실전! 스프링 데이터 JPA - v2021-04-13

#인강/jpa활용편/datajpa

인프런 강의: 실전! 스프링 부트와 JPA 활용1 - 웹 애플리케이션 개발
인프런: https://www.inflearn.com

버전 수정 이력

v2021-04-13

`@Query("select new study.datajpa.dto.MemberDto` 오타 수정

- 궁그미님 감사합니다.

v2021-04-02

- `@EntityGraph`에서 `@Param("username")` 잘못 사용된 부분 제거
  - 최준성님 감사합니다^^

v2021-03-07

- [오타수정] `select m from Team t` ➡ `select t from Team t`
  - 류재준님 감사합니다^^

v1.2 - 2020년 3월 28일

- BaseEntity extends BaseEntity PDF 문장 오류 수정
  - sangyong choi님 감사합니다^^

v1.1 - 2020년 3월 13일

- 사용자 정의 리포지토리 구현 - 사용자 정의 리포지토리 구현 최신 방식 추가

v1.0 - 2019년 11월 26일

- v1.0 최초 배포

목차

    - 라이브러리 살펴보기
    - H2 데이터베이스 설치
    - 스프링 데이터 JPA와 DB 설정, 동작확인
- 예제 도메인 모델
    - 예제 도메인 모델과 동작확인
- 공통 인터페이스 기능
    - 순수 JPA 기반 리포지토리 만들기
    - 공통 인터페이스 설정
    - 공통 인터페이스 적용
    - 공통 인터페이스 분석
- 쿼리 메소드 기능
    - 메소드 이름으로 쿼리 생성
    - JPA NamedQuery
    - @Query, 리포지토리 메소드에 쿼리 정의하기
    - @Query, 값, DTO 조회하기
    - 파라미터 바인딩
    - 반환 타입
    - 순수 JPA 페이징과 정렬
    - 스프링 데이터 JPA 페이징과 정렬
    - 벌크성 수정 쿼리
    - @EntityGraph
    - JPA Hint & Lock
- 확장 기능
    - 사용자 정의 리포지토리 구현
    - Auditing
    - Web 확장 - 도메인 클래스 컨버터
    - Web 확장 - 페이징과 정렬
- 스프링 데이터 JPA 분석
    - 스프링 데이터 JPA 구현체 분석
    - 새로운 엔티티를 구별하는 방법
- 나머지 기능들
    - Specifications (명세)
    - Query By Example
    - Projections
    - 네이티브 쿼리

스프링 데이터 JPA 소개

소개

강의 자료

프로젝트 환경설정

프로젝트 생성

- 스프링 부트 스타터(https://start.spring.io/)
- 사용 기능: web, jpa, h2, lombok
  - SpringBootVersion: **2.2.1**
  - groupId: study
  - artifactId: data-jpa

Gradle 전체 설정

```
plugins {
  id 'org.springframework.boot' version '2.2.1.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'
  implementation 'com.github.gavlyukovskiy:p6spy-spring-boot-starter:1.5.7'
  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.datajpa.controller;

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

@RestController
public class HelloController {

    @RequestMapping("/hello")
    public String hello() {
        return "hello";
```

```
    }
}
```

> 참고: 최근 IntelliJ 버전은 Gradle로 실행을 하는 것이 기본 설정이다. 이렇게 하면 실행속도가 느리다.
> 다음과 같이 변경하면 자바로 바로 실행하므로 좀 더 빨라진다.
>
> Preferences ➜ Build, Execution, Deployment ➜ Build Tools ➜ Gradle
> Build and run using: Gradle ➜ IntelliJ IDEA
> 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

라이브러리 살펴보기

gradle 의존관계 보기

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

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
    - 스프링 ORM
    - JPA, 하이버네이트
    - 스프링 데이터 JPA
- 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:~/datajpa` (최소 한번)
    - `~/datajpa.mv.db` 파일 생성 확인
    - 이후 부터는 `jdbc:h2:tcp://localhost/~/datajpa` 이렇게 접속

> 참고: H2 데이터베이스의 MVCC 옵션은 H2 1.4.198 버전부터 제거되었습니다. 사용 버전이 1.4.199 이므로 옵션 없이 사용하면 됩니다.

스프링 데이터 JPA와 DB 설정, 동작확인

`application.yml`

```yaml
spring:
  datasource:
    url: jdbc:h2:tcp://localhost/~/datajpa
    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.

실제 동작하는지 확인하기

회원 엔티티

```
@Entity
@Getter @Setter
public class Member {

    @Id @GeneratedValue
    private Long id;
    private String username;
    ...
}
```

회원 JPA 리포지토리

```
@Repository
public class MemberJpaRepository {

    @PersistenceContext
    private EntityManager em;

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

    public Member find(Long id) {
        return em.find(Member.class, id);
    }
}
```

JPA 기반 테스트

```
@SpringBootTest
@Transactional
@Rollback(false)
public class MemberJpaRepositoryTest {

    @Autowired
    MemberJpaRepository memberJpaRepository;

    @Test
    public void testMember() {
        Member member = new Member("memberA");
        Member savedMember = memberJpaRepository.save(member);

        Member findMember = memberJpaRepository.find(savedMember.getId());

        assertThat(findMember.getId()).isEqualTo(member.getId());
        assertThat(findMember.getUsername()).isEqualTo(member.getUsername());

        assertThat(findMember).isEqualTo(member); //JPA 엔티티 동일성 보장
    }
}
```

스프링 데이터 JPA 리포지토리

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

}
```

스프링 데이터 JPA 기반 테스트

```
@SpringBootTest
@Transactional
@Rollback(false)
public class MemberRepositoryTest {
```

```
    @Autowired
    MemberRepository memberRepository;

    @Test
    public void testMember() {
        Member member = new Member("memberA");
        Member savedMember = memberRepository.save(member);

        Member findMember =
memberRepository.findById(savedMember.getId()).get();

        Assertions.assertThat(findMember.getId()).isEqualTo(member.getId());

Assertions.assertThat(findMember.getUsername()).isEqualTo(member.getUsername())
;

        Assertions.assertThat(findMember).isEqualTo(member); //JPA 엔티티 동일성
보장
    }

}
```

- Entity, Repository 동작 확인
- jar 빌드해서 동작 확인

> 참고: 스프링 부트를 통해 복잡한 설정이 다 자동화 되었다. `persistence.xml` 도 없고, `LocalContainerEntityManagerFactoryBean` 도 없다. 스프링 부트를 통한 추가 설정은 스프링 부트 메뉴얼을 참고하고, 스프링 부트를 사용하지 않고 순수 스프링과 JPA 설정 방법은 자바 ORM 표준 JPA 프로그래밍 책을 참고하자.

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.

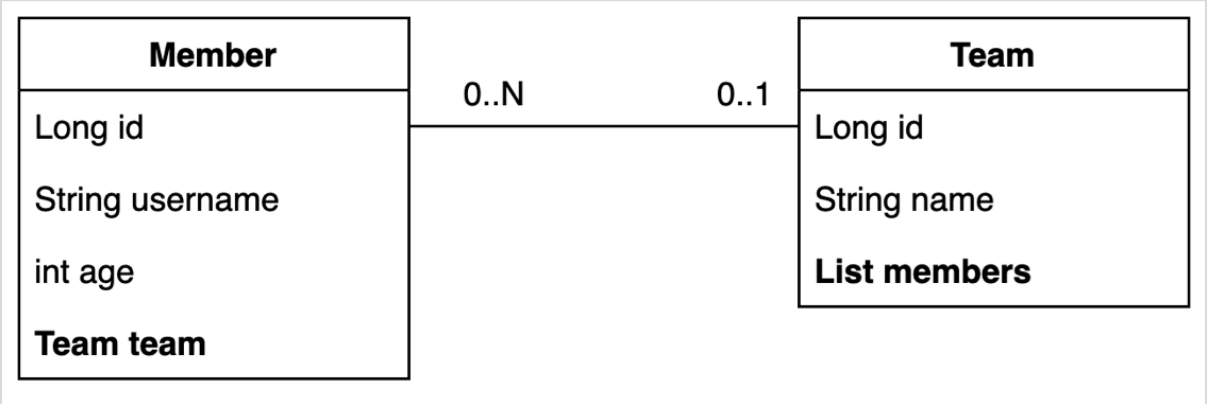
```
implementation 'com.github.gavlyukovskiy:p6spy-spring-boot-starter:1.5.7'
```

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

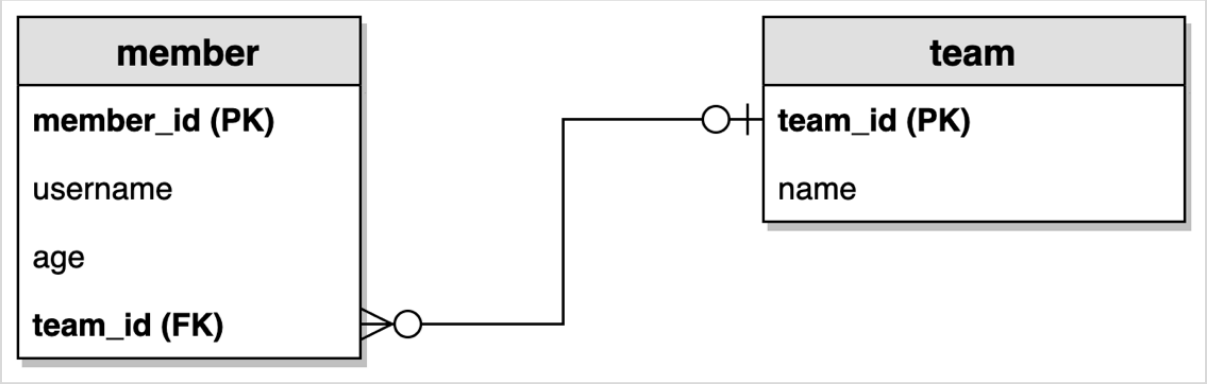
예제 도메인 모델

예제 도메인 모델과 동작확인

엔티티 클래스



ERD



Member 엔티티

```
package study.datajpa.entity;
```

```java
import lombok.AccessLevel;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import javax.persistence.*;

@Entity
@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.datajpa.entity;

import lombok.AccessLevel;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

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")
    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.datajpa.entity;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.annotation.Rollback;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Transactional;

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

@SpringBootTest
public class MemberTest {

    @PersistenceContext
    EntityManager em;

    @Test
    @Transactional
    @Rollback(false)
    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

공통 인터페이스 기능

- 순수 JPA 기반 리포지토리 만들기
- 스프링 데이터 JPA 공통 인터페이스 소개
- 스프링 데이터 JPA 공통 인터페이스 활용

순수 JPA 기반 리포지토리 만들기

- 순수한 JPA 기반 리포지토리를 만들자
- 기본 CRUD
  - 저장
  - 변경 ➡ 변경감지 사용
  - 삭제
  - 전체 조회
  - 단건 조회
  - 카운트

> 참고: JPA에서 수정은 변경감지 기능을 사용하면 된다.
> 트랜잭션 안에서 엔티티를 조회한 다음에 데이터를 변경하면, 트랜잭션 종료 시점에 변경감지 기능이 작동해서 변경된 엔티티를 감지하고 UPDATE SQL을 실행한다.

순수 JPA 기반 리포지토리 - 회원

```
package study.datajpa.repository;

import org.springframework.stereotype.Repository;
import study.datajpa.entity.Member;

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

@Repository
public class MemberJpaRepository {

    @PersistenceContext
    private EntityManager em;
```

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

    public void delete(Member member) {
        em.remove(member);
    }

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

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

    public long count() {
        return em.createQuery("select count(m) from Member m", Long.class)
                .getSingleResult();
    }

    public Member find(Long id) {
        return em.find(Member.class, id);
    }
}
```

순수 JPA 기반 리포지토리 - 팀

```
package study.datajpa.repository;

import org.springframework.stereotype.Repository;
import study.datajpa.entity.Team;

import javax.persistence.EntityManager;
```

```
import javax.persistence.PersistenceContext;
import java.util.List;
import java.util.Optional;

@Repository
public class TeamJpaRepository {

    @PersistenceContext
    private EntityManager em;

    public Team save(Team team) {
        em.persist(team);
        return team;
    }

    public void delete(Team team) {
        em.remove(team);
    }

    public List<Team> findAll() {
        return em.createQuery("select t from Team t", Team.class)
                .getResultList();
    }

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

    public long count() {
        return em.createQuery("select count(t) from Team t", Long.class)
                .getSingleResult();
    }

}
```

- 회원 리포지토리와 거의 동일하다.

순수 JPA 기반 리포지토리 테스트

```java
package study.datajpa.repository;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Transactional;
import study.datajpa.entity.Member;

import java.util.List;

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

@SpringBootTest
@Transactional
public class MemberJpaRepositoryTest {

    @Autowired
    MemberJpaRepository memberJpaRepository;

    @Test
    public void testMember() {
        Member member = new Member("memberA");
        Member savedMember = memberJpaRepository.save(member);

        Member findMember = memberJpaRepository.find(savedMember.getId());

        assertThat(findMember.getId()).isEqualTo(member.getId());
        assertThat(findMember.getUsername()).isEqualTo(member.getUsername());

        assertThat(findMember).isEqualTo(member); //JPA 엔티티 동일성 보장
    }

    @Test
    public void basicCRUD() {
        Member member1 = new Member("member1");
        Member member2 = new Member("member2");
        memberJpaRepository.save(member1);
```

```
        memberJpaRepository.save(member2);

        //단건 조회 검증
        Member findMember1 =
memberJpaRepository.findById(member1.getId()).get();
        Member findMember2 =
memberJpaRepository.findById(member2.getId()).get();
        assertThat(findMember1).isEqualTo(member1);
        assertThat(findMember2).isEqualTo(member2);

        //리스트 조회 검증
        List<Member> all = memberJpaRepository.findAll();
        assertThat(all.size()).isEqualTo(2);

        //카운트 검증
        long count = memberJpaRepository.count();
        assertThat(count).isEqualTo(2);

        //삭제 검증
        memberJpaRepository.delete(member1);
        memberJpaRepository.delete(member2);

        long deletedCount = memberJpaRepository.count();
        assertThat(deletedCount).isEqualTo(0);
    }
}
```

- 기본 CRUD를 검증한다.

공통 인터페이스 설정

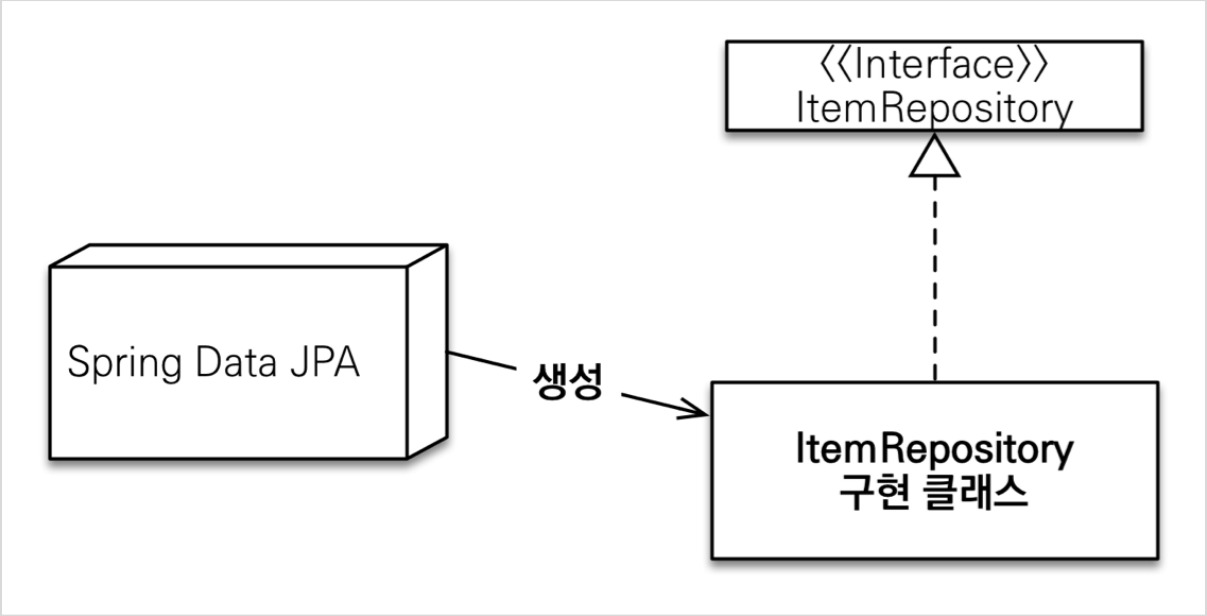
JavaConfig 설정- 스프링 부트 사용시 생략 가능

```
@Configuration
@EnableJpaRepositories(basePackages = "jpabook.jpashop.repository")
public class AppConfig {}
```

- 스프링 부트 사용시 `@SpringBootApplication` 위치를 지정(해당 패키지와 하위 패키지 인식)

- 만약 위치가 달라지면 `@EnableJpaRepositories` 필요

스프링 데이터 JPA가 구현 클래스 대신 생성



- `org.springframework.data.repository.Repository`를 구현한 클래스는 스캔 대상
  - MemberRepository 인터페이스가 동작한 이유
  - 실제 출력해보기(Proxy)
  - memberRepository.getClass() ➡ class com.sun.proxy.$ProxyXXX
- `@Repository` 애노테이션 생략 가능
  - 컴포넌트 스캔을 스프링 데이터 JPA가 자동으로 처리
  - JPA 예외를 스프링 예외로 변환하는 과정도 자동으로 처리

공통 인터페이스 적용

순수 JPA로 구현한 `MemberJpaRepository` 대신에 스프링 데이터 JPA가 제공하는 공통 인터페이스 사용

스프링 데이터 JPA 기반 MemberRepository

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

MemberRepository 테스트

```
package study.datajpa.repository;
```

```
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.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Transactional;
import study.datajpa.entity.Member;

import java.util.List;

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

@SpringBootTest
@Transactional
public class MemberRepositoryTest {

    @Autowired
    MemberRepository memberRepository;

    @Test
    public void testMember() {
        Member member = new Member("memberA");
        Member savedMember = memberRepository.save(member);

        Member findMember =
memberRepository.findById(savedMember.getId()).get();

        Assertions.assertThat(findMember.getId()).isEqualTo(member.getId());

Assertions.assertThat(findMember.getUsername()).isEqualTo(member.getUsername())
;

        Assertions.assertThat(findMember).isEqualTo(member); //JPA 엔티티 동일성
보장
    }

    @Test
    public void basicCRUD() {
```

```
        Member member1 = new Member("member1");
        Member member2 = new Member("member2");
        memberRepository.save(member1);
        memberRepository.save(member2);

        //단건 조회 검증
        Member findMember1 = memberRepository.findById(member1.getId()).get();
        Member findMember2 = memberRepository.findById(member2.getId()).get();
        assertThat(findMember1).isEqualTo(member1);
        assertThat(findMember2).isEqualTo(member2);

        //리스트 조회 검증
        List<Member> all = memberRepository.findAll();
        assertThat(all.size()).isEqualTo(2);

        //카운트 검증
        long count = memberRepository.count();
        assertThat(count).isEqualTo(2);

        //삭제 검증
        memberRepository.delete(member1);
        memberRepository.delete(member2);

        long deletedCount = memberRepository.count();
        assertThat(deletedCount).isEqualTo(0);
    }

}
```

기존 순수 JPA 기반 테스트에서 사용했던 코드를 그대로 스프링 데이터 JPA 리포지토리 기반 테스트로 변경해도 동일한 방식으로 동작

TeamRepository 생성

```
package study.datajpa.repository;

import org.springframework.data.jpa.repository.JpaRepository;
```

```
import study.datajpa.entity.Team;


public interface TeamRepository extends JpaRepository<Team, Long> {
}
```

- TeamRepository는 테스트 생략
- Generic
  - T: 엔티티 타입
  - ID: 식별자 타입(PK)

공통 인터페이스 분석

- JpaRepository 인터페이스: 공통 CRUD 제공
- 제네릭은 <엔티티 타입, 식별자 타입> 설정

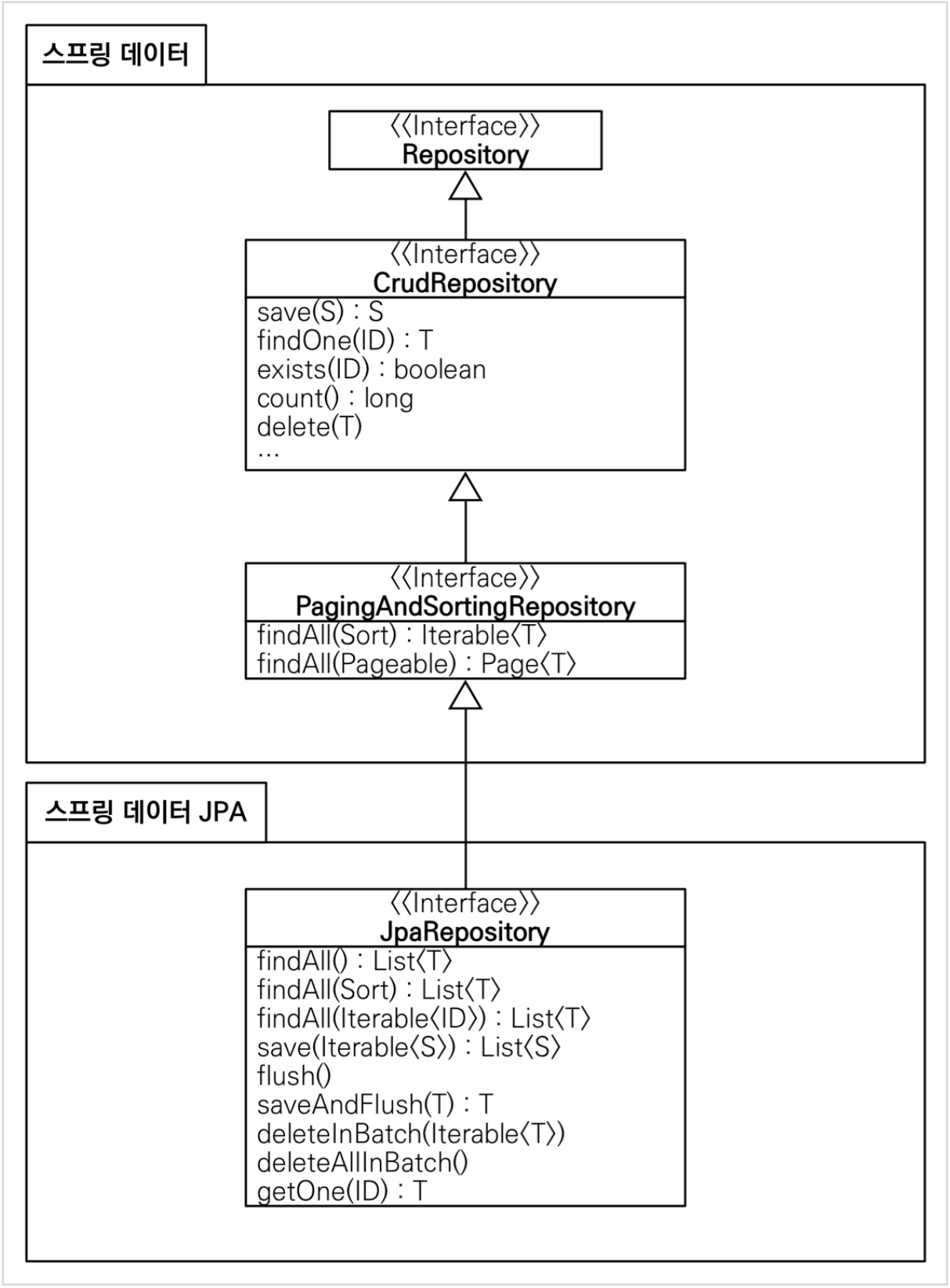
`JpaRepository` 공통 기능 인터페이스

```
public interface JpaRepository<T, ID extends Serializable>
                                    extends PagingAndSortingRepository<T, ID>
{
    ...
}
```

`JpaRepository` 를 사용하는 인터페이스

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

공통 인터페이스 구성



주의

- `T findOne(ID)` → `Optional<T> findById(ID)` 변경

제네릭 타입

- `T` : 엔티티

- `ID` : 엔티티의 식별자 타입
- `S` : 엔티티와 그 자식 타입

주요 메서드

- `save(S)` : 새로운 엔티티는 저장하고 이미 있는 엔티티는 병합한다.
- `delete(T)` : 엔티티 하나를 삭제한다. 내부에서 `EntityManager.remove()` 호출
- `findById(ID)` : 엔티티 하나를 조회한다. 내부에서 `EntityManager.find()` 호출
- `getOne(ID)` : 엔티티를 프록시로 조회한다. 내부에서 `EntityManager.getReference()` 호출
- `findAll(…)` : 모든 엔티티를 조회한다. 정렬(`Sort`)이나 페이징(`Pageable`) 조건을 파라미터로 제공할 수 있다.

> 참고: `JpaRepository` 는 대부분의 공통 메서드를 제공한다.

쿼리 메소드 기능

- 메소드 이름으로 쿼리 생성
- NamedQuery
- @Query - 리파지토리 메소드에 쿼리 정의
- 파라미터 바인딩
- 반환 타입
- 페이징과 정렬
- 벌크성 수정 쿼리
- @EntityGraph

스프링 데이터 JPA가 제공하는 마법 같은 기능

쿼리 메소드 기능 3가지

- 메소드 이름으로 쿼리 생성
- 메소드 이름으로 JPA NamedQuery 호출
- `@Query` 어노테이션을 사용해서 리파지토리 인터페이스에 쿼리 직접 정의

메소드 이름으로 쿼리 생성

메소드 이름을 분석해서 JPQL 쿼리 실행

이름과 나이를 기준으로 회원을 조회하려면?

순수 JPA 리포지토리

```
public List<Member> findByUsernameAndAgeGreaterThan(String username, int age) {
    return em.createQuery("select m from Member m where m.username = :username
and m.age > :age")
             .setParameter("username", username)
             .setParameter("age", age)
             .getResultList();
}
```

순수 JPA 테스트 코드

```
@Test
public void findByUsernameAndAgeGreaterThan() {
    Member m1 = new Member("AAA", 10);
    Member m2 = new Member("AAA", 20);
    memberJpaRepository.save(m1);
    memberJpaRepository.save(m2);

    List<Member> result =
memberJpaRepository.findByUsernameAndAgeGreaterThan("AAA", 15);
    assertThat(result.get(0).getUsername()).isEqualTo("AAA");
    assertThat(result.get(0).getAge()).isEqualTo(20);
    assertThat(result.size()).isEqualTo(1);
}
```

스프링 데이터 JPA

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

```
    List<Member> findByUsernameAndAgeGreaterThan(String username, int age);
}
```

- 스프링 데이터 JPA는 메소드 이름을 분석해서 JPQL을 생성하고 실행

쿼리 메소드 필터 조건

스프링 데이터 JPA 공식 문서 참고: (https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation)

스프링 데이터 JPA가 제공하는 쿼리 메소드 기능

- 조회: find…By ,read…By ,query…By get…By,
  - https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.query-methods.query-creation
  - 예:) findHelloBy 처럼 ...에 식별하기 위한 내용(설명)이 들어가도 된다.
- COUNT: count…By 반환타입 `long`
- EXISTS: exists…By 반환타입 `boolean`
- 삭제: delete…By, remove…By 반환타입 `long`
- DISTINCT: findDistinct, findMemberDistinctBy
- LIMIT: findFirst3, findFirst, findTop, findTop3
  - https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.limit-query-result

> 참고: 이 기능은 엔티티의 필드명이 변경되면 인터페이스에 정의한 메서드 이름도 꼭 함께 변경해야 한다.
> 그렇지 않으면 애플리케이션을 시작하는 시점에 오류가 발생한다.
> 이렇게 애플리케이션 로딩 시점에 오류를 인지할 수 있는 것이 스프링 데이터 JPA의 매우 큰 장점이다.

JPA NamedQuery

JPA의 NamedQuery를 호출할 수 있음

* `@NamedQuery` 어노테이션으로 Named 쿼리 정의*

```
@Entity
@NamedQuery(
```

```
        name="Member.findByUsername",
        query="select m from Member m where m.username = :username")
public class Member {
    ...
}
```

JPA를 직접 사용해서 Named 쿼리 호출

```
public class MemberRepository {

    public List<Member> findByUsername(String username) {
        ...
        List<Member> resultList =
            em.createNamedQuery("Member.findByUsername", Member.class)
                .setParameter("username", username)
                .getResultList();
    }
}
```

스프링 데이터 JPA로 NamedQuery 사용

```
@Query(name = "Member.findByUsername")
List<Member> findByUsername(@Param("username") String username);
```

`@Query`를 생략하고 메서드 이름만으로 Named 쿼리를 호출할 수 있다.

스프링 데이터 JPA로 Named 쿼리 호출

```
public interface MemberRepository
         extends JpaRepository<Member, Long>  { //** 여기 선언한 Member 도메인 클래스

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

- 스프링 데이터 JPA는 선언한 "도메인 클래스 + .(점) + 메서드 이름"으로 Named 쿼리를 찾아서 실행
- 만약 실행할 Named 쿼리가 없으면 메서드 이름으로 쿼리 생성 전략을 사용한다.
- 필요하면 전략을 변경할 수 있지만 권장하지 않는다.
  - 참고: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.query-methods.query-lookup-strategies

> 참고: 스프링 데이터 JPA를 사용하면 실무에서 Named Query를 직접 등록해서 사용하는 일은 드물다. 대신 `@Query` 를 사용해서 리파지토리 메소드에 쿼리를 직접 정의한다.

@Query, 리포지토리 메소드에 쿼리 정의하기

메서드에 JPQL 쿼리 작성

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

@Query("select m from Member m where m.username= :username and m.age = :age")
List<Member> findUser(@Param("username") String username, @Param("age") int
age);

}
```

- `@org.springframework.data.jpa.repository.Query` 어노테이션을 사용
- 실행할 메서드에 정적 쿼리를 직접 작성하므로 이름 없는 Named 쿼리라 할 수 있음
- JPA Named 쿼리처럼 애플리케이션 실행 시점에 문법 오류를 발견할 수 있음(매우 큰 장점!)

> 참고: 실무에서는 메소드 이름으로 쿼리 생성 기능은 파라미터가 증가하면 메서드 이름이 매우 지저분해진다. 따라서 `@Query` 기능을 자주 사용하게 된다.

@Query, 값, DTO 조회하기

단순히 값 하나를 조회

```
@Query("select m.username from Member m")
List<String> findUsernameList();
```

JPA 값 타입(`@Embedded`)도 이 방식으로 조회할 수 있다.

DTO로 직접 조회

```
@Query("select new study.datajpa.dto.MemberDto(m.id, m.username, t.name) " +
        "from Member m join m.team t")
List<MemberDto> findMemberDto();
```

주의! DTO로 직접 조회 하려면 JPA의 `new` 명령어를 사용해야 한다. 그리고 다음과 같이 생성자가 맞는 DTO가 필요하다. (JPA와 사용방식이 동일하다.)

```
package study.datajpa.repository;

import lombok.Data;

@Data
public class MemberDto {
    private Long id;
    private String username;
    private String teamName;

    public MemberDto(Long id, String username, String teamName) {
        this.id = id;
        this.username = username;
        this.teamName = teamName;
    }
}
```

파라미터 바인딩

- 위치 기반
- 이름 기반

```
select m from Member m where m.username = ?0 //위치 기반
select m from Member m where m.username = :name //이름 기반
```

파라미터 바인딩

```
import org.springframework.data.repository.query.Param

public interface MemberRepository extends JpaRepository<Member, Long> {

    @Query("select m from Member m where m.username = :name")
    Member findMembers(@Param("name") String username);
}
```

> 참고: 코드 가독성과 유지보수를 위해 이름 기반 파라미터 바인딩을 사용하자 (위치기반은 순서 실수가 바꾸면...)

컬렉션 파라미터 바인딩

`Collection` 타입으로 in절 지원

```
@Query("select m from Member m where m.username in :names")
List<Member> findByNames(@Param("names") List<String> names);
```

반환 타입

스프링 데이터 JPA는 유연한 반환 타입 지원

```
List<Member> findByUsername(String name); //컬렉션
Member findByUsername(String name); //단건
```

```
Optional<Member> findByUsername(String name); //단건 Optional
```

스프링 데이터 JPA 공식 문서: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repository-query-return-types

조회 결과가 많거나 없으면?

- 컬렉션
  - 결과 없음: 빈 컬렉션 반환
- 단건 조회
  - 결과 없음: `null` 반환
  - 결과가 2건 이상: `javax.persistence.NonUniqueResultException` 예외 발생

> 참고: 단건으로 지정한 메서드를 호출하면 스프링 데이터 JPA는 내부에서 JPQL의 `Query.getSingleResult()` 메서드를 호출한다. 이 메서드를 호출했을 때 조회 결과가 없으면 `javax.persistence.NoResultException` 예외가 발생하는데 개발자 입장에서 다루기가 상당히 불편하다. 스프링 데이터 JPA는 단건을 조회할 때 이 예외가 발생하면 예외를 무시하고 대신에 `null`을 반환한다.

순수 JPA 페이징과 정렬

JPA에서 페이징을 어떻게 할 것인가?

다음 조건으로 페이징과 정렬을 사용하는 예제 코드를 보자.

- 검색 조건: 나이가 10살
- 정렬 조건: 이름으로 내림차순
- 페이징 조건: 첫 번째 페이지, 페이지당 보여줄 데이터는 3건

JPA 페이징 리포지토리 코드

```
public List<Member> findByPage(int age, int offset, int limit) {
    return em.createQuery("select m from Member m where m.age = :age order by m.username desc")
            .setParameter("age", age)
```

```
                .setFirstResult(offset)
                .setMaxResults(limit)
                .getResultList();
    }

    public long totalCount(int age) {
        return em.createQuery("select count(m) from Member m where m.age = :age",
    Long.class)
                .setParameter("age", age)
                .getSingleResult();
    }
```

JPA 페이징 테스트 코드

```
    @Test
    public void paging() throws Exception {
        //given
        memberJpaRepository.save(new Member("member1", 10));
        memberJpaRepository.save(new Member("member2", 10));
        memberJpaRepository.save(new Member("member3", 10));
        memberJpaRepository.save(new Member("member4", 10));
        memberJpaRepository.save(new Member("member5", 10));

        int age = 10;
        int offset = 0;
        int limit = 3;

        //when
        List<Member> members = memberJpaRepository.findByPage(age, offset, limit);
        long totalCount = memberJpaRepository.totalCount(age);

        //페이지 계산 공식 적용...
        // totalPage = totalCount / size ...
        // 마지막 페이지 ...
        // 최초 페이지 ..

        //then
        assertThat(members.size()).isEqualTo(3);
```

```
    assertThat(totalCount).isEqualTo(5);
}
```

스프링 데이터 JPA 페이징과 정렬

페이징과 정렬 파라미터

- `org.springframework.data.domain.Sort` : 정렬 기능
- `org.springframework.data.domain.Pageable` : 페이징 기능 (내부에 `Sort` 포함)

특별한 반환 타입

- `org.springframework.data.domain.Page` : 추가 count 쿼리 결과를 포함하는 페이징
- `org.springframework.data.domain.Slice` : 추가 count 쿼리 없이 다음 페이지만 확인 가능 (내부적으로 limit + 1조회)
- `List` (자바 컬렉션): 추가 count 쿼리 없이 결과만 반환

페이징과 정렬 사용 예제

```
Page<Member> findByUsername(String name, Pageable pageable);  //count 쿼리 사용
Slice<Member> findByUsername(String name, Pageable pageable); //count 쿼리 사용
안함
List<Member> findByUsername(String name, Pageable pageable);  //count 쿼리 사용
안함
List<Member> findByUsername(String name, Sort sort);
```

다음 조건으로 페이징과 정렬을 사용하는 예제 코드를 보자.

- 검색 조건: 나이가 10살
- 정렬 조건: 이름으로 내림차순
- 페이징 조건: 첫 번째 페이지, 페이지당 보여줄 데이터는 3건

Page 사용 예제 정의 코드

```
public interface MemberRepository extends Repository<Member, Long> {
```

```
    Page<Member> findByAge(int age, Pageable pageable);
}
```

Page 사용 예제 실행 코드

```
//페이징 조건과 정렬 조건 설정
@Test
public void page() throws Exception {
    //given
    memberRepository.save(new Member("member1", 10));
    memberRepository.save(new Member("member2", 10));
    memberRepository.save(new Member("member3", 10));
    memberRepository.save(new Member("member4", 10));
    memberRepository.save(new Member("member5", 10));

    //when
    PageRequest pageRequest = PageRequest.of(0, 3, Sort.by(Sort.Direction.DESC,
"username"));
    Page<Member> page = memberRepository.findByAge(10, pageRequest);

    //then
    List<Member> content = page.getContent(); //조회된 데이터
    assertThat(content.size()).isEqualTo(3); //조회된 데이터 수
    assertThat(page.getTotalElements()).isEqualTo(5); //전체 데이터 수
    assertThat(page.getNumber()).isEqualTo(0); //페이지 번호
    assertThat(page.getTotalPages()).isEqualTo(2); //전체 페이지 번호
    assertThat(page.isFirst()).isTrue(); //첫번째 항목인가?
    assertThat(page.hasNext()).isTrue(); //다음 페이지가 있는가?
}
```

- 두 번째 파라미터로 받은 `Pagable`은 인터페이스다. 따라서 실제 사용할 때는 해당 인터페이스를 구현한 `org.springframework.data.domain.PageRequest` 객체를 사용한다.
- `PageRequest` 생성자의 첫 번째 파라미터에는 현재 페이지를, 두 번째 파라미터에는 조회할 데이터 수를 입력한다. 여기에 추가로 정렬 정보도 파라미터로 사용할 수 있다. 참고로 페이지는 0부터 시작한다.

> 주의: Page는 1부터 시작이 아니라 0부터 시작이다.

Page 인터페이스

```
public interface Page<T> extends Slice<T> {
    int getTotalPages();     //전체 페이지 수
    long getTotalElements(); //전체 데이터 수
    <U> Page<U> map(Function<? super T, ? extends U> converter); //변환기
}
```

Slice 인터페이스

```
public interface Slice<T> extends Streamable<T> {
  int getNumber();             //현재 페이지
  int getSize();               //페이지 크기
  int getNumberOfElements();   //현재 페이지에 나올 데이터 수
  List<T> getContent();        //조회된 데이터
  boolean hasContent();        //조회된 데이터 존재 여부
  Sort getSort();              //정렬 정보
  boolean isFirst();           //현재 페이지가 첫 페이지 인지 여부
  boolean isLast();            //현재 페이지가 마지막 페이지 인지 여부
  boolean hasNext();           //다음 페이지 여부
  boolean hasPrevious();       //이전 페이지 여부
  Pageable getPageable();      //페이지 요청 정보
  Pageable nextPageable();     //다음 페이지 객체
  Pageable previousPageable();//이전 페이지 객체
  <U> Slice<U> map(Function<? super T, ? extends U> converter); //변환기
}
```

참고: count 쿼리를 다음과 같이 분리할 수 있음

```
@Query(value = “select m from Member m”,
       countQuery = “select count(m.username) from Member m”)
Page<Member> findMemberAllCountBy(Pageable pageable);
```

Top, First 사용 참고

https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.limit-query-result

```
List<Member> findTop3By();
```

페이지를 유지하면서 엔티티를 DTO로 변환하기

```
Page<Member> page = memberRepository.findByAge(10, pageRequest);
Page<MemberDto> dtoPage = page.map(m -> new MemberDto());
```

실습

- Page
- Slice (count X) 추가로 limit + 1을 조회한다. 그래서 다음 페이지 여부 확인(최근 모바일 리스트 생각해보면 됨)
- List (count X)
- 카운트 쿼리 분리(이건 복잡한 sql에서 사용, 데이터는 left join, 카운트는 left join 안해도 됨)
  - 실무에서 매우 중요!!!

> 참고: 전체 count 쿼리는 매우 무겁다.

벌크성 수정 쿼리

JPA를 사용한 벌크성 수정 쿼리

```
public int bulkAgePlus(int age) {
    int resultCount = em.createQuery(
            "update Member m set m.age = m.age + 1" +
                    "where m.age >= :age")
            .setParameter("age", age)
            .executeUpdate();
    return resultCount;
}
```

JPA를 사용한 벌크성 수정 쿼리 테스트

```
@Test
public void bulkUpdate() throws Exception {
    //given
    memberJpaRepository.save(new Member("member1", 10));
    memberJpaRepository.save(new Member("member2", 19));
    memberJpaRepository.save(new Member("member3", 20));
    memberJpaRepository.save(new Member("member4", 21));
    memberJpaRepository.save(new Member("member5", 40));

    //when
    int resultCount = memberJpaRepository.bulkAgePlus(20);

    //then
    assertThat(resultCount).isEqualTo(3);
}
```

스프링 데이터 JPA를 사용한 벌크성 수정 쿼리

```
@Modifying
@Query("update Member m set m.age = m.age + 1 where m.age >= :age")
int bulkAgePlus(@Param("age") int age);
```

스프링 데이터 JPA를 사용한 벌크성 수정 쿼리 테스트

```
@Test
public void bulkUpdate() throws Exception {
    //given
    memberRepository.save(new Member("member1", 10));
    memberRepository.save(new Member("member2", 19));
    memberRepository.save(new Member("member3", 20));
    memberRepository.save(new Member("member4", 21));
```

```
        memberRepository.save(new Member("member5", 40));

        //when
        int resultCount = memberRepository.bulkAgePlus(20);

        //then
        assertThat(resultCount).isEqualTo(3);
    }
```

- 벌크성 수정, 삭제 쿼리는 `@Modifying` 어노테이션을 사용
  - 사용하지 않으면 다음 예외 발생
  - `org.hibernate.hql.internal.QueryExecutionRequestException: Not supported for DML operations`
- 벌크성 쿼리를 실행하고 나서 영속성 컨텍스트 초기화: `@Modifying(clearAutomatically = true)` (이 옵션의 기본값은 `false`)
  - 이 옵션 없이 회원을 `findById` 로 다시 조회하면 영속성 컨텍스트에 과거 값이 남아서 문제가 될 수 있다. 만약 다시 조회해야 하면 꼭 영속성 컨텍스트를 초기화 하자.

> 참고: 벌크 연산은 영속성 컨텍스트를 무시하고 실행하기 때문에, 영속성 컨텍스트에 있는 엔티티의 상태와 DB에 엔티티 상태가 달라질 수 있다.
> 권장하는 방안
> 1. 영속성 컨텍스트에 엔티티가 없는 상태에서 벌크 연산을 먼저 실행한다.
> 2. 부득이하게 영속성 컨텍스트에 엔티티가 있으면 벌크 연산 직후 영속성 컨텍스트를 초기화 한다.

@EntityGraph

연관된 엔티티들을 SQL 한번에 조회하는 방법

member → team은 지연로딩 관계이다. 따라서 다음과 같이 team의 데이터를 조회할 때 마다 쿼리가 실행된다. (N+1 문제 발생)

```
    @Test
    public void findMemberLazy() throws Exception {
        //given
        //member1 -> teamA
```

```
    //member2 -> teamB
    Team teamA = new Team("teamA");
    Team teamB = new Team("teamB");
    teamRepository.save(teamA);
    teamRepository.save(teamB);
    memberRepository.save(new Member("member1", 10, teamA));
    memberRepository.save(new Member("member2", 20, teamB));

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

    //when
    List<Member> members = memberRepository.findAll();

    //then
    for (Member member : members) {
        member.getTeam().getName();
    }
}
```

참고: 다음과 같이 지연 로딩 여부를 확인할 수 있다.

```
//Hibernate 기능으로 확인
Hibernate.isInitialized(member.getTeam())

//JPA 표준 방법으로 확인
PersistenceUnitUtil util =
em.getEntityManagerFactory().getPersistenceUnitUtil();
util.isLoaded(member.getTeam());
```

연관된 엔티티를 한번에 조회하려면 페치 조인이 필요하다.

JPQL 페치 조인

```
@Query("select m from Member m left join fetch m.team")
```

```
List<Member> findMemberFetchJoin();
```

스프링 데이터 JPA는 JPA가 제공하는 엔티티 그래프 기능을 편리하게 사용하게 도와준다. 이 기능을 사용하면 JPQL 없이 페치 조인을 사용할 수 있다. (JPQL + 엔티티 그래프도 가능)

EntityGraph

```
//공통 메서드 오버라이드
@Override
@EntityGraph(attributePaths = {"team"})
List<Member> findAll();

//JPQL + 엔티티 그래프
@EntityGraph(attributePaths = {"team"})
@Query("select m from Member m")
List<Member> findMemberEntityGraph();

//메서드 이름으로 쿼리에서 특히 편리하다.
@EntityGraph(attributePaths = {"team"})
List<Member> findByUsername(String username)
```

EntityGraph 정리

- 사실상 페치 조인(FETCH JOIN)의 간편 버전
- LEFT OUTER JOIN 사용

NamedEntityGraph 사용 방법

```
@NamedEntityGraph(name = "Member.all", attributeNodes =
@NamedAttributeNode("team"))
@Entity
public class Member {}
```

```
@EntityGraph("Member.all")
@Query("select m from Member m")
List<Member> findMemberEntityGraph();
```

JPA Hint & Lock

JPA Hint

JPA 쿼리 힌트(SQL 힌트가 아니라 JPA 구현체에게 제공하는 힌트)

쿼리 힌트 사용

```
@QueryHints(value = @QueryHint(name = "org.hibernate.readOnly", value =
"true"))
Member findReadOnlyByUsername(String username);
```

쿼리 힌트 사용 확인

```
@Test
public void queryHint() throws Exception {
    //given
    memberRepository.save(new Member("member1", 10));
    em.flush();
    em.clear();

    //when
    Member member = memberRepository.findReadOnlyByUsername("member1");
    member.setUsername("member2");

    em.flush(); //Update Query 실행X
}
```

쿼리 힌트 Page 추가 예제

```
@QueryHints(value = { @QueryHint(name = "org.hibernate.readOnly",
                                 value = "true")},
            forCounting = true)
```

```
Page<Member> findByUsername(String name, Pagable pageable);
```

- `org.springframework.data.jpa.repository.QueryHints` 어노테이션을 사용
- `forCounting` : 반환 타입으로 `Page` 인터페이스를 적용하면 추가로 호출하는 페이징을 위한 count 쿼리도 쿼리 힌트 적용(기본값 `true`)

Lock

```
@Lock(LockModeType.PESSIMISTIC_WRITE)
List<Member> findByUsername(String name);
```

- `org.springframework.data.jpa.repository.Lock` 어노테이션을 사용
- JPA가 제공하는 락은 JPA 책 16.1 트랜잭션과 락 절을 참고

확장 기능

사용자 정의 리포지토리 구현

- 스프링 데이터 JPA 리포지토리는 인터페이스만 정의하고 구현체는 스프링이 자동 생성
- 스프링 데이터 JPA가 제공하는 인터페이스를 직접 구현하면 구현해야 하는 기능이 너무 많음
- 다양한 이유로 인터페이스의 메서드를 직접 구현하고 싶다면?
  - JPA 직접 사용(`EntityManager`)
  - 스프링 JDBC Template 사용
  - MyBatis 사용
  - 데이터베이스 커넥션 직접 사용 등등...
  - Querydsl 사용

사용자 정의 인터페이스

```
public interface MemberRepositoryCustom {
    List<Member> findMemberCustom();
}
```

사용자 정의 인터페이스 구현 클래스

```
@RequiredArgsConstructor
public class MemberRepositoryImpl implements MemberRepositoryCustom {

    private final EntityManager em;

    @Override
    public List<Member> findMemberCustom() {
        return em.createQuery("select m from Member m")
                .getResultList();
    }
}
```

사용자 정의 인터페이스 상속

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

사용자 정의 메서드 호출 코드

```
List<Member> result = memberRepository.findMemberCustom();
```

사용자 정의 구현 클래스

- 규칙: 리포지토리 인터페이스 이름 + `Impl`
- 스프링 데이터 JPA가 인식해서 스프링 빈으로 등록

Impl 대신 다른 이름으로 변경하고 싶으면?

XML 설정

```
<repositories base-package="study.datajpa.repository"
              repository-impl-postfix="Impl" />
```

JavaConfig 설정

```
@EnableJpaRepositories(basePackages = "study.datajpa.repository",
                       repositoryImplementationPostfix = "Impl")
```

> 참고: 실무에서는 주로 QueryDSL이나 SpringJdbcTemplate을 함께 사용할 때 사용자 정의 리포지토리 기능 자주 사용

> 참고: 항상 사용자 정의 리포지토리가 필요한 것은 아니다. 그냥 임의의 리포지토리를 만들어도 된다. 예를들어 MemberQueryRepository를 인터페이스가 아닌 클래스로 만들고 스프링 빈으로 등록해서 그냥 직접 사용해도 된다. 물론 이 경우 스프링 데이터 JPA와는 아무런 관계 없이 별도로 동작한다.

사용자 정의 리포지토리 구현 최신 방식

(참고: 강의 영상에는 없는 내용입니다.)

스프링 데이터 2.x 부터는 사용자 정의 구현 클래스에 리포지토리 인터페이스 이름 + `Impl` 을 적용하는 대신에
사용자 정의 인터페이스 명 + `Impl` 방식도 지원한다.
예를 들어서 위 예제의 `MemberRepositoryImpl` 대신에 `MemberRepositoryCustomImpl` 같이 구현해도 된다.

최신 사용자 정의 인터페이스 구현 클래스 예제

```
@RequiredArgsConstructor
public class MemberRepositoryCustomImpl implements MemberRepositoryCustom {

    private final EntityManager em;

    @Override
    public List<Member> findMemberCustom() {
        return em.createQuery("select m from Member m")
                .getResultList();
    }
}
```

기존 방식보다 이 방식이 사용자 정의 인터페이스 이름과 구현 클래스 이름이 비슷하므로 더 직관적이다. 추가로 여러 인터페이스를 분리해서 구현하는 것도 가능하기 때문에 새롭게 변경된 이 방식을 사용하는 것을 더 권장한다.

Auditing

- 엔티티를 생성, 변경할 때 변경한 사람과 시간을 추적하고 싶으면?
  - 등록일
  - 수정일
  - 등록자
  - 수정자

순수 JPA 사용

우선 등록일, 수정일 적용

```
package study.datajpa.entity;

@MappedSuperclass
@Getter
public class JpaBaseEntity {

    @Column(updatable = false)
    private LocalDateTime createdDate;
    private LocalDateTime updatedDate;

    @PrePersist
    public void prePersist() {
        LocalDateTime now = LocalDateTime.now();
        createdDate = now;
        updatedDate = now;
    }

    @PreUpdate
    public void preUpdate() {
        updatedDate = LocalDateTime.now();
    }
}
```

```
public class Member extends JpaBaseEntity {}
```

확인 코드

```
@Test
public void JpaEventBaseEntity() throws Exception {
    //given
    Member member = new Member("member1");
    memberRepository.save(member); //@PrePersist

    Thread.sleep(100);
    member.setUsername("member2");

    em.flush(); //@PreUpdate
    em.clear();

    //when
    Member findMember = memberRepository.findById(member.getId()).get();

    //then
    System.out.println("findMember.createdDate = " +
findMember.getCreatedDate());
    System.out.println("findMember.updatedDate = " +
findMember.getUpdatedDate());
}
```

JPA 주요 이벤트 어노테이션

- @PrePersist, @PostPersist
- @PreUpdate, @PostUpdate

스프링 데이터 JPA 사용

설정

`@EnableJpaAuditing` ➜ 스프링 부트 설정 클래스에 적용해야함

`@EntityListeners(AuditingEntityListener.class)` ➜ 엔티티에 적용

사용 어노테이션

- `@CreatedDate`
- `@LastModifiedDate`
- `@CreatedBy`
- `@LastModifiedBy`

스프링 데이터 Auditing 적용 - 등록일, 수정일

```
package study.datajpa.entity;

@EntityListeners(AuditingEntityListener.class)
@MappedSuperclass
@Getter
public class BaseEntity {

    @CreatedDate
    @Column(updatable = false)
    private LocalDateTime createdDate;

    @LastModifiedDate
    private LocalDateTime lastModifiedDate;

}
```

스프링 데이터 Auditing 적용 - 등록자, 수정자

```
package jpabook.jpashop.domain;

@EntityListeners(AuditingEntityListener.class)
@MappedSuperclass
public class BaseEntity {

    @CreatedDate
    @Column(updatable = false)
    private LocalDateTime createdDate;
    @LastModifiedDate
```

```
    private LocalDateTime lastModifiedDate;

    @CreatedBy
    @Column(updatable = false)
    private String createdBy;
    @LastModifiedBy
    private String lastModifiedBy;

}
```

등록자, 수정자를 처리해주는 `AuditorAware` 스프링 빈 등록

```
@Bean
public AuditorAware<String> auditorProvider() {
    return () -> Optional.of(UUID.randomUUID().toString());
}
```

실무에서는 세션 정보나, 스프링 시큐리티 로그인 정보에서 ID를 받음

> 참고: 실무에서 대부분의 엔티티는 등록시간, 수정시간이 필요하지만, 등록자, 수정자는 없을 수도 있다.
> 그래서 다음과 같이 Base 타입을 분리하고, 원하는 타입을 선택해서 상속한다.

```
public class BaseTimeEntity {
    @CreatedDate
    @Column(updatable = false)
    private LocalDateTime createdDate;
    @LastModifiedDate
    private LocalDateTime lastModifiedDate;
}

public class BaseEntity extends BaseTimeEntity {
    @CreatedBy
    @Column(updatable = false)
    private String createdBy;
    @LastModifiedBy
    private String lastModifiedBy;
```

```
}
```

> 참고: 저장시점에 등록일, 등록자는 물론이고, 수정일, 수정자도 같은 데이터가 저장된다. 데이터가 중복 저장되는 것 같지만, 이렇게 해두면 변경 컬럼만 확인해도 마지막에 업데이트한 유저를 확인 할 수 있으므로 유지보수 관점에서 편리하다. 이렇게 하지 않으면 변경 컬럼이 `null` 일때 등록 컬럼을 또 찾아야 한다.
> 참고로 저장시점에 저장데이터만 입력하고 싶으면 `@EnableJpaAuditing(modifyOnCreate = false)` 옵션을 사용하면 된다.

전체 적용

`@EntityListeners(AuditingEntityListener.class)`를 생략하고 스프링 데이터 JPA 가 제공하는 이벤트를 엔티티 전체에 적용하려면 orm.xml에 다음과 같이 등록하면 된다.

`META-INF/orm.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/
orm http://xmlns.jcp.org/xml/ns/persistence/orm_2_2.xsd"
                 version="2.2">

    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <entity-listeners>
                <entity-listener
class="org.springframework.data.jpa.domain.support.AuditingEntityListener"/>
            </entity-listeners>
        </persistence-unit-defaults>
    </persistence-unit-metadata>

</entity-mappings>
```

Web 확장 - 도메인 클래스 컨버터

HTTP 파라미터로 넘어온 엔티티의 아이디로 엔티티 객체를 찾아서 바인딩

도메인 클래스 컨버터 사용 전

```
@RestController
@RequiredArgsConstructor
public class MemberController {

    private final MemberRepository memberRepository;

    @GetMapping("/members/{id}")
    public String findMember(@PathVariable("id") Long id) {
        Member member = memberRepository.findById(id).get();
        return member.getUsername();
    }
}
```

도메인 클래스 컨버터 사용 후

```
@RestController
@RequiredArgsConstructor
public class MemberController {

    private final MemberRepository memberRepository;

    @GetMapping("/members/{id}")
    public String findMember(@PathVariable("id") Member member) {
        return member.getUsername();
    }

}
```

- HTTP 요청은 회원 `id` 를 받지만 도메인 클래스 컨버터가 중간에 동작해서 회원 엔티티 객체를 반환
- 도메인 클래스 컨버터도 리파지토리를 사용해서 엔티티를 찾음

> **주의:** 도메인 클래스 컨버터로 엔티티를 파라미터로 받으면, 이 엔티티는 단순 조회용으로만 사용해야 한다.
> (트랜잭션이 없는 범위에서 엔티티를 조회했으므로, 엔티티를 변경해도 DB에 반영되지 않는다.)

Web 확장 - 페이징과 정렬

스프링 데이터가 제공하는 페이징과 정렬 기능을 스프링 MVC에서 편리하게 사용할 수 있다.

페이징과 정렬 예제

```
@GetMapping("/members")
public Page<Member> list(Pageable pageable) {
    Page<Member> page = memberRepository.findAll(pageable);
    return page;
}
```

- 파라미터로 `Pageable` 을 받을 수 있다.
- `Pageable` 은 인터페이스, 실제는 `org.springframework.data.domain.PageRequest` 객체 생성

요청 파라미터

- 예) `/members?page=0&size=3&sort=id,desc&sort=username,desc`
- page: 현재 페이지, **0부터 시작한다.**
- size: 한 페이지에 노출할 데이터 건수
- sort: 정렬 조건을 정의한다. 예) 정렬 속성,정렬 속성...(ASC | DESC), 정렬 방향을 변경하고 싶으면 `sort` 파라미터 추가 (`asc` 생략 가능)

기본값

- 글로벌 설정: 스프링 부트

```
spring.data.web.pageable.default-page-size=20 /# 기본 페이지 사이즈/
spring.data.web.pageable.max-page-size=2000 /# 최대 페이지 사이즈/
```

- 개별 설정

  `@PageableDefault` 어노테이션을 사용

```
@RequestMapping(value = "/members_page", method = RequestMethod.GET)
public String list(@PageableDefault(size = 12, sort = “username”,
                    direction = Sort.Direction.DESC) Pageable pageable) {
    ...
}
```

접두사

- 페이징 정보가 둘 이상이면 접두사로 구분
- `@Qualifier`에 접두사명 추가 "{접두사명}_xxx”
- 예제: `/members?member_page=0&order_page=1`

```
public String list(
    @Qualifier("member") Pageable memberPageable,
    @Qualifier("order") Pageable orderPageable, ...
```

Page 내용을 DTO로 변환하기

- 엔티티를 API로 노출하면 다양한 문제가 발생한다. 그래서 엔티티를 꼭 DTO로 변환해서 반환해야 한다.
- Page는 `map()`을 지원해서 내부 데이터를 다른 것으로 변경할 수 있다.

Member DTO

```
@Data
public class MemberDto {
    private Long id;
    private String username;

    public MemberDto(Member m) {
        this.id = m.getId();
        this.username = m.getUsername();
```

```
    }
}
```

`Page.map()` 사용

```
@GetMapping("/members")
public Page<MemberDto> list(Pageable pageable) {
    Page<Member> page = memberRepository.findAll(pageable);
    Page<MemberDto> pageDto = page.map(MemberDto::new);
    return pageDto;
}
```

`Page.map()` 코드 최적화

```
@GetMapping("/members")
public Page<MemberDto> list(Pageable pageable) {
    return memberRepository.findAll(pageable).map(MemberDto::new);
}
```

Page를 1부터 시작하기

- 스프링 데이터는 Page를 0부터 시작한다.
- 만약 1부터 시작하려면?

- 1. Pageable, Page를 파리미터와 응답 값으로 사용히지 않고, 직접 클래스를 만들어서 처리한다. 그리고 직접 PageRequest(Pageable 구현체)를 생성해서 리포지토리에 넘긴다. 물론 응답값도 Page 대신에 직접 만들어서 제공해야 한다.
- 2. `spring.data.web.pageable.one-indexed-parameters`를 `true`로 설정한다. 그런데 이 방법은 web에서 `page` 파라미터를 `-1` 처리 할 뿐이다. 따라서 응답값인 `Page`에 모두 0 페이지 인덱스를 사용하는 한계가 있다.

`one-indexed-parameters` Page 1요청 (`http://localhost:8080/members?page=1`)

```
{
```

```
    "content": [
      ...
    ],
    "pageable": {
        "offset": 0,
        "pageSize": 10,
        "pageNumber": 0 //0 인덱스
    },
    "number": 0, //0 인덱스
    "empty": false
}
```

스프링 데이터 JPA 분석

스프링 데이터 JPA 구현체 분석

- 스프링 데이터 JPA가 제공하는 공통 인터페이스의 구현체
- `org.springframework.data.jpa.repository.support.SimpleJpaRepository`

리스트 12.31 SimpleJpaRepository

```
@Repository
@Transactional(readOnly = true)
public class SimpleJpaRepository<T, ID> ...{

    @Transactional
    public <S extends T> S save(S entity) {

        if (entityInformation.isNew(entity)) {
            em.persist(entity);
            return entity;
        } else {
            return em.merge(entity);
```

```
        }
    }
    ...
}
```

- `@Repository` 적용: JPA 예외를 스프링이 추상화한 예외로 변환
- `@Transactional` 트랜잭션 적용
  - JPA의 모든 변경은 트랜잭션 안에서 동작
  - 스프링 데이터 JPA는 변경(등록, 수정, 삭제) 메서드를 트랜잭션 처리
  - 서비스 계층에서 트랜잭션을 시작하지 않으면 리파지토리에서 트랜잭션 시작
  - 서비스 계층에서 트랜잭션을 시작하면 리파지토리는 해당 트랜잭션을 전파 받아서 사용
  - 그래서 스프링 데이터 JPA를 사용할 때 트랜잭션이 없어도 데이터 등록, 변경이 가능했음(사실은 트랜잭션이 리포지토리 계층에 걸려있는 것임)

- `@Transactional(readOnly = true)`
  - 데이터를 단순히 조회만 하고 변경하지 않는 트랜잭션에서 `readOnly = true` 옵션을 사용하면 플러시를 생략해서 약간의 성능 향상을 얻을 수 있음
  - 자세한 내용은 JPA 책 15.4.2 읽기 전용 쿼리의 성능 최적화 참고

매우 중요!!!

- *`save()` 메서드*
  - 새로운 엔티티면 저장(`persist`)
  - 새로운 엔티티가 아니면 병합(`merge`)

새로운 엔티티를 구별하는 방법

매우 중요!!!

- *`save()` 메서드*
  - 새로운 엔티티면 저장(`persist`)
  - 새로운 엔티티가 아니면 병합(`merge`)

- 새로운 엔티티를 판단하는 기본 전략
  - 식별자가 객체일 때 `null` 로 판단
  - 식별자가 자바 기본 타입일 때 `0` 으로 판단
  - `Persistable` 인터페이스를 구현해서 판단 로직 변경 가능

* `Persistable` *

```
package org.springframework.data.domain;

public interface Persistable<ID> {
    ID getId();
    boolean isNew();
}
```

> 참고: JPA 식별자 생성 전략이 `@GenerateValue` 면 `save()` 호출 시점에 식별자가 없으므로 새로운 엔티티로 인식해서 정상 동작한다. 그런데 JPA 식별자 생성 전략이 `@Id` 만 사용해서 직접 할당이면 이미 식별자 값이 있는 상태로 `save()` 를 호출한다. 따라서 이 경우 `merge()` 가 호출된다. `merge()` 는 우선 DB를 호출해서 값을 확인하고, DB에 값이 없으면 새로운 엔티티로 인지하므로 매우 비효율 적이다. 따라서 `Persistable` 를 사용해서 새로운 엔티티 확인 여부를 직접 구현하게는 효과적이다.
> 참고로 등록시간(`@CreatedDate`)을 조합해서 사용하면 이 필드로 새로운 엔티티 여부를 편리하게 확인할 수 있다. (@CreatedDate에 값이 없으면 새로운 엔티티로 판단)

Persistable 구현

```
package study.datajpa.entity;

import lombok.AccessLevel;
import lombok.NoArgsConstructor;
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.domain.Persistable;
import org.springframework.data.jpa.domain.support.AuditingEntityListener;

import javax.persistence.Entity;
import javax.persistence.EntityListeners;
import javax.persistence.Id;
import java.time.LocalDateTime;

@Entity
```

```java
@EntityListeners(AuditingEntityListener.class)
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class Item implements Persistable<String> {

    @Id
    private String id;

    @CreatedDate
    private LocalDateTime createdDate;

    public Item(String id) {
        this.id = id;
    }

    @Override
    public String getId() {
        return id;
    }

    @Override
    public boolean isNew() {
        return createdDate == null;
    }
}
```

나머지 기능들

Specifications (명세)

책 도메인 주도 설계(Domain Driven Design)는 SPECIFICATION(명세)라는 개념을 소개
스프링 데이터 JPA는 JPA Criteria를 활용해서 이 개념을 사용할 수 있도록 지원

술어(predicate)

- 참 또는 거짓으로 평가
- AND OR 같은 연산자로 조합해서 다양한 검색조건을 쉽게 생성(컴포지트 패턴)
- 예) 검색 조건 하나하나
- 스프링 데이터 JPA는 `org.springframework.data.jpa.domain.Specification` 클래스로 정의

명세 기능 사용 방법

* `JpaSpecificationExecutor` 인터페이스 상속*

```
public interface MemberRepository extends JpaRepository<Member, Long>,

JpaSpecificationExecutor<Member> {

}
```

* `JpaSpecificationExecutor` 인터페이스*

```
public interface JpaSpecificationExecutor<T> {

    Optional<T> findOne(@Nullable Specification<T> spec);
    List<T> findAll(Specification<T> spec);
    Page<T> findAll(Specification<T> spec, Pageable pageable);
    List<T> findAll(Specification<T> spec, Sort sort);
    long count(Specification<T> spec);
}
```

`Specification`을 파라미터로 받아서 검색 조건으로 사용

명세 사용 코드

```
@Test
public void specBasic() throws Exception {
    //given
    Team teamA = new Team("teamA");
    em.persist(teamA);

    Member m1 = new Member("m1", 0, teamA);
```

```
    Member m2 = new Member("m2", 0, teamA);
    em.persist(m1);
    em.persist(m2);
    em.flush();
    em.clear();

    //when
    Specification<Member> spec =
MemberSpec.username("m1").and(MemberSpec.teamName("teamA"));
    List<Member> result = memberRepository.findAll(spec);

    //then
    Assertions.assertThat(result.size()).isEqualTo(1);
}
```

- `Specification`을 구현하면 명세들을 조립할 수 있음. `where()`, `and()`, `or()`, `not()` 제공
- `findAll`을 보면 회원 이름 명세(`username`)와 팀 이름 명세(`teamName`)를 `and`로 조합해서 검색 조건으로 사용

`MemberSpec` 명세 정의 코드

```
public class MemberSpec {

    public static Specification<Member> teamName(final String teamName) {
        return (Specification<Member>) (root, query, builder) -> {

            if (StringUtils.isEmpty(teamName)) {
                return null;
            }

            Join<Member, Team> t = root.join("team", JoinType.INNER); //회원과
조인
            return builder.equal(t.get("name"), teamName);
        };
    }

    public static Specification<Member> username(final String username) {
        return (Specification<Member>) (root, query, builder) ->
```

```
                builder.equal(root.get("username"), username);
        }
    }
```

- 명세를 정의하려면 `Specification` 인터페이스를 구현
- 명세를 정의할 때는 `toPredicate(...)` 메서드만 구현하면 되는데 JPA Criteria의 `Root`, `CriteriaQuery`, `CriteriaBuilder` 클래스를 파라미터 제공
- 예제에서는 편의상 람다를 사용

> **참고: 실무에서는 JPA Criteria를 거의 안쓴다! 대신에 QueryDSL을 사용하자.**

Query By Example

- https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#query-by-example

```
package study.datajpa.repository;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.data.domain.Example;
import org.springframework.data.domain.ExampleMatcher;
import org.springframework.transaction.annotation.Transactional;
import study.datajpa.entity.Member;
import study.datajpa.entity.Team;

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

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

@SpringBootTest
@Transactional
```

```java
public class QueryByExampleTest {

    @Autowired MemberRepository memberRepository;
    @Autowired EntityManager em;

    @Test
    public void basic() throws Exception {
        //given
        Team teamA = new Team("teamA");
        em.persist(teamA);

        em.persist(new Member("m1", 0, teamA));
        em.persist(new Member("m2", 0, teamA));
        em.flush();

        //when
        //Probe 생성
        Member member = new Member("m1");
        Team team = new Team("teamA"); //내부조인으로 teamA 가능
        member.setTeam(team);

        //ExampleMatcher 생성, age 프로퍼티는 무시
        ExampleMatcher matcher = ExampleMatcher.matching()
                .withIgnorePaths("age");

        Example<Member> example = Example.of(member, matcher);

        List<Member> result = memberRepository.findAll(example);

        //then
        assertThat(result.size()).isEqualTo(1);

    }
}
```

- Probe: 필드에 데이터가 있는 실제 도메인 객체
- ExampleMatcher: 특정 필드를 일치시키는 상세한 정보 제공, 재사용 가능

- Example: Probe와 ExampleMatcher로 구성, 쿼리를 생성하는데 사용

장점

- 동적 쿼리를 편리하게 처리
- 도메인 객체를 그대로 사용
- 데이터 저장소를 RDB에서 NOSQL로 변경해도 코드 변경이 없게 추상화 되어 있음
- 스프링 데이터 JPA `JpaRepository` 인터페이스에 이미 포함

단점

- 조인은 가능하지만 내부 조인(INNER JOIN)만 가능함 외부 조인(LEFT JOIN) 안됨
- 다음과 같은 중첩 제약조건 안됨
  - `firstname = ?0 or (firstname = ?1 and lastname = ?2)`
- 매칭 조건이 매우 단순함
  - 문자는 `starts/contains/ends/regex`
  - 다른 속성은 정확한 매칭(`=`)만 지원

정리

- **실무에서 사용하기에는 매칭 조건이 너무 단순하고, LEFT 조인이 안됨**
- **실무에서는 QueryDSL을 사용하자**

Projections

- https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#projections

엔티티 대신에 DTO를 편리하게 조회할 때 사용
전체 엔티티가 아니라 만약 회원 이름만 딱 조회하고 싶으면?

```
public interface UsernameOnly {
    String getUsername();
}
```

- 조회할 엔티티의 필드를 getter 형식으로 지정하면 해당 필드만 선택해서 조회(Projection)

```
public interface MemberRepository ... {
```

```
    List<UsernameOnly> findProjectionsByUsername(String username);
}
```

- 메서드 이름은 자유, 반환 타입으로 인지

```
@Test
public void projections() throws Exception {
    //given
    Team teamA = new Team("teamA");
    em.persist(teamA);

    Member m1 = new Member("m1", 0, teamA);
    Member m2 = new Member("m2", 0, teamA);
    em.persist(m1);
    em.persist(m2);
    em.flush();
    em.clear();

    //when
    List<UsernameOnly> result =
memberRepository.findProjectionsByUsername("m1");

    //then
    Assertions.assertThat(result.size()).isEqualTo(1);
}
```

```
select m.username from member m
 where m.username='m1';
```

SQL에서도 select절에서 username만 조회(Projection)하는 것을 확인

인터페이스 기반 Closed Projections

프로퍼티 형식(getter)의 인터페이스를 제공하면, 구현체는 스프링 데이터 JPA가 제공

```
public interface UsernameOnly {
```

```
    String getUsername();
}
```

인터페이스 기반 Open Proejctions

다음과 같이 스프링의 SpEL 문법도 지원

```
public interface UsernameOnly {

    @Value("#{target.username + ' ' + target.age + ' ' + target.team.name}")
    String getUsername();
}
```

단! 이렇게 SpEL문법을 사용하면, DB에서 엔티티 필드를 다 조회해온 다음에 계산한다! 따라서 JPQL SELECT 절 최적화가 안된다.

클래스 기반 Projection

다음과 같이 인터페이스가 아닌 구체적인 DTO 형식도 가능

생성자의 파라미터 이름으로 매칭

```
public class UsernameOnlyDto {

    private final String username;

    public UsernameOnlyDto(String username) {
        this.username = username;
    }

    public String getUsername() {
        return username;
    }
}
```

동적 Projections

다음과 같이 Generic type을 주면, 동적으로 프로젝션 데이터 번경 가능

```

```

```
<T> List<T> findProjectionsByUsername(String username, Class<T> type);
```

사용코드

```
List<UsernameOnly> result = memberRepository.findProjectionsByUsername("m1",
UsernameOnly.class);
```

중첩 구조 처리

```
public interface NestedClosedProjection {

    String getUsername();
    TeamInfo getTeam();

    interface TeamInfo {
        String getName();
    }
}
```

```
select
    m.username as col_0_0_,
    t.teamid as col_1_0_,
    t.teamid as teamid1_2_,
    t.name as name2_2_
from
    member m
left outer join
    team t
        on m.teamid=t.teamid
where
    m.username=?
```

주의

- 프로젝션 대상이 root 엔티티면, JPQL SELECT 절 최적화 가능
- 프로젝션 대상이 ROOT가 아니면
  - LEFT OUTER JOIN 처리
  - 모든 필드를 SELECT해서 엔티티로 조회한 다음에 계산

정리

- **프로젝션 대상이 root 엔티티면 유용하다.**
- **프로젝션 대상이 root 엔티티를 넘어가면 JPQL SELECT 최적화가 안된다!**
- **실무의 복잡한 쿼리를 해결하기에는 한계가 있다.**
- **실무에서는 단순할 때만 사용하고, 조금만 복잡해지면 QueryDSL을 사용하자**

네이티브 쿼리

가급적 네이티브 쿼리는 사용하지 않는게 좋음, 정말 어쩔 수 없을 때 사용

최근에 나온 궁극의 방법 ➜ 스프링 데이터 Projections 활용

스프링 데이터 JPA 기반 네이티브 쿼리

- 페이징 지원
- 반환 타입
  - Object[]
  - Tuple
  - DTO(스프링 데이터 인터페이스 Projections 지원)
- 제약
  - Sort 파라미터를 통한 정렬이 정상 동작하지 않을 수 있음(믿지 말고 직접 처리)
  - JPQL처럼 애플리케이션 로딩 시점에 문법 확인 불가
  - 동적 쿼리 불가

JPA 네이티브 SQL 지원

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

    @Query(value = "select * from member where username = ?", nativeQuery =
true)
    Member findByNativeQuery(String username);
```

```
}
```

- JPQL은 위치 기반 파리미터를 1부터 시작하지만 네이티브 SQL은 0부터 시작
- 네이티브 SQL을 엔티티가 아닌 DTO로 변환은 하려면
  - DTO 대신 JPA TUPLE 조회
  - DTO 대신 MAP 조회
  - @SqlResultSetMapping ➡ 복잡
  - Hibernate ResultTransformer를 사용해야함 ➡ 복잡
  - https://vladmihalcea.com/the-best-way-to-map-a-projection-query-to-a-dto-with-jpa-and-hibernate/
  - 네이티브 SQL을 DTO로 조회할 때는 JdbcTemplate or myBatis 권장

Projections 활용

예) 스프링 데이터 JPA 네이티브 쿼리 + 인터페이스 기반 Projections 활용

```
@Query(value = "SELECT m.member_id as id, m.username, t.name as teamName " +
               "FROM member m left join team t",
               countQuery = "SELECT count(*) from member",
               nativeQuery = true)
Page<MemberProjection> findByNativeProjection(Pageable pageable);
```

동적 네이티브 쿼리

- 하이버네이트를 직접 활용
- 스프링 JdbcTemplate, myBatis, jooq같은 외부 라이브러리 사용

예) 하이버네이트 기능 사용

```
//given
String sql = "select m.username as username from member m";

List<MemberDto> result = em.createNativeQuery(sql)
        .setFirstResult(0)
        .setMaxResults(10)
        .unwrap(NativeQuery.class)
        .addScalar("username")
        .setResultTransformer(Transformers.aliasToBean(MemberDto.class))
        .getResultList();
```

```
}
```