4. 쿼리 메소드 기능

#1.인강/jpa활용편/datajpa/강의

- /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

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 `Pageable`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.

스프링 부트 3 - 하이버네이트 6 left join 최적화 설명 추가

스프링 부트 3 이상을 사용하면 하이버네이트 6이 적용된다.

이 경우 하이버네이트 6에서 의미없는 left join을 최적화 해버린다. 따라서
다음을 실행하면 SQL이 LEFT JOIN을 하지 않는 것으로 보인다.

```
@Query(value = "select m from Member m left join m.team t")
    Page<Member> findByAge(int age, Pageable pageable);
```

실행 결과 - SQL

```
select
        m1_0.member_id,
        m1_0.age,
        m1_0.team_id,
        m1_0.username
    from
        member m1_0
```

하이버네이트 6은 이런 경우 왜 left join을 제거하는 최적화를 할까?

실행한 JPQL을 보면 left join을 사용하고 있다.

`select m from Member m left join m.team t`

`Member`와 `Team`을 조인을 하지만 사실 이 쿼리를 `Team`을 전혀 사용하지 않는다. `select` 절이나, `where`절에서 사용하지 않는 다는 뜻이다. 그렇다면 이 JPQL은 사실상 다음과 같다.

`select m from Member m`

`left join`이기 때문에 왼쪽에 있는 `member` 자체를 다 조회한다는 뜻이 된다.
만약 `select`나, `where`에 `team`의 조건이 들어간다면 정상적인 `join`문이 보인다.
JPA는 이 경우 최적화를 해서 `join`없이 해당 내용만으로 SQL을 만든다.

여기서 만약 `Member`와 `Team`을 하나의 SQL로 한번에 조회하고 싶으시다면 JPA가 제공하는 `fetch join`을 사용해야 한다. (`fetch join`은 JPA 기본편 참고)

`select m from Member m left join fetch m.team t`

이 경우에도 SQL에서 join문은 정상 수행된다.

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, Pageable 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