https://freedeveloper.tistory.com/28

실습 내용이나 자세한 소스코드는 GitHub에 있습니다  
포스팅 내용은 간략하게 추린 핵심 내용만 포스팅되어 있습니다

<https://github.com/freespringbook/start-spring-boot-study/tree/chap01-03-spring_data_jpa_uses>

해당 포스팅 참고 서적 URL

<https://book.naver.com/bookdb/book_detail.nhn?bid=12247655>

SpringBoot Version

책이 출간된지 좀 지나고 공부를 해서 가장 최신의 버전으로 실습을 진행함

* Book Version: 2.0.0
* 실습 Version: 2.1.2.RELEASE

1. Spring Boot Spring Data JPA 사용해보기

3. Spring Data JPA를 이용한 단순 게시물의 처리

더미 데이터 생성

@Test

@TestDescription("더미데이터 생성")

**public** **void** **testInsert200**() {

IntStream.rangeClosed(1, 200).forEach(i -> {

Board board = Board.builder()

.title("제목.." + i)

.content("내용 ...." + i + " 채우기 ")

.writer("user0" + (i % 10))

.build();

repo.save(board);

});

}

쿼리 메소드 이용하기

find…By 로 쿼리 메소드 작성 시  
find 뒤에는 엔티티 타입  
By 뒤에는 칼럼명으로 구성

ex) findBoardByTitle

쿼리 메소드 리턴 타입은 **Page<T>**, **Slice<T>**, **List<T>** 와 같은 Collection<T> 형태

// Board 엔티티에서 title에 해당하는 값을 조회해서 List 컬렉션 타입으로 리턴

List<Board> **findBoardByTitle**(String title);

@Test

@TestDescription("쿼리메소드 테스트")

**public** **void** **testByTitle**() {

repo.findBoardByTitle("제목..177").forEach(board -> System.out.println(board));

}

메소드 이름 내에 지원되는 키워드

| **Keyword** | **Sample** | **JPQL snippet** |
| --- | --- | --- |
| And | findByLastnameAndFirstname | … where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | … where x.lastname = ?1 or x.firstname = ?2 |
| Is,Equals | findByFirstname,findByFirstnameIs,findByFirstnameEquals | … where x.firstname = ?1 |
| Between | findByStartDateBetween | … where x.startDate between ?1 and ?2 |
| LessThan | findByAgeLessThan | … where x.age < ?1 |
| LessThanEqual | findByAgeLessThanEqual | … where x.age <= ?1 |
| GreaterThan | findByAgeGreaterThan | … where x.age > ?1 |
| GreaterThanEqual | findByAgeGreaterThanEqual | … where x.age >= ?1 |
| After | findByStartDateAfter | … where x.startDate > ?1 |
| Before | findByStartDateBefore | … where x.startDate < ?1 |
| IsNull | findByAgeIsNull | … where x.age is null |
| IsNotNull,NotNull | findByAge(Is)NotNull | … where x.age not null |
| Like | findByFirstnameLike | … where x.firstname like ?1 |
| NotLike | findByFirstnameNotLike | … where x.firstname not like ?1 |
| StartingWith | findByFirstnameStartingWith | … where x.firstname like ?1 (parameter bound with appended %) |
| EndingWith | findByFirstnameEndingWith | … where x.firstname like ?1 (parameter bound with prepended %) |
| Containing | findByFirstnameContaining | … where x.firstname like ?1 (parameter bound wrapped in %) |
| OrderBy | findByAgeOrderByLastnameDesc | … where x.age = ?1 order by x.lastname desc |
| Not | findByLastnameNot | … where x.lastname <> ?1 |
| In | findByAgeIn(Collection<Age> ages) | … where x.age in ?1 |
| NotIn | findByAgeNotIn(Collection<Age> ages) | … where x.age not in ?1 |
| True | findByActiveTrue() | … where x.active = true |
| False | findByActiveFalse() | … where x.active = false |
| IgnoreCase | findByFirstnameIgnoreCase | … where UPPER(x.firstame) = UPPER(?1) |

findBy를 이용한 특정 칼럼

SQL문에서 특정한 칼럼의 값을 조회할 때는 쿼리 메소드의 이름을 findBy로 시작하는 방식을 이용

Collection<T> findBy + 속성 이름(속성 타입)

// writer가 작성한 모든 데이터를 조회

Collection<Board> **findByWriter**(String writer);

@Test

@TestDescription("게시물에서 user00 이라는 작성자의 모든 데이터를 구한다")

**public** **void** **testByWriter**() {

Collection<Board> results = repo.findByWriter("user00");

results.forEach(board -> System.out.println(board));

}

like 구문 처리

| **형태** | **쿼리 메소드** |
| --- | --- |
| 단순 like | Like |
| 키워드 + '%' | StartingWith |
| '%' + 키워드 | EndingWith |
| '%' + 키워드 + '%' | Containing |

@Test

@TestDescription("키워드 % 검색 테스트")

**public** **void** **testByWriterStartingWith**() {

Collection<Board> results = repo.findByWriterStartingWith("user");

results.forEach(board -> System.out.println(board));

}

@Test

@TestDescription(" % 키워드 검색 테스트")

**public** **void** **testByWriterEndingWith**() {

Collection<Board> results = repo.findByWriterEndingWith("00");

results.forEach(board -> System.out.println(board));

}

@Test

@TestDescription(" % 키워드 % 검색 테스트")

**public** **void** **testByWriterContaining**() {

Collection<Board> results = repo.findByWriterContaining("05");

results.forEach(board -> System.out.println(board));

}

// 작성자에 대한 like % 키워드 %

Collection<Board> **findByWriterContaining**(String writer);

// 작성자에 대한 like 키워드 %

Collection<Board> **findByWriterStartingWith**(String writer);

// 작성자에 대한 like % 키워드

Collection<Board> **findByWriterEndingWith**(String writer);

and 혹은 or 조건 처리

2개 이상의 속성을 이용해서 엔티티들을 검색해야 할때

게시글의 title과 content 속성에 특정한 문자열이 들어있는 게시물을 검색  
findBy + TitleContaining + Or + ContentContaining

// OR 조건의 처리

Collection<Board> **findByTitleContainingOrContentContaining**(String title, String content);

@Test

@TestDescription(" And Or 조건 테스트")

**public** **void** **findByTitleContainingOrContentContaining**() {

Collection<Board> results = repo.findByTitleContainingOrContentContaining("09", "09");

results.forEach(board -> System.out.println(board));

}

부등호 처리

쿼리 메소드에서는 > 와 < 같은 부등호는 **GreaterThan**과 **LessThan**을 이용해서 처리

// title LIKE % ? % AND BNO > ?

Collection<Board> **findByTitleContainingAndBnoGreaterThan**(String keywoard, Long num);

@Test

@TestDescription("title LIKE % ? % AND BNO > ? 부등호 처리 테스트")

**public** **void** **testByTitleAndBno**() {

Collection<Board> results = repo.findByTitleContainingAndBnoGreaterThan("5", 50L);

results.forEach(board -> System.out.println(board));

}

order by 처리

가져오는 데이터의 순서를 지정하기 위해서 이용  
OrderBy + 속성 + Asc or Desc

// bno > ? ORDER BY bno DESC

Collection<Board> **findByBnoGreaterThanOrderByBnoDesc**(Long bno);

@Test

@TestDescription("bno > ? ORDER BY bno DESC")

**public** **void** **testBnoOrderBy**() {

Collection<Board> results = repo.findByBnoGreaterThanOrderByBnoDesc(90L);

getBoardList(results);

}

페이징 처리와 정렬

모든 쿼리 메소드의 마지막 파라미터로 페이지 처리를 할 수 있는 Pageable 인터페이스와 정렬을 처리하는 Sort 인터페이스를 사용할 수 있음

페이징 처리에 필요한 정보 제공 구현 클래스중 PageRequest.of()를 가장 많이 사용

Pageable 인터페이스가 적용되면  
리턴타입은

* org.springframework.data.domain.Slice
* org.springframework.data.domain.Page
* java.util.List

// bno > ? ORDER BY bno DESC Paging

List<Board> **findByBnoGreaterThanOrderByBnoDesc**(Long bno, Pageable paging);

@Test

@TestDescription("bno > ? ORDER BY bno DESC Paging")

**public** **void** **testBnoOrderByPaging**() {

// 첫 번째 페이지(인덱스 번호는 0부터 시작)이고 10건의 데이터를 가져오도록 설정

Pageable paging = PageRequest.of(0, 10);

Collection<Board> results = repo.findByBnoGreaterThanOrderByBnoDesc(0L, paging);

results.forEach(board -> System.out.println(board));

}

정렬 처리에는 Pageable 인터페이스와 같이 Sort 클래스 이용

PageRequest 생성자

| **생성자** | **설명** |
| --- | --- |
| PageRequest(int page, int size) | 페이지 번호(0부터 시작), 페이지당 데이터의 수 |
| PageRequest(int page, int size, Sort.Direction direction, String … props) | 페이지 번호, 페이지당 데이터의 수, 정렬 방향, 속성(칼럼)들 |
| PageRequest(int page, int size, Sort sort) | 페이지 번호, 페이지당 데이터의 수, 정렬 방향 |

PageRequest 생성자를 이용한 정렬

List<Board> **findByBnoGreaterThan**(Long bno, Pageable paging);

@Test

@TestDescription("PageRequst 생성자를 이용한 정렬")

**public** **void** **testBnoPagingSort**() {

Pageable paging = PageRequest.of(0, 10, Sort.Direction.ASC, "bno");

Collection<Board> results = repo.findByBnoGreaterThan(0L, paging);

getBoardList(results);

}

Page 타입

Page<T> 타입을 이용하면 Spring MVC와 연동할 때 상당한 편리함을 제공

Page<Board> **findByBnoGreaterThan**(Long bno, Pageable paging);

@Test

@TestDescription("PageRequst 생성자를 이용한 정렬")

**public** **void** **testBnoPagingSort**() {

Pageable paging = PageRequest.of(0, 10, Sort.Direction.ASC, "bno");

Page<Board> result = repo.findByBnoGreaterThan(0L, paging);

System.out.println("PAGE SIZE: " + result.getSize());

System.out.println("TOTAL PAGES: " + result.getTotalPages());

System.out.println("TOTAL COUNT: " + result.getTotalElements());

System.out.println("NEXT: " + result.nextPageable());

List<Board> list = result.getContent();

results.forEach(board -> System.out.println(board));

}

웹에서 필요한 데이터들을 추가적으로 처리

| **메소드** | **설명** |
| --- | --- |
| int getNumber() | 현재 페이지의 정보 |
| int getSize() | 한 페이지의 크기 |
| int getTotalPages() | 전체 페이지의 수 |
| int getNumberOfElements() | 결과 데이터 수 |
| boolean hasPreviousPage() | 이전 페이지의 존재 여부 |
| boolean hasNextPage() | 다음 페이지의 존재 여부 |
| boolean isLastPage() | 마지막 페이지 여부 |
| Pageable nextPageable() | 다음 페이지 객체 |
| Pageable previousPageable() | 이전 페이지 객체 |
| List getContent() | 조회된 데이터 |
| boolean hasContent() | 결과 존재 여부 |
| Sort getSort() | 검색 시 사용된 Sort 정보 |

[@Query](https://github.com/Query)를 이용하기

좀 더 구체적인 조건등을 지정하기 위해 @Query를 이용  
**JPQL**이라는 JPA에서 사용하는 쿼리 문법을 이용하거나 순수한 데이터베이스에 맞는 SQL을 사용

단순 게시물의 처리를 위한 [@Query](https://github.com/Query) 작성

* **JPQL**
  + 테이블 대신 엔티티 타입 이용
  + 칼럼명 대신 엔티티 속성 이용

@Query("SELECT b FROM Board b WHERE b.title LIKE %?1% AND b.bno > 0 ORDER BY b.bno DESC")

List<Board> **findByTitle**(String title);

'?1'은 첫 번째로 전달되는 파라미터

@Test

@TestDescription("@Query 를 사용해 JPQL로 title 조회")

**public** **void** **testByTitle2**() {

repo.findByTitle("17").forEach(board -> System.out.println(board));

}

내용에 대한 검색 처리 [@Param](https://github.com/Param)

여러 개의 파라미터를 전달할 떄 이름을 이용해 쉽게 구분해서 전달

@Query("SELECT b FROM Board b WHERE b.content LIKE %:content% AND b.bno > 0 ORDER BY b.bno DESC")

List<Board> **findByContent**(@Param("content") String content);

작성자에 대한 검색 처리 #{#entityName}

#{#entityName}은 Repository 인터페이스를 정의할 때 <엔티티 타입, PK 타입>에서 엔티티 타입을 자동으로 참고  
유사한 상속 구조의 Repository 인터페이스를 여러 개 생성하는 경우라면 유용하게 사용

@Query("SELECT b FROM #{#entityName} WHERE b.content LIKE %?1% AND b.bno > 0 ORDER BY b.bno DESC")

List<Board> **findByWriter**(String writer);

[@Query](https://github.com/Query)의 활용

@Query를 활용하면 얻을 수 있는 장점

* 리턴 값이 반드시 엔티티 타입이 아니라 필요한 몇 개의 칼럼 값들만 추출할 수 있음
* nativeQuery 속성을 지정해서 데이터베이스에 사용하는 SQL을 그대료 사용할 수 있음
* Repository에 지정된 엔티티 타입 뿐 아니라 필요한 엔티티 타입을 다양하게 사용할 수 있음

필요한 칼럼만 추출하는 경우

// content 칼럼을 제외

@Query("SELECT board.bno, board.title, board.writer, board.regdate FROM Board board WHERE board.title LIKE %?1% AND board.bno > 0 ORDER BY board.bno DESC")

List<Object[]> findByTitle2(String title);

@Test

@TestDescription("content 칼럼을 제외")

**public** **void** **testByTitle17**() {

repo.findByTitle2("17").forEach(arr -> System.out.println(Arrays.toString(arr)));

}

nativeQuery 사용

복잡한 쿼리 작성시 유용하지만 데이터베이스에 독립적이라는 장점은 포기  
@Query에 nativeQuery=true 를 지정하면 value 값을 그대로 실행

//nativeQuery 사용

@Query(value = "SELECT bno, title, writer FROM tbl\_boards WHERE title LIKE CONCAT('%', ?1, '%') AND bno > 0 ORDER BY bno DESC", nativeQuery=**true**)

List<Object[]> findByTitle3(String title);

@Test

@TestDescription("nativeQuery 사용")

**public** **void** **testByTitle3**() {

repo.findByTitle3("17").forEach(arr -> System.out.println(Arrays.toString(arr)));

}

[@Query](https://github.com/Query)와 Paging 처리/정렬

@Query를 이용하더라도 **Pageable** 인터페이스 활용 가능  
**Pageable** 타입 사용시 '@Query로 작성한 내용 + 페이징 처리' 형태

//@Query와 Paging 처리/정렬

@Query("SELECT board FROM Board board WHERE board.bno > 0 ORDER BY board.bno DESC")

List<Board> **findBypage**(Pageable pageable);

@Test

@TestDescription("@Query와 Paging 처리/정렬")

**public** **void** **testByPaging**() {

Pageable pageable = PageRequest.of(0, 10);

repo.findBypage(pageable).forEach(board -> System.out.println(board));

}

Querydsl을 이용한 동적 SQL의 처리

동적인 상황에 대한 처리를 위해서 Querydsl(Query Domain Specific Language)를 이용

QueryDSL 의존성 추가

QueryDSL은 스프링 부트가 의존성을 관리해주므로 버전을 명시하지 않아도 됨  
apt모듈은 QueryDSL이 Entity모델을 보고 Query용 Specific Language(특정 언어)를 만들어 주는 모듈

<**dependency**>

<**groupId**>com.querydsl</**groupId**>

<**artifactId**>querydsl-apt</**artifactId**>

</**dependency**>

<**dependency**>

<**groupId**>com.querydsl</**groupId**>

<**artifactId**>querydsl-jpa</**artifactId**>

</**dependency**>

QueryDSL용 빌드 설정

**Querydsl**은 JPA를 처리하기 위해 엔티티 클래스를 생성하는 방식을 이용  
이를 **'Qdomain'**이라 하며 동적 쿼리를 생성해 낼 때 이를 이용

<**plugin**>

<**groupId**>com.mysema.maven</**groupId**>

<**artifactId**>apt-maven-plugin</**artifactId**>

<**version**>1.1.3</**version**>

<**executions**>

<**execution**>

<**goals**>

<**goal**>process</**goal**>

</**goals**>

<**configuration**>

<**outputDirectory**>target/generated-sources/java</**outputDirectory**>

<**processor**>com.querydsl.apt.jpa.JPAAnnotationProcessor</**processor**>

</**configuration**>

</**execution**>

</**executions**>

</**plugin**>

Predicate 준비

**Predicate** '이 조건이 맞다'고 판단하는 근거를 함수로 제공하는 것  
Repository에서 **Predicate**를 파라미터로 전달하기 위해서는 QueryDslPredicateExecutor<T>  
인터페이스를 Repository에 추가 해주어야 함  
**Predicate**는 필요한 곳에서 생성하는 방식을 이용하기도 하지만 별도의 클래스 등으로 만들어서 사용할 수도 있음

QueryDslPredicateExecutor 인터페이스 메소드

| **메소드** | **설명** |
| --- | --- |
| long count(Predicate) | 데이터의 전체 개수 |
| boolean exists(Predicate) | 데이터의 존재 여부 |
| Iterable findAll(Predicate) | 조건에 맞는 모든 데이터 |
| Page findAll(Predicate) | 조건에 맞는 모든 데이터 |
| Iterable findAll(Predicate, Sort) | 조건에 맞는 모든 데이터와 정렬 |
| T findOne(Predicate) | 조건에 맞는 하나의 데이터 |

Repository 변경

Repository 인터페이스에는 QueryDslPredicateExecutor<T> 인터페이스를 상속하도록 추가

**public** **interface** **BoardRepository** **extends** **CrudRepository**<**Board**, **Long**>, **QuerydslPredicateExecutor**<**Board**> {

Predicate 생성 및 테스트

**Predicate** '단언하다, 확신하다'는 의미  
boolean으로 리턴되는 결과 데이터를 만드는데 **BooleanBuilder**를 이용해서 생성

@Test

@TestDescription("@Querydsl를 이용한 Predicate 생성 및 테스트")

**public** **void** **testPredicate**() {

String type = "t";

String keyword = "17";

BooleanBuilder builder = **new** BooleanBuilder();

QBoard board = QBoard.board;

**if** (type.equals("t")) {

builder.and(board.title.like("%" + keyword + "%"));

}

// bno > 0

builder.and(board.bno.gt(0L));

Pageable pageable = PageRequest.of(0, 10);

Page<Board> result = repo.findAll(builder, pageable);

System.out.println("PAGE SIZE: " + result.getSize());

System.out.println("TOTAL PAGES: " + result.getTotalPages());

System.out.println("TOTAL COUNT: " + result.getTotalElements());

System.out.println("NEXT: " + result.nextPageable());

List<Board> list = result.getContent();

list.forEach(b -> System.out.println(b));

}