Query Methods

1. 기본적인 쿼리 메소드

Spring Data JPA는 `Repository` 인터페이스를 사용해 데이터베이스의 데이터를 쿼리하는 데 필요한 메소드들을 제공. `JpaRepository`나 `CrudRepository`를 상속받은 인터페이스에서 자동으로 쿼리 메소드를 정의할 수 있다.

1.1. 메소드 이름 규칙

Spring Data JPA에서 쿼리 메소드는 메소드 이름으로 쿼리를 생성.

예를 들어:

- `findBy`: 특정 필드로 데이터를 찾을 때 사용

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

  → `username` 값에 해당하는 `User` 객체들을 반환하는 쿼리 생성.

- `findAll`: 모든 엔티티를 조회

  ```
  List<User> findAll();
  ```

- `countBy`: 조건에 맞는 데이터의 개수를 세는 메소드

  ```
  long countByUsername(String username);
  ```

- `deleteBy`: 조건에 맞는 데이터를 삭제하는 메소드

  ```
  void deleteByUsername(String username);
  ```

1.2. 특수한 쿼리 메소드

- `And` / `Or`: 여러 조건을 결합

  ```
  List<User> findByUsernameAndAge(String username, int age);
  List<User> findByUsernameOrAge(String username, int age);
  ```

- `Like`: 부분 일치 검색

```
List<User> findByUsernameLike(String pattern);
```

- `Between`: 범위 검색

```
List<User> findByAgeBetween(int start, int end);
```

- `IsNull` / `IsNotNull`: NULL값 여부로 검색

```
List<User> findByEmailIsNull();
List<User> findByEmailIsNotNull();
```

- `OrderBy`: 정렬

```
List<User> findByAgeOrderByUsernameAsc(int age);
```

---

2. @Query 어노테이션을 활용한 커스텀 쿼리

`@Query` 어노테이션을 사용하면 JPQL(Java Persistence Query Language)이나 네이티브 SQL 쿼리를 직접 작성하여 복잡한 쿼리를 수행할 수 있다.

2.1. JPQL 사용 예시

```
@Query("SELECT u FROM User u WHERE u.age > :age")
List<User> findUsersByAgeGreaterThan(@Param("age") int age);
```

2.2. 네이티브 SQL 사용 예시

```
@Query(value = "SELECT * FROM user WHERE age > ?1", nativeQuery = tr
ue)
List<User> findUsersByNativeSQL(int age);
```

2.3. 다중 결과 반환

- **단일 값 반환**: `@Query`를 사용하여 단일 값 반환

```
@Query("SELECT COUNT(u) FROM User u WHERE u.age > :age")
```

```
long countUsersByAgeGreaterThan(@Param("age") int age);
```

- **DTO로 반환**: 특정 데이터를 DTO 객체로 매핑하여 반환

```
@Query("SELECT new com.example.dto.UserDTO(u.username, u.age) FROM User u WHERE u.age > :age")
List<UserDTO> findUserDTOsByAgeGreaterThan(@Param("age") int age);
```

---

3. 페이징(Paging)과 정렬(Sorting)

Spring Data JPA는 페이징과 정렬을 손쉽게 처리할 수 있도록 지원합니다. 이를 위해 `Pageable`과 `Sort` 객체를 활용.

3.1. 페이징

```
Page<User> findByAgeGreaterThan(int age, Pageable pageable);
```

`Pageable` 객체를 사용하면 페이지 번호, 크기, 정렬 등을 설정할 수 있다.

3.2. 정렬

```
List<User> findByAgeGreaterThan(int age, Sort sort);
```

`Sort` 객체를 사용하여 정렬 조건을 설정할 수 있다.

---

4. 명시적 쿼리 메소드

쿼리 메소드를 사용할 때, 보다 복잡한 조건을 명시적으로 지정해야 하는 경우 `@Query` 어노테이션을 사용하는 방법 외에도 `Criteria API`나 `Specification`을 활용할 수 있다.

4.1. Criteria API (동적 쿼리 생성)

`CriteriaBuilder`를 사용하여 동적으로 쿼리를 생성할 수 있다.

예를 들어:

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<User> query = cb.createQuery(User.class);
```

```
Root<User> user = query.from(User.class);
query.select(user).where(cb.equal(user.get("username"), "john_doe"));
```

4.2. **Specification 사용**

`Specification`은 복잡한 조건을 처리할 때 유용하며, `JpaSpecificationExecutor` 인터페이스를 사용한다.

```
public class UserSpecification implements Specification<User> {
    public Predicate toPredicate(Root<User> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder) {
        return criteriaBuilder.equal(root.get("username"), "john_doe");
    }
}
```

`Specification`을 사용하여 복합적인 조건을 정의할 수 있다.

---

5. 쿼리 메소드에서 자주 쓰이는 키워드 정리

- `Distinct` : 중복된 데이터를 제거하여 반환

  ```
  List<User> findDistinctByUsername(String username);
  ```

- `Exists` : 조건을 만족하는 데이터가 존재하는지 확인

  ```
  boolean existsByUsername(String username);
  ```

- `Not` : 부정 조건

  ```
  List<User> findByAgeNot(int age);
  ```

- `In` : 여러 값에 대해 검색

  ```
  List<User> findByAgeIn(List<Integer> ages);
  ```

- `NotIn` : 주어진 값들 외의 값들을 검색

  ```
  List<User> findByAgeNotIn(List<Integer> ages);
  ```

- `OrderBy`: 정렬 순서 지정

```
List<User> findByAgeOrderByUsernameDesc(int age);
```

---

6. 쿼리 메소드 최적화

쿼리 메소드 사용 시 성능을 최적화하기 위한 몇 가지 팁:

- **최소화된 쿼리 반환**: 필요한 데이터만 반환하여 성능을 높다.
- **인덱스 활용**: 자주 조회되는 필드에 인덱스를 설정.
- **페이징 처리**: 한 번에 너무 많은 데이터를 가져오지 않도록 적절한 페이징을 사용.

뚱. Summary

Spring Data JPA의 쿼리 메소드는 데이터베이스와의 상호작용을 직관적이고 편리하게 만들어 줌. 쿼리 메소드의 이름만으로 복잡한 SQL을 작성할 수 있으며, 커스텀 쿼리, 페이징 처리, 동적 쿼리 등 다양한 방법을 통해 유연한 데이터 조회가 가능.