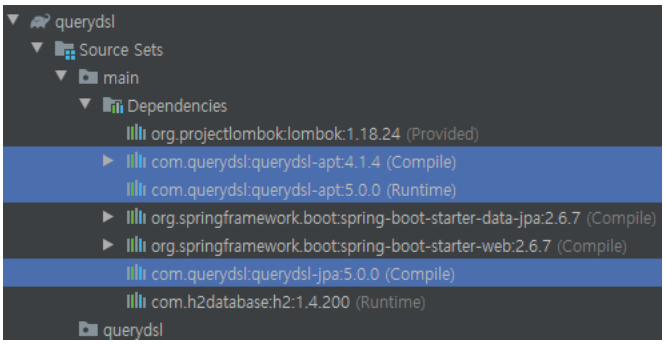
▼ querydsl

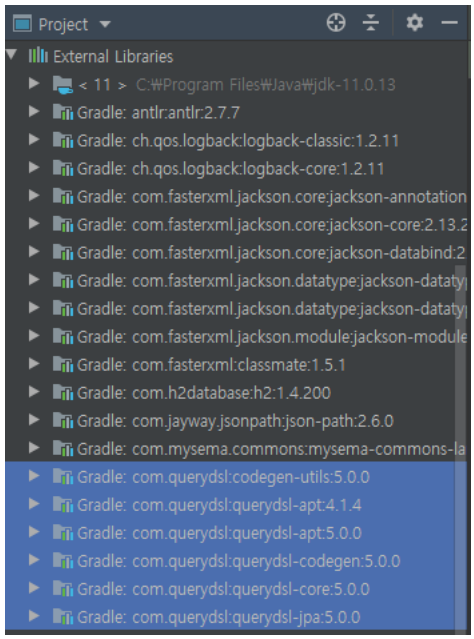
- querydsl-apt

  To generate Qtype

- querydsl-jpa / core

  For writing codes





▼ starter-web

tomcat

webmvc

▼ data-jpa

- hibernate
- jdbc
- transaction
- hikariCP

  Database connection pooling
- aop/starter/logging

  slf4: interface

  logback : implementer

▼ test

assertj

mockito

▼ JPAQueryFactory

Choose on your preference

1. make instance and inject

   easier for testing

```
@Repository
public class MemberJpaRepository {

  private final EntityManager em;
  private final JPAQueryFactory queryFactory;

  // Spring inject the argument
  public MemberJpaRepository(EntityManager em) {
    this.em = em;
    this.queryFactory = new JPAQueryFactory(em);
  }
}
```

2. Bean registration

   - Advantage

     lombok @RequiredArgsConstructor is applicable

   - Disadvantage

     a bit troublesome when testing due to injection

   [NOTE]

   Even thought it is a singleton, there are no concurrency problem.

   Because spring binds it by each transaction

```
package study.querydsl;

@SpringBootApplication
public class QuerydslApplication {

  public static void main(String[] args) {
    SpringApplication.run(QuerydslApplication.class, args);
  }

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

```
@Repository
public class MemberJpaRepository {

  private final EntityManager em;
  private final JPAQueryFactory queryFactory;

  public MemberJpaRepository(EntityManager em, JPAQueryFactory queryFactory) {
```

```
    this.em = em;
    this.queryFactory = queryFactory;
  }
}
```

OR

```
private final EntityManager em;

public MemberRepositoryImpl(JPAQueryFactory queryFactory) {
  this.queryFactory = queryFactory;
}
```

▼ Dynamic query

if there are no conditions, all data are fetched. → much data, much wasting

dynamic query should have some conditions or paging query.

▼ Grammar - basic

▼ JPAQueryFactory

use in field level is recommended

EntityManager that spring injects, is designed to be no concurrency problem.

→ no problem in multi-threaded form.

Even if multi threads come in, they are distributed to be bound to corresponding transaction depending on where the transaction is.

```
@SpringBootTest
@Transactional
public class QuerydslBasicTest {

  @Autowired EntityManager em;

  JPAQueryFactory queryFactory;

  @BeforeEach
  public void becore() {
    queryFactory = new JPAQueryFactory(em);
    ...
  }

  @Test
  public void startQuerydsl() {
    //JPAQueryFactory queryFactory = new JPAQueryFactory(em);

    QMember m = new QMember("m");

    Member findMember = queryFactory
        .select(m)
        .from(m)
        .where(m.username.eq("member1")) // make preparedStatement and bind parameter
        .fetchOne();

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

▼ Q type

1. default instance with static import (recommended)

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

@Test
public void startQuerydsl() {
  Member findMember = queryFactory
      .select(member)
      .from(member)
      .where(member.username.eq("member1")) // make preparedStatement and bind parameter
      .fetchOne();

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

2. manual instance

   Used when same tables need to be joined.

   parameter used as alias in jpql.

   ```
   QMember member = new QMember("m");
   ```

3. default instance with variable

   ```
   QMember member = QMember.member;
   ```

   ```
   @Generated("com.querydsl.codegen.DefaultEntitySerializer")
   public class QMember extends EntityPathBase<Member> {
       ...
       public static final QMember member = new QMember("member1");
   ```

▼ Search condition query

▼ provides all the conditions provided by JPQL

```
member.username.eq("member1") // username = 'member1'
member.username.ne("member1") //username != 'member1'
member.username.eq("member1").not() // username != 'member1'

member.username.isNotNull() //이름이 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) //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%") //like 검색
member.username.contains("member") // like ‘%member%’ 검색
member.username.startsWith("member") //like ‘member%’ 검색
...
```

▼ and

1. ,

```
Member findMember = queryFactory
    .selectFrom(member)
    .where(
        member.username.eq("member1"),
        member.age.eq(10))
    .fetchOne();
```

2. and()

```
Member findMember = queryFactory
    .selectFrom(member)
    .where(
        member.username.eq("member1").and(
        member.age.eq(10))
    )
    .fetchOne();
```

▼ fetch types for results

method	return	Exception/note
fetch()	list empty list if empty	
fetchOne()	single result null if no result exception if over 1	com.querydsl.core .NonUniqueResultException
fetchFirst()	limit(1).fetchOne()	
~~fetchResults ()~~	~~result with paging info~~	~~deprecated additional total count query dispatched.~~
~~fetchCount()~~	~~count~~	~~deprecated~~

count()

simple is the best for count query.

many conditions, bad search performance.

```
queryFactory
 //.select(Wildcard.count) //select count(*)
 .select(member.count()) //select count(member.id)
 .from(member)
 .fetchOne();
```

▼ sort

desc(), asc()

nullsLast(), nullsFirst()

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

▼ paging

offset(), limit()

count() → refer to ‘fetch types for results’

▼ grouping

▼ aggregation methods()

count(), sum(), avg(), max(), min()

- return type tuple

  when return types diverse, data return tuple

  not used much, better get in DTO

  details in ‘projection and return‘ below

```
@Test
  public void aggregation() { // 집합, 통계
    List<Tuple> result = queryFactory
        .select(
            member.count(),
            member.age.sum(),
            member.age.avg(),
            member.age.max(),
            member.age.min()
            )
        .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);
  }
```

▼ groupBy()

having() : conditions for groupBy

```
/**
   * get team name and average age of each team
   */
  @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)
        .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); // (10 + 20 ) / 2

    assertThat(teamB.get(team.name)).isEqualTo("teamB");
    assertThat(teamB.get(member.age.avg())).isEqualTo(35); // (30 + 40 ) / 2
  }
```

▼ Join

▼ basic

- inner join

  join() = innerJoin()

- outer join

  leftJoin() - used a lot

  outerJoin()

▼ theta join

join unrelated fields

enumerate needed tables in from clause.

filter in where clause

NOTE

outer join impossible → use join on

```
/** theta join
   * find memberName == teamName
   */
  @Test
  public void theta_join() {
    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");
  }
```

▼ on

▼ 1. Filtering join target

- inner join

  USE where clause. → better recognition

```
List<Tuple> result = queryFactory
        .select(member, team)
        .from(member)
        .join(member.team, team)
        .where(team.name.eq("teamA"))
        .fetch();
```

- outer join (leftJoin)

```
List<Tuple> result = queryFactory
        .select(member, team)
        .from(member)
        .leftJoin(member.team, team)
        .on(team.name.eq("teamA"))
        .fetch();
```

▼ 2. Joining entities with no relations

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

▼ fetchJoin()

fetch related entity together

`add fetchJoin() after join()`

```
@PersistenceUnit EntityManagerFactory emf;
@Test
  public void fetchJoinUse(){
    em.flush();
    em.clear();

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

    boolean loaded = emf.getPersistenceUnitUtil().isLoaded(findMember.getTeam());
    assertThat(loaded).as("fetch join not applied").isTrue();
  }
```

▼ subquery

import static com.querydsl.jpa.JPAExpressions.*

make instance manually if same table needs to be used in subquery.

▼ [NOTE] inline view(subquery from clause) is provided.

- Solution steps
  1. change subquery into join
  2. divide query into two
  3. nativeSQL

! DO NOT include logic in query.

query focuses on fetching data.

logic process in backend

related rendering like date format in view.

one long query is not always good

refer to ‘BOOK: SQL antipatterns’

```
/** find oldest member */
@Test
public void subQuery(){
  QMember memberSub = new QMember("memberSub");

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

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

▼ case clause

NOT RECOMMENDED

query is for fetching data or calculating.

these logic should be dealt with in application or presentation layer

- simple

```
List<String> result = queryFactory
        .select(member.age
            .when(10).then("열살")
            .when(20).then("스무살")
            .otherwise("기타"))
        .from(member)
        .fetch();
```

- complex

```
List<String> result = queryFactory
        .select(new CaseBuilder()
            .when(member.age.between(0, 20)).then("0~20살")
            .when(member.age.between(21, 30)).then("21~30살")
            .otherwise("기타")
        )
        .from(member)
        .fetch();
```

▼ constant addition

com.querydsl.core.types.dsl.Expressions

```
List<Tuple> result = queryFactory
        .select(member.username, Expressions.constant("A"))
        .from(member)
        .fetch();
```

▼ concat

strinbValue() → change ENUM type to String

```
List<String> result = queryFactory
        .select(member.username.concat("_").concat(member.age.stringValue()))
        .from(member)
        .where(member.username.eq("member1"))
        .fetch();
```

▼ Grammar - intermediate

▼ Projection and result return

▼ basic

▼ One projection target

set equivalent target

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

▼ More than two projection target

1. get in Tuple
2. get in DTO

Tuple example)

```
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);
    }
```

DO NOT let upper layer that what technology sub layer implements.

jdbc result set should only be used in repository or DAO.

upper layer: service, controller

sub layer: jpa, querydsl

using Tuple in service or controller layer is at not good structure because tuple is subordinate to querydsl

▼ DTO

qeurydsl bean creation (bean population)

if there are changes to querydsl into other technology or

if pure DTO needed, choose 1~3.

▼ 1. property access using setter

Projections.bean()

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

▼ 2. direct field access

Projections.fields()

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

▼ when field name is different.

1. as() : for simple query
2. ExpressionUtils.as() : for subquery

```
List<UserDto> result = queryFactory
        .select(Projections.fields(UserDto.class,
            member.username.as("name"),
            ExpressionUtils.as(member.age, "age1")))
        .from(member)
        .fetch();
```

```
List<UserDto> result = queryFactory
        .select(Projections.fields(UserDto.class,
            member.username.as("name"),
            ExpressionUtils.as(JPAExpressions
              .select(memberSub.age.max())
              .from(memberSub), "age")
        ))
        .from(member)
        .fetch();
```

▼ 3. constructor

Projections.constructor()

- Disadvantage

  error occurs at run time

  can’t see parameter

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

▼ 4. @QueryProjection

use when no change to change querydsl into other techonology

1. set annotation
2. compileQuerydsl → Q file generated

- Advantage
  1. error occurs on compile
  2. easy to check parameters.

- Disadvantage
  1. Q file generation
  2. DTO is dependent on querydsl (not pure dto)

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

▼ Dynamic query

▼ BooleanBuilder

add conditions to builder

- Disadvantage

  required to look into builder for conditions.

```
@Test
public void dynamicQuery_WhereParam() {
String usernameParam = "member1";
Integer ageParam = 10;      // where member0_.username=? and member0_.age=?
//Integer ageParam = null;  // where member0_.username=?
```

```
List<Member> result = searchMember2(usernameParam, ageParam);

assertThat(result.size()).isEqualTo(1);
}
```

```
private List<Member> searchMember1(String usernameCond, Integer ageCond) {
    //BooleanBuilder builder = new BooleanBuilder(member.username.eq(usernameCond)); // username not null
    Bo oleanBuilder 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();
  }
```

▼ Where multi parameter (recommended)

null in where() be ignored.

- advantage

  easy to read querydsl.

  conditions are reusable → projection(select clause) can be flexibly changed

  conditions can be combined (null check !) → allEq()

```
@Test
public void dynamicQuery_WhereParam() {
  String usernameParam = "member1";
  Integer ageParam = 10;      // where member0_.username=? and member0_.age=?
  //Integer ageParam = null;  // where member0_.username=?

  List<Member> result = searchMember2(usernameParam, ageParam);

  assertThat(result.size()).isEqualTo(1);
}
```

```
private List<Member> searchMember2(String usernameCond, Integer ageCond) {
  return queryFactory
      .selectFrom(member)
      .where(usernameEq(usernameCond), ageEq(ageCond)) // if param in where null, the param is ignored
      //.where(allEq(usernameCond,ageCond))
      .fetch();
}

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

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

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

▼ Bulk operation for update and remove

DON'T FORGET to init persistentContext after bulk operation

- update

```
@Test
@Commit
public void bulkUpdate() {
  // member1 = 10 -> DB & PersistentContext : member1
  // member2 = 20 -> DB & PersistentContext : member2
  // member3 = 30 -> DB & PersistentContext : member3
  // member4 = 40 -> DB & PersistentContext : member4

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

  // member1 = 10 -> DB 비회원    / PersistentContext member1
  // member2 = 20 -> DB 비회원    / PersistentContext member2
  // member3 = 30 -> DB member3  / PersistentContext member3
  // member4 = 40 -> DB member4  / PersistentContext member4

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

  List<Member> result = queryFactory
      .selectFrom(member)
      .fetch();

  for (Member member : result) {
    System.out.println("member1 = " + member);
  }
}
```

- delete

```
@Test
public void bulkDelete() {
  queryFactory
      .delete(member)
      .where(member.age.gt(18))
      .execute();
}
```

▼ SQL function call

call based on dialect

querydsl provides functions that ANSI standard does.

for dialect functions use Expressions.stringTemplate()

```
@Test
public void sqlFunction() {
  List<String> result = queryFactory
      .select(Expressions.stringTemplate(
          "function('replace', {0}, {1}, {2})",
          member.username, "member", "m"))
      .where(//member.username.eq(Expressions.stringTemplate("function('lower, {0})", member.username)))
          member.username.eq(member.username.lower())) // querydsl
      .from(member)
```

```
      .fetch();

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

▼ Querydsl with Pure JPA

▼ Dynamic query performance optimization - Builder

StringUtils.hasText() → null & “” check

```
// Dynamic query performance optimization - Builder
public List<MemberTeamDto> searchByBuilder(MemberSearchCondition condition) {

  BooleanBuilder builder = new BooleanBuilder();
  if (hasText(condition.getUsername())) {
    builder.and(member.username.eq(condition.getUsername()));
  }
  if (hasText(condition.getTeamName())) {
    builder.and(team.name.eq(condition.getTeamName()));
  }
  if (condition.getAgeGoe() != null ) {
    builder.and(member.age.goe(condition.getAgeGoe()));
  }
  if (condition.getAgeLoe() != null ) {
    builder.and(member.age.loe(condition.getAgeLoe()));
  }

  return queryFactory
      .select(new QMemberTeamDto(
          member.id.as("memberId"),
          member.username,
          member.age,
          team.id.as("teamId"),
          team.name.as("teamName")
      ))
      .from(member)
      .leftJoin(member.team, team)
      .where(builder)
      .fetch();
}
```

▼ Dynamic query performance optimization - where paramter

```
// Dynamic query performance optimization - where paramter
public List<MemberTeamDto> search(MemberSearchCondition condition) {
  return queryFactory
      .select(new QMemberTeamDto(
          member.id.as("memberId"),
          member.username,
          member.age,
          team.id.as("teamId"),
          team.name.as("teamName")))
      .from(member)
      .leftJoin(member.team, team)
      .where(
          usernameEq(condition.getUsername()),
          teamNameEq(condition.getTeamName()),
          ageGoe(condition.getAgeGoe()),
          ageLoe(condition.getAgeLoe())
      )
      .fetch();
}
```

```
private BooleanExpression usernameEq(String username) {
  return hasText(username) ? member.username.eq(username) : null;
}

private BooleanExpression teamNameEq(String teamName) {
  return hasText(teamName) ? team.name.eq(teamName) : null;
}

private BooleanExpression ageGoe(Integer ageGoe) {
  return ageGoe != null ? member.age.goe(ageGoe) : null;
}

private Predicate ageLoe(Integer ageLoe) {
  return ageLoe != null ? member.age.loe(ageLoe) : null;
}
```

▼ Test

```
// Dynamic query performance optimization - Builder. where parameter
@Test
public void searchTest() {
  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);

  MemberSearchCondition condition = new MemberSearchCondition();
  condition.setAgeGoe(35);
  condition.setAgeLoe(40);
  condition.setTeamName("teamB");

  // List<MemberTeamDto> result = memberJpaRepository.searchByBuilder(condition);
  List<MemberTeamDto> result = memberJpaRepository.search(condition); // where parameter

  assertThat(result).extracting("username").containsExactly("member4");
}
```

▼ Initialize sample data

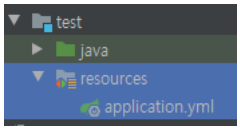
Set profiles so that the sample data does not affect the test case execution.

▼ Separating the tomcat profile from the test case profile.

1. Add profiles in yml
   - local
   - dev or development
   - real (operation server)

```
spring:
  profiles:
    active: local
```

2. make resources directory and copy the yml above.



3. change ‘active’ value into test

```
spring:
  profiles:
    active: test
```

▼ Initialize

After setting, check the tomcat log at the top


```
@Profile("local")
@Component
@RequiredArgsConstructor
public class InitMember {

  private final initMemberService initMemberService;

  @PostConstruct
  public void init() {
    initMemberService.init();
  }

  @Component
  static class initMemberService {
    @PersistenceContext // @Autowired : spring injection
    private EntityManager em;

    @Transactional
    public void init() {
      Team teamA = new Team("teamA");
      Team teamB = new Team("teamB");
      em.persist(teamA);
      em.persist(teamB);

      for (int i = 0; i <100; i++) {
        Team selectedTeam = i % 2 == 0 ? teamA : teamB;
        em.persist(new Member("member"+1, i, selectedTeam));
      }
    }
  }
}
```

Due to spring life cycle, @PostConstruct cannot co-exist @Transactional.

So below is not possible.

```
@PostConstruct
@Transactional
public void init() {
    Team teamA = new Team("teamA");
```

```
    Team teamB = new Team("teamB");
    em.persist(teamA);
    em.persist(teamB);

    for (int i = 0; i <100; i++) {
      Team selectedTeam = i % 2 == 0 ? teamA : teamB;
      em.persist(new Member("member"+1, i, selectedTeam));
    }
}
```

▼ Querydsl with Spring Data JPA

▼ Change into Spring Data JPA repository

No nee to implement

- save()
- findById()
- findAll()

```
public interface MemberRepository extends JpaRepository<Member, Long> {
  // change into spring data jpa repository
  List<Member> findByUsername(String username);
}
```

▼ custom repository

To use querydsl, implementing code should be written. However, Spring Data JPA is running as an inetrface.

There are two ways to use own codes.

▼ 1. custom repository interface (priority)

For core business logic that has high potential for reuse. (entity search, etc...)

▼ 1. Make custom interface

```
public interface MemberRepositoryCustom {
  List<MemberTeamDto> search(MemberSearchCondition condition);
}
```

▼ 2. Implement the custom interface.

class name must be [repository interface name] + Impl

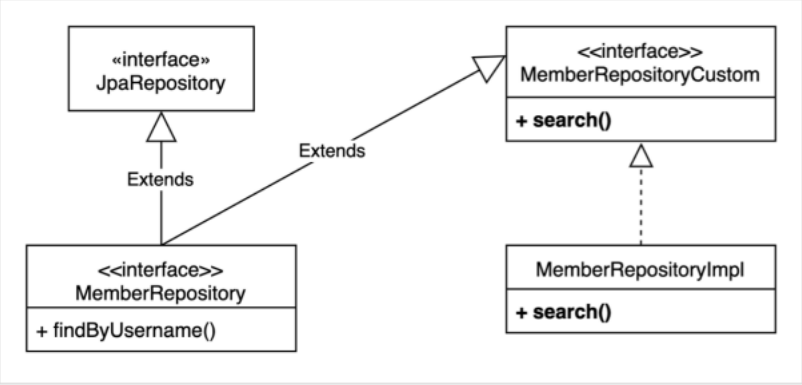
```
 public class MemberRepositoryImpl implements MemberRepositoryCustom{

  private final JPAQueryFactory queryFactory;

  public MemberRepositoryImpl(EntityManager em) {
    this.queryFactory = new JPAQueryFactory(em);
  }

  // if registered as spring bean
  /*public MemberRepositoryImpl(JPAQueryFactory queryFactory) {
```

```
    this.queryFactory = queryFactory;
  }*/

  @Override
  public List<MemberTeamDto> search(MemberSearchCondition condition) {
    return queryFactory
        .select(new QMemberTeamDto(
            member.id.as("memberId"),
            member.username,
            member.age,
            team.id.as("teamId"),
            team.name.as("teamName")))
        .from(member)
        .leftJoin(member.team, team)
        .where(
            usernameEq(condition.getUsername()),
            teamNameEq(condition.getTeamName()),
            ageGoe(condition.getAgeGoe()),
            ageLoe(condition.getAgeLoe())
        )
        .fetch();
  }

  private BooleanExpression usernameEq(String username) {
    return hasText(username) ? member.username.eq(username) : null;
  }

  private BooleanExpression teamNameEq(String teamName) {
    return hasText(teamName) ? team.name.eq(teamName) : null;
  }

  private BooleanExpression ageGoe(Integer ageGoe) {
    return ageGoe != null ? member.age.goe(ageGoe) : null;
  }

  private Predicate ageLoe(Integer ageLoe) {
    return ageLoe != null ? member.age.loe(ageLoe) : null;
  }
}
```

▼ 3. Extends the custom interface from spring data repository

```
public interface MemberRepository extends JpaRepository<Member, Long>, MemberRepositoryCustom {
  ...
}
```

- Custom repository structure



▼ 2. Separate repository class from spring data jpa (XxxQueryRepository)

spring data jpa doesn't always need to be used.

- Choose this pattern for

Queries that have no commonness.

Queries that are subordinate to specific API or view.

(In those cases, modifying life cycle is set to API or view.)

```
@Repository
public class MemberQueryRepository {

  private final JPAQueryFactory queryFactory;

  public MemberQueryRepository(EntityManager em) {
    this.queryFactory = new JPAQueryFactory(em);
  }

  public List<MemberTeamDto> search(MemberSearchCondition condition) {
    return queryFactory
        .select( ... )
        .from(member)
        .leftJoin(member.team, team)
        .where( ... )
        .fetch();
  }

  ...
}
```

```
class MemberRepositoryTest {
  @Autowired MemberRepository memberRepository;
  @Autowired MemberQueryRepository memberQueryRepository;
}
```

▼ paging

▼ interconnection with querydsl paging

```
public interface MemberRepositoryCustom {
  ...
  Page<MemberTeamDto> searchPage(MemberSearchCondition condition, Pageable pageable);
}
```

```
/** saerchPageSimple & saerchPageComplex ==> searchPage
 * because fetchResults() deprecated */
@Override
public Page<MemberTeamDto> searchPage(MemberSearchCondition condition, Pageable pageable) {
  List<MemberTeamDto> content = queryFactory
      .select(new QMemberTeamDto(
          member.id.as("memberId"),
          member.username,
          member.age,
          team.id.as("teamId"),
          team.name.as("teamName")))
      .from(member)
      .leftJoin(member.team, team)
      .where(
```

```
          usernameEq(condition.getUsername()),
          teamNameEq(condition.getTeamName()),
          ageGoe(condition.getAgeGoe()),
          ageLoe(condition.getAgeLoe())
      )
      .offset(pageable.getOffset())
      .limit(pageable.getPageSize())
      .fetch();

  Long total = queryFactory
      .select(member.count())
      .from(member)
      .leftJoin(member.team, team)
      .where(
          usernameEq(condition.getUsername()),
          teamNameEq(condition.getTeamName()),
          ageGoe(condition.getAgeGoe()),
          ageLoe(condition.getAgeLoe()))
      .fetchOne();

  return new PageImpl<>(content, pageable, total);
}
```

▼ countQuery optimization

- PageableExecutionUtils.getPage()

count query is dispatched only need.

(!isLastPage && contentSize > pageSize)

or else, get the total count from content

```
JPAQuery<Long> countQuery = queryFactory
    .select(member.count())
    .from(member)
    .leftJoin(member.team, team)
    .where(
        usernameEq(condition.getUsername()),
        teamNameEq(condition.getTeamName()),
        ageGoe(condition.getAgeGoe()),
        ageLoe(condition.getAgeLoe()));

// return PageableExecutionUtils.getPage(content, pageable, () -> countQuery.fetchOne());
return PageableExecutionUtils.getPage(content, pageable, countQuery::fetchOne);
```

refer to method definition for condition details.

▼ controller

```
// http://localhost:8080/v2/members?
// http://localhost:8080/v2/members?page=0&size=5
@GetMapping("/v2/members")
public Page<MemberTeamDto> searchMemberV2(MemberSearchCondition condition, Pageable pageable) {
  return memberRepository.searchPage(condition, pageable);
}
```

▼ sorting

Spring Data Paging 3 - developing controller_back part

It’s difficult to user sort() in Pageable if dynamic sorting is required that goes beyond the bounds of loot entity.

- Solution steps

1. get parameter and process them yourself

   refer to ‘Grammar - basic / Search condition query / sort’
2. Manual querydsl support class / sort
3. convert spring data sort into OrderSpecifier of querydsl

```
JPAQuery<Member> query = queryFactory
   .selectFrom(member);

for (Sort.Order o : pageable.getSort()) {
   PathBuilder pathBuilder = new PathBuilder(member.getType(), member.getMetadata());
   query.orderBy(new OrderSpecifier(o.isAscending() ? Order.ASC : Order.DESC,pathBuilder.get(o.getProperty())));
}

List<Member> result = query.fetch();
```

▼ Querydsl functions that Spring Data JPA provides

inadequate in working level

applicable for simple logic.

- Limitation

left join is not possible, only implied join in provided.

client or service class should rely on implementation technology(querydsl).

→ check ‘reason to use repository’

▼ Interface support - QuerydslPredicateExecutor (ref.)

Pageable, sort are provided and works fine.

Add ‘QuerydslPredicateExecutor<T>’ interface in repository

```
public interface MemberRepository extends JpaRepository<Member, Long>,
                                          MemberRepositoryCustom,
                                          QuerydslPredicateExecutor<Member> {
  ...
}
```

Make use in client or service class

```
@Test
public void querydslPredicateExecutorTest() {
  ...
  QMember member = QMember.member;
  Iterable<Member> result = memberRepository.findAll(member.age.between(10, 40).and(member.username.eq("member")));
  ...
}
```

▼ Querydsl web support (ref.)

No content. refer to textbook and lecture.

▼ Repository support - QuerydslRepositorySupport

No content. refer to textbook and lecture.

▼ Manual querydsl support class

refer to text book and lecture when needed.

- better pagination than spring data jpa provides.
- separate count from paging
- sort
- etc.