

JPA Entity Relation

1. 엔티티 매핑

JPA(Java Persistence API)를 사용하면, 자바 클래스를 데이터베이스 테이블에 매핑할 수 있다. 이 과정에서 사용하는 클래스를 엔티티(Entity)라고 하며, 이 엔티티 클래스의 인스턴스가 데이터베이스의 행(row)에 해당한다. 이러한 매핑을 통해, 자바 애플리케이션에서 객체 지향적인 방식으로 데이터베이스 작업을 수행할 수 있게 된다. JPA는 이 매핑 정보를 사용하여 애플리케이션과 데이터베이스 사이의 상호작용을 처리한다.

[JPA 주요 어노테이션]

@Entity : 클래스가 JPA 엔티티임을 나타낸다. 클래스 이름이 DB 테이블 이름에 매핑된다.
@Table : 엔티티 클래스가 매핑될 테이블의 정보를 명시한다. (name, catalog, schema 등의 속성을 가질 수 있음)
@Id : 엔티티의 기본 키(Primary Key)를 나타낸다.
@GeneratedValue : 기본 키의 값을 자동으로 생성할 전략을 명시한다.
(AUTO, IDENTITY, SEQUENCE, TABLE 중 선택)
@Column : 필드가 매핑될 테이블의 컬럼을 명시한다.
(name, nullable, length 등의 속성을 가질 수 있음)
@ManyToOne, @OneToMany, @OneToOne, @ManyToMany : 엔티티 간의 관계를 명시한다. (@JoinColumn과 함께 사용되는 경우가 많음)
@JoinColumn : 외래 키(Foreign Key)를 매핑할 때 사용한다.
(name, referencedColumnName 등의 속성을 가질 수 있음)
@Transient : 필드가 영속성 컨텍스트에 저장되거나 검색되지 않음을 나타낸다.
@Temporal : 날짜 타입(java.util.Date, java.util.Calendar)의 매핑을 명시한다.
(TemporalType.DATE, TemporalType.TIME, TemporalType.TIMESTAMP 중 선택)

1) 엔티티와 테이블 매핑

@Entity 어노테이션은 클래스가 JPA 엔티티임을 나타낸다.
@Table 어노테이션은 엔티티가 매핑될 데이터베이스 테이블의 이름을 지정한다. 만약 @Table 어노테이션을 생략하면, 클래스 이름이 테이블 이름으로 사용된다.

[예시]

- 이 코드에서 User 클래스는 users 테이블에 매핑된다.

```
@Entity
@Table(name = "users")
public class User {

}
```

2) 기본 키 매핑

각 엔티티는 고유한 식별자를 가져야 하는데, 이를 기본 키(Primary Key)라고 한다. @Id 어노테이션을 사용하여 필드를 기본 키로 지정한다. @GeneratedValue 어노테이션은 기본 키 값의 생성 전략을 지정할 때 사용한다.

[예시]

- 이 코드에서 id 필드는 자동으로 생성되는 기본 키로 지정된다.

```
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

}
```

3) 필드와 컬럼 매핑

@Column 어노테이션을 사용하여 엔티티의 필드가 데이터베이스의 어떤 컬럼에 매핑될지 지정할 수 있다. 이 어노테이션을 생략하면 필드 이름이 컬럼 이름으로 사용된다.

[예시]

- 이 코드에서 username 필드는 user_name 컬럼에 매핑되며, null 값을 허용하지 않고 최대 길이가 50인 컬럼으로 지정된다.

```
@Entity
public class User {
```

```

@Id
private Long id;

@Column(name = "user_name", nullable = false, length = 50)
private String username;

}

```

2. 관계매핑

관계 매핑은 객체 간의 관계를 데이터베이스의 테이블 간의 관계에 매핑하는 과정이다. 이를 통해 객체 지향 프로그래밍에서의 복잡한 관계를 데이터베이스에 효율적으로 반영할 수 있다.

1) 단방향 관계 , 양방향 관계

단방향 관계: 단방향 관계는 한 엔티티가 다른 엔티티를 참조할 수 있지만, 반대 방향으로의 참조가 이루어지지 않는 관계를 말한다.

[예시]

```

@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToOne
    @JoinColumn(name = "author_id")
    private Author author;
}

```

이 예시에서 Book 엔티티는 저자(Author)를 참조하지만, Author 엔티티는 Book에 대한 정보를 가지고 있지 않는다.

양방향 관계: 양쪽 클래스에서 서로 접근할 수 있는 관계이다. 양방향 관계는 참조를 양쪽에 모두 유지해야 한다. 즉 양방향 관계는 두 엔티티가 서로를 참조할 수 있는 관계를 말한다.

[예시]

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

    private String name;

    @OneToMany(mappedBy = "author")
    private List<Book> books = new ArrayList<>();
}
```

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

    private String title;

    @ManyToOne
    @JoinColumn(name = "author_id")
    private Author author;
}
```

이 예시에서는 Author 엔티티가 자신이 쓴 Book들을 List<Book> 형태로 가지고 있으며, Book 엔티티는 자신의 Author를 참조한다. 이로 인해 양방향 관계가 형성된다.

- 단방향 관계는 구현이 간단하고 명확하지만, 양쪽 엔티티 간의 관계를 완전히 파악하기 어려운 반면 양방향 관계는 관계를 보다 명확하게 표현할 수 있지만 양쪽 엔티티에서 서로를 참조해야 하므로 관리가 더 복잡해질 수 있다. 따라서 실제 애플리케이션 개발에서는 필요에 따라 적절한 관계 유형을 선택해야 한다.

2) 지연 로딩(Lazy Loading) vs 즉시 로딩(Eager Loading)

- 지연 로딩과 즉시 로딩은 JPA에서 연관된 엔티티를 언제 로딩할지 결정하는 두 가지 전략

지연 로딩: 엔티티의 속성 값이 실제로 사용될 때까지 로딩을 지연시키는 방법이다. 실제로 필요한 시점까지 데이터 로딩을 연기함으로써 불필요한 데이터베이스 접근을 줄일수 있으므로 성능 최적화에 도움이 될 수 있다.

즉시 로딩: 엔티티를 로딩할 때 연관된 엔티티까지 모두 즉시 로딩하는 방법이다. 필요한 모든 데이터를 한 번에 가져오지만, 불필요한 데이터까지 로딩할 위험이 있다.

[예시] 저자와 책의 관계

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

    private String name;

    @OneToMany(fetch = FetchType.LAZY) // 지연 로딩 설정
    private List<Book> books = new ArrayList<>();
}
```

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

    private String title;

    @ManyToOne(fetch = FetchType.EAGER) // 즉시 로딩 설정
    private Author author;
}
```

지연 로딩 설정 시: Author 엔티티를 조회할 때, 연관된 Book 엔티티는 즉시 로드되지 않는다. 대신, 실제로 books 컬렉션에 접근(예: author.getBooks().get(0))할 때 데이터베이스에서 해당 책들을 로드한다.

즉시 로딩 설정 시: Book 엔티티를 조회할 때, 연관된 Author 엔티티는 즉시 함께 로드된다. Book 엔티티를 데이터베이스에서 가져올 때, 연관된 Author 정보도 바로 조회하여 엔티티에 포함시킨다.

3) 매핑 관계

@ManyToOne (다대일) 관계 매핑

하나의 엔티티가 다른 엔티티의 여러 인스턴스와 관련될 수 있는 관계이다. 예를 들어, 여러 개의 주문(Order)이 하나의 고객(Customer)에게 속하는 관계이다.

@Entity

```
public class Order {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    @ManyToOne  
    @JoinColumn(name = "customer_id")  
    private Customer customer;  
}
```

@Entity

```
public class Customer {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    @OneToMany(mappedBy = "customer")  
    private List<Order> orders = new ArrayList<>();  
}
```

@OneToMany (일대다) 관계 매핑

하나의 엔티티가 다른 엔티티의 여러 인스턴스와 관련될 수 있는 관계의 반대 방향이다. 예를 들어, 하나의 저자(Author)가 여러 권의 책(Book)을 출판한 관계이다.

@Entity

```
public class Author {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```

private Long id;

@OneToMany
@JoinColumn(name = "author_id")
private List<Book> books = new ArrayList<>();
}

```

@OneToOne (일대일) 관계 매핑

두 엔티티 간에 서로 하나만 관련될 수 있는 관계이다. 예를 들어, 한 명의 사용자(User)가 하나의 프로필(Profile)을 가지는 관계이다.

```

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

    @OneToOne
    @JoinColumn(name = "profile_id")
    private Profile profile;
}

```

@ManyToMany (다대다) 관계 매핑

두 엔티티 그룹이 서로 복잡하게 연결된 관계를 가질 때 사용된다. 예를 들어, 여러 학생(Student)이 여러 강좌(Course)에 등록할 수 있는 관계이다.

```

@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToMany
    @JoinTable(
        name = "enrollment",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private List<Course> courses = new ArrayList<>();
}

```

3. Fetch 조인

Fetch 조인은 JPA에서 연관된 엔티티나 컬렉션을 SQL의 한 번의 조인을 사용하여 함께 로딩하는 기능이다. 이 방법을 사용하면, 지연 로딩으로 인해 발생할 수 있는 N+1 문제를 해결하고, 성능을 향상시킬 수 있다.

Fetch 조인은 JPA를 사용하여 애플리케이션의 성능을 최적화할 수 있는 강력한 도구이다. 하지만, Fetch 조인을 사용할 때는 데이터의 양과 쿼리의 복잡성을 고려해야 한다. 또한, 필요하지 않은 데이터까지 가져오는 과도한 Fetch 조인의 사용은 오히려 성능 저하를 일으킬 수 있으므로 주의해야 한다.

[예시]

도서(Book)와 저자(Author) 엔티티가 있고 한 저자가 여러 권의 책을 저술할 수 있다고 가정

Fetch 조인을 사용하기 전에는 연관된 엔티티를 로드하기 위해 여러 번의 쿼리가 필요했지만, Fetch 조인을 사용한 후에는 단 하나의 쿼리로 필요한 모든 정보를 빠르게 로드할 수 있게 된다. 이로 인해 애플리케이션의 성능이 향상되고, 데이터베이스와의 불필요한 통신이 줄어들게 된다.

[Fetch 조인 사용 전]

저자를 조회할 때, 저자의 책들은 지연 로딩(Lazy Loading)으로 설정되어 있어서 저자 정보만 먼저 로드되고, 각 책의 정보는 접근할 때마다 별도의 쿼리로 로드된다.

@Entity

```
public class Author {
```

```
    @Id
```

```
    @GeneratedValue
```

```
    private Long id;
```

```
    private String name;
```

```
    @OneToMany(mappedBy = "author", fetch = FetchType.LAZY)
```

```
    private List<Book> books;
```

```
}
```

```
List<Author> authors = entityManager.createQuery("SELECT a FROM Author a", Author.class).getResultList();
```

```
    for (Author author : authors) {
```

```
        author.getBooks().forEach(book -> System.out.println(book.getTitle())); //
```

여기서 각 책에 대한 추가 쿼리가 실행됨}

[Fetch 조인 사용 후]

Fetch 조인을 사용하여 저자와 그의 책들을 한 번의 쿼리로 함께 로드한다. 이로 인해 추가적인 쿼리 없이 필요한 모든 정보에 접근할 수 있다.

```
List<Author> authors = entityManager.createQuery(
    "SELECT a FROM Author a JOIN FETCH a.books", Author.class
).getResultList();
```

```
for (Author author : authors) {
    author.getBooks().forEach(book -> System.out.println(book.getTitle())); // 추가 쿼리 없이 책 정보 접근
}
```

4. 고급 매핑 전략

1) 상속 관계 매핑

JPA에서는 객체 지향의 상속 구조를 데이터베이스 테이블 구조에 매핑할 수 있는 여러 전략을 제공한다. 가장 일반적인 상속 매핑 전략은 단일 테이블 전략(SINGLE_TABLE), 조인된 테이블 전략(JOINED), 그리고 구분 컬럼 테이블 전략(TABLE_PER_CLASS)이다.

단일 테이블 전략(SINGLE_TABLE): 모든 엔티티를 하나의 테이블에 매핑한다. 가장 간단하지만, null이 허용되는 컬럼이 많아질 수 있다.

조인된 테이블 전략(JOINED): 각 엔티티를 별도의 테이블로 분리하고, 상속 구조에서는 조인을 사용하여 관련 데이터를 조회한다. 데이터 정규화가 잘 되어 있지만, 조인으로 인한 성능 저하가 발생할 수 있다.

구분 컬럼 테이블 전략(TABLE_PER_CLASS): 각 엔티티를 자신의 테이블로 매핑한다. 상속받은 필드들이 각 테이블에 중복되어 저장된다. 이 전략은 데이터베이스의 다형성 쿼리를 지원하지 않는다.

2) 복합 키와 식별 관계 매핑

복합 키는 두 개 이상의 필드를 조합하여 고유 식별자를 생성하는 경우 사용된다. JPA에서는 @IdClass와 @EmbeddedId를 통해 복합 키를 매핑할 수 있다.

@IdClass: 복합 키를 위한 별도의 클래스를 정의하고, 엔티티 클래스에서 이를 사용한다.

@EmbeddedId: 복합 키 필드를 엔티티 내에 직접 임베딩한다.

```
@IdClass(PersonId.class)
@Entity
public class Person {
    @Id
    private String firstName;

    @Id
    private String lastName;
}

public class PersonId implements Serializable {
    private String firstName;
    private String lastName;
}
```

3) 엔티티 간의 관계 매핑 시 고려 사항

단방향 vs 양방향: 양방향 관계는 관리해야 할 참조가 더 많기 때문에, 정말 필요한 경우에만 사용하는 것이 좋다.

지연 로딩 vs 즉시 로딩: 가능한 지연 로딩을 기본으로 사용하여 성능 문제를 방지한다.

카디널리티 고려: 관계의 카디널리티(일대일, 일대다, 다대일, 다대다)에 따라 적절한 어노테이션을 사용한다.

연관관계 주인: 양방향 관계에서는 연관관계의 주인을 정확히 설정해야 한다.

이러한 고급 매핑 전략들은 JPA를 사용하여 복잡한 도메인 모델을 데이터베이스에 효과적으로 매핑하고 관리하는 데 도움이 된다. 각 전략의 선택은 애플리케이션의 요구 사항과 성능 고려사항에 따라 달라질 수 있다.

5. JPA 관계 매핑 최적화

JPA 관계 매핑 최적화는 애플리케이션의 성능을 크게 향상시킬 수 있는 중요한 작업이다.

N+1 문제를 해결하고, 적절한 로딩 전략을 선택하며, 데이터베이스와 JPA 설정을 최적화하는 것은 데이터 접근 계층의 성능과 확장성을 개선하는 데 큰 도움이 된다.

1) N+1 문제

N+1 문제 정의: N+1 문제는 연관된 엔티티를 로딩할 때 발생하는 성능 문제이다. 예를 들어, 한 번의 쿼리로 N개의 엔티티를 로드한 후, 각 엔티티에 대해 연관된 엔티티를 로드하기 위해 추가적인 N번의 쿼리가 실행되는 상황이다. 결과적으로 총 N+1번의 쿼리가 실행되어 성능이 저하된다.

예시: Post 엔티티와 Comment 엔티티가 있고, 한 Post에 여러 Comment가 연관되어 있을 때, 모든 Post를 조회하고 각 Post에 대한 Comment를 로딩하려고 하면 N+1 문제가 발생할 수 있다.

2) N+1 문제 해결 전략

Fetch 조인 사용: JOIN FETCH를 사용하여 관련 엔티티를 미리 로드하면 추가적인 쿼리 없이 필요한 데이터를 한 번에 가져올 수 있다.

```
String jpql = "SELECT p FROM Post p JOIN FETCH p.comments";
```

배치 사이즈 설정: @BatchSize 어노테이션 또는 hibernate.default_batch_fetch_size 설정을 사용하여 한 번에 로드할 연관 엔티티의 수를 설정할 수 있다. 이 방법은 지연 로딩을 사용할 때 유용하다.

```
@OneToMany
@BatchSize(size = 10)
private List<Comment> comments;
또는 application.properties에서 설정:
```

```
spring.jpa.properties.hibernate.default_batch_fetch_size=10
```

엔티티 그래프 사용: JPA 2.1부터 추가된 엔티티 그래프 기능을 사용하여 조회 시점에 로딩할 속성을 동적으로 정의할 수 있다.

```
EntityGraph<Post> graph = entityManager.createEntityGraph(Post.class);
graph.addAttributeNodes("comments");
List<Post> posts = entityManager.createQuery("select p from Post p",
Post.class)
    .setHint("javax.persistence.loadgraph", graph)
```

.getResultList();

3) 성능 최적화 전략

적절한 로딩 전략 선택: 필요에 따라 지연 로딩과 즉시 로딩을 적절히 선택한다. 가능한 지연 로딩을 기본으로 사용하여 불필요한 데이터 로딩을 방지한다.

쿼리 힌트 사용: 데이터베이스에 최적화된 쿼리를 실행하기 위해 JPA 구현체에서 제공하는 쿼리 힌트를 사용할 수 있다.

응답 크기 최소화: DTO(Data Transfer Object)를 사용하여 클라이언트에 전송할 데이터의 크기를 최소화한다. 필요한 데이터만 선택적으로 로드하여 성능을 향상시킬 수 있다.

인덱스 활용: 데이터베이스의 쿼리 성능을 향상시키기 위해 적절한 인덱스를 생성한다.

쓰기 지연 활용: JPA의 쓰기 지연 기능을 사용하여 엔티티의 변경 사항을 즉시 데이터베이스에 반영하지 않고, 트랜잭션의 끝에서 한 번에 반영하여 성능을 최적화할 수 있다.