실전! Querydsl v2023-05-26

#1.인강/JPA활용편/querydsl/강의#

인프런 강의: 실전! Querydsl
인프런: https://www.inflearn.com

버전 수정 이력

v2023-05-26

- 스프링 부트 3.0 - p6spy 1.9.0 사용(DH L님 도움)
- PageableExecutionUtils 최적화 조건 보충(박진영님 도움)
- QueryDSL 설정과 검증 백틱 `junit-vintage-engine` 백틱 오타 수정(cwh73090님 도움)

v2023-01-03

- 스프링 부트 3.0 - p6spy 쿼리 파라미터 로그 남기기 - 스프링 부트 3.0 사용법 추가

v2022-11-28

- 스프링 부트 3.0 내용 추가
- 프로젝트 선택에서 `Gradle - Groovy` 추가

v2022-05-01

- 스프링 부트 버전 표시 `'2.2.2.RELEASE'` 오타(박용훈님 도움)

v2022-01-12

- 스프링 부트 2.6 이상, Querydsl 5.0 지원 방법 추가

v2021-08-08

- private 추가, 도움주신 분: 류재준

v2021-04-10

- @QueryProjection에서 as 잘못 사용하는 부분 수정
  - 도움주신 분: Young.K님

v1.1 - 2020-10-02

- CASE 문 ➡ orderBy에서 Case 문 함께 사용하기 예제 추가

v1.0

- 최초 배포

목차

    - 동적 쿼리 - BooleanBuilder 사용
    - 동적 쿼리 - Where 다중 파라미터 사용
    - 수정, 삭제 벌크 연산
    - SQL function 호출하기
- 실무 활용 - 순수 JPA와 Querydsl
    - 순수 JPA 리포지토리와 Querydsl
    - 동적 쿼리와 성능 최적화 조회 - Builder 사용
    - 동적 쿼리와 성능 최적화 조회 - Where절 파라미터 사용
    - 조회 API 컨트롤러 개발
- 실무 활용 - 스프링 데이터 JPA와 Querydsl
    - 스프링 데이터 JPA 리포지토리로 변경
    - 사용자 정의 리포지토리
    - 스프링 데이터 페이징 활용1 - Querydsl 페이징 연동
    - 스프링 데이터 페이징 활용2 - CountQuery 최적화
    - 스프링 데이터 페이징 활용3 - 컨트롤러 개발
- 스프링 데이터 JPA가 제공하는 Querydsl 기능
    - 인터페이스 지원 - QuerydslPredicateExecutor
    - Querydsl Web 지원
    - 리포지토리 지원 - QuerydslRepositorySupport
    - Querydsl 지원 클래스 직접 만들기
- 스프링 부트 2.6 이상, Querydsl 5.0 지원 방법

Querydsl 소개

소개

영상 참고

강의 자료

현재 파일

프로젝트 환경설정

프로젝트 생성

- 스프링 부트 스타터(https://start.spring.io/)

- Project: **Gradle - Groovy** Project
- 사용 기능: Spring Web, jpa, h2, lombok
  - SpringBootVersion: **2.2.2**
  - groupId: study
  - artifactId: querydsl

주의! - 스프링 부트 3.0

스프링 부트 3.0을 선택하게 되면 다음 부분을 꼭 확인해주세요.

- **1. Java 17 이상**을 사용해야 합니다.
- **2. javax 패키지 이름을 jakarta로 변경**해야 합니다.
  - 오라클과 자바 라이센스 문제로 모든 javax 패키지를 jakarta로 변경하기로 했습니다.
- **3. H2 데이터베이스를 2.1.214 버전 이상 사용해주세요.**

패키지 이름 변경 예)

- **JPA 애노테이션**
  - javax.persistence.Entity ➡ jakarta.persistence.Entity
- **스프링에서 자주 사용하는 @PostConstruct 애노테이션**
  - javax.annotation.PostConstruct ➡ jakarta.annotation.PostConstruct
- **스프링에서 자주 사용하는 검증 애노테이션**
  - javax.validation ➡ jakarta.validation

- Querydsl 스프링 부트 3.0 설정은 다음을 참고해주세요.
  - https://www.inflearn.com/chats/700670

스프링 부트 3.0 관련 자세한 내용은 다음 링크를 확인해주세요: https://bit.ly/springboot3

Gradle 전체 설정

```
plugins {
  id 'org.springframework.boot' version '2.2.2.RELEASE'
  id 'io.spring.dependency-management' version '1.0.8.RELEASE'
  id 'java'
}

group = 'study'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'
```

```
configurations {
  compileOnly {
      extendsFrom annotationProcessor
  }
}

repositories {
  mavenCentral()
}

dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
  implementation 'org.springframework.boot:spring-boot-starter-web'
  compileOnly 'org.projectlombok:lombok'
  runtimeOnly 'com.h2database:h2'
  annotationProcessor 'org.projectlombok:lombok'
  testImplementation('org.springframework.boot:spring-boot-starter-test') {
      exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
  }
}

test {
  useJUnitPlatform()
}
```

- 동작 확인
  - 기본 테스트 케이스 실행
  - 스프링 부트 메인 실행 후 에러페이지로 간단하게 동작 확인(`http://localhost:8080')
  - 테스트 컨트롤러를 만들어서 spring web 동작 확인(http://localhost:8080/hello)

테스트 컨트롤러

```
package study.querydsl.controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public String hello() {
        return "hello!";
    }
}
```

IntelliJ Gradle 대신에 자바로 바로 실행하기

최근 IntelliJ 버전은 Gradle로 실행을 하는 것이 기본 설정이다. 이렇게 하면 실행속도가 느리다. 다음과 같이 변경하면 자바로 바로 실행하므로 좀 더 빨라진다.

1. Preferences ➜ Build, Execution, Deployment ➜ Build Tools ➜ Gradle
2. Build and run using: Gradle ➜ IntelliJ IDEA
3. Run tests using: Gradle ➜ IntelliJ IDEA

롬복 적용

1. Preferences ➜ plugin ➜ lombok 검색 실행 (재시작)
2. Preferences ➜ Annotation Processors 검색 ➜ Enable annotation processing 체크 (재시작)
3. 임의의 테스트 클래스를 만들고 @Getter, @Setter 확인

Querydsl 설정과 검증

`build.gradle`에 주석을 참고해서 querydsl 설정 추가

```
plugins {
    id 'org.springframework.boot' version '2.2.2.RELEASE'
    id 'io.spring.dependency-management' version '1.0.8.RELEASE'
    //querydsl 추가
    id "com.ewerk.gradle.plugins.querydsl" version "1.0.10"
    id 'java'
```

```
}

group = 'study'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

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

    //querydsl 추가
    implementation 'com.querydsl:querydsl-jpa'

    compileOnly 'org.projectlombok:lombok'
    runtimeOnly 'com.h2database:h2'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
    }
}

test {
    useJUnitPlatform()
}

//querydsl 추가 시작
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 추가 끝
```

Querydsl 환경설정 검증

검증용 엔티티 생성

```
package study.querydsl.entity;

import lombok.Getter;
import lombok.Setter;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
@Getter @Setter
public class Hello {

    @Id @GeneratedValue
    private Long id;
}
```

검증용 Q 타입 생성

Gradle IntelliJ 사용법

- Gradle ➜ Tasks ➜ build ➜ clean
- Gradle ➜ Tasks ➜ other ➜ compileQuerydsl

Gradle 콘솔 사용법

- ./gradlew clean compileQuerydsl

Q 타입 생성 확인

- build ➜ generated ➜ querydsl
  - study.querydsl.entity.QHello.java 파일이 생성되어 있어야 함

> 참고: Q타입은 컴파일 시점에 자동 생성되므로 버전관리(GIT)에 포함하지 않는 것이 좋다. 앞서 설정에서 생성 위치를 gradle build 폴더 아래 생성되도록 했기 때문에 이 부분도 자연스럽게 해결된다. (대부분 gradle build 폴더를 git에 포함하지 않는다.)

테스트 케이스로 실행 검증

```
package study.querydsl;

import com.querydsl.jpa.impl.JPAQueryFactory;
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.annotation.Commit;
import org.springframework.transaction.annotation.Transactional;
import study.querydsl.entity.Hello;
import study.querydsl.entity.QHello;

import javax.persistence.EntityManager;
import java.util.List;

@SpringBootTest
@Transactional
class QuerydslApplicationTests {

  @Autowired
  EntityManager em;
```

```
    @Test
    void contextLoads() {

        Hello hello = new Hello();
        em.persist(hello);

        JPAQueryFactory query = new JPAQueryFactory(em);
        QHello qHello = QHello.hello; //Querydsl Q타입 동작 확인

        Hello result = query
                .selectFrom(qHello)
                .fetchOne();

        Assertions.assertThat(result).isEqualTo(hello);
        //lombok 동작 확인 (hello.getId())
        Assertions.assertThat(result.getId()).isEqualTo(hello.getId());
    }
}
```

- Querydsl Q타입이 정상 동작하는가?
- lombok이 정상 동작 하는가?

> 참고: 스프링 부트에 아무런 설정도 하지 않으면 h2 DB를 메모리 모드로 JVM안에서 실행한다.

라이브러리 살펴보기

gradle 의존관계 보기

```
./gradlew dependencies --configuration compileClasspath
```

Querydsl 라이브러리 살펴보기

- querydsl-apt: Querydsl 관련 코드 생성 기능 제공
- querydsl-jpa: querydsl 라이브러리

스프링 부트 라이브러리 살펴보기

- spring-boot-starter-web
  - spring-boot-starter-tomcat: 톰캣 (웹서버)
  - spring-webmvc: 스프링 웹 MVC
- spring-boot-starter-data-jpa
  - spring-boot-starter-aop
  - spring-boot-starter-jdbc
    - HikariCP 커넥션 풀 (부트 2.0 기본)
  - hibernate + JPA: 하이버네이트 + JPA
  - spring-data-jpa: 스프링 데이터 JPA
- spring-boot-starter(공통): 스프링 부트 + 스프링 코어 + 로깅
  - spring-boot
    - spring-core
  - spring-boot-starter-logging
    - logback, slf4j

테스트 라이브러리

- spring-boot-starter-test
  - junit: 테스트 프레임워크, 스프링 부트 2.2부터 junit5(`jupiter`) 사용
    - 과거 버전은 `vintage`
  - mockito: 목 라이브러리
  - assertj: 테스트 코드를 좀 더 편하게 작성하게 도와주는 라이브러리
    - https://joel-costigliola.github.io/assertj/index.html
  - spring-test: 스프링 통합 테스트 지원
- 핵심 라이브러리
  - 스프링 MVC
  - JPA, 하이버네이트
  - 스프링 데이터 JPA
  - Querydsl
- 기타 라이브러리
  - H2 데이터베이스 클라이언트
  - 커넥션 풀: 부트 기본은 HikariCP
  - 로깅 SLF4J & LogBack
  - 테스트

H2 데이터베이스 설치

개발이나 테스트 용도로 가볍고 편리한 DB, 웹 화면 제공

- https://www.h2database.com
- 다운로드 및 설치
- h2 데이터베이스 버전은 스프링 부트 버전에 맞춘다.
- 권한 주기: `chmod 755 h2.sh`
- 데이터베이스 파일 생성 방법
  - `jdbc:h2:~/querydsl` (최소 한번)
  - `~/querydsl.mv.db` 파일 생성 확인
  - 이후 부터는 `jdbc:h2:tcp://localhost/~/querydsl` 이렇게 접속

> 참고: H2 데이터베이스의 MVCC 옵션은 H2 1.4.198 버전부터 제거되었습니다. 이후 부터는 옵션 없이 사용하면 됩니다.

> 주의: 가급적 안정화 버전을 사용하세요. 1.4.200 버전은 몇가지 오류가 있습니다.
> 현재 안정화 버전은 1.4.199(2019-03-13) 입니다.
> 다운로드 링크: https://www.h2database.com/html/download.html

스프링 부트 설정 - JPA, DB

`application.yml`

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

  jpa:
    hibernate:
      ddl-auto: create
    properties:
```

```
      hibernate:
#        show_sql: true
        format_sql: true

logging.level:
  org.hibernate.SQL: debug
#  org.hibernate.type: trace
```

- spring.jpa.hibernate.ddl-auto: create
  - 이 옵션은 애플리케이션 실행 시점에 테이블을 drop 하고, 다시 생성한다.

> 참고: 모든 로그 출력은 가급적 로거를 통해 남겨야 한다.
> `show_sql` : 옵션은 `System.out` 에 하이버네이트 실행 SQL을 남긴다.
> `org.hibernate.SQL` : 옵션은 logger를 통해 하이버네이트 실행 SQL을 남긴다.

쿼리 파라미터 로그 남기기

- 로그에 다음을 추가하기 `org.hibernate.type` : SQL 실행 파라미터를 로그로 남긴다.
- 외부 라이브러리 사용
  - https://github.com/gavlyukovskiy/spring-boot-data-source-decorator

스프링 부트를 사용하면 이 라이브러리만 추가하면 된다.

```
implementation 'com.github.gavlyukovskiy:p6spy-spring-boot-starter:1.5.8'
```

> 참고: 쿼리 파라미터를 로그로 남기는 외부 라이브러리는 시스템 자원을 사용하므로, 개발 단계에서는 편하게 사용해도 된다. 하지만 운영시스템에 적용하려면 꼭 성능테스트를 하고 사용하는 것이 좋다.

쿼리 파라미터 로그 남기기 - 스프링 부트 3.0

스프링 부트 3.0 이상을 사용하면 라이브러리 버전을 1.9.0 이상을 사용해야 한다.

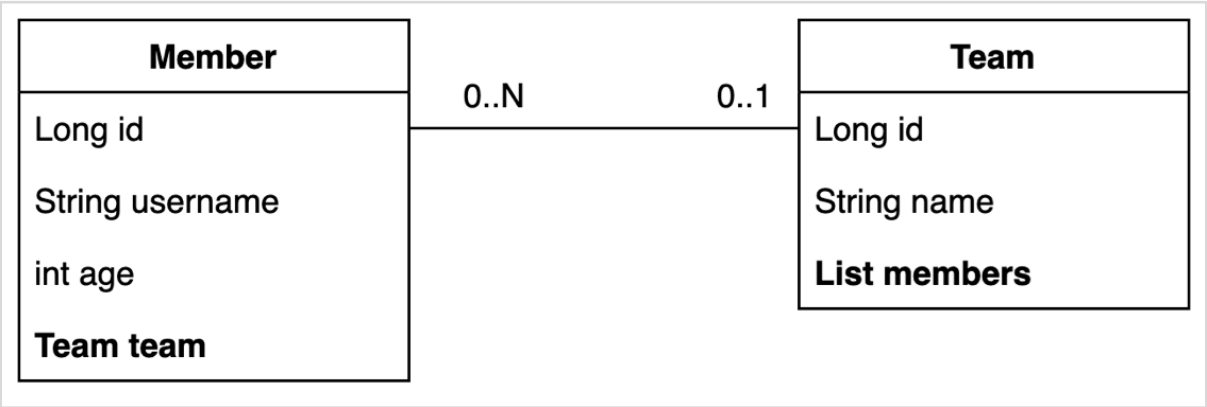
```
implementation 'com.github.gavlyukovskiy:p6spy-spring-boot-starter:1.9.0'
```

예제 도메인 모델

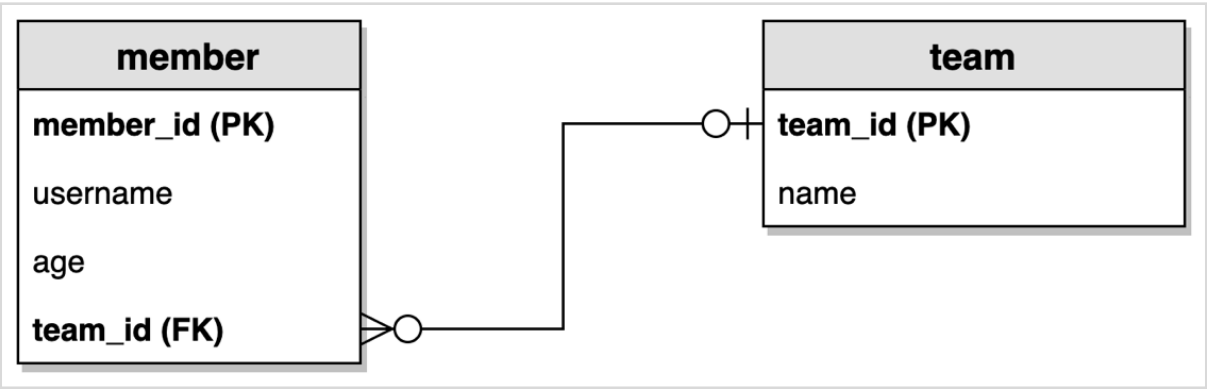
> 참고: 스프링 데이터 JPA와 동일한 예제 도메인 모델을 사용합니다. 스프링 데이터 JPA 강의를 들으신 분은 엔티티 코드만 작성하고 다음으로 넘어가도 됩니다.

예제 도메인 모델과 동작확인

엔티티 클래스



ERD



Member 엔티티

```
package study.querydsl.entity;

import lombok.*;
import javax.persistence.*;

@Entity
```

```java
@Getter @Setter
@NoArgsConstructor(access = AccessLevel.PROTECTED)
@ToString(of = {"id", "username", "age"})
public class Member {

    @Id
    @GeneratedValue
    @Column(name = "member_id")
    private Long id;
    private String username;
    private int age;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "team_id")
    private Team team;

    public Member(String username) {
        this(username, 0);
    }

    public Member(String username, int age) {
        this(username, age, null);
    }

    public Member(String username, int age, Team team) {
        this.username = username;
        this.age = age;
        if (team != null) {
            changeTeam(team);
        }
    }

    public void changeTeam(Team team) {
        this.team = team;
        team.getMembers().add(this);
    }
}
```

- 롬복 설명
  - @Setter: 실무에서 가급적 Setter는 사용하지 않기
  - @NoArgsConstructor AccessLevel.PROTECTED: 기본 생성자 막고 싶은데, JPA 스팩상 PROTECTED로 열어두어야 함
  - @ToString은 가급적 내부 필드만(연관관계 없는 필드만)
- `changeTeam()` 으로 양방향 연관관계 한번에 처리(연관관계 편의 메소드)

Team 엔티티

```
package study.querydsl.entity;

import lombok.*;

import javax.persistence.*;
import java.util.ArrayList;
import java.util.List;

@Entity
@Getter @Setter
@NoArgsConstructor(access = AccessLevel.PROTECTED)
@ToString(of = {"id", "name"})
public class Team {

    @Id @GeneratedValue
    @Column(name = "team_id")
    private Long id;
    private String name;

    @OneToMany(mappedBy = "team")
    private List<Member> members = new ArrayList<>();

    public Team(String name) {
        this.name = name;
    }
}
```

- Member와 Team은 양방향 연관관계, `Member.team` 이 연관관계의 주인, `Team.members` 는 연관관계의

주인이 아님, 따라서 `Member.team` 이 데이터베이스 외래키 값을 변경, 반대편은 읽기만 가능

데이터 확인 테스트

```
package study.querydsl.entity;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.annotation.Commit;
import org.springframework.transaction.annotation.Transactional;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.List;

@SpringBootTest
@Transactional
@Commit
public class MemberTest {

    @PersistenceContext
    EntityManager em;

    @Test


    public void testEntity() {
        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);

        //초기화
        em.flush();
        em.clear();

        //확인
        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.team=" + member.getTeam());
        }
    }
}
```

- 가급적 순수 JPA로 동작 확인 (뒤에서 변경)
- db 테이블 결과 확인
- 지연 로딩 동작 확인

기본 문법

시작 - JPQL vs Querydsl

테스트 기본 코드

```
package study.querydsl;

import com.querydsl.jpa.impl.JPAQueryFactory;
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
```

```java
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.transaction.annotation.Transactional;
import study.querydsl.entity.Member;
import study.querydsl.entity.QMember;
import study.querydsl.entity.Team;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.List;

import static org.assertj.core.api.Assertions.*;
import static study.querydsl.entity.QMember.*;


@SpringBootTest
@Transactional
public class QuerydslBasicTest {

    @PersistenceContext
    EntityManager em;

    @BeforeEach
    public void before() {
        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);
    }
```

```
}
```

- 지금부터는 이 예제로 실행

Querydsl vs JPQL

```
@Test
public void startJPQL() {
    //member1을 찾아라.
    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() {
    //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`로 `JPAQueryFactory` 생성
- Querydsl은 JPQL 빌더
- JPQL: 문자(실행 시점 오류), Querydsl: 코드(컴파일 시점 오류)
- JPQL: 파라미터 바인딩 직접, Querydsl: 파라미터 바인딩 자동 처리

JPAQueryFactory를 필드로

```
package study.querydsl;

import com.querydsl.jpa.impl.JPAQueryFactory;
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.transaction.annotation.Transactional;
import study.querydsl.entity.Member;
import study.querydsl.entity.QMember;
import study.querydsl.entity.Team;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.List;

import static org.assertj.core.api.Assertions.*;
import static study.querydsl.entity.QMember.*;

@SpringBootTest
@Transactional
public class QuerydslBasicTest {

    @PersistenceContext
    EntityManager em;

    JPAQueryFactory queryFactory;
```

```
    @BeforeEach
    public void before() {
        queryFactory = new JPAQueryFactory(em);
        //…
    }

    @Test
    public void startQuerydsl2() {
        //member1을 찾아라.
        QMember m = new QMember("m");

        Member findMember = queryFactory
                .select(m)
                .from(m)
                .where(m.username.eq("member1"))
                .fetchOne();
        assertThat(findMember.getUsername()).isEqualTo("member1");
    }

}
```

- JPAQueryFactory를 필드로 제공하면 동시성 문제는 어떻게 될까? 동시성 문제는 JPAQueryFactory를 생성할 때 제공하는 EntityManager(em)에 달려있다. 스프링 프레임워크는 여러 쓰레드에서 동시에 같은 EntityManager에 접근해도, 트랜잭션 마다 별도의 영속성 컨텍스트를 제공하기 때문에, 동시성 문제는 걱정하지 않아도 된다.

기본 Q-Type 활용

Q클래스 인스턴스를 사용하는 2가지 방법

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

기본 인스턴스를 static import와 함께 사용

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

@Test
public void startQuerydsl3() {
    //member1을 찾아라.
    Member findMember = queryFactory
            .select(member)
            .from(member)
            .where(member.username.eq("member1"))
            .fetchOne();

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

다음 설정을 추가하면 실행되는 JPQL을 볼 수 있다.

```
spring.jpa.properties.hibernate.use_sql_comments: true
```

> 참고: 같은 테이블을 조인해야 하는 경우가 아니면 기본 인스턴스를 사용하자

검색 조건 쿼리

기본 검색 쿼리

```
@Test
public void search() {
    Member findMember = queryFactory
            .selectFrom(member)
            .where(member.username.eq("member1")
                    .and(member.age.eq(10)))
            .fetchOne();
```

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

- 검색 조건은 `.and()`, `.or()`를 메서드 체인으로 연결할 수 있다.

> 참고: `select`, `from`을 `selectFrom`으로 합칠 수 있음

JPQL이 제공하는 모든 검색 조건 제공

```
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 조건을 파라미터로 처리

```
@Test
public void searchAndParam() {
    List<Member> result1 = queryFactory
            .selectFrom(member)
            .where(member.username.eq("member1"),
                    member.age.eq(10))
```

```
            .fetch();
    assertThat(result1.size()).isEqualTo(1);
}
```

- `where()`에 파라미터로 검색조건을 추가하면 `AND` 조건이 추가됨
- 이 경우 `null` 값은 무시 ➜ 메서드 추출을 활용해서 동적 쿼리를 깔끔하게 만들 수 있음 ➜ 뒤에서 설명

결과 조회

- `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()`
- `fetchResults()` : 페이징 정보 포함, total count 쿼리 추가 실행
- `fetchCount()` : count 쿼리로 변경해서 count 수 조회

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

//단 건
Member findMember1 = queryFactory
        .selectFrom(member)
        .fetchOne();

//처음 한 건 조회
Member findMember2 = queryFactory
        .selectFrom(member)
        .fetchFirst();

//페이징에서 사용
QueryResults<Member> results = queryFactory
        .selectFrom(member)
```

```
        .fetchResults();

//count 쿼리로 변경
long count = queryFactory
        .selectFrom(member)
        .fetchCount();
```

정렬

```
/**
 * 회원 정렬 순서
 * 1. 회원 나이 내림차순(desc)
 * 2. 회원 이름 올림차순(asc)
 * 단 2에서 회원 이름이 없으면 마지막에 출력(nulls last)
 */
@Test
public void sort() {
    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

```
@Test
public void paging1() {
    List<Member> result = queryFactory
            .selectFrom(member)
            .orderBy(member.username.desc())
            .offset(1) //0부터 시작(zero index)
            .limit(2) //최대 2건 조회
            .fetch();

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

전체 조회 수가 필요하면?

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

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

주의: count 쿼리가 실행되니 성능상 주의!

참고: 실무에서 페이징 쿼리를 작성할 때, 데이터를 조회하는 쿼리는 여러 테이블을 조인해야 하지만, count 쿼리는 조인이 필요 없는 경우도 있다. 그런데 이렇게 자동화된 count 쿼리는 원본 쿼리와 같이 모두 조인을 해버리기 때문에 성능이 안나올 수 있다. count 쿼리에 조인이 필요없는 성능 최적화가 필요하다면, count 전용 쿼리를 별도로 작성해야 한다.

집합

집합 함수

```
/**
 * JPQL
 * select
 *    COUNT(m),   //회원수
 *    SUM(m.age), //나이 합
 *    AVG(m.age), //평균 나이
 *    MAX(m.age), //최대 나이
 *    MIN(m.age)  //최소 나이
 * from Member m
 */
@Test
public void aggregation() throws Exception {
    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);
}
```

- JPQL이 제공하는 모든 집합 함수를 제공한다.
- tuple은 프로젝션과 결과반환에서 설명한다.

GroupBy 사용

```
/**
 * 팀의 이름과 각 팀의 평균 연령을 구해라.
 */
@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);

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

`groupBy`, 그룹화된 결과를 제한하려면 `having`

groupBy(), having() 예시

```
…
.groupBy(item.price)
.having(item.price.gt(1000))
…
```

조인 - 기본 조인

기본 조인

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

```
join(조인 대상, 별칭으로 사용할 Q타입)
```

기본 조인

```
/**
 * 팀 A에 소속된 모든 회원
 */
@Test
public void join() throws Exception {
    QMember member = QMember.member;
    QTeam team = QTeam.team;

    List<Member> result = queryFactory
            .selectFrom(member)
            .join(member.team, team)
            .where(team.name.eq("teamA"))
            .fetch();

    assertThat(result)
            .extracting("username")
```

```
                .containsExactly("member1", "member2");
    }
```

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

세타 조인

연관관계가 없는 필드로 조인

```
/**
 * 세타 조인(연관관계가 없는 필드로 조인)
 * 회원의 이름이 팀 이름과 같은 회원 조회
 */
@Test
public void theta_join() throws Exception {
    em.persist(new Member("teamA"));
    em.persist(new Member("teamB"));

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

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

- from 절에 여러 엔티티를 선택해서 세타 조인
- 외부 조인 불가능 ➜ 다음에 설명할 조인 on을 사용하면 외부 조인 가능

조인 - on절

- ON절을 활용한 조인(JPA 2.1부터 지원)
  1. 조인 대상 필터링
  2. 연관관계 없는 엔티티 외부 조인

1. 조인 대상 필터링

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

```
/**
 * 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'
 * SQL: SELECT m.*, t.* FROM Member m LEFT JOIN Team t ON m.TEAM_ID=t.id and
t.name='teamA'
 */
@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);
    }

}
```

결과

```
t=[Member(id=3, username=member1, age=10), Team(id=1, name=teamA)]
```

```
t=[Member(id=4, username=member2, age=20), Team(id=1, name=teamA)]
t=[Member(id=5, username=member3, age=30), null]
t=[Member(id=6, username=member4, age=40), null]
```

> 참고: on 절을 활용해 조인 대상을 필터링 할 때, 외부조인이 아니라 내부조인(inner join)을 사용하면, where 절에서 필터링 하는 것과 기능이 동일하다. 따라서 on 절을 활용한 조인 대상 필터링을 사용할 때, 내부조인 이면 익숙한 where 절로 해결하고, 정말 외부조인이 필요한 경우에만 이 기능을 사용하자.

2. 연관관계 없는 엔티티 외부 조인

예) 회원의 이름과 팀의 이름이 같은 대상 **외부 조인**

```
/**
 * 2. 연관관계 없는 엔티티 외부 조인
 * 예) 회원의 이름과 팀의 이름이 같은 대상 외부 조인
 * JPQL: SELECT m, t FROM Member m LEFT JOIN Team t on m.username = t.name
 * SQL: SELECT m.*, t.* FROM  Member m LEFT JOIN Team t ON m.username = t.name
 */
@Test
public void join_on_no_relation() throws Exception {
    em.persist(new Member("teamA"));
    em.persist(new Member("teamB"));

    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("t=" + tuple);
    }

}
```

- 하이버네이트 5.1부터 `on` 을 사용해서 서로 관계가 없는 필드로 외부 조인하는 기능이 추가되었다. 물론 내

부 조인도 가능하다.

- 주의! 문법을 잘 봐야 한다. **leftJoin()** 부분에 일반 조인과 다르게 엔티티 하나만 들어간다.
  - 일반조인: `leftJoin(member.team, team)`
  - on조인: `from(member).leftJoin(team).on(xxx)`

결과

```
t=[Member(id=3, username=member1, age=10), null]
t=[Member(id=4, username=member2, age=20), null]
t=[Member(id=5, username=member3, age=30), null]
t=[Member(id=6, username=member4, age=40), null]
t=[Member(id=7, username=teamA, age=0), Team(id=1, name=teamA)]
t=[Member(id=8, username=teamB, age=0), Team(id=2, name=teamB)]
```

조인 - 페치 조인

페치 조인은 SQL에서 제공하는 기능은 아니다. SQL조인을 활용해서 연관된 엔티티를 SQL 한번에 조회하는 기능이다. 주로 성능 최적화에 사용하는 방법이다.

페치 조인 미적용

지연로딩으로 Member, Team SQL 쿼리 각각 실행

```
@PersistenceUnit
EntityManagerFactory emf;

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

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

    boolean loaded =
emf.getPersistenceUnitUtil().isLoaded(findMember.getTeam());
```

```
    assertThat(loaded).as("페치 조인 미적용").isFalse();
}
```

페치 조인 적용

즉시로딩으로 Member, Team SQL 쿼리 조인으로 한번에 조회

```
@Test
public void fetchJoinUse() throws Exception {
    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("페치 조인 적용").isTrue();
}
```

사용방법

- `join(), leftJoin()` 등 조인 기능 뒤에 `fetchJoin()` 이라고 추가하면 된다.

> 참고: 페치 조인에 대한 자세한 내용은 JPA 기본편이나, 활용2편을 참고하자

서브 쿼리

`com.querydsl.jpa.JPAExpressions` 사용

서브 쿼리 eq 사용

```
/**
 * 나이가 가장 많은 회원 조회
 */
@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);
}
```

서브 쿼리 goe 사용

```
/**
 * 나이가 평균 나이 이상인 회원
 */
@Test
public void subQueryGoe() throws Exception {
    QMember memberSub = new QMember("memberSub");

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

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

서브쿼리 여러 건 처리 in 사용

```
/**
 * 서브쿼리 여러 건 처리, in 사용
 */
@Test
public void subQueryIn() throws Exception {
    QMember memberSub = new QMember("memberSub");

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

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

select 절에 subquery

```
List<Tuple> fetch = queryFactory
        .select(member.username,
                JPAExpressions
                        .select(memberSub.age.avg())
                        .from(memberSub)
        ).from(member)
```

```
        .fetch();

for (Tuple tuple : fetch) {
    System.out.println("username = " + tuple.get(member.username));
    System.out.println("age = " +
tuple.get(JPAExpressions.select(memberSub.age.avg())
            .from(memberSub)));
}
```

static import 활용

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

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

from 절의 서브쿼리 한계

JPA JPQL 서브쿼리의 한계점으로 from 절의 서브쿼리(인라인 뷰)는 지원하지 않는다. 당연히 Querydsl 도 지원하지 않는다. 하이버네이트 구현체를 사용하면 select 절의 서브쿼리는 지원한다. Querydsl도 하이버네이트 구현체를 사용하면 select 절의 서브쿼리를 지원한다.

from 절의 서브쿼리 해결방안

1. 서브쿼리를 join으로 변경한다. (가능한 상황도 있고, 불가능한 상황도 있다.)
2. 애플리케이션에서 쿼리를 2번 분리해서 실행한다.
3. nativeSQL을 사용한다.

Case 문

select, 조건절(where), order by에서 사용 가능

단순한 조건

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

복잡한 조건

```
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();
```

orderBy에서 Case 문 함께 사용하기 예제

> **참고: 강의 이후 추가된 내용입니다.**

예를 들어서 다음과 같은 임의의 순서로 회원을 출력하고 싶다면?

1. 0 ~ 30살이 아닌 회원을 가장 먼저 출력
2. 0 ~ 20살 회원 출력
3. 21 ~ 30살 회원 출력

```
NumberExpression<Integer> rankPath = new CaseBuilder()
        .when(member.age.between(0, 20)).then(2)
        .when(member.age.between(21, 30)).then(1)
```

```
        .otherwise(3);

List<Tuple> result = queryFactory
        .select(member.username, member.age, rankPath)
        .from(member)
        .orderBy(rankPath.desc())
        .fetch();

for (Tuple tuple : result) {
    String username = tuple.get(member.username);
    Integer age = tuple.get(member.age);
    Integer rank = tuple.get(rankPath);
    System.out.println("username = " + username + " age = " + age + " rank = "
+ rank);
}
```

Querydsl은 자바 코드로 작성하기 때문에 `rankPath` 처럼 복잡한 조건을 변수로 선언해서 `select` 절, `orderBy` 절에서 함께 사용할 수 있다.

```
결과
username = member4 age = 40 rank = 3
username = member1 age = 10 rank = 2
username = member2 age = 20 rank = 2
username = member3 age = 30 rank = 1
```

상수, 문자 더하기

상수가 필요하면 `Expressions.constant(xxx)` 사용

```
Tuple result = queryFactory
        .select(member.username, Expressions.constant("A"))
        .from(member)
        .fetchFirst();
```

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

문자 더하기 concat

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

- 결과: member1_10

> 참고: `member.age.stringValue()` 부분이 중요한데, 문자가 아닌 다른 타입들은 `stringValue()` 로 문자로 변환할 수 있다. 이 방법은 ENUM을 처리할 때도 자주 사용한다.

중급 문법

프로젝션과 결과 반환 - 기본

프로젝션: select 대상 지정

프로젝션 대상이 하나

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

- 프로젝션 대상이 하나면 타입을 명확하게 지정할 수 있음

- 프로젝션 대상이 둘 이상이면 튜플이나 DTO로 조회

튜플 조회

프로젝션 대상이 둘 이상일 때 사용

`com.querydsl.core.Tuple`

```
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);
    System.out.println("username=" + username);
    System.out.println("age=" + age);
}
```

프로젝션과 결과 반환 - DTO 조회

순수 JPA에서 DTO 조회

MemberDto

```
package study.querydsl.dto;

import lombok.Data;

@Data
public class MemberDto {
    private String username;
    private int age;

    public MemberDto() {
    }
```

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

순수 JPA에서 DTO 조회 코드

```
List<MemberDto> result = em.createQuery(
        "select new study.querydsl.dto.MemberDto(m.username, m.age) " +
                "from Member m", MemberDto.class)
        .getResultList();
```

- 순수 JPA에서 DTO를 조회할 때는 new 명령어를 사용해야함
- DTO의 package이름을 다 적어줘야해서 지저분함
- 생성자 방식만 지원함

Querydsl 빈 생성(Bean population)

결과를 DTO 반환할 때 사용
다음 3가지 방법 지원

- 프로퍼티 접근
- 필드 직접 접근
- 생성자 사용

프로퍼티 접근 - Setter

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

필드 직접 접근

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

별칭이 다를 때

```
package study.querydsl.dto;

import lombok.Data;

@Data
public class UserDto {
    private String name;
    private int age;
}
```

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

- 프로퍼티나, 필드 접근 생성 방식에서 이름이 다를 때 해결 방안

- `ExpressionUtils.as(source,alias)` : 필드나, 서브 쿼리에 별칭 적용
- `username.as("memberName")` : 필드에 별칭 적용

생성자 사용

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

프로젝션과 결과 반환 - @QueryProjection

생성자 + @QueryProjection

```
package study.querydsl.dto;

import com.querydsl.core.annotations.QueryProjection;
import lombok.Data;

@Data
public class MemberDto {
    private String username;
    private int age;

    public MemberDto() {
    }

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

```
        this.age = age;
    }
}
```

- `./gradlew compileQuerydsl`
- `QMemberDto` 생성 확인

@QueryProjection 활용

```
List<MemberDto> result = queryFactory
        .select(new QMemberDto(member.username, member.age))
        .from(member)
        .fetch();
```

이 방법은 컴파일러로 타입을 체크할 수 있으므로 가장 안전한 방법이다. 다만 DTO에 QueryDSL 어노테이션을 유지해야 하는 점과 DTO까지 Q 파일을 생성해야 하는 단점이 있다.

distinct

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

> 참고: distinct는 JPQL의 distinct와 같다.

동적 쿼리 - BooleanBuilder 사용

동적 쿼리를 해결하는 두가지 방식

- BooleanBuilder
- Where 다중 파라미터 사용

```
@Test
public void 동적쿼리_BooleanBuilder() throws Exception {
    String usernameParam = "member1";
```

```
    Integer ageParam = 10;

    List<Member> result = searchMember1(usernameParam, ageParam);
    Assertions.assertThat(result.size()).isEqualTo(1);
}

private List<Member> searchMember1(String usernameCond, Integer ageCond) {
    BooleanBuilder 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();
}
```

동적 쿼리 - Where 다중 파라미터 사용

```
@Test
public void 동적쿼리_WhereParam() throws Exception {
    String usernameParam = "member1";
    Integer ageParam = 10;

    List<Member> result = searchMember2(usernameParam, ageParam);
    Assertions.assertThat(result.size()).isEqualTo(1);
}


private List<Member> searchMember2(String usernameCond, Integer ageCond) {
    return queryFactory
            .selectFrom(member)
            .where(usernameEq(usernameCond), ageEq(ageCond))
```

```
            .fetch();
}

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

}

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

- `where` 조건에 `null` 값은 무시된다.
- 메서드를 다른 쿼리에서도 재활용 할 수 있다.
- 쿼리 자체의 가독성이 높아진다.

조합 가능

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

- `null` 체크는 주의해서 처리해야함

수정, 삭제 벌크 연산

쿼리 한번으로 대량 데이터 수정

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

기존 숫자에 1 더하기

```
long count = queryFactory
        .update(member)
        .set(member.age, member.age.add(1))
        .execute();
```

곱하기: `multiply(x)`

```
update member
   set age = age + 1
```

쿼리 한번으로 대량 데이터 삭제

```
long count = queryFactory
        .delete(member)
        .where(member.age.gt(18))
        .execute();
```

> 주의: JPQL 배치와 마찬가지로, 영속성 컨텍스트에 있는 엔티티를 무시하고 실행되기 때문에 배치 쿼리를 실행하고 나면 영속성 컨텍스트를 초기화 하는 것이 안전하다.

SQL function 호출하기

SQL function은 JPA와 같이 Dialect에 등록된 내용만 호출할 수 있다.

member ➡ M으로 변경하는 replace 함수 사용

```
String result = queryFactory
        .select(Expressions.stringTemplate("function('replace', {0}, {1},
{2})", member.username, "member", "M"))
```

```
        .from(member)
        .fetchFirst();
```

소문자로 변경해서 비교해라.

```
.select(member.username)
.from(member)
.where(member.username.eq(Expressions.stringTemplate("function('lower', {0})",
member.username)))
```

lower 같은 ansi 표준 함수들은 querydsl이 상당부분 내장하고 있다. 따라서 다음과 같이 처리해도 결과는 같다.

```
.where(member.username.eq(member.username.lower()))
```

실무 활용 - 순수 JPA와 Querydsl

- 순수 JPA 리포지토리와 Querydsl
- 동적쿼리 Builder 적용
- 동적쿼리 Where 적용
- 조회 API 컨트롤러 개발

순수 JPA 리포지토리와 Querydsl

순수 JPA 리포지토리

```
package study.querydsl.repository;

import com.querydsl.core.BooleanBuilder;
import com.querydsl.core.types.dsl.BooleanExpression;
```

```
import com.querydsl.jpa.impl.JPAQueryFactory;
import org.springframework.stereotype.Repository;
import study.querydsl.dto.MemberSearchCondition;
import study.querydsl.dto.MemberTeamDto;
import study.querydsl.dto.QMemberTeamDto;
import study.querydsl.entity.Member;

import javax.persistence.EntityManager;
import java.util.List;
import java.util.Optional;

import static org.springframework.util.StringUtils.hasText;
import static org.springframework.util.StringUtils.isEmpty;
import static study.querydsl.entity.QMember.member;
import static study.querydsl.entity.QTeam.team;

@Repository
public class MemberJpaRepository {

    private final EntityManager em;
    private final JPAQueryFactory queryFactory;

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

    public void save(Member member) {
        em.persist(member);
    }

    public Optional<Member> findById(Long id) {
        Member findMember = em.find(Member.class, id);
        return Optional.ofNullable(findMember);
    }

    public List<Member> findAll() {
        return em.createQuery("select m from Member m", Member.class)
                .getResultList();
```

```
    }

    public List<Member> findByUsername(String username) {
        return em.createQuery("select m from Member m where m.username
= :username", Member.class)
                .setParameter("username", username)
                .getResultList();
    }

}
```

순수 JPA 리포지토리 테스트

```
package study.querydsl.repository;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.transaction.annotation.Transactional;
import study.querydsl.dto.MemberSearchCondition;
import study.querydsl.dto.MemberTeamDto;
import study.querydsl.entity.Member;
import study.querydsl.entity.Team;

import javax.persistence.EntityManager;
import java.util.List;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest
@Transactional
class MemberJpaRepositoryTest {

    @Autowired
    EntityManager em;
    @Autowired
    MemberJpaRepository memberJpaRepository;
```

```
@Test
public void basicTest() {
    Member member = new Member("member1", 10);
    memberJpaRepository.save(member);

    Member findMember = memberJpaRepository.findById(member.getId()).get();
    assertThat(findMember).isEqualTo(member);

    List<Member> result1 = memberJpaRepository.findAll();
    assertThat(result1).containsExactly(member);

    List<Member> result2 = memberJpaRepository.findByUsername("member1");
    assertThat(result2).containsExactly(member);
}
```

Querydsl 사용

순수 JPA 리포지토리 - Querydsl 추가

```
public List<Member> findAll_Querydsl() {
    return queryFactory
            .selectFrom(member).fetch();
}

public List<Member> findByUsername_Querydsl(String username) {
    return queryFactory
            .selectFrom(member)
            .where(member.username.eq(username))
            .fetch();
}
```

Querydsl 테스트 추가

```
@Test
public void basicQuerydslTest() {
    Member member = new Member("member1", 10);
    memberJpaRepository.save(member);

    Member findMember = memberJpaRepository.findById(member.getId()).get();
    assertThat(findMember).isEqualTo(member);

    List<Member> result1 = memberJpaRepository.findAll_Querydsl();
    assertThat(result1).containsExactly(member);

    List<Member> result2 =
memberJpaRepository.findByUsername_Querydsl("member1");
    assertThat(result2).containsExactly(member);
}
```

JPAQueryFactory 스프링 빈 등록

다음과 같이 `JPAQueryFactory`를 스프링 빈으로 등록해서 주입받아 사용해도 된다.

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

> 참고: 동시성 문제는 걱정하지 않아도 된다. 왜냐하면 여기서 스프링이 주입해주는 엔티티 매니저는 실제 동작 시점에 진짜 엔티티 매니저를 찾아주는 프록시용 가짜 엔티티 매니저이다. 이 가짜 엔티티 매니저는 실제 사용 시점에 트랜잭션 단위로 실제 엔티티 매니저(영속성 컨텍스트)를 할당해준다.
> 더 자세한 내용은 자바 ORM 표준 JPA 책 13.1 트랜잭션 범위의 영속성 컨텍스트를 참고하자.

동적 쿼리와 성능 최적화 조회 - Builder 사용

MemberTeamDto - 조회 최적화용 DTO 추가

```
package study.querydsl.dto;
```

```
import com.querydsl.core.annotations.QueryProjection;
import lombok.Data;

@Data
public class MemberTeamDto {
    private Long memberId;
    private String username;
    private int age;
    private Long teamId;
    private String teamName;

    @QueryProjection
    public MemberTeamDto(Long memberId, String username, int age, Long teamId,
String teamName) {
        this.memberId = memberId;
        this.username = username;
        this.age = age;
        this.teamId = teamId;
        this.teamName = teamName;
    }
}
```

- `@QueryProjection`을 추가했다. `QMemberTeamDto`를 생성하기 위해 `./gradlew compileQuerydsl`을 한번 실행하자.

> 참고: `@QueryProjection`을 사용하면 해당 DTO가 Querydsl을 의존하게 된다. 이런 의존이 싫으면, 해당 에노테이션을 제거하고, `Projection.bean(), fields(), constructor()`을 사용하면 된다.

회원 검색 조건

```
package study.querydsl.dto;

import lombok.Data;

@Data
public class MemberSearchCondition {
    //회원명, 팀명, 나이(ageGoe, ageLoe)
```

```
    private String username;
    private String teamName;
    private Integer ageGoe;
    private Integer ageLoe;
}
```

- 이름이 너무 길면 `MemberCond` 등으로 줄여 사용해도 된다.

동적쿼리 - Builder 사용

Builder를 사용한 예제

```
//Builder 사용
//회원명, 팀명, 나이(ageGoe, ageLoe)
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,
                    member.username,
                    member.age,
                    team.id,
                    team.name))
            .from(member)
            .leftJoin(member.team, team)
```

```
                .where(builder)
                .fetch();
    }
```

오류 정정

강의 영상에서는 `member.id.as("memberId")` 라고 적었는데, `QMemberTeamDto` 는 생성자를 사용하기 때문에 필드 이름을 맞추지 않아도 된다. 따라서 `member.id` 만 적으면 된다.

조회 예제 테스트

```
@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);

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

```
}
```

동적 쿼리와 성능 최적화 조회 - Where절 파라미터 사용

Where절에 파라미터를 사용한 예제

```
//회원명, 팀명, 나이(ageGoe, ageLoe)
public List<MemberTeamDto> search(MemberSearchCondition condition) {
    return queryFactory
            .select(new QMemberTeamDto(
                    member.id,
                    member.username,
                    member.age,
                    team.id,
                    team.name))
            .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 isEmpty(username) ? null : member.username.eq(username);
}

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

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

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

참고: where 절에 파라미터 방식을 사용하면 조건 재사용 가능

```
//where 파라미터 방식은 이런식으로 재사용이 가능하다.
public List<Member> findMember(MemberSearchCondition condition) {
    return queryFactory
            .selectFrom(member)
            .leftJoin(member.team, team)
            .where(usernameEq(condition.getUsername()),
                    teamNameEq(condition.getTeamName()),
                    ageGoe(condition.getAgeGoe()),
                    ageLoe(condition.getAgeLoe()))
            .fetch();
}
```

조회 API 컨트롤러 개발

편리한 데이터 확인을 위해 샘플 데이터를 추가하자.
샘플 데이터 추가가 테스트 케이스 실행에 영향을 주지 않도록 다음과 같이 프로파일을 설정하자

프로파일 설정

`src/main/resources/application.yml`

```
spring:
  profiles:
```

```
    active: local
```

테스트는 기존 application.yml을 복사해서 다음 경로로 복사하고, 프로파일을 test로 수정하자

`src/test/resources/application.yml`

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

이렇게 분리하면 main 소스코드와 테스트 소스 코드 실행시 프로파일을 분리할 수 있다.

샘플 데이터 추가

```
package study.querydsl;

import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;
import study.querydsl.entity.Member;
import study.querydsl.entity.Team;

import javax.annotation.PostConstruct;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

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

    private final InitMemberService initMemberService;

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

```
    @Component
    static class InitMemberService {

        @PersistenceContext
        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" + i, i, selectedTeam));
            }

        }
    }
}
```

조회 컨트롤러

```
package study.querydsl.controller;

import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import study.querydsl.dto.MemberSearchCondition;
import study.querydsl.dto.MemberTeamDto;
import study.querydsl.repository.MemberJpaRepository;
```

```
import java.util.List;

@RestController
@RequiredArgsConstructor
public class MemberController {

    private final MemberJpaRepository memberJpaRepository;

    @GetMapping("/v1/members")
    public List<MemberTeamDto> searchMemberV1(MemberSearchCondition condition)
{
        return memberJpaRepository.search(condition);
    }

}
```

- 예제 실행(postman)
- `http://localhost:8080/v1/members?teamName=teamB&ageGoe=31&ageLoe=35`

실무 활용 - 스프링 데이터 JPA와 Querydsl

스프링 데이터 JPA 리포지토리로 변경

스프링 데이터 JPA - MemberRepository 생성

```
package study.querydsl.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import study.querydsl.entity.Member;

import java.util.List;

public interface MemberRepository extends JpaRepository<Member, Long> {

    List<Member> findByUsername(String username);
```

```
}
```

스프링 데이터 JPA 테스트

```
package study.querydsl.repository;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.transaction.annotation.Transactional;
import study.querydsl.dto.MemberSearchCondition;
import study.querydsl.dto.MemberTeamDto;
import study.querydsl.entity.Member;
import study.querydsl.entity.Team;

import javax.persistence.EntityManager;
import java.util.List;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest
@Transactional
class MemberRepositoryTest {

    @Autowired
    EntityManager em;
    @Autowired
    MemberRepository memberRepository;

    @Test
    public void basicTest() {
        Member member = new Member("member1", 10);
        memberRepository.save(member);

        Member findMember = memberRepository.findById(member.getId()).get();
        assertThat(findMember).isEqualTo(member);
```

```
        List<Member> result1 = memberRepository.findAll();
        assertThat(result1).containsExactly(member);

        List<Member> result2 = memberRepository.findByUsername("member1");
        assertThat(result2).containsExactly(member);
    }
}
```

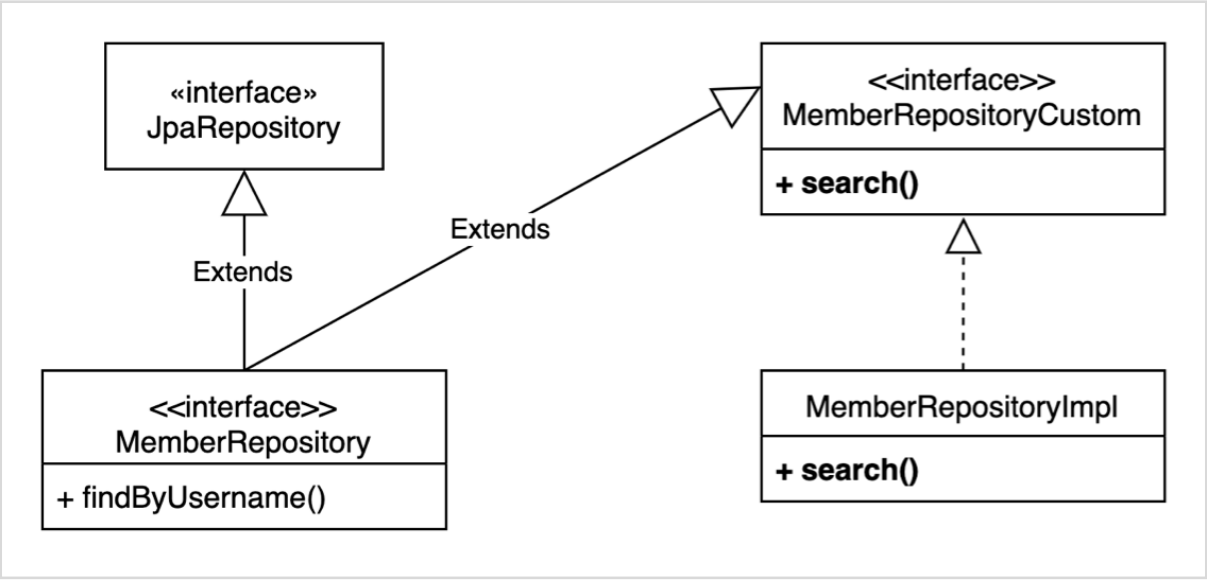
- Querydsl 전용 기능인 회원 search를 작성할 수 없다. ➡ 사용자 정의 리포지토리 필요

사용자 정의 리포지토리

사용자 정의 리포지토리 사용법

1. 사용자 정의 인터페이스 작성
2. 사용자 정의 인터페이스 구현
3. 스프링 데이터 리포지토리에 사용자 정의 인터페이스 상속

사용자 정의 리포지토리 구성



1. 사용자 정의 인터페이스 작성

```
package study.querydsl.repository;
```

```
import study.querydsl.dto.MemberSearchCondition;
import study.querydsl.dto.MemberTeamDto;

import java.util.List;

public interface MemberRepositoryCustom {

    List<MemberTeamDto> search(MemberSearchCondition condition);
}
```

2. 사용자 정의 인터페이스 구현

```
package study.querydsl.repository;

import com.querydsl.core.types.dsl.BooleanExpression;
import com.querydsl.jpa.impl.JPAQueryFactory;
import study.querydsl.dto.MemberSearchCondition;
import study.querydsl.dto.MemberTeamDto;
import study.querydsl.dto.QMemberTeamDto;

import javax.persistence.EntityManager;
import java.util.List;

import static org.springframework.util.StringUtils.isEmpty;
import static study.querydsl.entity.QMember.member;
import static study.querydsl.entity.QTeam.team;

public class MemberRepositoryImpl implements MemberRepositoryCustom {

    private final JPAQueryFactory queryFactory;

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

    @Override
    //회원명, 팀명, 나이(ageGoe, ageLoe)
    public List<MemberTeamDto> search(MemberSearchCondition condition) {
```

```
        return queryFactory
                .select(new QMemberTeamDto(
                        member.id,
                        member.username,
                        member.age,
                        team.id,
                        team.name))
                .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 isEmpty(username) ? null : member.username.eq(username);
    }

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

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

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

3. 스프링 데이터 리포지토리에 사용자 정의 인터페이스 상속

```
package study.querydsl.repository;
```

```
import org.springframework.data.jpa.repository.JpaRepository;
import study.querydsl.entity.Member;

import java.util.List;

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

    List<Member> findByUsername(String username);
}
```

커스텀 리포지토리 동작 테스트 추가

```
@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 = memberRepository.search(condition);
```

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

스프링 데이터 페이징 활용1 - Querydsl 페이징 연동

- 스프링 데이터의 Page, Pageable을 활용해보자.
- 전체 카운트를 한번에 조회하는 단순한 방법
- 데이터 내용과 전체 카운트를 별도로 조회하는 방법

사용자 정의 인터페이스에 페이징 2가지 추가

```
package study.querydsl.repository;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import study.querydsl.dto.MemberSearchCondition;
import study.querydsl.dto.MemberTeamDto;

import java.util.List;

public interface MemberRepositoryCustom {

    List<MemberTeamDto> search(MemberSearchCondition condition);

    Page<MemberTeamDto> searchPageSimple(MemberSearchCondition condition,
Pageable pageable);
    Page<MemberTeamDto> searchPageComplex(MemberSearchCondition condition,
Pageable pageable);
}
```

전체 카운트를 한번에 조회하는 단순한 방법

searchPageSimple(), fetchResults() 사용

```
/**
 * 단순한 페이징, fetchResults() 사용
 */
@Override
public Page<MemberTeamDto> searchPageSimple(MemberSearchCondition condition,
Pageable pageable) {

    QueryResults<MemberTeamDto> results = queryFactory
            .select(new QMemberTeamDto(
                    member.id,
                    member.username,
                    member.age,
                    team.id,
                    team.name))
            .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())
            .fetchResults();

    List<MemberTeamDto> content = results.getResults();
    long total = results.getTotal();

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

- Querydsl이 제공하는 `fetchResults()` 를 사용하면 내용과 전체 카운트를 한번에 조회할 수 있다.(실제 쿼리는 2번 호출)
- `fetchResult()` 는 카운트 쿼리 실행시 필요없는 `order by` 는 제거한다.

데이터 내용과 전체 카운트를 별도로 조회하는 방법

searchPageComplex()

```
/**
 * 복잡한 페이징
 * 데이터 조회 쿼리와, 전체 카운트 쿼리를 분리
 */
@Override
public Page<MemberTeamDto> searchPageComplex(MemberSearchCondition condition,
Pageable pageable) {

    List<MemberTeamDto> content = queryFactory
            .select(new QMemberTeamDto(
                    member.id,
                    member.username,
                    member.age,
                    team.id,
                    team.name))
            .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)
            .from(member)
            .leftJoin(member.team, team)
            .where(usernameEq(condition.getUsername()),
                    teamNameEq(condition.getTeamName()),
                    ageGoe(condition.getAgeGoe()),
                    ageLoe(condition.getAgeLoe()))
            .fetchCount();

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

```
}
```

- 전체 카운트를 조회 하는 방법을 최적화 할 수 있으면 이렇게 분리하면 된다. (예를 들어서 전체 카운트를 조회할 때 조인 쿼리를 줄일 수 있다면 상당한 효과가 있다.)
- 코드를 리펙토링해서 내용 쿼리과 전체 카운트 쿼리를 읽기 좋게 분리하면 좋다.

스프링 데이터 페이징 활용2 - CountQuery 최적화

PageableExecutionUtils.getPage()로 최적화

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

//        return new PageImpl<>(content, pageable, total);
        return PageableExecutionUtils.getPage(content, pageable,
countQuery::fetchCount);
```

- 스프링 데이터 라이브러리가 제공
- count 쿼리가 생략 가능한 경우 생략해서 처리
  - 페이지 시작이면서 컨텐츠 사이즈가 페이지 사이즈보다 작을 때
  - 마지막 페이지 일 때 (offset + 컨텐츠 사이즈를 더해서 전체 사이즈 구함, 더 정확히는 마지막 페이지이면서 컨텐츠 사이즈가 페이지 사이즈보다 작을 때)

스프링 데이터 페이징 활용3 - 컨트롤러 개발

실제 컨트롤러

```
package study.querydsl.controller;

import lombok.RequiredArgsConstructor;
```

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import study.querydsl.dto.MemberSearchCondition;
import study.querydsl.dto.MemberTeamDto;
import study.querydsl.repository.MemberJpaRepository;
import study.querydsl.repository.MemberRepository;

import java.util.List;

@RestController
@RequiredArgsConstructor
public class MemberController {

    private final MemberJpaRepository memberJpaRepository;
    private final MemberRepository memberRepository;

    @GetMapping("/v1/members")
    public List<MemberTeamDto> searchMemberV1(MemberSearchCondition condition)
{
        return memberJpaRepository.search(condition);
    }
    @GetMapping("/v2/members")
    public Page<MemberTeamDto> searchMemberV2(MemberSearchCondition condition,
Pageable pageable) {
        return memberRepository.searchPageSimple(condition, pageable);
    }
    @GetMapping("/v3/members")
    public Page<MemberTeamDto> searchMemberV3(MemberSearchCondition condition,
Pageable pageable) {
        return memberRepository.searchPageComplex(condition, pageable);
    }
}
```

- `http://localhost:8080/v2/members?size=5&page=2`

스프링 데이터 정렬(Sort)

스프링 데이터 JPA는 자신의 정렬(Sort)을 Querydsl의 정렬(OrderSpecifier)로 편리하게 변경하는 기능을 제공한다. 이 부분은 뒤에 스프링 데이터 JPA가 제공하는 Querydsl 기능에서 살펴보겠다.

스프링 데이터의 정렬을 Querydsl의 정렬로 직접 전환하는 방법은 다음 코드를 참고하자.

스프링 데이터 Sort를 Querydsl의 OrderSpecifier로 변환

```
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();
```

> 참고: 정렬(`Sort`)은 조건이 조금만 복잡해져도 `Pageable` 의 `Sort` 기능을 사용하기 어렵다. 루트 엔티티 범위를 넘어가는 동적 정렬 기능이 필요하면 스프링 데이터 페이징이 제공하는 `Sort` 를 사용하기 보다는 파라미터를 받아서 직접 처리하는 것을 권장한다.

스프링 데이터 JPA가 제공하는 Querydsl 기능

여기서 소개하는 기능은 제약이 커서 복잡한 실무 환경에서 사용하기에는 많이 부족하다. 그래도 스프링 데이터에서 제공하는 기능이므로 간단히 소개하고, 왜 부족한지 설명하겠다.

인터페이스 지원 - QuerydslPredicateExecutor

- 공식 URL: https://docs.spring.io/spring-data/jpa/docs/2.2.3.RELEASE/reference/html/#core.extensions.querydsl

QuerydslPredicateExecutor 인터페이스

```
public interface QuerydslPredicateExecutor<T> {
```

```
  Optional<T> findById(Predicate predicate);
  Iterable<T> findAll(Predicate predicate);
  long count(Predicate predicate);
  boolean exists(Predicate predicate);

  // … more functionality omitted.
}
```

리포지토리에 적용

```
interface MemberRepository extends JpaRepository<User, Long>,
QuerydslPredicateExecutor<User> {
}
```

```
Iterable result = memberRepository.findAll(
        member.age.between(10, 40)
        .and(member.username.eq("member1"))
);
```

한계점

- 조인X (묵시적 조인은 가능하지만 left join이 불가능하다.)
- 클라이언트가 Querydsl에 의존해야 한다. 서비스 클래스가 Querydsl이라는 구현 기술에 의존해야 한다.
- 복잡한 실무환경에서 사용하기에는 한계가 명확하다.

> 참고: `QuerydslPredicateExecutor`는 Pagable, Sort를 모두 지원하고 정상 동작한다.

Querydsl Web 지원

- 공식 URL: https://docs.spring.io/spring-data/jpa/docs/2.2.3.RELEASE/reference/html/#core.web.type-safe

한계점

- 단순한 조건만 가능
- 조건을 커스텀하는 기능이 복잡하고 명시적이지 않음
- 컨트롤러가 Querydsl에 의존
- 복잡한 실무환경에서 사용하기에는 한계가 명확

리포지토리 지원 - QuerydslRepositorySupport

QuerydslRepositorySupport

장점

- `getQuerydsl().applyPagination()` 스프링 데이터가 제공하는 페이징을 Querydsl로 편리하게 변환 가능(단! Sort는 오류발생)
- `from()`으로 시작 가능(최근에는 QueryFactory를 사용해서 `select()`로 시작하는 것이 더 명시적)
- EntityManager 제공

한계

- Querydsl 3.x 버전을 대상으로 만듬
- Querydsl 4.x에 나온 JPAQueryFactory로 시작할 수 없음
  - select로 시작할 수 없음 (from으로 시작해야함)
- `QueryFactory`를 제공하지 않음
- 스프링 데이터 Sort 기능이 정상 동작하지 않음

Querydsl 지원 클래스 직접 만들기

스프링 데이터가 제공하는 `QuerydslRepositorySupport`가 지닌 한계를 극복하기 위해 직접 Querydsl 지원 클래스를 만들어보자.

장점

- 스프링 데이터가 제공하는 페이징을 편리하게 변환

- 페이징과 카운트 쿼리 분리 가능
- 스프링 데이터 Sort 지원
- `select()`, `selectFrom()`으로 시작 가능
- `EntityManager`, `QueryFactory` 제공

Querydsl4RepositorySupport

```
package study.querydsl.repository.support;

import com.querydsl.core.types.EntityPath;
import com.querydsl.core.types.Expression;
import com.querydsl.core.types.dsl.PathBuilder;
import com.querydsl.jpa.impl.JPAQuery;
import com.querydsl.jpa.impl.JPAQueryFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.support.JpaEntityInformation;
import
org.springframework.data.jpa.repository.support.JpaEntityInformationSupport;
import org.springframework.data.jpa.repository.support.Querydsl;
import org.springframework.data.querydsl.SimpleEntityPathResolver;
import org.springframework.data.repository.support.PageableExecutionUtils;
import org.springframework.stereotype.Repository;
import org.springframework.util.Assert;

import javax.annotation.PostConstruct;
import javax.persistence.EntityManager;
import java.util.List;
import java.util.function.Function;

/**
 * Querydsl 4.x 버전에 맞춘 Querydsl 지원 라이브러리
 *
 * @author Younghan Kim
 * @see
org.springframework.data.jpa.repository.support.QuerydslRepositorySupport
 */
```

```
@Repository
public abstract class Querydsl4RepositorySupport {

    private final Class domainClass;
    private Querydsl querydsl;
    private EntityManager entityManager;
    private JPAQueryFactory queryFactory;

    public Querydsl4RepositorySupport(Class<?> domainClass) {
        Assert.notNull(domainClass, "Domain class must not be null!");
        this.domainClass = domainClass;
    }

    @Autowired
    public void setEntityManager(EntityManager entityManager) {
        Assert.notNull(entityManager, "EntityManager must not be null!");

        JpaEntityInformation entityInformation =
JpaEntityInformationSupport.getEntityInformation(domainClass, entityManager);
        SimpleEntityPathResolver resolver = SimpleEntityPathResolver.INSTANCE;
        EntityPath path = resolver.createPath(entityInformation.getJavaType());
        this.entityManager = entityManager;
        this.querydsl = new Querydsl(entityManager, new
PathBuilder<>(path.getType(), path.getMetadata()));
        this.queryFactory = new JPAQueryFactory(entityManager);
    }

    @PostConstruct
    public void validate() {
        Assert.notNull(entityManager, "EntityManager must not be null!");
        Assert.notNull(querydsl, "Querydsl must not be null!");
        Assert.notNull(queryFactory, "QueryFactory must not be null!");
    }

    protected JPAQueryFactory getQueryFactory() {
        return queryFactory;
    }

    protected Querydsl getQuerydsl() {
```

```
        return querydsl;
    }

    protected EntityManager getEntityManager() {
        return entityManager;
    }

    protected <T> JPAQuery<T> select(Expression<T> expr) {
        return getQueryFactory().select(expr);
    }

    protected <T> JPAQuery<T> selectFrom(EntityPath<T> from) {
        return getQueryFactory().selectFrom(from);
    }

    protected <T> Page<T> applyPagination(Pageable pageable,
Function<JPAQueryFactory, JPAQuery> contentQuery) {
        JPAQuery jpaQuery = contentQuery.apply(getQueryFactory());
        List<T> content = getQuerydsl().applyPagination(pageable,
jpaQuery).fetch();
        return PageableExecutionUtils.getPage(content, pageable,
jpaQuery::fetchCount);
    }

    protected <T> Page<T> applyPagination(Pageable pageable,
Function<JPAQueryFactory, JPAQuery> contentQuery, Function<JPAQueryFactory,
JPAQuery> countQuery) {
        JPAQuery jpaContentQuery = contentQuery.apply(getQueryFactory());
        List<T> content = getQuerydsl().applyPagination(pageable,
jpaContentQuery).fetch();

        JPAQuery countResult = countQuery.apply(getQueryFactory());
        return PageableExecutionUtils.getPage(content, pageable,
countResult::fetchCount);
    }

}
```

Querydsl4RepositorySupport 사용 코드

```
package study.querydsl.repository;

import com.querydsl.core.types.dsl.BooleanExpression;
import com.querydsl.jpa.impl.JPAQuery;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.support.PageableExecutionUtils;
import org.springframework.stereotype.Repository;
import study.querydsl.dto.MemberSearchCondition;
import study.querydsl.entity.Member;
import study.querydsl.repository.support.Querydsl4RepositorySupport;

import java.util.List;

import static org.springframework.util.StringUtils.isEmpty;
import static study.querydsl.entity.QMember.member;
import static study.querydsl.entity.QTeam.team;

@Repository
public class MemberTestRepository extends Querydsl4RepositorySupport {

    public MemberTestRepository() {
        super(Member.class);
    }

    public List<Member> basicSelect() {
        return select(member)
                .from(member)
                .fetch();
    }

    public List<Member> basicSelectFrom() {
        return selectFrom(member)
                .fetch();
    }
```

```
    public Page<Member> searchPageByApplyPage(MemberSearchCondition condition,
Pageable pageable) {

        JPAQuery<Member> query = selectFrom(member)
                .leftJoin(member.team, team)
                .where(usernameEq(condition.getUsername()),
                        teamNameEq(condition.getTeamName()),
                        ageGoe(condition.getAgeGoe()),
                        ageLoe(condition.getAgeLoe()));

        List<Member> content = getQuerydsl().applyPagination(pageable, query)
                .fetch();

        return PageableExecutionUtils.getPage(content, pageable,
query::fetchCount);
    }

    public Page<Member> applyPagination(MemberSearchCondition condition,
Pageable pageable) {
        return applyPagination(pageable, contentQuery -> contentQuery
                .selectFrom(member)
                .leftJoin(member.team, team)
                .where(usernameEq(condition.getUsername()),
                        teamNameEq(condition.getTeamName()),
                        ageGoe(condition.getAgeGoe()),
                        ageLoe(condition.getAgeLoe())));
    }

    public Page<Member> applyPagination2(MemberSearchCondition condition,
Pageable pageable) {
        return applyPagination(pageable, contentQuery -> contentQuery
                        .selectFrom(member)
                        .leftJoin(member.team, team)
                        .where(usernameEq(condition.getUsername()),
                                teamNameEq(condition.getTeamName()),
                                ageGoe(condition.getAgeGoe()),
                                ageLoe(condition.getAgeLoe())),
                countQuery -> countQuery
```

```
                        .selectFrom(member)
                        .leftJoin(member.team, team)
                        .where(usernameEq(condition.getUsername()),
                                teamNameEq(condition.getTeamName()),
                                ageGoe(condition.getAgeGoe()),
                                ageLoe(condition.getAgeLoe()))
        );
    }


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

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

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

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

}
```

스프링 부트 2.6 이상, Querydsl 5.0 지원 방법

> 참고: 해당 내용은 강의 이후에 추가된 내용입니다.

최신 스프링 부트 2.6부터는 Querydsl 5.0을 사용한다.

스프링 부트 2.6 이상 사용시 다음과 같은 부분을 확인해야 한다.

- 1. `build.gradle` 설정 변경
- 2. `PageableExecutionUtils` Deprecated(향후 미지원) 패키지 변경
- 3. Querydsl `fetchResults()` , `fetchCount()` Deprecated(향후 미지원)

build.gradle 설정 방법

`build.gradle`

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

plugins {
    id 'org.springframework.boot' version '2.6.2'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    //querydsl 추가
    id "com.ewerk.gradle.plugins.querydsl" version "1.0.10"
    id 'java'
}

group = 'com.example'
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'

    //querydsl 추가
    implementation "com.querydsl:querydsl-jpa:${queryDslVersion}"
    annotationProcessor "com.querydsl:querydsl-apt:${queryDslVersion}"

    compileOnly 'org.projectlombok:lombok'
    runtimeOnly 'com.h2database:h2'
    annotationProcessor 'org.projectlombok:lombok'

    //테스트에서 lombok 사용
    testCompileOnly 'org.projectlombok:lombok'
    testAnnotationProcessor 'org.projectlombok:lombok'

    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
    }

}

test {
    useJUnitPlatform()
}


//querydsl 추가 시작
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 추가 끝
```

- `querydsl-jpa`, `querydsl-apt`를 추가하고 버전을 명시해야 한다.

참고로 위 설정은 JUnit은 4를 사용한다.
JUnit 5를 사용하려면 다음 설정을 제거하면 된다.
`exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'`

`PageableExecutionUtils` Deprecated(향후 미지원) 패키지 변경

PageableExecutionUtils 클래스 사용 패키지 변경
기능이 Deprecated 된 것은 아니고, 사용 패키지 위치가 변경되었습니다. 기존 위치를 신규 위치로 변경해 주시면 문제 없이 사용할 수 있습니다.

- 기존: `org.springframework.data.repository.support.PageableExecutionUtils`
- 신규: `org.springframework.data.support.PageableExecutionUtils`

Querydsl `fetchResults()`, `fetchCount()` Deprecated(향후 미지원)

Querydsl의 `fetchCount()`, `fetchResult()`는 개발자가 작성한 select 쿼리를 기반으로 count용 쿼리를 내부에서 만들어서 실행합니다.
그런데 이 기능은 강의에서 설명드린 것 처럼 select 구문을 단순히 count 처리하는 용도로 바꾸는 정도입니다. 따라서 단순한 쿼리에서는 잘 동작하지만, 복잡한 쿼리에서는 제대로 동작하지 않습니다.
Querydsl은 향후 `fetchCount()`, `fetchResult()`를 지원하지 않기로 결정했습니다.
참고로 Querydsl의 변화가 빠르지는 않기 때문에 당장 해당 기능을 제거하지는 않을 것입니다.

따라서 count 쿼리가 필요하면 다음과 같이 별도로 작성해야 합니다.

count 쿼리는 예제

```
@Test
public void count() {
    Long totalCount = queryFactory
            //.select(Wildcard.count) //select count(*)
```

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

        System.out.println("totalCount = " + totalCount);
    }
```

- `count(*)`을 사용하고 싶으면 예제의 주석처럼 `Wildcard.count`를 사용하시면 됩니다.
- `member.count()`를 사용하면 `count(member.id)`로 처리됩니다.
- 응답 결과는 숫자 하나이므로 `fetchOne()`을 사용합니다.

`MemberRepositoryImpl.searchPageComplex()` 예제에서 보여드린 것 처럼 select 쿼리와는 별도로 count 쿼리를 작성하고 `fetch()`를 사용해야 합니다. 다음은 최신 버전에 맞추어 수정된 예제입니다.

수정된 searchPageComplex 예제

```
import org.springframework.data.support.PageableExecutionUtils; //패키지 변경

public Page<MemberTeamDto> searchPageComplex(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();
```

```
    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);
}
```