

스프링부트로 RestFulAPI 구현하기

10장 JPA 심화 1 JPQL 연관관계

- JPQL
- OneToOne
- OneToMany
- ManyToOne
- ManyToMany
- cascade
- fetch Lazy eager

박명회

10장 Generatedvalue 애노테이션

키워드	설명
GenerationType.AUTO	JPA 구현체가 자동으로 생성전략 결정
GenerationType.IDENTITY	기본키 생성을 데이터베이스에 위임 MYSQL인 경우 AUTO_INCREMENT로 사용하여 기본키 생성
GenerationType.SEQUENCE	데이터베이스 시퀀스 오브젝트를 이용한 기본키 생성
GenerationType.TABLE	키 생성용 테이블 사용

10장 JPA 쿼리 메소드 및 JPQL snippet

키워드	쿼리메소드	JPQL snippet
and	findByEmailAndName	where tb.email=?1 and tb.name=?2
or	findByEmailOrName	where tb.email=?1 or tb.name=?2
is,Equals	findByNames findByNameEquals	where tb.name =?1
between	findByCreatedDateBetween	where tb.createdDate between ?1 and ?2
lessthan	findByAgeLessThan	where tb.age < ?1
lessthenEqual	findByAgeLessThanEqual	where tb.age <= ?1
GreaterThen	findByGreaterThen	where tb.age > ?1
GreaterThenEqual	findByGreaterThenEqual	where tb.age >= ?1
After	findByCreatedDateAfter	where tb.createdDate > ?1
Before	findByCreatedDateBefore	where tb.createdDate < ?1

10장 JPA 쿼리 메소드 및 JPQL snippet

키워드	쿼리메소드	JPQL snippet
orderBy	findAllOrderByldDesc	order by tb.id desc
Null	findByNameNull	where tb.name is null
in	findByNameIn(Collection<Name> names)	tb.name in ?1
NotIn	findByNameNotIn(Collection<Name> names)	tb.name not in ?1
True	findByActiveTrue	where active = true
False	findByActiveFalse	where active = false
Containing	findByNameContaining	where name like ?1

10장 쿼리메서드 와 JPQL 테스트

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import java.util.Optional;

public interface UserRepository extends JpaRepository<User, Long> {

    // JPQL
    @Query("SELECT u FROM User u WHERE u.name = :nameOrEmail OR u.email = :nameOrEmail")
    Optional<User> findByNameOrEmail(@Param("nameOrEmail") String nameOrEmail);

    // 쿼리 메서드
    // Optional<User> findByUsernameOrEmail(String username, String email);

    List<User> findByAgeLessThan(int age);

}
```

10장 Name과 email 테스트 코드 만들어보기

@SpringBootTest

```
public class UserRepositoryTest {
```

```
    @Test
```

```
    public void testFindByUsernameOrEmail() {
```

```
        // given
```

```
        User user = new User();
```

```
        user.setUsername("testuser");
```

```
        user.setEmail("testuser@example.com");
```

```
        userRepository.save(user);
```

```
        // when
```

```
        Optional<User> foundByUsername = userRepository.findByUsernameOrEmail("testuser");
```

```
        Optional<User> foundByEmail = userRepository.findByUsernameOrEmail("testuser@example.com");
```

```
        // then
```

```
        assertThat(foundByUsername).isPresent();
```

```
        assertThat(foundByUsername.get().getUsername()).isEqualTo("testuser");
```

```
        assertThat(foundByEmail).isPresent();
```

```
        assertThat(foundByEmail.get().getEmail()).isEqualTo("testuser@example.com");
```

```
    }
```

```
}
```

10장 lessthan 테스트 코드 만들어보기

@Test

```
public void testFindByAgeLessThan() {
```

```
    // given
```

```
    User user1 = new User();
```

```
    user1.setUsername("younguser");
```

```
    user1.setEmail("younguser@example.com");
```

```
    user1.setAge(20);
```

```
    userRepository.save(user1);
```

```
    User user2 = new User();
```

```
    user2.setUsername("olderuser");
```

```
    user2.setEmail("olderuser@example.com");
```

```
    user2.setAge(30);
```

```
    userRepository.save(user2);
```

```
    // when
```

```
    List<User> foundUsers = userRepository.findByAgeLessThan(25);
```

```
    // then
```

```
    assertThat(foundUsers).hasSize(1);
```

```
    assertThat(foundUsers.get(0).getUsername()).isEqualTo("younguser");
```

```
}
```

10장 findAllByOrderByIdAsc() findAllByOrderByIdDesc() 테스트

```
List<User> findAllByOrderByIdAsc();  
List<User> findAllByOrderByIdDesc();
```

@Test

```
public void testFindAllByOrderByIdAsc() {  
    // when List<User> foundUsersAsc = userRepository.findAllByOrderByIdAsc();  
    // then assertThat(foundUsersAsc).hasSize(3);  
    assertThat(foundUsersAsc.get(0).getId()).isEqualTo(user1.getId());  
    assertThat(foundUsersAsc.get(1).getId()).isEqualTo(user2.getId());  
    assertThat(foundUsersAsc.get(2).getId()).isEqualTo(user3.getId());  
}
```

@Test

```
public void testFindAllByOrderByIdDesc() {  
    // when List<User> foundUsersAsc = userRepository.findAllByOrderByIdDesc();  
    // then  
    assertThat(foundUsersAsc).hasSize(3);  
    assertThat(foundUsersAsc.get(0).getId()).isEqualTo(user1.getId());  
    assertThat(foundUsersAsc.get(1).getId()).isEqualTo(user2.getId());  
    assertThat(foundUsersAsc.get(2).getId()).isEqualTo(user3.getId());  
}
```


10장 findAllByOrderByIdAsc() findAllByOrderByIdDesc() 테스트

// Containing 메서드

```
List<User> findByNameContaining(String keyword);
```

// JPQL 메서드

```
@Query("SELECT u FROM User u WHERE u.name LIKE %:keyword%")
```

```
List<User> findByNameContainingJPQL(@Param("keyword") String keyword);
```

10장 native 쿼리 사용해보기

```
@Query(value = "SELECT * FROM User u WHERE u.name LIKE %:keyword%", nativeQuery = true)  
List<User> findByNameContainingNative(@Param("keyword") String keyword);
```

10장 연관관계 맵핑

일대일 @OneToOne

일대다 @OneToMany

다대일 @ManyToOne

다대다 @ManyToMany

연관관계 맵핑 방법은 위와 같이 4가지가 존재합니다.

연관관계 매핑은 객체 지향 프로그래밍과 관계형 데이터베이스 간의 데이터를 매핑하기 위해 사용되는 기법입니다. 이는 ORM(Object-Relational Mapping)의 중요한 부분으로, 엔티티 간의 관계를 데이터베이스의 테이블 관계로 변환합니다.

cascade: 연관된 엔티티에 대해 특정 작업(영속성 전이)을 전파합니다.

예를 들어, **CascadeType.ALL**은 모든 작업(저장, 삭제, 병합 등)을 전파합니다.

fetch: 연관된 엔티티를 가져오는 전략을 정의합니다.

FetchType.EAGER는 즉시 로딩, **FetchType.LAZY**는 지연 로딩을 의미합니다.

mappedBy: 양방향 관계에서 주인을 지정합니다. 주인이 아닌 쪽에서 **mappedBy**를 사용하여 관계의 주인을 지정합니다.

10장 영속성 전이

영속성 전이란

영속성 전이 즉, `cascade`의 사전적 정의는 ‘작은폭포’, ‘폭포처럼 흐른다’입니다. 영속성 적이란 엔티티의 상태를 변경할때 해당 엔티티와 연관 엔티티의 상태 변화를 전파시키는 옵션입니다. 이때 부모는 `One`에 해당하고 자식은 `Many`에 해당합니다.

즉 `User`엔티티가 삭제되었을때 해당 엔티티와 연관되어 있는 `FreeBoard`엔티티가 함께 삭제 됩니다.

`User`엔티티를 저장할때 `FreeBoard`엔티티를 한꺼번에 저장할 수도 있습니다.

<code>cascade</code> 타입	설명
<code>PERSIST</code>	부모 엔티티가 영속화될때 자식엔티티도 영속화
<code>MERGE</code>	부모 엔티티가 병합될때 자식엔티티도 병합
<code>REMOVE</code>	부모 엔티티가 삭제될때 연관된 자식 엔티티도 삭제
<code>REFRESH</code>	부모 엔티티가 <code>refresh</code> 되면 연관된 자식 엔티티도 <code>refresh</code>
<code>DETACH</code>	부모 엔티티가 <code>detech</code> 되면 연관된 자식 엔티티도 <code>detach</code> 상태로 변경
<code>ALL</code>	부모 엔티티의 영속성 상태 변화를 자식엔티티도 모두 전이

10장 @OneToOne

@Entity

```
public class User {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String name;  
    @OneToOne(mappedBy = "user", cascade = CascadeType.ALL,  
                orphanRemoval=true)  
    private UserProfile profile;  
}
```

@Entity

```
public class UserProfile {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String address;  
    private String phoneNumber;  
  
    @OneToOne  
    @JoinColumn(name = "user_id")  
    private User user;  
}
```

단방향

User삭제시 변화 없음

양방향

cascadeType.ALL이면

User객체의 UserProfile변경해서 User객체
저장시 UserProfile도 함께 저장됨

고아객체제거

orphanRemoval=true이면

User객체 삭제시 UserProfile이 먼저
삭제되고 User객체가 삭제된다.

@OneToOne과 @OneToMany에서도
동일한 규칙을 지니고 있다.

10장 @OneToMany @ManyToOne

이번에는 **User**와 **FreeBoard** 테이블 간의 일대다(OneToMany) 및 다대일(ManyToOne) 관계를 설정하는 예제를 제공할 것입니다. **FreeBoard** 테이블은 사용자가 작성하는 게시물들을 나타냅니다.

— @OneToMany

User.class

```
import javax.persistence.*;
import java.util.ArrayList;
import java.util.List;
```

@OneToMany는 기본 fetch가 지연로딩
(LAZY)

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    private List<FreeBoard> freeBoards = new ArrayList<>();
}
```

10장 @OneToMany @ManyToOne

— @ManyToOne

FreeBoard.class

```
import javax.persistence.*;
```

```
@Entity
```

```
public class FreeBoard {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
```

```
    private String title;
```

```
    private String content;
```

```
    @ManyToOne(fetch = FetchType.EAGER)
```

```
    @JoinColumn(name = "user_id")
```

```
    private User user;
```

```
}
```

@ManyToOne은 기본 fetch가 즉시로딩
(EAGER)

10장 @OneToMany @ManyToOne 테스트 코드

— @ManyToOne

@Test

```
void oneToManyCascadeTest(){
    User user = userRepository.findById(2l).orElseThrow();
    List<FreeBoard> list = user.getList();
    list.add(
        FreeBoard.builder()
            .title("새로운글")
            .content("새로운 테스트")
            .author("새로운사람")
            .user(user)
            .build());
    list.remove(1);
    userRepository.save(user);
}
```

cascadeType.ALL속성으로 인하여
List<FreeBoard> list의 내용을 삭제 및
변경해서

userRepository.save(user);
할시에 freeboard내용도 함께 변경
되는 것을 알 수 있다.

orphanRemoval = true속성으로
인하여
User객체 삭제시 User가 작성한
FreeBoard도 함께 삭제 되는 것을 알
수 있다.

10장 @ManyToMany

User와 FreeBoard 간의 다대다(ManyToMany) 관계를 설정하는 예제를 제공할 것입니다. 이 예제에서는 사용자가 여러 게시물에 좋아요를 누를 수 있고, 하나의 게시물도 여러 사용자에게 좋아요를 받을 수 있는 시나리오를 다룹니다.

@Entity

```
public class User {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
```

```
    private String name;
```

```
    @ManyToMany
```

```
    @JoinTable(
```

```
        name = "user_freeboard_likes",
```

```
        joinColumns = @JoinColumn(name = "user_id"),
```

```
        inverseJoinColumns = @JoinColumn(name = "freeboard_id")
```

```
    )
```

```
    private List<FreeBoard> likedFreeBoards = new ArrayList<>();
```

```
}
```

10장 @ManyToMany

@Entity

```
public class FreeBoard {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String title;  
    private String content;  
  
    @ManyToMany(mappedBy = "likedFreeBoards")  
    private List<User> likedByUsers = new ArrayList<>();  
}
```

User 엔티티:

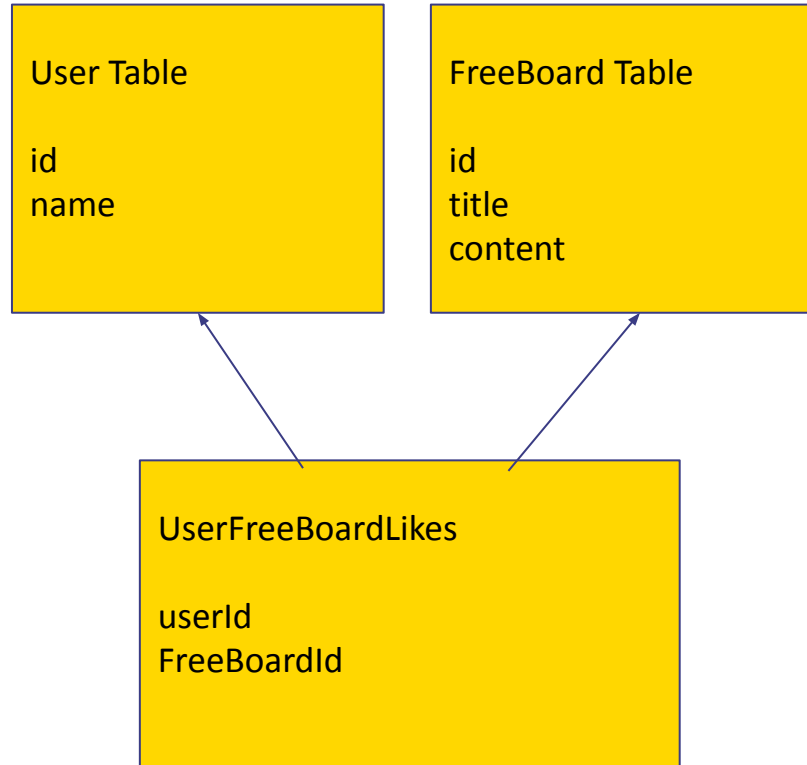
@ManyToMany를 통해 User 엔티티가 FreeBoard 엔티티와 다대다 관계를 맺고 있음을 나타냅니다.

@JoinTable은 중간 테이블을 정의합니다. name 속성은 중간 테이블의 이름을 지정하고, joinColumns와 inverseJoinColumns는 중간 테이블에서 사용할 외래 키를 지정합니다.

FreeBoard 엔티티:

@ManyToMany(mappedBy = "likedFreeBoards")를 통해 FreeBoard 엔티티가 User 엔티티와 다대다 관계를 맺고 있으며, 관계의 주인이 아님을 나타냅니다. 이 경우 mappedBy 속성은 User 엔티티의 likedFreeBoards 필드를 참조합니다.

10장 @ManyToMany



user_freeboard_likes 테이블 :
`userId`, `freeboardId` 필드를 가지며,

이는 User와 FreeBoard 간의 다대다 관계를 나타내는 외래 키입니다.

10장 @ManyToMany

-- Insert users

```
INSERT INTO User (name) VALUES ('Alice'); -- ID 1
```

```
INSERT INTO User (name) VALUES ('Bob'); -- ID 2
```

-- Insert freeboards

```
INSERT INTO FreeBoard (title, content) VALUES ('Post 1', 'This is the first post.');
```

```
INSERT INTO FreeBoard (title, content) VALUES ('Post 2', 'This is the second post.');
```

-- Insert likes into user_freeboard_likes

```
INSERT INTO user_freeboard_likes (user_id, freeboard_id) VALUES (1, 1); -- Alice likes Post 1
```

```
INSERT INTO user_freeboard_likes (user_id, freeboard_id) VALUES (1, 2); -- Alice likes Post 2
```

```
INSERT INTO user_freeboard_likes (user_id, freeboard_id) VALUES (2, 1); -- Bob likes Post 1
```

Alice가 Post 1과 Post 2에 좋아요를 누르고,
Bob이 Post 1에 좋아요를 누른다고 가정합니다.

감사합니다