7. 나머지 기능들

#1.인강/jpa활용편/datajpa/강의

- /Specifications (명세)
- /Query By Example
- /Projections
- /네이티브 쿼리

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;
```

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

@SpringBootTest
@Transactional
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 ON m.team_id = t.team_id",
            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();

}
```