JPA

ORM(객체-관계 매핑, Object-Relational Mapping)이란?

ORM은 객체지향 프로그래밍 언어(Java, Python 등)에서 사용하는 객체와 관계형 데이터베이스의 테이블을 매핑(연결)하는 기술.

ORM을 사용하면 **개발자가 객체를 사용하여 데이터베이스 작업을 처리**할 수 있으며, 직접 SQL 쿼리를 작성하지 않아도 데이터를 저장, 수정, 삭제, 조회할 수 있음.

ORM의 주요 목적

1. **객체와 관계형 데이터베이스의 간극 해소**
   - 객체지향 언어의 **클래스, 객체**와 관계형 데이터베이스의 **테이블, 레코드** 간의 차이를 극복하기 위해 사용함.
2. **SQL 의존도를 낮춤**
   - 개발자는 SQL 쿼리를 직접 작성하는 대신, ORM 라이브러리가 자동으로 SQL을 생성하고 실행함.

ORM의 동작 방식

- **엔티티(Entity)**: 객체지향 프로그래밍에서 데이터베이스의 테이블에 매핑되는 클래스.
- **매핑(Mapping)**: 클래스의 속성(필드)과 데이터베이스의 열(컬럼)을 연결하는 과정.
- **ORM 라이브러리**: 개발자가 작성한 객체 작업(Java 객체 생성, 수정 등)을 SQL로 변환하고 데이터베이스에 실행.

ORM의 장점

1. **생산성 향상**:
   - 개발자는 객체지향적으로 작업하며 데이터베이스와의 상호작용을 처리할 수 있어 생산성이 높아짐.
   - SQL 쿼리를 직접 작성하지 않아도 됨.
2. **유지보수 용이**:
   - 데이터베이스 변경(예: 테이블 이름, 필드 이름 변경)이 있어도 ORM 설정만 수정하면 됨.
   - 코드가 데이터베이스 스키마와 분리되어 관리됨.
3. **DBMS 독립성**:
   - 동일한 ORM 코드를 사용하면 DBMS(MySQL, PostgreSQL 등)가 변경되어도 큰 수정 없이 작업이 가능함.
   - JPA 같은 ORM은 다양한 데이터베이스에 대한 드라이버를 지원함.
4. **객체지향적 접근**:
   - 데이터베이스 작업을 객체를 사용해 처리하므로 객체지향 프로그래밍의 장점을 활용할 수 있음.
   - **지연 로딩(Lazy Loading)**, **캐싱** 등의 고급 기능 지원.
5. **표준화**:
   - JPA(Java Persistence API)와 같은 표준화된 ORM 인터페이스가 있어 확장성과 유연성이 뛰어남.

ORM의 단점

1. **학습 곡선**:
   - ORM 사용법을 익히고 설정하는 데 시간이 필요함.
   - 특히, 복잡한 쿼리 작성 시 ORM만으로 해결하기 어려운 경우도 있음.
2. **성능 이슈**:
   - 단순 쿼리는 직접 작성한 SQL보다 ORM이 약간 느릴 수 있음.
   - 복잡한 쿼리나 대량의 데이터 처리를 위해 **Native SQL**을 사용하는 경우도 생김.
3. **추상화된 쿼리의 한계**:

- ORM은 복잡한 비즈니스 로직에 적합하지 않을 수 있음.
- 성능 최적화가 필요한 경우 SQL 튜닝을 직접 해야 함.

왜 ORM을 사용하나요?

1. **객체지향 개발 환경 유지**:
   - ORM은 객체지향 프로그래밍의 철학을 유지하며 데이터베이스와 상호작용을 가능하게 함.
2. **코드 간결화**:
   - 데이터베이스에 관련된 복잡한 코드를 줄이고, 객체 지향적 비즈니스 로직에 집중할 수 있음.
3. **표준화된 API 활용**:
   - JPA 같은 표준을 사용하면, 코드의 이식성이 높아지고 다양한 데이터베이스에서 재사용이 가능.

언제 ORM을 사용하지 않아야 할까요?

- **고성능 SQL이 중요한 경우**:
  - 매우 복잡한 쿼리나 대규모 데이터 처리 작업에서는 SQL을 직접 작성하는 것이 더 효율적임.
- **간단한 프로젝트**:
  - 프로젝트가 매우 단순한 경우 ORM 설정이 오히려 오버헤드가 될 수 있음.

JPA(Java Persistence API)의 특징

JPA는 **Java 표준 ORM(Object-Relational Mapping) 기술**로, 객체지향 프로그래밍 언어와 관계형 데이터베이스 간의 데이터 작업을 간소화하고 추상화하는 데 중점을 둡니다. JPA는 ORM에 대한 명세를 제공하며, Hibernate와 EclipseLink 같은 다양한 구현체가 있음.

JPA의 주요 특징

1. **표준 API**

- JPA는 Java EE(EJB 3.0)의 일부로 처음 등장했으며, 현재는 독립적으로도 사용 가능.
- 다양한 ORM 프레임워크(Hibernate, EclipseLink 등)의 공통 인터페이스 역할.

2. **객체-관계 매핑 (ORM)**

- 엔티티 클래스와 데이터베이스 테이블을 매핑.
- 객체 지향적인 방식으로 데이터를 처리.

3. **영속성 컨텍스트 (Persistence Context)**

- 엔티티의 상태를 관리하고, 데이터베이스와 동기화하는 **캐시 메모리** 역할.
- 1차 캐시, 더티 체킹 등을 통해 효율적인 데이터 관리 가능.

4. **JPQL(Java Persistence Query Language)**

- SQL과 유사한 쿼리 언어로 엔티티 객체를 대상으로 데이터 조회/수정 가능.
- 테이블이나 컬럼이 아닌 **엔티티와 필드 기반**으로 작동.

5. **자동 SQL 생성**

- 데이터를 삽입, 조회, 수정, 삭제할 때 SQL을 자동으로 생성.
- 기본 CRUD 작업에서 SQL 작성 부담 감소.

6. **지연 로딩 (Lazy Loading) 및 즉시 로딩 (Eager Loading)**

- 연관된 엔티티를 필요할 때 로드하거나(Lazy), 즉시 로드(Eager)할 수 있는 옵션 제공.

7. **트랜잭션 관리**

- 데이터의 일관성과 안정성을 보장하기 위해 트랜잭션 API와 통합.
- `@Transactional`과 같은 스프링 프레임워크 지원으로 간단하게 관리 가능.

8. **애노테이션 기반 구성**

- Java 코드 내에서 직접 매핑 정의 (`@Entity`, `@Table`, `@Id` 등).
- XML 설정 없이 애노테이션으로 간편하게 관리 가능.

9. **DBMS 독립성**

- 특정 데이터베이스에 종속되지 않고, JPA 프로바이더(Hibernate 등)가 SQL 변환을 처리.

10. **엔티티 생명주기 관리**

- 엔티티의 상태(Persist, Detached, Removed 등)를 JPA가 관리.
- 객체의 상태에 따라 자동으로 SQL 실행.

JPA가 가지는 현재 스프링 프레임워크에서의 위치

1. **Spring Data JPA의 핵심 기술**

- JPA는 스프링 프레임워크에서 데이터 계층을 관리하기 위한 **표준 기술**로 자리잡음.
- **Spring Data JPA**는 JPA의 기능을 확장하여 Repository 기반 개발을 쉽게 만듦.
  - CRUD 메서드 자동 생성.
  - 메서드 이름 기반 쿼리 생성.
  - JPQL 및 Native Query 지원.

2. **스프링 프레임워크와의 통합**

- 스프링의 **트랜잭션 관리**(@Transactional)와 깊이 통합.
- 애플리케이션의 엔티티 관리를 JPA로 처리하고, 비즈니스 로직은 스프링 서비스 계층에서 구현.

3. **ORM 기술의 사실상 표준**

- Hibernate, EclipseLink 같은 구현체가 JPA 명세를 따르기 때문에, JPA는 Java 생태계의 ORM 표준으로 자리 잡음.
- 특히, 스프링 프레임워크에서는 Hibernate(JPA 구현체)가 기본 제공.

4. **Spring Boot와의 밀접한 통합**

- Spring Boot는 JPA를 쉽게 설정하고 사용할 수 있도록 자동 설정 기능 제공.
- `spring-boot-starter-data-jpa` 의존성만 추가하면 JPA 환경이 자동으로 구성됨.
- **HikariCP, Hibernate 설정 자동화** 및 **데이터베이스 초기화** 지원.

5. **데이터 접근 계층의 주류 기술**

- 과거 JDBC Template과 MyBatis가 주요 데이터 접근 방식이었다면, 현재는 JPA/Spring Data JPA가 데이터 계층 개발의 주류.
- 유지보수성과 생산성이 향상됨.

6. **현대 애플리케이션 아키텍처에 적합**

- JPA는 **클라우드 네이티브 애플리케이션**, **마이크로서비스 아키텍처** 등 현대적인 애플리케이션 구조에서 데이터 계층 관리를 효율적으로 처리.

JPA를 사용하는 이유

- **생산성 향상**: CRUD 작업과 기본 매핑 작업을 자동화하여 개발 속도 증가.
- **코드 간결화**: SQL 대신 객체 지향 코드를 작성.
- **유지보수성 증가**: 데이터베이스 변경 시 애플리케이션 코드 수정 최소화.
- **데이터 일관성 관리**: 영속성 컨텍스트를 통해 데이터 상태를 자동으로 동기화.

- **현대적 애플리케이션 요구사항에 부합**: 성능과 확장성을 동시에 고려한 설계.

JPA와 Hibernate의 관계

- **JPA는 인터페이스, Hibernate는 구현체**:
  - JPA는 구현체 없이 동작하지 않음. Hibernate는 JPA 명세를 구현한 대표적인 ORM 프레임워크.
  - 즉, Hibernate는 JPA 표준을 따르면서 추가적인 기능을 제공.
- **JPA는 Hibernate의 상위 개념**:
  - JPA는 ORM을 사용할 때 반드시 따라야 할 규칙(명세)이며, Hibernate는 이를 실제로 구현하여 실행 가능한 기능으로 만듦.
  - Hibernate 외에도 EclipseLink, OpenJPA 등 다른 JPA 구현체가 존재.
- **Spring Data JPA에서의 역할**:
  - 스프링 프레임워크는 Hibernate를 기본 구현체로 사용.
  - `spring-boot-starter-data-jpa`를 사용하면 내부적으로 Hibernate가 JPA 구현체로 자동 설정됨.

JPA와 Hibernate의 차이점

구분	JPA	Hibernate
정의	ORM을 위한 **Java 표준 API**	JPA를 구현한 **프레임워크**
역할	명세 제공 (인터페이스)	JPA 명세를 구현하고, 추가 기능 제공
기능 제공	자체적으로는 아무런 기능도 제공하지 않음	캐싱, Native SQL, 고급 성능 최적화 등 다양한 추가 기능 제공
종속성	특정 구현체에 의존	독립적 사용 가능하나, JPA와 함께 사용하는 것이 일반적
확장성	모든 JPA 구현체에서 동일하게 작동	Hibernate 고유의 기능은 다른 구현체와 호환되지 않을 수 있음

결론

- **JPA는 인터페이스이고 Hibernate는 구현체**:
  - JPA는 ORM을 위한 규칙을 정의한 **표준 API**이고, Hibernate는 이를 구현한 대표적인 프레임워크.
  - 스프링 부트에서 JPA를 사용하면 Hibernate가 기본 구현체로 동작.
- **Hibernate를 통한 JPA 사용**:
  - 대부분의 개발자들은 Hibernate를 통해 JPA를 간접적으로 사용.
  - 즉, JPA 명세를 기반으로 코드를 작성하지만, 실제 작업은 Hibernate가 처리.
- **사용자의 입장**:
  - JPA 명세에 따라 코드를 작성하면 Hibernate 외의 다른 구현체로도 쉽게 전환 가능.
  - 하지만 Hibernate의 고급 기능이 필요할 경우 이를 활용할 수 있음.

Hibernate (하이버네이트)

1. Hibernate의 탄생

- **탄생 배경**:
  - 2001년 Gavin King이 개발한 오픈소스 프로젝트로 시작.
  - 당시의 EJB 2.x(Entity Bean)는 복잡한 설정과 성능 문제가 많았고, ORM 솔루션으로 적합하지 않았음.
  - 객체지향적 방식으로 데이터베이스와 상호작용을 간소화하려는 목표로 개발됨.

- **초기 목표**:
  - 기존의 EJB(Entity Bean) 기술을 대체할 더 간단하고 효율적인 ORM 솔루션 제공.
  - SQL 작성 없이 객체를 데이터베이스에 저장하고 조회할 수 있도록 지원.

---

2. 주요 발전 과정

1. **Hibernate 1.x (2001~2002)**:
   - 초기 버전은 단순히 객체와 관계형 데이터베이스의 매핑에 초점을 맞춤.
   - XML 기반 매핑 설정 제공.
   - Java 개발자들 사이에서 점차 인기를 얻음.
2. **Hibernate 2.x (2003~2004)**:
   - 더욱 정교한 ORM 기능 도입.
   - 복합 키, 상속 매핑, 컬렉션 매핑 등 고급 매핑 기능 추가.
   - 캐싱 메커니즘 도입으로 성능 개선.
   - EJB 2.x에 대한 강력한 대안으로 자리 잡음.
3. **Hibernate 3.x (2005~2009)**:
   - JPA(Java Persistence API)가 등장하면서 JPA를 지원하는 구현체로 발전.
   - JPQL(Java Persistence Query Language) 도입.
   - XML 매핑 외에 어노테이션 매핑 지원.
   - 고급 캐싱과 배치 처리 기능 강화.
   - Spring Framework와의 통합이 강화되며 ORM 표준으로 자리 잡음.
4. **Hibernate 4.x (2011~2015)**:
   - JPA 2.0 스펙 준수.
   - 복합 키 매핑 개선 및 엔티티 라이프사이클 관리 강화.
   - HQL(Hibernate Query Language) 최적화 및 다양한 데이터베이스 지원.
   - 하위 호환성을 유지하며 안정적인 기능 제공.
5. **Hibernate 5.x (2015~현재)**:
   - JPA 2.1/2.2 표준 준수.
   - 부트스트래핑 개선 및 JDK 8 지원.
   - Java 8의 `Optional`, `Stream`, `CompletableFuture` 등과의 통합.
   - 엔티티 그래프(Entity Graph)와 배치 처리(Batch Processing) 성능 최적화.
   - 스프링 부트(Spring Boot)와 완벽한 통합.
6. **Hibernate 6.x (2021~현재)**:
   - JPA 3.0 표준 지원.
   - SQL 변환 성능 개선 및 다양한 벤더 확장 기능 추가.
   - JSON 데이터 타입 및 NoSQL 데이터베이스 지원 강화.
   - 최신 Java 기능(JDK 11+)과의 호환성 확보.

---

3. Hibernate의 현재

- **대표적인 JPA 구현체**:
  - JPA 명세 이전부터 ORM 시장에서 사용되었으며, JPA 등장 이후 이를 가장 효과적으로 구현.
- **스프링 프레임워크와의 통합**:

- Spring Data JPA의 기본 ORM 엔진으로 사용되며, 설정과 사용이 간단.

- **활발한 커뮤니티**:
  - Hibernate는 가장 큰 사용자 커뮤니티와 방대한 문서를 보유.
  - 다양한 데이터베이스와의 호환성, 성능 최적화, 안정성을 제공.

---

4. Hibernate의 의의

- **ORM 표준화의 선구자**:
  - 객체지향 프로그래밍과 관계형 데이터베이스 간의 간극을 줄이는 데 크게 기여.
  - JPA 표준의 기반이 된 기술적 아이디어를 제공.
- **현대적 개발 방식의 토대**:
  - 현대의 자바 개발에서 데이터베이스 작업을 간소화하고 유지보수성을 향상시키는 데 중요한 역할.

초급

1. **Spring Boot와 JPA의 기본 설정**
   - Spring Boot 프로젝트에 JPA 의존성을 추가하고 application.properties 파일에 데이터베이스 연결 설정하기.
2. **Entity와 Table 매핑**
   - @Entity와 @Table 어노테이션을 사용해 클래스와 데이터베이스 테이블을 매핑하는 기본 방법.
3. **기본 CRUD Repository 사용법**
   - JpaRepository나 CrudRepository를 이용해 기본적인 Create, Read, Update, Delete 작업 수행.
4. **데이터베이스 연결 및 MySQL 사용**
   - 간단한 개발 환경에서 MySQL 데이터베이스를 설정하고 이를 활용하여 테스트하는 방법.
5. **Spring Boot DevTools와 JPA 디버깅**
   - 개발 편의를 위한 DevTools 사용 및 SQL 쿼리 로깅 설정(@EnableJpaRepositories).
6. **JPQL (Java Persistence Query Language) 기초**
   - 데이터베이스 테이블 대신 엔티티 객체를 대상으로 하는 간단한 JPQL 쿼리 작성.
7. **스프링 데이터 JPA 메서드 쿼리 작성법**
   - 메서드 이름만으로 쿼리를 생성하는 규칙(@Query 사용 없이 메서드 정의).
8. **기본 필드 매핑**
   - JPA 어노테이션으로 데이터베이스 테이블 필드와 클래스 필드를 매핑하고 기본 키 설정.

초급-2 기본 어노테이션

1. 엔티티 정의와 기본 매핑

어노테이션	설명
`@Entity`	JPA 엔티티임을 선언. 해당 클래스는 데이터베이스 테이블과 매핑됨.
`@Table`	엔티티와 매핑될 테이블 이름을 지정. (기본적으로 클래스 이름이 테이블 이름으로 사용됨)
`@Id`	해당 필드가 엔티티의 기본 키임을 선언.
`@GeneratedValue`	기본 키 생성 전략을 지정. (예: `GenerationType.IDENTITY`, `AUTO`, `SEQUENCE`, `TABLE`)
`@Column`	필드를 데이터베이스의 특정 컬럼과 매핑. 이름, 길이, nullable 여부 등 세부 설정 가능.
`@Embedded`	내장 타입(Embeddable 클래스)을 포함할 때 사용.

`@Embeddable`	내장 타입 클래스를 정의할 때 사용.
`@Transient`	해당 필드는 데이터베이스에 매핑하지 않음을 선언.
`@Lob`	대용량 데이터를 저장할 때 사용. (예: CLOB, BLOB)

2. 기본 키 및 복합 키 매핑

어노테이션	설명
`@IdClass`	복합 키를 별도의 클래스에서 정의.
`@EmbeddedId`	복합 키를 내장 타입(Embeddable)으로 정의.
`@MapsId`	기본 키를 매핑할 때 다른 엔티티의 필드와 공유하도록 설정.

3. 관계 매핑

어노테이션	설명
`@OneToOne`	1:1 관계를 정의.
`@ManyToOne`	N:1 관계를 정의. (다수가 하나와 연관)
`@OneToMany`	1:N 관계를 정의. (하나가 다수와 연관)
`@ManyToMany`	N:N 관계를 정의.
`@JoinColumn`	외래 키를 매핑할 때 사용. (관계 매핑 시 주로 사용)
`@JoinTable`	다대다 관계에서 조인 테이블을 정의.
`@MappedBy`	관계의 역방향을 정의. 주로 양방향 매핑에서 사용.

4. 페치(Fetch) 및 캐스케이드(Cascade) 전략

어노테이션	설명
`@Fetch` (Hibernate)	데이터 페치 전략을 세부적으로 설정.
`@Cascade` (Hibernate)	엔티티 상태 변경 시 연관 엔티티에 전파할 동작 지정.
`@FetchType.LAZY`	지연 로딩(Lazy Loading)을 명시적으로 설정.
`@FetchType.EAGER`	즉시 로딩(Eager Loading)을 명시적으로 설정.
`@CascadeType.ALL`	모든 상태 변화를 연관 엔티티에 전파.

5. 기타 고급 어노테이션

어노테이션	설명
`@MappedSuperclass`	부모 클래스가 직접 테이블로 매핑되지 않고, 자식 클래스에 필드 매핑을 전달할 때 사용.
`@Inheritance`	상속 전략 지정. (예: `SINGLE_TABLE`, `JOINED`, `TABLE_PER_CLASS`)
`@DiscriminatorColumn`	상속 엔티티에서 부모 테이블에 구분 컬럼을 정의.
`@DiscriminatorValue`	상속 엔티티에서 구분 컬럼의 값을 정의.
`@Version`	낙관적 락(Optimistic Lock)을 위한 버전 필드 매핑.
`@NamedQuery`	정적 쿼리를 이름으로 정의.
`@NamedNativeQuery`	네이티브 쿼리를 이름으로 정의.

@Column 주요 옵션

옵션	기본값	설명
name	필드 이름	데이터베이스 컬럼 이름을 지정. 클래스의 필드 이름과 다르게 설정하고 싶을 때 사용.
nullable	`true`	컬럼이 `NULL` 값을 허용하는지 여부를 설정.
unique	`false`	컬럼에 유니크 제약 조건을 추가.
insertable	`true`	SQL `INSERT`에 해당 필드를 포함할지 여부를 지정.
updatable	`true`	SQL `UPDATE`에 해당 필드를 포함할지 여부를 지정.
columnDefinition	자동 생성됨	SQL 문에서 사용할 컬럼 정의를 직접 지정. (예: `VARCHAR(255) NOT NULL`)

table	기본 테이블명	필드가 매핑될 테이블 이름을 지정. 다중 테이블 매핑 시 사용.
length	255	`VARCHAR` 타입 컬럼의 최대 길이를 설정.
precision	0	`BigDecimal` 또는 `BigInteger` 타입의 전체 자릿수를 설정.
scale	0	`BigDecimal` 타입의 소수 자릿수를 설정.

@Column 옵션별 설명

1. `name`

   데이터베이스 컬럼 이름이 필드 이름과 다를 경우 지정.

   ```
   java
   코드 복사
   @Column(name = "user_name")
   private String username;
   ```

2. `nullable`

   기본값은 `true`로, 컬럼이 `NULL` 값을 허용할지 여부를 설정.

   ```
   java
   코드 복사
   @Column(nullable = false)
   private String email;
   ```

3. `unique`

   컬럼에 유니크 제약 조건을 추가합니다.

   > 주의: 유니크 인덱스 생성은 데이터베이스 벤더에 따라 다르게 처리될 수 있으므로, 복합 유니크 조건은 @Table의 uniqueConstraints를 권장.

   ```
   java
   코드 복사
   @Column(unique = true)
   private String phoneNumber;
   ```

4. `insertable`

   SQL `INSERT` 문에 해당 필드를 포함할지 여부를 설정.

   > 보통 읽기 전용 컬럼이나 기본값을 데이터베이스에서 처리할 때 사용.

   ```
   java
   코드 복사
   @Column(insertable = false)
   private String createdBy;
   ```

5. `updatable`

   SQL `UPDATE` 문에 해당 필드를 포함할지 여부를 설정.

   > 생성 시 고정된 값을 변경하지 않도록 설정할 때 유용.

   ```
   java
   코드 복사
   ```

```
@Column(updatable = false)
private LocalDateTime createdAt;
```

6. `columnDefinition`

SQL 문에서 사용할 컬럼 정의를 직접 지정.

> JPA가 자동으로 생성하는 타입을 무시하고 사용자 정의 스키마를 제공할 수 있음.

```
java
코드 복사
@Column(columnDefinition = "TEXT NOT NULL")
private String description;
```

7. `table`

필드가 특정 테이블에 매핑될 경우 테이블 이름을 지정.

> 주로 다중 테이블 매핑 시 사용.

```
java
코드 복사
@Column(table = "audit_log")
private String logDetails;
```

8. `length`

문자 타입(`String`) 컬럼의 최대 길이를 설정.

> 기본값은 255로 설정.

```
java
코드 복사
@Column(length = 100)
private String shortDescription;
```

9. `precision`

숫자 타입(`BigDecimal` 또는 `BigInteger`)의 전체 자릿수를 설정.

```
java
코드 복사
@Column(precision = 10, scale = 2)
private BigDecimal price;
```

10. `scale`

숫자 타입(`BigDecimal`)의 소수 자릿수를 설정.

> 예: precision=10, scale=2일 경우 최대 10자리 숫자 중 소수점 아래 2자리까지 저장.

```
java
코드 복사
@Column(precision = 8, scale = 3)
private BigDecimal weight;
```

초급-3 기본 CRUD Repository 사용법

Spring Data JPA는 데이터베이스의 CRUD(Create, Read, Update, Delete) 작업을 간단히 처리할 수 있도록 `CrudRepository` 또는 `JpaRepository` 인터페이스를 제공. 이를 활용하면 별도의 쿼리 작성 없이 기본적인 데이터 작업을 수행할 수 있음.

기본 설정

1. **JpaRepository 인터페이스 상속**
   - 엔티티와 기본 키 타입을 제네릭으로 지정하여 레포지토리를 생성.

```
java
코드 복사
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
}
```

2. **엔티티 클래스와 매핑**
   - 예를 들어, `User` 엔티티 클래스는 다음과 같이 정의.

```
java
코드 복사
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

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

    // Getter, Setter, Constructor
}
```

CRUD 메서드 사용 예시

1. **Create (저장)**
   - 새로운 데이터를 저장.

```
java
코드 복사
@Autowired
private UserRepository userRepository;

public void saveUser() {
    User user = new User();
    user.setName("John Doe");
    user.setEmail("john.doe@example.com");
```

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

2. **Read (조회)**
   - 데이터베이스에서 데이터를 조회.

```
java
코드 복사
public void findUserById(Long id) {
    User user = userRepository.findById(id).orElse(null);
    if (user != null) {
        System.out.println("User Name: " + user.getName());
    }
}

public void findAllUsers() {
    List<User> users = userRepository.findAll();
    users.forEach(user -> System.out.println(user.getName()));
}
```

3. **Update (수정)**
   - 수정 작업은 `save` 메서드를 재사용. JPA는 엔티티가 이미 존재하면 업데이트로 처리.

```
java
코드 복사
public void updateUser(Long id, String newEmail) {
    User user = userRepository.findById(id).orElse(null);
    if (user != null) {
        user.setEmail(newEmail);
        userRepository.save(user);
    }
}
```

4. **Delete (삭제)**
   - 데이터를 삭제할 때는 기본 키를 기준으로 작업.

```
java
코드 복사
public void deleteUser(Long id) {
    userRepository.deleteById(id);
}
```

JpaRepository의 주요 메서드

메서드	설명
`save(Entity entity)`	엔티티 저장 (Create/Update)
`findById(ID id)`	기본 키로 데이터 조회
`findAll()`	모든 데이터 조회
`delete(Entity entity)`	엔티티를 삭제
`deleteById(ID id)`	기본 키로 데이터 삭제
`count()`	저장된 데이터 개수 반환
`existsById(ID id)`	기본 키 존재 여부 확인

이렇게 기본 CRUD 작업을 손쉽게 처리할 수 있어 개발 생산성이 크게 향상됨. 필요 시 메서드 이름 기반 쿼리 작성법을 추가로 사용할 수도 있음.

초급-4 데이터베이스 연결 및 MySQL 사용

Spring Boot에서 JPA를 통해 MySQL 데이터베이스를 사용하려면 적절한 의존성 추가, 데이터베이스 설정, 그리고 간단한 CRUD 작업 구현이 필요.

1. 의존성 추가

`Spring Boot`, `JPA`, `MySQL`을 사용하기 위해 필요한 의존성을 `pom.xml`에 추가.

```
xml
코드 복사
<dependencies>
    <!-- Spring Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- MySQL Driver -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>

    <!-- Spring Boot Starter -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
</dependencies>
```

---

2. MySQL 데이터베이스 설정

`application.properties` 또는 `application.yml` 파일에 MySQL 데이터베이스 연결 설정을 추가.

application.properties

```
properties
코드 복사
spring.datasource.url=jdbc:mysql://localhost:3306/my_database
spring.datasource.username=root
spring.datasource.password=your_password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# JPA 설정
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

주요 설정 설명:

- `spring.datasource.url`: MySQL 데이터베이스 URL.
- `spring.datasource.username`: 데이터베이스 사용자 이름.
- `spring.datasource.password`: 데이터베이스 비밀번호.

- `spring.jpa.hibernate.ddl-auto`: 스키마 자동 생성 옵션 (`update`, `create`, `create-drop`, `none` 등).
- `spring.jpa.show-sql`: 실행되는 SQL을 콘솔에 출력.

---

3. **MySQL 데이터베이스 준비**

MySQL에서 사용할 데이터베이스를 생성.

---

4. **엔티티 생성 및 매핑**

Spring Boot JPA에서 테이블과 매핑될 엔티티 클래스를 생성.

```
java
코드 복사
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

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

    private String name;
    private String email;

    // Getter, Setter, Constructor
    public User() {}
    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
}
```

---

5. **Repository 인터페이스 생성**

CRUD 작업을 수행하기 위해 `JpaRepository` 인터페이스를 생성.

```
java
코드 복사
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
}
```

---

6. **Service 또는 Controller에서 CRUD 작업 구현**

데이터 저장

```
java
코드 복사
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    public void saveUser(String name, String email) {
        User user = new User(name, email);
        userRepository.save(user); // MySQL에 저장
    }
}
```

데이터 조회

```
java
코드 복사
public void findUserById(Long id) {
    userRepository.findById(id).ifPresent(user -> {
        System.out.println("User Name: " + user.getName());
    });
}
```

데이터 삭제

```
java
코드 복사
public void deleteUser(Long id) {
    userRepository.deleteById(id);
}
```

---

7. **실행 및 확인**

- Spring Boot 애플리케이션을 실행.
- MySQL Workbench 또는 CLI에서 `my_database`의 내용을 확인.

결론

Spring Boot와 JPA를 활용하면 복잡한 SQL 쿼리 작성 없이도 MySQL과 같은 관계형 데이터베이스와 쉽게 상호작용할 수 있음. 위의 예제를 통해 간단한 CRUD 작업을 구현하고 Spring Boot JPA의 강력한 기능을 체험할 수 있음.

데이터베이스 스키마(Schema)
데이터베이스에서
테이블과 그 구조를 정의한 명세

스키마는 데이터베이스 설계의 핵심적인 문서로, 다음을 포함함:

- **데이터베이스 구조**: 테이블 이름, 열 이름, 데이터 타입 등.
- **제약 조건**: 각 열의 데이터 제약 조건.
- **관계 정의**: 테이블 간의 관계를 명시.

- **데이터베이스 객체**: 트리거, 함수, 프로시저 등.

⇒ 보통 SQL문서, ERD문서 그리고 각종 설계문서 및 Wiki문서들이 사용됨.

Spring JPA에서 `spring.jpa.hibernate.ddl-auto` 옵션은 Hibernate가 애플리케이션 시작 시 데이터베이스 스키마를 어떻게 처리할지 지정하는 데 사용됨. 이 값은 데이터베이스 테이블과 엔티티 간의 스키마 동기화를 제어.

---

1. `update`

- **의미**: 기존 스키마를 변경 없이 유지하면서 엔티티와 데이터베이스 스키마를 동기화.
  - 엔티티에 추가된 컬럼은 테이블에 자동으로 추가되지만, 기존 컬럼은 삭제되지 않음.
  - 데이터는 유지됨.
- **사용 상황**:
  - 개발 단계에서 엔티티 수정 사항을 기존 데이터베이스에 반영하고 싶을 때.
  - 운영 환경에서는 데이터 손실 위험이 없도록 권장되지 않음.

---

2. `create`

- **의미**: 애플리케이션 실행 시 기존 데이터베이스 스키마를 삭제하고 새로 생성.
  - 테이블이 모두 삭제되고 엔티티 정의에 따라 다시 생성.
  - 기존 데이터는 손실.
- **사용 상황**:
  - 테스트 및 초기 개발 환경에서 데이터베이스를 빈 상태로 시작하고 싶을 때.
  - 운영 환경에서는 사용하지 않음.

---

3. `create-drop`

- **의미**: 애플리케이션 실행 시 스키마를 생성하고, 애플리케이션 종료 시 삭제.
  - 임시 데이터베이스를 생성해 테스트를 반복 수행할 때 유용.
- **사용 상황**:
  - 테스트 환경에서 데이터베이스를 매번 초기화하고 싶을 때.
  - 운영 환경에서는 절대 사용하지 않음.

---

4. `none`

- **의미**: Hibernate가 데이터베이스 스키마에 아무 작업도 수행하지 않음.
  - 애플리케이션은 데이터베이스에 대해 읽기/쓰기를 수행하지만, 테이블 생성이나 수정은 수동으로 처리.
- **사용 상황**:
  - 운영 환경에서 데이터베이스 스키마 변경이 제어되어야 할 때.
  - 데이터베이스 초기화가 필요 없을 때.

추가 옵션 (참고용)

`validate`

- **의미**: 엔티티와 데이터베이스 스키마가 일치하는지 확인만 하고, 수정 작업은 하지 않음.
  - 스키마가 엔티티와 다르면 애플리케이션 실행이 실패.
- **사용 상황**:
  - 스키마가 이미 수동으로 설정되어 있고, 애플리케이션 실행 시 정확성을 보장하고 싶을 때.

정리

옵션	기존 데이터 유지	테이블 생성	테이블 삭제	주요 용도
`update`	O	필요 시 추가	X	개발 단계에서 데이터 유지
`create`	X	O	O	초기 개발 환경
`create-drop`	X	O	O (종료 시)	테스트 환경
`none`	O	X	X	운영 환경 (스키마 유지)
`validate`	O	X	X	운영 환경 (스키마 검증)

권장 사항

1. **개발 환경**
   - 주로 `update` 또는 `create`를 사용하여 개발 속도를 높임.
   - 데이터베이스를 매번 초기화해야 하는 경우 `create-drop`도 사용 가능.
2. **운영 환경**
   - 운영에서는 **절대 `update`, `create`, `create-drop`을 사용하지 않음**.
   - 운영 환경에서는 일반적으로 `none` 또는 `validate`를 사용하여 데이터와 스키마를 안전하게 유지.

이 옵션은 **데이터 손실 위험**을 동반할 수 있으므로 환경에 따라 신중히 선택.

초급-5 Spring Boot DevTools와 JPA 디버깅

Spring Boot DevTools와 JPA 디버깅은 개발 중에 애플리케이션의 효율적인 테스트와 디버깅을 돕는 유용한 도구.

1. Spring Boot DevTools

Spring Boot DevTools는 개발 중에 애플리케이션을 빠르게 리로드하고, 디버깅, 로그 등을 더욱 쉽게 관리할 수 있도록 도와주는 개발 도구. DevTools는 개발 환경에서만 사용되며, 프로덕션 환경에서는 비활성화됨.

Spring Boot DevTools 설치

Spring Boot DevTools는 의존성만 추가하면 됨. `pom.xml` 파일에 아래와 같이 DevTools 의존성을 추가.

```
xml
코드 복사
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>development</scope>  <!-- 개발 환경에서만 활성화 -->
    </dependency>
</dependencies>
```

DevTools의 주요 기능

1. **자동 리로딩 (Automatic Restart)**
   - 애플리케이션 코드나 리소스 파일이 변경될 때 자동으로 애플리케이션을 재시작.
   - 이는 개발 중에 애플리케이션을 수동으로 재시작할 필요 없이 즉시 변경 사항을 반영할 수 있게 함.
2. **LiveReload 지원**
   - `LiveReload`는 웹 애플리케이션에서 HTML, CSS, JavaScript 변경을 감지하고 브라우저를 자동으로 리프레시하여 실시간으로 결과를 확인할 수 있도록 함.
   - 브라우저에 `LiveReload` 플러그인을 설치하면 이 기능을 사용할 수 있음.
3. **디버깅 로그 (Enhanced Logging)**
   - DevTools는 애플리케이션의 로그 레벨을 쉽게 설정할 수 있도록 하며, 특히 개발 중에는 `DEBUG` 레벨의 로그를 출력하여 시스템의 상태를 더 쉽게 추적할 수 있음.

2. JPA 디버깅

JPA를 사용할 때는 SQL 쿼리가 어떻게 실행되는지, 엔티티가 어떻게 매핑되는지, 성능 문제가 있는지 등을 확인할 필요가 있음. JPA 디버깅은 이를 보다 쉽게 추적하고 이해할 수 있게 도와줌.

JPA SQL 로그 활성화

JPA가 실행하는 SQL 쿼리를 콘솔에 출력하려면 `application.properties` 파일에 아래와 같이 설정을 추가함.

```
properties
코드 복사
# JPA 쿼리 출력
spring.jpa.show-sql=true

# SQL 쿼리 포맷을 보기 좋게 출력
spring.jpa.properties.hibernate.format_sql=true

# JPA 쿼리 실행 시 사용하는 파라미터 출력
spring.jpa.properties.hibernate.type=trace
```

쿼리 실행 시간 로깅

JPA에서 실행된 쿼리의 실행 시간을 로깅하려면, `hibernate`의 `statistics` 기능을 사용하여 쿼리 성능을 추적할 수 있음.

```
properties
코드 복사
# Hibernate 통계 활성화
spring.jpa.properties.hibernate.generate_statistics=true
```

위 설정을 활성화한 후, 애플리케이션에서 Hibernate의 통계 정보를 출력할 수 있음. 예를 들어, 실행된 쿼리의 수, 실행된 시간 등을 출력할 수 있음.

JPA 디버깅 도구 사용

1. **Hibernate Profiler**
   - Hibernate Profiler는 Hibernate의 내부 작업을 디버깅하고 성능을 분석할 수 있는 도구. 이 도구를 사용하면 SQL 쿼리, 세션 처리 시간, 캐시 사용 등을 추적할 수 있음.
2. **Spring Boot DevTools와 함께 사용**
   - DevTools와 함께 사용하면 애플리케이션을 자동으로 재시작하고, JPA 쿼리를 확인하며, 로깅된 SQL을 통해 디버깅을 더욱 빠르게 할 수 있음.

JPA 디버깅을 위한 트릭

1. **쿼리 최적화**
   - 실행되는 SQL 쿼리를 디버깅하면서 비효율적인 쿼리나 불필요한 쿼리가 실행되는지 확인. 예를 들어, `N+1` 문제를 해결하기 위해 `@OneToMany` 연관 관계에서 `fetch = FetchType.LAZY`를 사용하거나, `join fetch`를 사용하여 연관된 데이터를 미리 로드할 수 있음.
2. **기본 Fetch 전략 확인**
   - JPA의 기본 Fetch 전략이 `LAZY`인지 `EAGER`인지 확인하고, 이를 변경하여 성능을 최적화할 수 있음. `LAZY` 로딩을 사용할 때는 지연 로딩된 데이터를 호출할 때 발생하는 쿼리 수가 많은지, `EAGER` 로딩을 사용할 때는 불필요한 데이터까지 로드되지 않는지 확인.
3. **트랜잭션 확인**
   - JPA는 트랜잭션 내에서 데이터베이스 작업을 처리. 트랜잭션이 제대로 관리되고 있는지, 필요한 경우에만 커밋되고 있는지 확인.

---

3. Spring Boot DevTools와 JPA 디버깅 결합

Spring Boot DevTools와 JPA 디버깅을 결합하면, 개발 중에 애플리케이션의 상태를 실시간으로 추적하고, SQL 쿼리 실행을 디버깅하며, 애플리케이션의 성능 문제를 쉽게 찾을 수 있음.

실시간 리로딩과 SQL 로그 예시

1. 애플리케이션에서 엔티티 변경 후 DevTools에 의해 자동으로 리로딩이 발생하고, 변경된 사항에 대한 SQL 쿼리가 콘솔에 출력.
2. `application.properties`에서 활성화된 SQL 로깅 설정을 통해 실제 실행된 SQL을 실시간으로 확인.
3. 쿼리 실행 시간 및 쿼리의 수를 `hibernate.generate_statistics` 설정을 통해 확인하고, 성능 이슈를 디버깅.

결론

- **Spring Boot DevTools**는 개발 환경에서 빠르게 리로딩하고, 실시간 디버깅을 돕는 도구로, 개발 과정에서 효율성을 크게 높여줌.
- **JPA 디버깅**은 실행되는 SQL 쿼리, 파라미터, 성능 등을 추적하고 분석하는 데 유용함.
- DevTools와 JPA 디버깅을 결합하면 애플리케이션의 동작을 더욱 빠르게 이해하고, 디버깅하며 성능을 최적화할 수 있음.

초급-6 JPQL (Java Persistence Query Language) 기초

- *JPQL (Java Persistence Query Language)**은 JPA (Java Persistence API)에서 사용하는 객체 지향 쿼리 언어. SQL과 유사하지만, 데이터베이스의 테이블이 아니라 **엔티티 객체**를 대상으로 쿼리를 작성하는 특징이 있음. JPQL을 사용하면 데이터베이스와 객체 간의 매핑을 유지하면서도 효율적인 쿼리 작업을 할 수 있음.

---

1. JPQL의 특징

- **객체 지향 쿼리 언어**: JPQL은 데이터베이스 테이블이 아닌, 엔티티 클래스(자바 객체)를 기준으로 쿼리를 작성. 따라서 `Entity` 객체의 필드를 조회하거나 조작할 수 있음.
- **테이블 기반 쿼리 아님**: JPQL에서 사용하는 것은 실제 테이블 이름이 아니라 **엔티티 클래스 이름임**.
- **쿼리 문법**: SQL과 문법이 유사하지만, SQL과의 차이점은 `SELECT`, `FROM`, `WHERE` 등 주요 SQL 문법 요소가 엔티티 클래스와 매핑된 속성(필드)에 대해 동작한다는 점.

---

2. JPQL 기본 문법

JPQL의 기본 구조는 SQL과 매우 비슷함. 기본적으로 `SELECT`, `FROM`, `WHERE` 절을 사용.

기본 예시: 모든 `User` 엔티티를 조회하는 JPQL 쿼리

```
java
코드 복사
String jpql = "SELECT u FROM User u";
```

위의 JPQL에서 `u`는 `User` 엔티티의 인스턴스를 의미하며, SQL의 테이블 이름 대신 엔티티 클래스 이름을 사용.

---

3. JPQL 쿼리 예시

1) SELECT 쿼리

```
java
코드 복사
// User 엔티티에서 모든 데이터를 조회
String jpql = "SELECT u FROM User u";
List<User> users = entityManager.createQuery(jpql, User.class).getResultList();
```

2) 조건을 추가한 쿼리 (WHERE)

```
java
코드 복사
```

```
// User 엔티티에서 특정 조건을 만족하는 데이터 조회
String jpql = "SELECT u FROM User u WHERE u.age > :age";
List<User> users = entityManager.createQuery(jpql, User.class)
                                .setParameter("age", 30)
                                .getResultList();
```

위 쿼리는 `age`가 30보다 큰 사용자들을 조회합니다. `:age`는 파라미터 바인딩을 의미함.

3) ORDER BY 절

```
java
코드 복사
// User 엔티티에서 나이(age)를 기준으로 내림차순 정렬
String jpql = "SELECT u FROM User u ORDER BY u.age DESC";
List<User> users = entityManager.createQuery(jpql, User.class).getResultList();
```

4) 특정 컬럼만 조회 (프로젝션)

```
java
코드 복사
// User 엔티티에서 name만 조회
String jpql = "SELECT u.name FROM User u";
List<String> names = entityManager.createQuery(jpql, String.class).getResultList();
```

이 경우, 전체 `User` 객체를 조회하는 것이 아니라, `name` 속성만 선택적으로 조회.

5) DISTINCT (중복 제거)

```
java
코드 복사
// 중복된 User의 이름을 제거하고 조회
String jpql = "SELECT DISTINCT u.name FROM User u";
List<String> distinctNames = entityManager.createQuery(jpql, String.class).getResultLi
st();
```

---

4. JPQL에서 사용하는 주요 키워드

1) SELECT

- `SELECT` 절을 사용하여 원하는 속성을 선택합니다. 전체 엔티티를 조회할 수도 있고, 특정 필드만 선택할 수도 있음.

  예시:

  ```
  java
  코드 복사
  SELECT u FROM User u // 전체 User 엔티티 조회
  SELECT u.name FROM User u // User의 name 속성만 조회
  ```

2) WHERE

- `WHERE` 절을 사용하여 조건을 추가할 수 있음.

  예시:

  ```
  java
  코드 복사
  SELECT u FROM User u WHERE u.age > 30
  ```

3) JOIN

- JPQL에서는 `JOIN`을 사용하여 연관된 엔티티를 조회할 수 있습니다. 일반적으로 `JOIN FETCH`를 사용해 `N+1`문제를 방지.

  예시:

  ```
  java
  코드 복사
  SELECT o FROM Order o JOIN FETCH o.customer c WHERE c.name = :name
  ```

- `JOIN FETCH`는 `Order`와 `Customer` 엔티티를 조인하여 한 번에 데이터를 가져오는 방식.

4) GROUP BY

- `GROUP BY`를 사용하여 데이터를 그룹화할 수 있음.

  예시:

  ```
  java
  코드 복사
  SELECT u.age, COUNT(u) FROM User u GROUP BY u.age
  ```

5) HAVING

- `HAVING`은 `GROUP BY`와 함께 사용되며, 그룹화된 데이터에 조건을 추가할 수 있음.

  예시:

  ```
  java
  코드 복사
  SELECT u.age, COUNT(u) FROM User u GROUP BY u.age HAVING COUNT(u) > 5
  ```

6) LIKE

- `LIKE` 연산자를 사용하여 문자열 패턴을 매칭할 수 있음.

  예시:

  ```
  java
  코드 복사
  SELECT u FROM User u WHERE u.name LIKE :name
  ```

---

5. JPQL 쿼리 실행 방법

JPQL을 실행하려면 `EntityManager`를 사용. `EntityManager`는 JPA에서 객체를 관리하는 주요 인터페이스.

쿼리 실행 예시

```
java
코드 복사
String jpql = "SELECT u FROM User u WHERE u.age > :age";
TypedQuery<User> query = entityManager.createQuery(jpql, User.class);
query.setParameter("age", 30);
List<User> users = query.getResultList();
```

- `createQuery`: JPQL 쿼리를 생성.
- `setParameter`: 쿼리에서 사용되는 파라미터를 설정.
- `getResultList`: 쿼리 결과를 리스트로 반환.

`@Query` 어노테이션을 사용하여 **`EntityManager`**를 통해 작성한 JPQL 예제들을 **`@Query`**를 사용하는 방식으로 변경할 수 있음.

1. EntityManager 사용 예제 1: 기본 조회

`EntityManager`를 이용한 예제:

```
java
코드 복사
String jpql = "SELECT u FROM User u WHERE u.name = :name";
List<User> users = entityManager.createQuery(jpql, User.class)
                                .setParameter("name", "Alice")
                                .getResultList();
```

`@Query` 어노테이션을 이용한 예제:

```
java
코드 복사
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("SELECT u FROM User u WHERE u.name = :name")
    List<User> findByName(@Param("name") String name);
}
```

2. EntityManager 사용 예제 2: 여러 조건으로 조회

`EntityManager`를 이용한 예제:

```
java
코드 복사
String jpql = "SELECT u FROM User u WHERE u.name = :name AND u.age = :age";
List<User> users = entityManager.createQuery(jpql, User.class)
                                .setParameter("name", "Alice")
                                .setParameter("age", 30)
                                .getResultList();
```

`@Query` 어노테이션을 이용한 예제:

```
java
코드 복사
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("SELECT u FROM User u WHERE u.name = :name AND u.age = :age")
    List<User> findByNameAndAge(@Param("name") String name, @Param("age") int age);
}
```

3. EntityManager 사용 예제 3: 범위 검색

`EntityManager`를 이용한 예제:

```
java
코드 복사
String jpql = "SELECT u FROM User u WHERE u.age BETWEEN :startAge AND :endAge";
List<User> users = entityManager.createQuery(jpql, User.class)
                                .setParameter("startAge", 20)
                                .setParameter("endAge", 30)
                                .getResultList();
```

`@Query` 어노테이션을 이용한 예제:

```
java
코드 복사
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("SELECT u FROM User u WHERE u.age BETWEEN :startAge AND :endAge")
    List<User> findByAgeBetween(@Param("startAge") int startAge, @Param("endAge") int
endAge);
}
```

---

4. EntityManager 사용 예제 4: 정렬

`EntityManager`**를 이용한 예제:**

```
java
코드 복사
String jpql = "SELECT u FROM User u WHERE u.age > :age ORDER BY u.name ASC";
List<User> users = entityManager.createQuery(jpql, User.class)
                                .setParameter("age", 20)
                                .getResultList();
```

`@Query` **어노테이션을 이용한 예제:**

```
java
코드 복사
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("SELECT u FROM User u WHERE u.age > :age ORDER BY u.name ASC")
    List<User> findByAgeGreaterThanOrderByNameAsc(@Param("age") int age);
}
```

---

5. EntityManager 사용 예제 5: 조인

`EntityManager`**를 이용한 예제:**

```
java
코드 복사
String jpql = "SELECT u FROM User u JOIN u.orders o WHERE o.status = :status";
List<User> users = entityManager.createQuery(jpql, User.class)
                                .setParameter("status", "DELIVERED")
                                .getResultList();
```

`@Query` **어노테이션을 이용한 예제:**

```
java
코드 복사
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("SELECT u FROM User u JOIN u.orders o WHERE o.status = :status")
    List<User> findByOrderStatus(@Param("status") String status);
}
```

결론

위 예제들처럼 `@Query` 어노테이션을 사용하면 **`EntityManager`**를 사용하여 직접 **JPQL 쿼리**를 작성하는 대신, **리포지토리 인터페이스**에서 **간단하게 쿼리 메서드를 작성**할 수 있음. 이렇게 하면 **쿼리 메서드**와 **매개변수**를 명확히 정의할 수 있어 코드가 간결하고

유지보수하기 쉬워짐.

Spring Data JPA에서는 **메서드 이름만으로 쿼리를 자동으로 생성**하는 기능을 제공함. 이를 통해, **복잡한 JPQL 쿼리**를 직접 작성하지 않고도, 메서드 이름에 따라 자동으로 쿼리를 생성할 수 있음. 이는 **쿼리 메서드(Query Method)**라고도 불림.

초급-7 Spring Data JPA 메서드 쿼리 작성법

1. **기본 규칙**

- 메서드 이름에 포함된 키워드를 통해 **자동으로 쿼리를 생성**.
- 메서드 이름은 **명확하고 일관된 규칙**을 따라야 하며, **조건에 맞는 데이터 조회**를 위한 **쿼리를 자동으로 생성**.
- 메서드 이름에 포함된 **속성 이름**과 **연산자**(예: `And`, `Or`, `Between`) 등을 통해 쿼리를 자동으로 만듦.

2. **기본 규칙 예시**

1. `findBy`: `findBy` 뒤에 **속성 이름**을 적어주면, 해당 속성에 대한 **조건**으로 쿼리가 생성.

```
java
코드 복사
List<User> findByName(String name);
```

   - **쿼리**: `SELECT u FROM User u WHERE u.name = :name`

2. `findBy` + **여러 조건**: 여러 조건을 추가하려면 `And`, `Or` 등을 사용하여 **복합 조건**을 설정할 수 있음.

```
java
코드 복사
List<User> findByNameAndAge(String name, int age);
```

   - **쿼리**: `SELECT u FROM User u WHERE u.name = :name AND u.age = :age`

3. `findBy` + `Not`: **부정 조건**을 나타낼 때는 `Not`을 사용.

```
java
코드 복사
List<User> findByNameNot(String name);
```

   - **쿼리**: `SELECT u FROM User u WHERE u.name != :name`

4. `findBy` + `Like`: **부분 일치** 검색을 할 때는 `Like`를 사용합니다. `Like`는 **SQL의** `LIKE` **연산자**와 동일하게 작동.

```
java
코드 복사
List<User> findByNameLike(String name);
```

   - **쿼리**: `SELECT u FROM User u WHERE u.name LIKE :name`

5. `findBy` + `GreaterThan`, `LessThan`: **숫자**나 **날짜**와 같은 비교 연산자에 사용.

```
java
코드 복사
List<User> findByAgeGreaterThan(int age);
```

   - **쿼리**: `SELECT u FROM User u WHERE u.age > :age`

6. `findBy` + `Between`: **범위 검색**을 할 때는 `Between`을 사용.

```
java
코드 복사
List<User> findByAgeBetween(int startAge, int endAge);
```

- **쿼리:** `SELECT u FROM User u WHERE u.age BETWEEN :startAge AND :endAge`

7. `findBy` + `OrderBy`: **정렬**을 할 때는 `OrderBy`를 사용.

```
java
코드 복사
List<User> findByAgeGreaterThanOrderByNameAsc(int age);
```

- **쿼리:** `SELECT u FROM User u WHERE u.age > :age ORDER BY u.name ASC`

---

3. **메서드 쿼리 예시**

예시 1: 기본 조회

```
java
코드 복사
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByName(String name);  // name으로 조회
}
```

- **쿼리:** `SELECT u FROM User u WHERE u.name = :name`

예시 2: 여러 조건으로 조회

```
java
코드 복사
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByNameAndAge(String name, int age);  // name과 age로 조회
}
```

- **쿼리:** `SELECT u FROM User u WHERE u.name = :name AND u.age = :age`

예시 3: 범위 검색

```
java
코드 복사
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByAgeBetween(int startAge, int endAge);  // 나이 범위로 조회
}
```

- **쿼리:** `SELECT u FROM User u WHERE u.age BETWEEN :startAge AND :endAge`

예시 4: 정렬

```
java
코드 복사
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByAgeGreaterThanOrderByNameAsc(int age);  // 나이가 특정 값보다 큰 사용
자들을 이름순으로 정렬
}
```

- **쿼리:** `SELECT u FROM User u WHERE u.age > :age ORDER BY u.name ASC`

---

4. `@Query` **어노테이션과의 차이점**

- **메서드 쿼리**는 **간단한 조회** 및 **조건 검색**에 적합.
- 복잡한 쿼리나 **`JOIN`*이 필요한 경우, `@Query` 어노테이션을 사용하여 **JPQL** 쿼리를 직접 작성하는 것이 좋음.

예시: `@Query` **사용**

```
java
코드 복사
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("SELECT u FROM User u JOIN u.orders o WHERE o.status = :status")
    List<User> findByOrderStatus(@Param("status") String status);
}
```

- **쿼리:** `SELECT u FROM User u JOIN u.orders o WHERE o.status = :status`

결론

- **메서드 쿼리**는 Spring Data JPA에서 쿼리를 작성하는 편리한 방법으로, 메서드 이름을 통해 **자동으로 쿼리를 생성**.
- 기본적인 조회, 조건 검색, 범위 검색, 정렬 등 대부분의 **단순한 쿼리**를 메서드 이름으로 쉽게 처리할 수 있음.
- 복잡한 쿼리나 특정 조건을 사용하는 쿼리는 `@Query` 어노테이션을 사용해 **JPQL 쿼리**를 직접 작성하는 방식이 필요할 수 있음.

초급-8 기본 필드 매핑

1. `@Entity`

- **목적**: 이 클래스가 **JPA 엔티티**임을 선언.
- **설명**: `@Entity` 어노테이션은 **JPA**에 의해 관리되는 엔티티 클래스를 정의할 때 사용. 이 어노테이션이 붙은 클래스는 데이터베이스 테이블에 매핑됨.
- **사용 예시**:

```
java
코드 복사
@Entity
public class User {
    @Id
    private Long id;
    private String name;
}
```

2. `@Table`

- **목적**: **엔티티 클래스**와 **데이터베이스 테이블**의 매핑을 설정.
- **설명**: `@Table` 어노테이션을 사용하면 엔티티가 매핑되는 테이블의 이름이나 설정을 정의할 수 있음. 기본적으로 엔티티 클래스 이름이 테이블 이름으로 사용.
- **주요 속성**:
  - `name`: 테이블의 이름을 지정.
  - `schema`: 테이블이 속한 스키마를 지정.
  - `catalog`: 테이블이 속한 카탈로그를 지정.
- **사용 예시**:

```
java
코드 복사
@Entity
@Table(name = "user_table")
public class User {
    @Id
    private Long id;
    private String name;
}
```

3. `@Id`

- **목적**: 해당 필드를 **기본 키**로 지정.
- **설명**: `@Id` 어노테이션을 사용하여 해당 필드를 **기본 키**로 설정. 기본 키는 데이터베이스에서 유일한 값을 가져야 하며, 각 엔티티 인스턴스를 고유하게 식별할 수 있음.

```
java
코드 복사
@Entity
public class User {
    @Id
    private Long id;
    private String name;
}
```

4. `@GeneratedValue`

- **목적**: 기본 키의 값이 **자동으로 생성**되도록 설정.
- **설명**: `@GeneratedValue`는 **기본 키 생성 전략**을 지정. 예를 들어, `GenerationType.IDENTITY`는 **자동 증가** 방식을 사용하고, `GenerationType.SEQUENCE`는 **시퀀스**를 사용하여 값을 생성.

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

5. `@Column`

- **목적**: 엔티티 필드와 **데이터베이스 컬럼**을 매핑.
- **설명**: `@Column`은 필드와 테이블 컬럼 간의 매핑을 설정. 기본적으로 필드명과 컬럼명이 동일하게 매핑되지만, 이 어노테이션을 사용하여 컬럼명을 다르게 설정하거나, 컬럼의 속성들을 지정할 수 있음.
- **주요 속성**:
  - `name`: 컬럼 이름을 지정.
  - `nullable`: NULL 허용 여부를 설정.
  - `length`: 문자열 타입 컬럼의 길이를 설정.
  - `unique`: 컬럼 값의 유일성을 지정.

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

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

6. `@ManyToOne`, `@OneToMany`, `@OneToOne`, `@ManyToMany`

- **목적**: **엔티티 간 관계**를 설정.
- **설명**: JPA에서는 다양한 관계를 정의할 수 있음. 이를 위해 **연관 관계**를 설정하는 어노테이션들이 사용됨.

@ManyToOne

- **목적**: **다대일** 관계를 설정.
- **사용 예시**: 여러 명의 **`User`*가 하나의 **`Team`*에 속할 수 있을 때.

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

    @ManyToOne
    @JoinColumn(name = "team_id")
    private Team team;
}
```

@OneToMany

- **목적**: **일대다** 관계를 설정.
- **사용 예시**: 하나의 **`Team`*이 여러 명의 **`User`*를 가질 수 있을 때.

```
java
코드 복사
@Entity
public class Team {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToMany(mappedBy = "team")
    private List<User> users;
}
```

@OneToOne

- **목적**: **일대일** 관계를 설정.
- **사용 예시**: 각 **`User`*가 하나의 **`Profile`*을 가질 때.

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

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

@ManyToMany

- **목적**: **다대다** 관계를 설정.
- **사용 예시**: 여러 명의 **`User`*가 여러 개의 **`Course`*를 수강할 수 있을 때.

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

    @ManyToMany
    @JoinTable(
        name = "user_course",
        joinColumns = @JoinColumn(name = "user_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private List<Course> courses;
}
```

7. `@JoinColumn`

- **목적**: **연관 관계**에서 **외래 키**를 설정.
- **설명**: `@JoinColumn` 어노테이션은 **연관 관계**에서 외래 키를 정의할 때 사용. `@ManyToOne`이나 `@OneToOne` 관계에서 외래 키를 지정할 때 사용.

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

    @ManyToOne
    @JoinColumn(name = "team_id", referencedColumnName = "id")
    private Team team;
}
```

8. `@Transient`

- **목적**: 데이터베이스 테이블에 매핑되지 않는 필드를 지정.
- **설명**: `@Transient` 어노테이션을 사용하면 해당 필드가 **JPA 엔티티와 테이블 간 매핑에서 제외**되며, 이 필드는 데이터베이스에 저장되지 않음.

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

    @Transient
    private String temporaryField;  // 데이터베이스에 저장되지 않음
}
```

9. `@Embeddable` **와** `@Embedded`

- **목적**: **복합값 타입**을 재사용할 수 있도록 설정.
- **설명**: `@Embeddable`은 **복합값 타입** 클래스를 정의하고, `@Embedded`는 이를 엔티티에서 사용할 때 사용.

```
java
코드 복사
@Embeddable
public class Address {
    private String city;
    private String street;
}

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

    @Embedded
    private Address address;
}
```

10. `@Version`

- **목적**: **버전 관리**를 설정하여 **낙관적 락(Optimistic Locking)**을 사용.
- **설명**: `@Version` 어노테이션은 데이터의 **버전**을 관리하여 여러 트랜잭션이 동시에 데이터를 수정할 때 **경합**을 방지하는 데 사용.

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

    @Version
    private Long version;
```

```
    private String name;
}
```

`@OneToOne`과 `@ManyToOne` 관계에서 **단방향**과 **양방향**에 따른 사용법을 정리하면 다음과 같음:

1. `@OneToOne` 관계

- *`@OneToOne`*은 두 엔티티가 **1:1** 관계일 때 사용. 이 관계는 **단방향**과 **양방향**으로 설정할 수 있음.

(1) 양방향 관계에서 `@OneToOne` 사용법

- 양방향 관계에서는 **양쪽 엔티티에 모두 어노테이션을 선언**하고, **한쪽에서는 `@JoinColumn`을 사용하여 외래 키**를 관리하며, 다른 한쪽에서는 **`mappedBy`*를 사용하여 **외래 키 관리 주체**를 지정함.

예시 - `@OneToOne` 양방향 관계

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

    @OneToOne
    @JoinColumn(name = "address_id")  // 외래 키 관리
    private Address address;
}
```

```
java
코드 복사
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne(mappedBy = "address")  // "address" 필드를 관리하는 엔티티(User)를 지정
    private User user;
}
```

- `User` 엔티티는 **`Address`*를 참조하고, **`Address`*는 **`User`*를 참조.
- *`@JoinColumn`*은 `User` 엔티티에서 외래 키를 관리하고, **`mappedBy`*는 **`Address`*에서 외래 키 관리 책임이 **`User`*에 있음을 나타냄.

(2) 단방향 관계에서 `@OneToOne` 사용법

- 단방향 관계에서는 **한쪽에서만 `@OneToOne`과 `@JoinColumn`*을 사용하고, 다른 한쪽은 관계를 정의하지 않음.

예시 - `@OneToOne` 단방향 관계

```
java
코드 복사
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;

    @OneToOne
    @JoinColumn(name = "address_id")  // 외래 키 관리
    private Address address;
}
```

```
java
코드 복사
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // "Address" 쪽에는 아무 어노테이션도 선언하지 않음
}
```

- `User` 엔티티는 **`Address`*를 참조하고, **`Address`*는 **`User`*를 참조하지 않습니다. 단방향 관계만 설정됨.

---

2. `@ManyToOne` 관계

- *`@ManyToOne`*은 **다대일(Many-to-One)** 관계에서 사용. 여러 엔티티가 하나의 엔티티를 참조하는 경우. 이 관계 역시 **단방향**과 **양방향**으로 설정할 수 있음.

(1) 양방향 관계에서 `@ManyToOne` 사용법

- 양방향 관계에서는 **한쪽에서 `@ManyToOne`*을 사용하고, **반대쪽에서 `@OneToMany`*를 사용. **`@OneToMany`*는 `mappedBy` 속성을 사용하여 외래 키 관리 주체를 지정.

예시 - `@ManyToOne` 양방향 관계

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

    @ManyToOne
    @JoinColumn(name = "team_id")  // 외래 키 관리
    private Team team;
}
```

```
java
코드 복사
@Entity
public class Team {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToMany(mappedBy = "team")  // "team" 필드를 관리하는 엔티티(User)를 지정
    private List<User> users;
}
```

- *`User`*는 여러 **`Team`*에 속할 수 있고, **`Team`*은 여러 **`User`*를 가질 수 있는 **양방향 관계**.
- *`@ManyToOne`*은 **User** 엔티티에서 **Team**을 참조하고, **`@OneToMany(mappedBy = "team")`*는 **Team** 엔티티에서 **User** 엔티티와의 관계를 설정.

(2) 단방향 관계에서 `@ManyToOne` 사용법

- 단방향 관계에서는 **한쪽에서만 `@ManyToOne`*을 사용하고, 다른 쪽에서는 관계를 정의하지 않음.

예시 - `@ManyToOne` 단방향 관계

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

    @ManyToOne
    @JoinColumn(name = "team_id")  // 외래 키 관리
    private Team team;
}
```

```
java
코드 복사
@Entity
public class Team {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // "Team" 쪽에는 아무 어노테이션도 선언하지 않음
}
```

- `User` 엔티티는 **`Team`*을 참조하고, **`Team`*은 **`User`*를 참조하지 않게 되어 **단방향 관계**만 설정.

정리

관계 유형	단방향 관계 사용법	양방향 관계 사용법
`@OneToOne`	한쪽에만 `@OneToOne` 과 `@JoinColumn` 사용	양쪽에 `@OneToOne`, 한쪽에 `@JoinColumn`, 다른 한쪽에 `mappedBy` 사용
`@ManyToOne`	한쪽에만 `@ManyToOne` 과 `@JoinColumn` 사용	`@ManyToOne`은 "다"쪽에, `@OneToMany`는 "일"쪽에 사용, `mappedBy`로 외래 키 관리 주체 지정

- **단방향 관계**는 **한쪽에만 어노테이션을 선언**하여 관계를 설정하고, **양방향 관계**는 **양쪽에 어노테이션을 선언**하여 상호 참조가 가능하도록 설정.

다대다 관계에서의 `@JoinTable` 사용

- **다대다 관계**는 기본적으로 **두 개의 테이블**을 **중간 테이블**을 통해 연결. 이때 `@JoinTable` 어노테이션을 사용하여 연결 테이블을 정의하고, **`@JoinColumn`*으로 외래 키 컬럼을 설정.
- **조인 테이블**은 실제로 **새로운 테이블**로, 두 엔티티의 **기본 키**들을 외래 키로 가지고 있음.

예시 - 다대다 관계에서 `@JoinTable` 사용법

1. 양방향 관계

다대다 관계에서 **양방향 관계**를 설정할 때, `@ManyToMany` 어노테이션을 양쪽 엔티티에 선언하고, **`@JoinTable`**로 조인 테이블을 설정. **`mappedBy`**는 **연결 관계의 주체**가 되는 엔티티를 지정.

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

    @ManyToMany
    @JoinTable(
        name = "student_course", // 조인 테이블의 이름
        joinColumns = @JoinColumn(name = "student_id"), // student 테이블의 외래 키
        inverseJoinColumns = @JoinColumn(name = "course_id") // course 테이블의 외래 키
    )
    private List<Course> courses;
}
```

```
java
코드 복사
@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToMany(mappedBy = "courses") // "courses" 필드를 관리하는 엔티티(Student) 지정
    private List<Student> students;
}
```

- *`@JoinTable`*은 두 테이블(`student`와 `course`)을 연결하는 중간 테이블인 **`student_course`*를 만듦.
- *`joinColumns`*와 **`inverseJoinColumns`*는 각각 **`Student`*와 **`Course`*의 외래 키 컬럼을 정의.

2. 단방향 관계

단방향 다대다 관계에서도 **`@ManyToMany`**와 **`@JoinTable`**을 사용하여 조인 테이블을 정의. 이때 **`mappedBy`**는 생략되며, **한쪽에서만 관계를 정의**.

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

    @ManyToMany
    @JoinTable(
        name = "student_course", // 조인 테이블의 이름
        joinColumns = @JoinColumn(name = "student_id"), // 외래 키 설정
        inverseJoinColumns = @JoinColumn(name = "course_id") // 외래 키 설정
    )
```

```
    private List<Course> courses;
}
```

```
java
코드 복사
@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // "Course" 엔티티는 단방향 관계이므로 다른 쪽에는 어노테이션 생략
}
```

- **조인 테이블**이 **`student_course`*로 설정되어 **`Student`*와 **`Course`** 테이블을 연결.

조인 테이블이 만들어지는 방식

- **조인 테이블**은 두 엔티티의 **기본 키**를 외래 키로 사용하여 생성. 이 테이블은 두 엔티티 간의 관계를 관리하는 역할을 하며, 실제 데이터베이스 테이블에 존재하게 됨.
- 예를 들어, 위의 예제에서 **`student_course`** 테이블은 다음과 같은 형태로 생성:

```
sql
코드 복사
CREATE TABLE student_course (
    student_id BIGINT NOT NULL,
    course_id BIGINT NOT NULL,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES student(id),
    FOREIGN KEY (course_id) REFERENCES course(id)
);
```

- **`student_course`** 테이블은 **`student_id`*와 **`course_id`** 컬럼을 가지고 있으며, 이는 각각 **`Student`*와 **`Course`** 엔티티의 기본 키임.
- *`PRIMARY KEY`*는 **`student_id`*와 **`course_id`*를 묶어서 설정함. 이를 통해 두 엔티티 간의 관계가 **다대다**임을 나타냄.

정리

- **다대다 관계**에서 **조인 컬럼(`@JoinColumn`)** 대신 **조인 테이블(`@JoinTable`)**을 사용.
- *`@JoinTable`*을 사용하여 **중간 테이블**을 정의하고, 그 테이블에서 두 엔티티의 기본 키를 외래 키로 연결.
- **조인 테이블**은 실제 데이터베이스에서 **가상의 테이블**로 생성되어 두 엔티티의 관계를 관리.

JPA 순환참조의 문제

JPA에서 순환참조 문제는 주로 양방향 연관관계를 설정할 때 발생. 이는 **`Entity A`가 `Entity B`를 참조하고, 동시에 `Entity B`도 다시 `Entity A`를 참조하는 구조**에서 나타날 수 있음. 이런 상황은 특히 **JSON 직렬화 과정**에서 문제가 될 수 있음.

---

1. 순환참조 문제의 원인

- **양방향 연관관계**

  JPA에서 양방향 연관관계를 사용하면, 한 객체가 다른 객체를 참조하고, 그 객체가 다시 원래 객체를 참조하는 형태가 됨.

  ```
  java
  코드 복사
  ```

```
@Entity
public class Parent {
    @Id
    @GeneratedValue
    private Long id;

    @OneToMany(mappedBy = "parent")
    private List<Child> children = new ArrayList<>();
}

@Entity
public class Child {
    @Id
    @GeneratedValue
    private Long id;

    @ManyToOne
    @JoinColumn(name = "parent_id")
    private Parent parent;
}
```

- `Parent`는 `children` 필드로 `Child`를 참조.
- `Child`는 `parent` 필드로 `Parent`를 참조.

- **JSON 직렬화 라이브러리**

  `Jackson`과 같은 JSON 직렬화 라이브러리는 객체를 직렬화할 때 참조를 따라가며 데이터를 생성.

  이때 양방향 관계를 가진 엔티티를 직렬화하면, 무한 루프가 발생할 수 있음.

```
json
코드 복사
{
    "id": 1,
    "children": [
        {
            "id": 2,
            "parent": {
                "id": 1,
                "children": [ ... ] // 무한 반복
            }
        }
    ]
}
```

---

2. 순환참조 문제 해결 방법

(1) `@JsonIgnore` 사용

한쪽 방향의 필드에 `@JsonIgnore`를 적용하여 직렬화에서 제외.

```
java
코드 복사
@Entity
public class Child {
    @Id
    @GeneratedValue
```

```
    private Long id;

    @ManyToOne
    @JoinColumn(name = "parent_id")
    @JsonIgnore // parent 필드를 직렬화하지 않음
    private Parent parent;
}
```

- **장점**: 간단히 적용 가능.
- **단점**: 특정 상황에서 데이터를 잃을 수 있음.

---

(2) `@JsonManagedReference`와 `@JsonBackReference` 사용

Jackson에서 제공하는 어노테이션으로, 한 방향을 직렬화하고 반대 방향은 직렬화하지 않도록 설정.

```
java
코드 복사
@Entity
public class Parent {
    @Id
    @GeneratedValue
    private Long id;

    @OneToMany(mappedBy = "parent")
    @JsonManagedReference // 직렬화 허용
    private List<Child> children = new ArrayList<>();
}

@Entity
public class Child {
    @Id
    @GeneratedValue
    private Long id;

    @ManyToOne
    @JoinColumn(name = "parent_id")
    @JsonBackReference // 직렬화 제외
    private Parent parent;
}
```

- **장점**: 순환참조를 명확히 처리.
- **단점**: Jackson에 종속적.

---

(3) DTO 사용

직접적으로 엔티티를 반환하지 않고, **DTO(Data Transfer Object)**를 사용해 필요한 데이터만 반환.

```
java
코드 복사
public class ParentDTO {
    private Long id;
    private List<ChildDTO> children;
}

public class ChildDTO {
```

```
    private Long id;
}
```

- **장점**: 엔티티 구조를 숨기며 API 설계가 깔끔해짐.
- **단점**: 추가적인 코딩 작업 필요.

---

(4) Hibernate의 `@Fetch` 전략 조정

양방향 관계에서 FetchType을 `LAZY`로 설정하여, 필요한 시점에만 데이터를 로드하도록 설정.

```
java
코드 복사
@OneToMany(mappedBy = "parent", fetch = FetchType.LAZY)
private List<Child> children = new ArrayList<>();
```

- **장점**: 성능 최적화.
- **단점**: LazyInitializationException 주의 필요.

---

3. 순환참조를 방지하기 위한 권장 사항

- **필요하지 않다면 양방향 연관관계를 피함 -** 단방향 연관관계로 설계하는 것이 더 단순하고 안전할 수 있음
- **서비스 계층에서 데이터를 가공.**엔티티를 직접 반환하기보다 DTO를 생성하여 반환하는 방식을 권장함.
- **직렬화 설정을 명확히 함.** `@JsonIgnore`나 `@JsonManagedReference`를 적절히 사용.

중급

1. **Lazy와 Eager 로딩의 이해 및 활용**
   - 관계 데이터를 지연 로딩(Lazy Loading)과 즉시 로딩(Eager Loading)으로 가져오는 방법 및 장단점.
2. **다양한 관계 매핑 (ManyToOne, ManyToMany)**
   - @ManyToOne, @ManyToMany 어노테이션을 활용해 복잡한 관계를 매핑.
3. **복합 키 매핑 (EmbeddedId, IdClass)**
   - 여러 필드가 기본 키를 구성하는 경우 이를 매핑하는 방법.
4. **NamedQuery와 NativeQuery 사용**
   - NamedQuery로 재사용 가능한 JPQL 정의, NativeQuery로 SQL 직접 실행.
5. **Entity Life Cycle (Persist, Merge, Remove, Detach)**
   - JPA의 엔티티 상태와 이를 제어하는 방법(영속, 비영속, 준영속 상태).
6. **Cascade와 OrphanRemoval의 활용**
   - 부모-자식 관계에서의 연속적인 작업 처리와 고아 객체 제거 설정.
7. **DTO를 사용한 데이터 전환 및 최적화**
   - 엔티티 대신 DTO(Data Transfer Object)를 이용해 필요한 데이터만 전송하고 성능 최적화.
8. **QueryDSL을 이용한 동적 쿼리 작성**
   - QueryDSL 라이브러리를 활용해 복잡한 조건의 동적 쿼리 작성.
9. **트랜잭션 관리 및 @Transactional**
   - 트랜잭션의 시작과 종료를 제어하고 @Transactional로 처리 범위 지정.
10. **페이징 및 정렬 기능 활용**

- Spring Data JPA의 Pageable, Sort 객체를 활용해 페이징 처리 및 정렬 구현.

11. **N+1 문제 이해와 해결 (Fetch Join)**
    - 다중 로딩으로 인한 성능 문제를 파악하고 Fetch Join을 통해 해결.
12. **Custom Repository 구현**
    - JPA Repository에 기본 메서드 외 추가 메서드를 정의하고 구현.
13. **EntityGraph 사용법**
    - @EntityGraph를 활용해 특정 관계를 Fetch Join 없이 명시적으로 로딩.
14. **JPA와 DB 성능 튜닝 기법 (Query Plan 분석 등)**
    - 쿼리 성능을 개선하기 위해 인덱스, Query Plan 분석, Batch Insert 등을 활용.

중급-1 Lazy와 Eager 로딩의 이해 및 활용

Lazy와 **Eager** 로딩은 JPA에서 연관된 엔티티를 조회할 때 데이터베드에 어떻게 접근할지를 결정하는 방식입니다. 이를 이해하는 것은 성능 최적화 및 올바른 데이터 조회를 위한 중요한 개념입니다.

1. Eager Loading (즉시 로딩)

Eager loading은 연관된 엔티티를 **즉시** 로딩하는 방식입니다. 즉, **주 엔티티**를 조회할 때 **연관된 엔티티**도 함께 즉시 조회됩니다. 이 방식은 주로 필요한 데이터를 한번에 모두 가져오고자 할 때 사용됩니다.

특징

- 연관된 엔티티를 **즉시 로딩**하여 **한 번의 쿼리로 모든 데이터를** 조회합니다.
- 기본적으로 **연관된 엔티티를 항상 가져옵니다**.
- **N+1 쿼리 문제**를 발생시킬 수 있기 때문에 주의가 필요합니다.

예시 (Eager Loading)

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

    @OneToMany(fetch = FetchType.EAGER) // Eager 로딩 설정
    private List<Book> books;

    // getters and setters
}
```

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

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

```
    // getters and setters
}
```

- `Author` 엔티티를 조회할 때 연관된 `Book` 엔티티도 즉시 로딩됩니다. 이 경우, `Book` 엔티티는 `Author` 엔티티와 함께 **즉시 로딩**됩니다.

장점

- 연관된 데이터를 **즉시** 가져오므로 **데이터가 즉시 필요할 때** 유용합니다.

단점

- **N+1 문제**가 발생할 수 있습니다. 예를 들어, 하나의 `Author` 엔티티를 조회할 때 여러 `Book` 엔티티를 조회하는데, `Author`마다 추가 쿼리가 실행되므로 **성능 저하**를 일으킬 수 있습니다.

---

2. Lazy Loading (지연 로딩)

Lazy loading은 연관된 엔티티를 **필요할 때만 로딩**하는 방식입니다. 즉, **주 엔티티**만 먼저 조회하고, **연관된 엔티티**는 실제로 사용될 때만 **지연 로딩**됩니다.

특징

- 연관된 엔티티를 **필요할 때만** 로딩하므로 **성능 최적화**에 유리합니다.
- **프록시 객체**를 사용하여 실제로 데이터를 가져오기 전에 연관된 엔티티를 **로딩하지 않습니다**.
- **N+1 문제**를 피할 수 있지만, 연관된 엔티티를 사용하기 전에 **세션이 닫혀 있으면 LazyInitializationException**이 발생할 수 있습니다.

예시 (Lazy Loading)

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

    @OneToMany(fetch = FetchType.LAZY) // Lazy 로딩 설정
    private List<Book> books;

    // getters and setters
}
```

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

    @ManyToOne(fetch = FetchType.LAZY) // Lazy 로딩 설정
    private Author author;

    // getters and setters
```

```
}
```

- `Author` 엔티티를 조회할 때, 연관된 `Book` 엔티티는 **지연 로딩**되어 실제로 사용될 때만 데이터가 조회됩니다.

장점

- 연관된 엔티티가 **필요할 때만 로딩**되므로, 불필요한 데이터 조회를 피할 수 있습니다.
- 성능 **최적화**가 가능합니다.

단점

- **LazyInitializationException**이 발생할 수 있습니다. 이는 엔티티가 **세션 밖에서 접근될 때** 발생하는 문제로, 세션이 닫힌 상태에서 연관된 데이터를 조회하려 할 때 발생합니다.

---

3. Lazy와 Eager 로딩 설정 방법

FetchType을 사용하여 **Lazy** 또는 **Eager** 로딩 방식을 설정할 수 있습니다.

- `@OneToMany`, `@ManyToOne`, `@OneToOne`, **`@ManyToMany`*에 `fetch` 속성으로 `FetchType.LAZY` 또는 **`FetchType.EAGER`*를 설정합니다.

기본값

- **OneToMany, ManyToMany**: 기본적으로 **`FetchType.LAZY`*가 적용됩니다.
- **ManyToOne, OneToOne**: 기본적으로 **`FetchType.EAGER`*가 적용됩니다.

`@OneToMany`와 `@ManyToMany` 어노테이션의 경우, 여러 엔티티가 연관될 수 있기 때문에 성능상의 이유로 기본값이 지연 로딩(Lazy Loading)으로 설정되어 있습니다. 반면, `@ManyToOne`과 `@OneToOne`은 연관된 엔티티가 하나이기 때문에 즉시 로딩(Eager Loading)이 기본값입니다.

Eager 로딩

```
java
코드 복사
@OneToMany(fetch = FetchType.EAGER)
```

Lazy 로딩

```
java
코드 복사
@OneToMany(fetch = FetchType.LAZY)
```

4. N+1 문제

Eager 로딩을 사용하면 연관된 데이터를 한번에 로딩하려고 할 수 있습니다. 그러나 많은 연관 데이터가 있을 경우, **여러 번의 쿼리가 실행되며 성능 문제가 발생할 수 있습니다**.

예시 (N+1 문제):

```
java
코드 복사
List<Author> authors = authorRepository.findAll();
for (Author author : authors) {
    System.out.println(author.getBooks());  // 여러 번의 쿼리가 실행될 수 있습니다.
}
```

위 코드에서는 `authors`를 조회할 때, **각각의** `author` **마다** `books` **를 조회**하게 되어 **N+1 문제**가 발생합니다.

이 문제를 해결하려면 **JPQL**을 사용하거나 `@Query` 어노테이션을 사용하여 ** `JOIN FETCH` **를 통해 **모든 데이터를 한 번에 로딩**하도록 할 수 있습니다.

N+1 문제 해결법 (JOIN FETCH 사용)

```
java
코드 복사
@Query("SELECT a FROM Author a JOIN FETCH a.books")
List<Author> findAllAuthorsWithBooks();
```

이 쿼리는 `Author` **와** `Book` **을 조인하여** 한 번의 쿼리로 **모든 데이터를 조회**합니다.

⇒ 지연로딩은 N+1 문제를 완벽히 해결하는 방법이 아님!!

지연 로딩을 사용하더라도 N+1 문제가 발생할 수 있습니다. 많은 사람들이 지연 로딩이 N+1 문제를 완전히 해결해 준다고 오해하지만, 이는 사실이 아닙니다. 오히려 지연 로딩은 N+1 문제의 발생 가능성을 *줄여주는* 것이지, 완전히 없애는 것은 아닙니다.

좀 더 정확히 말하자면, 지연 로딩은 N+1 문제의 *발생 시점을 지연시키는* 역할을 합니다. 즉, 데이터를 조회하는 시점에는 연관된 엔티티를 함께 가져오지 않고, 실제로 해당 엔티티에 접근하는 시점에 추가적인 쿼리를 발생시키는 것입니다.

---

5. 언제 Eager 또는 Lazy 로딩을 사용해야 하는지?

- **Eager 로딩**:
  - **항상 필요한 데이터**가 있을 때 사용 (예: 연관된 엔티티가 항상 필요하고, 연관된 데이터를 조회하는 데 시간이 크게 소요되지 않는 경우).
  - 예시: **사용자**가 여러 **글**을 쓴 경우, 사용자 정보를 조회할 때 글 목록도 자주 필요하다면 Eager 로딩을 사용해도 괜찮습니다.
- **Lazy 로딩**:
  - **조건에 따라 필요한 데이터**만 조회할 때 사용 (예: 연관된 엔티티가 항상 필요하지 않거나, 성능 최적화가 필요한 경우).
  - 예시: **사용자**가 작성한 **댓글** 목록을 조회할 때, 댓글이 항상 필요한 경우가 아니라면 Lazy 로딩을 사용하여 필요한 시점에만 데이터를 조회합니다.

---

6. 결론

- **Eager 로딩**은 모든 연관 데이터를 즉시 조회하므로 성능상 유리하지 않을 수 있으며, **N+1 문제**를 초래할 수 있습니다.
- **Lazy 로딩**은 연관 데이터를 실제로 필요할 때만 조회하므로 성능 최적화에 유리하지만, **LazyInitializationException**이 발생할 수 있습니다.
- 이 두 방식은 사용 시점에 따라 성능 및 기능적인 측면에서 적절하게 선택해야 합니다.

영속성 컨텍스트(Persistence Context) 문제

- *영속성 컨텍스트(Persistence Context)**는 JPA에서 객체를 관리하는 **1차 캐시**로, 데이터베이스와 애플리케이션 사이에서 엔티티를 관리하는 메모리 공간입니다. 영속성 컨텍스트는 **엔티티의 상태 변화**를 추적하고, **데이터베이스와의 동기화**를 담당합니다. 이 컨텍스트 내에서 객체는 **영속 상태**로 관리됩니다.

영속성 컨텍스트의 문제점은 **동일한 엔티티 객체를 여러 번 조회할 때** 발생하는 **성능 저하**와 **잘못된 객체 상태**를 관리하는 데 어려움이 있을 수 있다는 점입니다. 이 문제를 이해하려면 영속성 컨텍스트의 특성과 관련된 문제들을 살펴볼 필요가 있습니다.

영속성 컨텍스트의 상태

JPA 엔티티는 다음과 같은 **상태**를 가집니다:

1. **Transient (비영속 상태)**:
   - 데이터베이스와 연결되지 않은 객체 상태입니다.
   - 객체가 새로 생성되었거나 영속성 컨텍스트에 의해 관리되지 않는 상태입니다.
2. **Managed (영속 상태)**:

- 영속성 컨텍스트에 의해 관리되는 상태입니다.
- 엔티티가 영속성 컨텍스트에 의해 관리될 때, 데이터베이스와 동기화됩니다.

3. **Detached (분리 상태)**:
   - 영속성 컨텍스트에서 분리된 상태입니다.
   - 엔티티가 영속성 컨텍스트에서 분리되었거나 세션이 종료된 후 상태입니다.
4. **Removed (삭제 상태)**:
   - 영속성 컨텍스트에서 제거된 상태입니다.
   - 데이터베이스에서 삭제된 상태로 변경되었지만 아직 커밋되지 않은 상태입니다.

영속성 컨텍스트 문제

영속성 컨텍스트에서 발생할 수 있는 주요 문제는 **N+1 쿼리 문제**와 **지연 로딩(Lazy Loading)과 관련된 문제**입니다.

1. N+1 문제

N+1 문제는 연관된 엔티티들을 조회할 때, **N개의 개별 쿼리**가 발생하는 문제입니다. 이는 **지연 로딩**을 사용할 때 자주 발생합니다.

문제 설명:

영속성 컨텍스트는 엔티티를 **한 번만 조회**하고, 그 이후 **같은 엔티티**에 대해서는 **영속성 컨텍스트에서 관리된 객체**를 반환해야 합니다. 하지만 **지연 로딩**을 사용할 경우, 연관된 엔티티들을 조회할 때마다 **별도의 쿼리가 실행**됩니다.

예시 코드 (N+1 문제 발생):

```
java
코드 복사
@Entity
public class Author {
    @Id
    private Long id;
    private String name;

    @OneToMany(fetch = FetchType.LAZY)
    private List<Book> books;
}

@Entity
public class Book {
    @Id
    private Long id;
    private String title;
}
```

```
java
코드 복사
List<Author> authors = authorRepository.findAll();
for (Author author : authors) {
    System.out.println(author.getBooks().size()); // 책 목록을 조회할 때마다 추가 쿼리 발생
}
```

위 코드에서는 `Author` 목록을 조회할 때, 각 `Author`에 해당하는 `Book` 목록을 조회하기 위해 **추가 쿼리**가 실행됩니다. 이때 **N+1 문제**가 발생하며, 첫 번째 쿼리에서 `Author` 목록을 조회하고, 그 다음 각 `Author`에 대해 별도로 `Book` 목록을 조회하는 추가 쿼리가 실행됩니다.

해결 방법 (JOIN FETCH 사용):

```
java
코드 복사
@Query("SELECT a FROM Author a JOIN FETCH a.books")
List<Author> findAllAuthorsWithBooks();
```

이렇게 `JOIN FETCH`를 사용하면 **한 번의 쿼리**로 `Author`와 `Book`을 **조인**하여 모두 가져올 수 있습니다.

---

2. 영속성 컨텍스트와 트랜잭션

영속성 컨텍스트는 **트랜잭션이 활성화된 상태에서만** 제대로 동작합니다. 만약 트랜잭션이 끝나면, 영속성 컨텍스트에서 관리되던 객체는 **detached 상태**로 변하고, 그 이후에는 더 이상 데이터베이스와 동기화되지 않습니다.

문제 예시:

```
java
코드 복사
@Transactional
public void updateAuthorName(Long authorId, String newName) {
    Author author = authorRepository.findById(authorId).orElseThrow();
    author.setName(newName);  // 영속성 컨텍스트에서 관리
    // 트랜잭션 끝나면 영속성 컨텍스트가 종료되어 detached 상태로 변함
}

public void updateAuthorNameOutsideTransaction(Long authorId, String newName) {
    Author author = authorRepository.findById(authorId).orElseThrow();
    author.setName(newName);  // 트랜잭션 밖에서의 변경
    authorRepository.save(author);  // 저장할 때 detached 상태로 저장
}
```

`updateAuthorName` 메서드는 **트랜잭션 내**에서 작동하므로 영속성 컨텍스트가 객체를 관리합니다. 그러나 `updateAuthorNameOutsideTransaction` 메서드는 **트랜잭션 외부**에서 작동하면, 엔티티가 **detached 상태**가 되어 **영속성 컨텍스트에서 관리되지 않기 때문에 저장되지 않을 수 있습니다**.

이 문제를 해결하려면, **트랜잭션이 활성화된 상태에서 엔티티를 관리**해야 하며, **detached 상태의 엔티티는 다시 영속성 컨텍스트에 병합**해야 합니다.

```
java
코드 복사
@Transactional
public void updateAuthorNameOutsideTransaction(Long authorId, String newName) {
    Author author = authorRepository.findById(authorId).orElseThrow();
    author.setName(newName);  // detached 상태
    entityManager.merge(author);  // 병합하여 영속성 컨텍스트에 반영
}
```

---

3. LazyInitializationException

LazyInitializationException은 **지연 로딩**을 사용할 때 **영속성 컨텍스트가 닫힌 상태에서** 연관된 엔티티를 접근하려 할 때 발생합니다. 즉, 연관된 엔티티가 **영속성 컨텍스트 밖에 있을 때** 데이터를 조회하려고 하면 예외가 발생합니다.

문제 예시:

```
java
코드 복사
@Transactional
public void printBookDetails(Long authorId) {
    Author author = authorRepository.findById(authorId).orElseThrow();
    List<Book> books = author.getBooks();  // Lazy 로딩으로 연관된 엔티티를 조회
    entityManager.clear();  // 영속성 컨텍스트를 클리어
    books.forEach(book -> System.out.println(book.getTitle()));  // LazyInitialization
Exception 발생
}
```

이 코드에서 **`entityManager.clear()`**로 영속성 컨텍스트를 클리어한 후, `Book` 목록에 접근하면 **`LazyInitializationException`**이 발생합니다. 이유는 **연관된 엔티티가 영속성 컨텍스트 밖에 있기 때문**입니다.

해결 방법 (트랜잭션 내에서 데이터 접근)

```
java
코드 복사
@Transactional
public void printBookDetails(Long authorId) {
    Author author = authorRepository.findById(authorId).orElseThrow();
    List<Book> books = author.getBooks();  // 트랜잭션 내에서 Lazy 로딩이 동작
    books.forEach(book -> System.out.println(book.getTitle()));  // 예외 발생 안함
}
```

이 방법은 **트랜잭션 내에서 데이터를 조회**하므로, 영속성 컨텍스트가 유지되는 상태에서 **지연 로딩**이 가능합니다.

영속성 컨텍스트의 문제는 **서비스 계층에서 레포지토리**가 동작하고, **컨트롤러**에서 해당 엔티티를 클라이언트에 반환할 때 발생하는 경우가 많습니다. 대표적인 문제로는 **LazyInitializationException**이 있으며, 이는 **영속성 컨텍스트가 이미 종료된 상태**에서 지연 로딩이 시도되었을 때 발생합니다.

이 문제를 예시로 들어, **컨트롤러**에서 클라이언트로 응답을 반환할 때, **서비스 계층**에서 **영속성 컨텍스트**와 관련된 문제가 발생하는 상황을 만들어 보겠습니다.

예시: LazyInitializationException 발생

1. 엔티티 클래스

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

    @OneToMany(fetch = FetchType.LAZY)  // 지연 로딩
    private List<Book> books;

    // getters and setters
}

@Entity
```

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

    // getters and setters
}
```

`Author` 엔티티는 **`Book`**과 **1:N 관계**를 가지며, `Book` 엔티티는 **지연 로딩(Lazy Loading)**으로 설정되어 있습니다.

2. 서비스 계층

서비스 계층에서는 `Author`를 조회하고, 해당 `Author` 객체에 있는 `Book` 목록을 사용할 수 있어야 합니다.

```
java
코드 복사
@Service
public class AuthorService {
    @Autowired
    private AuthorRepository authorRepository;

    @Transactional  // 트랜잭션을 열어놓고 작업
    public Author getAuthor(Long authorId) {
        Author author = authorRepository.findById(authorId)
            .orElseThrow(() -> new RuntimeException("Author not found"));
        return author;  // books는 Lazy 로딩이므로 트랜잭션 내에서만 정상적으로 조회
    }
}
```

`AuthorService`에서는 `@Transactional`을 사용하여 트랜잭션을 시작하고, `Author`를 조회합니다. 이때 `books` 필드는 **Lazy Loading**이므로, `Author` 객체를 반환하는 동안 트랜잭션이 유지됩니다.

3. 컨트롤러 계층

컨트롤러에서 클라이언트의 요청을 받고, 해당 서비스를 호출하여 결과를 반환합니다. 이때 문제가 발생할 수 있습니다.

```
java
코드 복사
@RestController
@RequestMapping("/authors")
public class AuthorController {
    @Autowired
    private AuthorService authorService;

    @GetMapping("/{id}")
    public Author getAuthor(@PathVariable Long id) {
        Author author = authorService.getAuthor(id);  // 서비스에서 데이터를 가져옴
        return author;  // 여기서 Lazy 로딩된 books 필드에 접근하면 예외 발생
    }
}
```

위 코드에서는 `AuthorService.getAuthor(id)`를 호출하여 `Author` 객체를 반환받고 이를 그대로 클라이언트에 반환하고 있습니다. 그러나 중요한 점은 `Author` 엔티티에 **`books` 필드**가 지연 로딩으로 설정되어 있기 때문에, **영속성 컨텍스트가 종료된 후에** `books` 필드를 접근하려고 하면 **LazyInitializationException**이 발생합니다.

4. 발생하는 문제

컨트롤러에서 반환되는 `Author` 객체가 **트랜잭션이 끝난 후**에 `books` 필드를 **지연 로딩(Lazy Loading)**하려 하면, 영속성 컨텍스트가 종료된 상태에서 지연 로딩을 시도하게 되어 예외가 발생합니다.

LazyInitializationException 예시:

```
yaml
코드 복사
org.hibernate.LazyInitializationException: failed to lazily initialize a collection of
role: com.example.Author.books, no session or session was closed
```

이 오류는 영속성 컨텍스트가 닫힌 후에 **지연 로딩**을 시도했기 때문에 발생합니다.

5. 해결 방법

이 문제를 해결하려면, `Author` **객체를 반환하기 전에** 트랜잭션 내에서 연관된 `Book` 목록을 **미리 로딩**하거나, **`FetchType.EAGER`**를 사용하여 즉시 로딩을 하도록 해야 합니다.

해결 방법 1: `JOIN FETCH`를 사용하여 미리 로딩하기

```
java
코드 복사
@Service
public class AuthorService {
    @Autowired
    private AuthorRepository authorRepository;

    @Transactional
    public Author getAuthor(Long authorId) {
        // JOIN FETCH를 사용하여 books도 미리 로딩
        return authorRepository.findByIdWithBooks(authorId);
    }
}

public interface AuthorRepository extends JpaRepository<Author, Long> {
    @Query("SELECT a FROM Author a JOIN FETCH a.books WHERE a.id = :id")
    Author findByIdWithBooks(Long id);
}
```

위와 같이 `@Query`에 **`JOIN FETCH`**를 사용하여 `Author`와 `Book`을 **즉시 로딩(Eager Loading)** 하도록 변경하면, `LazyInitializationException`을 방지할 수 있습니다.

해결 방법 2: 트랜잭션을 **컨트롤러**에서 관리

```
java
코드 복사
@RestController
@RequestMapping("/authors")
public class AuthorController {
    @Autowired
    private AuthorService authorService;

    @Transactional  // 컨트롤러에서 트랜잭션을 관리
    @GetMapping("/{id}")
    public Author getAuthor(@PathVariable Long id) {
        Author author = authorService.getAuthor(id);  // 서비스에서 데이터를 가져옴
```

```
        return author;  // Lazy 로딩된 books 필드가 트랜잭션 내에서 조회됨
    }
}
```

위와 같이 **컨트롤러**에서 트랜잭션을 관리하면, `Author` 객체를 반환할 때 **영속성 컨텍스트가 유지**되어 `books` 필드를 지연 로딩할 수 있습니다.

결론

영속성 컨텍스트와 관련된 문제는 **트랜잭션의 범위**에 따라 발생할 수 있습니다. 주로 **서비스 계층**에서 연관된 엔티티를 로딩하고 **컨트롤러**에서 반환하는 과정에서 **Lazy Loading**으로 인한 `LazyInitializationException`이 발생합니다.

이 문제를 해결하려면:

1. **트랜잭션 범위를 잘 관리**해야 합니다. (`@Transactional`을 적절하게 사용)
2. **지연 로딩 대신 즉시 로딩**을 선택하거나, ** `JOIN FETCH`*를 사용하여 필요한 데이터를 미리 로딩합니다.

해결 방법 3: DTO 사용하여 응답

서비스 모듈에서 DTO를 만들어 데이터를 반환하는 방식은 **LazyLoading 문제**를 해결하고, **네트워크 응답의 효율성**을 높이기 위해 유용합니다. 아래에 DTO를 활용한 예제를 설명하겠습니다.

1. 엔티티 클래스

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

    @OneToMany(fetch = FetchType.LAZY)  // 지연 로딩
    private List<Book> books;

    // getters and setters
}

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

    // getters and setters
}
```

2. DTO 클래스

DTO를 사용하여 **엔티티**를 **데이터 전송 객체**로 변환합니다. `Author`와 `Book`에 대한 정보를 포함할 수 있습니다.

```
java
코드 복사
public class AuthorDTO {
```

```
    private Long id;
    private String name;
    private List<BookDTO> books;

    public AuthorDTO(Long id, String name, List<BookDTO> books) {
        this.id = id;
        this.name = name;
        this.books = books;
    }

    // getters and setters
}

public class BookDTO {
    private Long id;
    private String title;

    public BookDTO(Long id, String title) {
        this.id = id;
        this.title = title;
    }

    // getters and setters
}
```

3. 서비스 계층에서 DTO 변환

서비스 계층에서 엔티티를 조회한 후, **DTO**로 변환하여 반환합니다. **DTO 변환은 주로 서비스 계층에서 처리**합니다.

```
java
코드 복사
@Service
public class AuthorService {
    @Autowired
    private AuthorRepository authorRepository;

    @Transactional
    public AuthorDTO getAuthor(Long authorId) {
        Author author = authorRepository.findById(authorId)
            .orElseThrow(() -> new RuntimeException("Author not found"));

        // Author 엔티티를 AuthorDTO로 변환
        List<BookDTO> bookDTOs = author.getBooks().stream()
            .map(book -> new BookDTO(book.getId(), book.getTitle()))
            .collect(Collectors.toList());

        return new AuthorDTO(author.getId(), author.getName(), bookDTOs);
    }
}
```

4. 컨트롤러 계층

컨트롤러에서는 서비스에서 반환된 **DTO**를 클라이언트에 그대로 반환합니다.

```
java
코드 복사
@RestController
@RequestMapping("/authors")
public class AuthorController {
    @Autowired
    private AuthorService authorService;

    @GetMapping("/{id}")
    public AuthorDTO getAuthor(@PathVariable Long id) {
        return authorService.getAuthor(id);  // DTO로 반환
    }
}
```

5. 발생하지 않는 문제

위와 같은 방식으로 DTO를 사용하면, **LazyInitializationException**이 발생하지 않습니다. `AuthorDTO`가 반환되면서 **영속성 컨텍스트가 종료된 이후에도** 데이터를 사용할 수 있게 됩니다. 또한, 필요한 데이터만을 응답으로 보내므로 **성능 최적화**와 **보안** 측면에서도 유리합니다.

DTO 사용의 장점

1. **LazyLoading 문제 해결**: DTO를 반환하면, 연관된 엔티티를 **즉시 로딩**하거나 **수동으로 로딩**할 수 있기 때문에 LazyInitializationException을 피할 수 있습니다.
2. **불필요한 데이터 차단**: 엔티티에서 필요한 필드만 DTO로 변환하여 클라이언트에 전달할 수 있습니다. 예를 들어, `books` 정보가 필요 없다면, 그 데이터를 아예 DTO에 포함시키지 않을 수 있습니다.
3. **응답 데이터 최적화**: 필요한 데이터만 포함된 DTO를 사용하면, **네트워크 부하**를 줄이고, **성능 최적화**가 가능합니다.
4. **보안**: 엔티티에는 민감한 정보가 있을 수 있지만, DTO를 사용하면 이를 제외하고 필요한 정보만 전송할 수 있어 보안에 유리합니다.

결론

DTO를 사용하여 응답 데이터를 처리하는 방식은 **LazyLoading 문제**를 해결하고, **네트워크 효율성**과 **보안성**을 개선하는 데 매우 유용한 방법입니다. **서비스 계층에서 DTO로 변환**하여 **컨트롤러에서 응답**하는 방식은 **모든 엔티티에 대해 일관된 방식으로 데이터를 처리**할 수 있게 해줍니다.

중급-3 복합 키 매핑 (EmbeddedId, IdClass)

- *복합 키 매핑 (Composite Key Mapping)**은 **하나의 엔티티**가 여러 필드를 **조합하여 기본 키**로 사용할 때 사용됩니다. JPA에서는 복합 키를 매핑할 때 두 가지 방법을 제공합니다: **`@EmbeddedId`*와 **`@IdClass`*입니다.

이 두 방법은 **복합 키**를 매핑하는 방식이 다르지만, 결과적으로는 **하나 이상의 필드를 조합하여 기본 키**로 사용하는 목적이 동일합니다.

1. `@EmbeddedId` 사용

`@EmbeddedId`는 복합 키를 **하나의 임베디드 키 클래스**로 정의하고, 이 클래스를 엔티티의 필드로 사용하는 방법입니다.

1.1 `@EmbeddedId` **사용 방법**

1. 복합 키를 위한 별도의 임베디드 클래스를 정의합니다.
2. 이 임베디드 클래스는 `@Embeddable` 어노테이션을 사용하여 정의하고, 복합 키를 이루는 필드를 정의합니다.
3. 복합 키를 사용하는 엔티티 클래스에서는 `@EmbeddedId` 어노테이션을 사용하여 임베디드 키 클래스를 필드로 설정합니다.

1.2 **예제:** `@EmbeddedId` **사용**

```
java
코드 복사
// 복합 키 클래스
@Embeddable
public class OrderItemId implements Serializable {
    private Long orderId;
    private Long productId;

    // 기본 생성자, getter, setter, equals, hashCode 메서드 구현
    public OrderItemId() {}

    public OrderItemId(Long orderId, Long productId) {
        this.orderId = orderId;
        this.productId = productId;
    }

    // equals, hashCode 메서드는 복합 키를 비교할 때 사용되므로 반드시 재정의 해야 합니다.

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        OrderItemId that = (OrderItemId) o;
        return Objects.equals(orderId, that.orderId) && Objects.equals(productId, tha
t.productId);
    }

    @Override
    public int hashCode() {
        return Objects.hash(orderId, productId);
    }
}

// 엔티티 클래스
@Entity
public class OrderItem {
    @EmbeddedId  // 복합 키 필드를 임베디드로 지정
    private OrderItemId id;

    private int quantity;

    // getter, setter
    public OrderItemId getId() {
        return id;
    }

    public void setId(OrderItemId id) {
        this.id = id;
    }

    public int getQuantity() {
        return quantity;
    }
```

```
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
}
```

설명:

- `OrderItemId`는 `@Embeddable` 클래스로, `orderId`와 `productId`를 복합 키로 사용합니다.
- `OrderItem` 엔티티에서는 `@EmbeddedId`를 사용하여 복합 키 필드를 선언합니다.
- `equals()`와 `hashCode()` 메서드를 `OrderItemId` 클래스에 정의해야 합니다. 이 메서드들은 JPA가 복합 키를 비교할 때 사용합니다.

1.3 **장점**:

- 복합 키를 **하나의 클래스로** 정의하고 관리할 수 있어서 코드가 간결하고 명확해집니다.
- 재사용이 용이하며, 복합 키의 필드가 여러 엔티티에서 동일하게 사용될 경우 유리합니다.

2. `@IdClass` 사용

`@IdClass`는 복합 키를 **별도의 클래스**에서 정의하고, 그 클래스를 엔티티에 적용하는 방법입니다. 이 방법은 복합 키 클래스를 **별도의 단일 클래스**로 만들어 사용합니다.

2.1 `@IdClass` **사용 방법**

1. 복합 키를 위한 **별도의 클래스를 정의**하고, 해당 클래스에 `@IdClass` 어노테이션을 붙입니다.
2. 엔티티 클래스에서는 복합 키를 구성하는 **각각의 필드**에 `@Id` 어노테이션을 붙여줍니다.

2.2 **예제:** `@IdClass` **사용**

```
java
코드 복사
// 복합 키 클래스
public class OrderItemIdClass implements Serializable {
    private Long orderId;
    private Long productId;

    // 기본 생성자, getter, setter, equals, hashCode 메서드 구현
    public OrderItemIdClass() {}

    public OrderItemIdClass(Long orderId, Long productId) {
        this.orderId = orderId;
        this.productId = productId;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        OrderItemIdClass that = (OrderItemIdClass) o;
        return Objects.equals(orderId, that.orderId) && Objects.equals(productId, tha
t.productId);
    }

    @Override
    public int hashCode() {
        return Objects.hash(orderId, productId);
    }
```

```
}

// 엔티티 클래스
@Entity
@IdClass(OrderItemIdClass.class)  // @IdClass를 사용하여 복합 키 클래스를 지정
public class OrderItem {
    @Id
    private Long orderId;  // 복합 키의 첫 번째 필드

    @Id
    private Long productId;  // 복합 키의 두 번째 필드

    private int quantity;

    // getter, setter
    public Long getOrderId() {
        return orderId;
    }

    public void setOrderId(Long orderId) {
        this.orderId = orderId;
    }

    public Long getProductId() {
        return productId;
    }

    public void setProductId(Long productId) {
        this.productId = productId;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
}
```

설명:

- `OrderItemIdClass`는 복합 키를 정의하는 **별도의 클래스**입니다.
- `OrderItem` 엔티티에서는 `@IdClass` 어노테이션을 사용하여 복합 키 클래스를 지정하고, 각 필드(`orderId`, `productId`)에 `@Id`를 지정하여 복합 키 필드를 설정합니다.

2.3 **장점**:

- 복합 키 클래스를 **별도의 클래스로 분리**하여 관리할 수 있습니다.
- `@EmbeddedId`보다 **필드를 직접 엔티티에 포함**시키는 형태로 사용할 수 있어서, 더 직관적이고 쉽게 필드를 관리할 수 있습니다.

3. `@EmbeddedId`와 `@IdClass`의 **차이점**

Feature	`@EmbeddedId`	`@IdClass`
복합 키 정의 방식	임베디드 클래스 사용	복합 키를 별도의 클래스에서 정의

키 필드 위치	엔티티의 **하나의 필드**로 정의	엔티티의 **각각의 필드**에 `@Id` 어노테이션 적용
사용 용이성	임베디드 클래스는 별도로 정의하고 한 곳에 관리	복합 키 클래스와 엔티티 필드를 명시적으로 분리
`equals()` / `hashCode()`	임베디드 클래스에서 구현	복합 키 클래스에서 구현

결론

- `@EmbeddedId`: 복합 키를 **하나의 클래스**로 정의하여 엔티티에 **임베디드** 형태로 사용하는 방식입니다. 코드가 간결하고 복합 키를 관리하기 용이합니다.
- `@IdClass`: 복합 키를 **별도의 클래스**로 정의하고, 엔티티에서는 각 필드에 `@Id` 어노테이션을 사용하여 복합 키를 매핑하는 방식입니다.

둘 중 어느 방법을 선택할지는 **개발자의 선호도**와 **프로젝트 요구 사항**에 따라 결정하면 됩니다. 일반적으로 **`@EmbeddedId`**를 사용하는 경우가 많지만, **`@IdClass`**를 사용하는 경우도 복합 키 필드를 **직접** 정의할 수 있어 유용할 수 있습니다.

중급-4 NamedQuery와 NativeQuery 사용

NamedQuery와 NativeQuery

JPA는 데이터베이스와 상호작용할 때 JPQL 또는 SQL 쿼리를 사용할 수 있도록 **`@NamedQuery`**와 **`@NamedNativeQuery`**를 제공합니다. 이 두 어노테이션은 **정적인 쿼리**를 사전에 정의하고 재사용할 수 있게 합니다.

---

1. `@NamedQuery`

1.1 정의

- JPQL(Java Persistence Query Language)을 사용하여 **엔티티 클래스 기준**으로 쿼리를 작성합니다.
- SQL과 달리 데이터베이스 테이블이 아닌 **엔티티 클래스 및 필드**를 대상으로 동작합니다.

1.2 선언 위치

- 일반적으로 **엔티티 클래스**에 정의되며, `@Entity`와 함께 사용됩니다.
- XML로도 정의 가능하지만 주로 어노테이션 방식이 사용됩니다.

1.3 사용법

예제: `@NamedQuery` 사용

```
java
코드 복사
@Entity
@NamedQuery(
    name = "User.findByUsername",
    query = "SELECT u FROM User u WHERE u.username = :username"
)
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String username;
    private String email;

    // getters and setters
}
```

- `name`: 쿼리를 식별하기 위한 이름입니다.

- `query`: 실행할 JPQL 쿼리입니다.

사용 코드

```
java
코드 복사
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    @Query(name = "User.findByUsername")
    User findByUsername(@Param("username") String username);
}
```

---

2. `@NamedNativeQuery`

2.1 정의

- **네이티브 SQL 쿼리**를 사용하여 데이터베이스 테이블 기준으로 작성합니다.
- JPQL보다 복잡한 SQL 쿼리나 데이터베이스에 의존적인 기능을 사용할 때 적합합니다.

2.2 선언 위치

- `@NamedNativeQuery` 어노테이션은 엔티티 클래스에 정의됩니다.
- 반환 타입이 명확하다면 엔티티에 매핑된 결과를 반환할 수도 있습니다.

2.3 사용법

예제: `@NamedNativeQuery` 사용

```
java
코드 복사
@Entity
@NamedNativeQuery(
    name = "User.findByEmailNative",
    query = "SELECT * FROM users WHERE email = :email",
    resultClass = User.class
)
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String username;
    private String email;

    // getters and setters
}
```

- `name`: 쿼리를 식별하기 위한 이름입니다.
- `query`: 실행할 네이티브 SQL 쿼리입니다.
- `resultClass`: 쿼리 결과를 매핑할 엔티티 클래스입니다.

사용 코드

```
java
코드 복사
@Repository
```

```
public interface UserRepository extends JpaRepository<User, Long> {
    @Query(name = "User.findByEmailNative")
    User findByEmailNative(@Param("email") String email);
}
```

3. `@Query`와의 차이점

- `@NamedQuery`와 `@NamedNativeQuery`는 **정적 쿼리**로 미리 정의되어 재사용성을 높입니다.
- `@Query`는 **동적 정의**에 적합하며, 특정 메서드에 국한된 쿼리를 작성할 때 사용됩니다.

4. 정리

Feature	NamedQuery	NamedNativeQuery
쿼리 언어	JPQL	네이티브 SQL
기준	엔티티 클래스 및 필드	데이터베이스 테이블 및 컬럼
선언 위치	엔티티 클래스	엔티티 클래스
복잡도	비교적 간단한 JPQL	복잡한 SQL, DB 의존적 쿼리 작성 가능
반환 타입	엔티티 클래스	엔티티 클래스 또는 기타 결과

5. 참고: 동적 네이티브 쿼리 사용

`@NamedNativeQuery` 대신 `@Query`를 사용하여 네이티브 SQL 쿼리를 작성할 수도 있습니다. 이 방식은 동적 네이티브 쿼리에 적합합니다.

```
java
코드 복사
@Query(value = "SELECT * FROM users WHERE email = :email", nativeQuery = true)
User findByEmail(@Param("email") String email);
```

- `nativeQuery = true`를 설정하면 JPQL 대신 네이티브 SQL로 실행됩니다.

결론

- `@NamedQuery`와 `@NamedNativeQuery`는 **재사용이 필요한 정적 쿼리**에 적합합니다.
- 동적 쿼리가 필요하거나 복잡도가 낮다면 `@Query`를 사용하는 것이 더 유연합니다.
- 네이티브 쿼리가 필요한 경우 데이터베이스 의존성을 고려하여 적절히 선택하세요.

예제: 여러 `@NamedQuery` 정의

```
java
코드 복사
@Entity
@NamedQueries({
    @NamedQuery(
        name = "User.findByUsername",
        query = "SELECT u FROM User u WHERE u.username = :username"
    ),
    @NamedQuery(
        name = "User.findByEmail",
        query = "SELECT u FROM User u WHERE u.email = :email"
    )
})
```

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

    private String username;
    private String email;

    // getters and setters
}
```

NamedNativeQuery도 동일하게 사용 가능

예제: 여러 `@NamedNativeQuery` 정의

```
java
코드 복사
@Entity
@NamedNativeQueries({
    @NamedNativeQuery(
        name = "User.findByUsernameNative",
        query = "SELECT * FROM users WHERE username = :username",
        resultClass = User.class
    ),
    @NamedNativeQuery(
        name = "User.findByEmailNative",
        query = "SELECT * FROM users WHERE email = :email",
        resultClass = User.class
    )
})
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String username;
    private String email;

    // getters and setters
}
```

중급-5 Entity Life Cycle (Persist, Merge, Remove, Detach)

```
persist(entity)
```

비영속 → 영속 : 엔티티를 영속성 컨텍스트에 영속화

```
merge(entity)
```

준영속 → 영속 : 준영속 상태의 엔티티를 영속성 컨텍스트에 병합

```
find(entityClass, primaryKey)
```

(비영속/준영속) → 영속 : 주어진 기본 키로 엔티티를 조회. 영속성 컨텍스트에 없으면 데이터베이스에서 조회하여 영속 상태로 만듦. 있으면 영속 상태 유지.

```
remove(entity)
```

영속 → 삭제 예약 : 엔티티를 삭제 대상으로 지정. 실제 삭제는 플러시 또는 커밋 시점에 발생

```
detach(entity)
```

영속 → 준영속 : 특정 엔티티를 영속성 컨텍스트에서 분리

```
clear()
```

영속 → 준영속 (전체) : 영속성 컨텍스트를 초기화하여 모든 엔티티를 준영속 상태로 만듦

```
close()
```

영속 → 준영속 (전체) : 엔티티 매니저를 종료하여 연결된 영속성 컨텍스트를 닫음. 모든 엔티티는 준영속 상태가 됨

public class JpaExample {

```
public static void main(String[] args) {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpabook");
    EntityManager em = emf.createEntityManager();
    EntityTransaction tx = em.getTransaction();

    try {
        tx.begin();

        // 1. 새로운 데이터 삽입 (INSERT)
        Member member = new Member();
        member.setName("최초 멤버");
        em.persist(member);
        System.out.println("1. persist() 후: " + member);

        // 2. 데이터 조회 (SELECT)
        Member findMember = em.find(Member.class, member.getId());
        System.out.println("2. find() 후: " + findMember);

        // 3. 데이터 수정 (UPDATE) - 변경 감지
        findMember.setName("이름 변경됨");
        System.out.println("3. 이름 변경 후: " + findMember);

        // 4. 다시 데이터 조회 (SELECT)
        Member findMember2 = em.find(Member.class, member.getId());
        System.out.println("4. 다시 find() 후: " + findMember2);

        // --- merge() 사용 시나리오 시작 ---
        // 5. 준영속 상태 만들기 (em.detach 또는 em.clear 또는 em.close 이후)
        em.detach(findMember); // findMember를 준영속 상태로 만듦
        System.out.println("5. detach() 후: " + findMember);

        // 6. 준영속 상태의 엔티티 수정
        findMember.setName("준영속 상태에서 이름 변경"); // DB에는 반영되지 않음
        System.out.println("6. 준영속 상태에서 이름 변경 후: " + findMember);
```

```
        // 7. merge()를 사용하여 영속 상태로 병합
        Member mergedMember = em.merge(findMember);
        System.out.println("7. merge() 후: " + mergedMember);
        if (mergedMember == findMember){
            System.out.println("merge 후 mergedMember 와 findMember 는 같은 객체이다.");
        } else {
            System.out.println("merge 후 mergedMember 와 findMember 는 다른 객체이다.");
        }

        // 8. merge된 엔티티 수정 (영속 상태이므로 변경 감지)
        mergedMember.setName("merge 후 다시 이름 변경");
        System.out.println("8. merge 후 다시 이름 변경 후: " + mergedMember);

        // --- merge() 사용 시나리오 끝 ---

        // 9. 데이터 삭제 (REMOVE)
        em.remove(mergedMember); // merge된 객체를 삭제해야 함
        System.out.println("9. remove() 후: " + mergedMember);

        tx.commit();
        System.out.println("tx.commit() 이후");

        Member deletedMember = em.find(Member.class, member.getId());
        System.out.println("10. tx.commit() 이후 find() : deletedMember = " + deletedMe
mber);

    } catch (Exception e) {
        tx.rollback();
        e.printStackTrace();
    } finally {
        em.close();
        emf.close();
    }
}
```

}

스프링 부트에서 JPA를 사용할 때는 대부분 **Spring Data JPA**를 통해 `JpaRepository` 또는 커스텀 리포지토리를 사용하며, `EntityManager`를 직접 다루는 경우는 드뭅니다. 하지만 스프링 부트에서도 JPA의 **엔티티 라이프사이클** 개념은 여전히 동일하게 적용됩니다. 다만, **EntityManager의 역할을 스프링 데이터 JPA가 추상화**했기 때문에 이를 간접적으로 경험하게 됩니다.

스프링 부트에서의 엔티티 라이프사이클

1. 비영속 (Transient)

- 엔티티 객체를 생성했지만, 아직 **영속성 컨텍스트(EntityManager)**에 저장하지 않은 상태.
- 예를 들어, 스프링 부트에서 새 엔티티 객체를 생성만 하고 아직 `save()`를 호출하지 않은 경우.

```
java
코드 복사
User user = new User(); // 비영속 상태
user.setName("John Doe");
```

---

2. 영속 (Persistent)

- 엔티티가 **영속성 컨텍스트**에 등록된 상태.

- 스프링 부트에서 `JpaRepository.save(entity)`를 호출하면 엔티티가 영속 상태로 전환됩니다.
- 영속 상태의 엔티티는 변경 감지(Dirty Checking)를 통해 자동으로 데이터베이스와 동기화됩니다.

```
java
코드 복사
User user = new User();
user.setName("John Doe");
userRepository.save(user); // 영속 상태로 전환
```

3. 준영속 (Detached)

- 영속 상태였던 엔티티가 **영속성 컨텍스트에서 분리된 상태**.
- 스프링 부트에서 트랜잭션이 끝나거나, 명시적으로 분리(detach)된 경우 준영속 상태가 됩니다.
- 준영속 상태에서는 변경 감지가 작동하지 않습니다.

```
java
코드 복사
@Transactional
public void updateUser(Long userId, String newName) {
    User user = userRepository.findById(userId).orElseThrow();
    entityManager.detach(user); // 준영속 상태로 전환
    user.setName(newName); // 데이터베이스에 반영되지 않음
}
```

4. 삭제 (Removed)

- 영속성 컨텍스트에 등록된 엔티티를 삭제할 준비 상태로 변경.
- 스프링 부트에서는 `JpaRepository.delete(entity)`를 호출하여 삭제 상태로 변경합니다.
- 트랜잭션 종료 시 `DELETE` SQL이 실행됩니다.

```
java
코드 복사
User user = userRepository.findById(1L).orElseThrow();
userRepository.delete(user); // 삭제 상태로 전환
```

스프링 데이터 JPA와 엔티티 라이프사이클의 차이점

1. **EntityManager를 직접 사용하지 않음**
   - Spring Data JPA는 `EntityManager`를 추상화하여 `save`, `findById`, `delete` 등의 메서드로 라이프사이클 관리를 간소화합니다.
   - 직접 `persist`, `merge`, `detach` 등을 호출할 필요가 없습니다.
2. **트랜잭션 관리**
   - 스프링 부트에서는 대부분의 데이터베이스 작업이 **트랜잭션 안에서 수행**됩니다.
   - `@Transactional` 애너테이션이 사용된 서비스 계층에서 트랜잭션이 열리고, 트랜잭션 종료 시점에 영속성 컨텍스트가 `flush()` 및 `commit()`을 수행합니다.
3. **자동 변경 감지**
   - 영속성 컨텍스트에 있는 엔티티는 변경이 감지되어 자동으로 데이터베이스에 반영됩니다.

- 변경 감지를 명시적으로 호출할 필요 없이, 트랜잭션 종료 시점에 동기화됩니다.

스프링 부트에서 라이프사이클을 반영한 코드 흐름 예제

1. 기본 CRUD 흐름

```
java
코드 복사
@Transactional
public void performLifecycleDemo() {
    // 1. 비영속 상태
    User user = new User();
    user.setName("John Doe");

    // 2. 영속 상태 (save 호출)
    userRepository.save(user);

    // 3. 변경 감지 (영속 상태)
    user.setName("John Updated");

    // 4. 삭제 상태
    userRepository.delete(user);
    // 트랜잭션 종료 시 flush & commit
}
```

2. 준영속 상태 처리 흐름

```
java
코드 복사
@Transactional
public void demonstrateDetachedEntity(Long userId) {
    // 1. 영속 상태로 엔티티 조회
    User user = userRepository.findById(userId).orElseThrow();

    // 2. 준영속 상태로 전환
    entityManager.detach(user);

    // 3. 준영속 상태에서 변경
    user.setName("Updated Name"); // 변경 감지되지 않음

    // 4. 병합 (merge)
    entityManager.merge(user); // 영속 상태로 병합
}
```

스프링 부트에서 라이프사이클 관리에 대한 팁

1. **DTO 사용 추천**
   - 준영속 상태의 문제를 피하기 위해 엔티티를 그대로 반환하지 말고 DTO를 사용하세요.
2. **트랜잭션 명확히 관리**
   - 서비스 계층에서만 `@Transactional`을 사용하는 것이 일반적입니다.
3. **영속 상태 활용**
   - 가능하면 영속 상태의 엔티티를 트랜잭션 내에서만 사용하고, 트랜잭션 밖에서는 DTO로 변환해 사용합니다.

스프링 부트 환경에서 JPA 라이프사이클은 자동화된 방식으로 관리되지만, 엔티티 상태와 동작을 이해하는 것이 문제를 예방하고 최적화에 도움이 됩니다. 😊

- *더티 체킹 (Dirty Checking)**은 JPA의 중요한 개념 중 하나로, **영속성 컨텍스트**에서 관리되는 엔티티 객체의 변경 사항을 추적하고, 이를 데이터베이스에 반영하는 과정입니다. 이 용어는 "더티(dirty)"라는 단어에서 유래된 것으로, 엔티티의 상태가 변경되었음을 나타내기 위해 사용됩니다. 즉, **"더티" 상태란 엔티티 객체의 필드 값이 변경된 상태**를 의미합니다.

왜 '더티(dirty)'라는 이름이 붙었을까요?

"더티"는 원래 컴퓨터 과학에서 **값이 변경된** 혹은 **업데이트가 필요한** 상태를 지칭하는 용어입니다. 이는 엔티티가 영속성 컨텍스트에 의해 관리되고 있는데, 객체의 속성이 변경되어 더 이상 "깨끗한(clean)" 상태가 아니라는 것을 나타내기 위해 사용됩니다.
즉, **엔티티가 변경된 상태**는 "더티"라고 부르고, 이를 추적하는 과정이 **더티 체킹**입니다.

더티 체킹의 동작 원리

JPA는 영속성 컨텍스트 내에서 엔티티 객체의 상태를 관리하며, 트랜잭션이 끝날 때 변경된 값을 **자동으로 감지**하고 **DB에 반영**합니다. 이를 위해 **자체적으로 엔티티 객체의 상태를 추적**하고, 변경된 필드를 추적하여 SQL `UPDATE` 문을 실행합니다.

1. **객체 상태 관리**

- 엔티티 객체가 영속성 컨텍스트에 저장된 후, 해당 객체의 필드 값이 변경되면, 이를 JPA가 감지하고 추적합니다.
- 이 때 객체는 **"더티(dirty)" 상태로 표시**됩니다.

2. **트랜잭션이 커밋될 때**

- 트랜잭션이 커밋될 때, **영속성 컨텍스트**에서 "더티 상태"인 엔티티들을 찾아 **자동으로 DB에 반영**(update)합니다.

3. **변경 감지**

- JPA는 **트랜잭션이 끝날 때**만 데이터를 DB에 반영하므로, 변경된 엔티티를 DB에 바로 반영하지 않고, **변경된 상태를 추적**합니다.

예시

```
java
코드 복사
@Transactional
public void updateUser(Long userId, String newName) {
    // 엔티티 조회
    User user = userRepository.findById(userId).orElseThrow();

    // 변경
    user.setName(newName);  // "더티 상태" 발생

    // 트랜잭션 종료 시점에 변경사항 자동 반영
    // JPA가 변경된 엔티티를 감지하고 UPDATE SQL을 실행
}
```

위 코드에서 `user.setName(newName)` 가 호출되면, 엔티티의 상태는 "더티" 상태로 변하고, 트랜잭션이 끝날 때 **UPDATE 쿼리**가 실행되어 데이터베이스에 반영됩니다. **별도로 `update()` 메서드를 호출할 필요는 없습니다**. 이는 JPA가 엔티티의 변경 사항을 자동으로 추적하고 반영하는 방식 덕분입니다.

왜 더티 체킹이 중요한가요?

1. **수동 업데이트를 피할 수 있음**:
   - 엔티티 상태 변경 후 명시적으로 `update()` 쿼리를 호출할 필요 없이, JPA가 자동으로 변경 사항을 추적하고 업데이트합니다.
2. **자동화된 상태 관리**:
   - 복잡한 트랜잭션이나 엔티티 상태를 자동으로 관리할 수 있어 개발자가 수동으로 데이터를 처리할 필요가 줄어듭니다.

3. **성능 향상**:
    - 여러 번의 명시적 SQL 쿼리를 줄일 수 있어 성능을 최적화할 수 있습니다. 엔티티 상태 변경이 있을 때만 데이터베이스에 반영되므로 효율적입니다.