

# 7. 데이터 접근 계층 (Spring Data)

## JDBC 연결

### 1. JDBC란?

JDBC는 Java에서 **관계형 데이터베이스(RDB)**와 통신할 수 있도록 해주는 **표준 API(인터페이스)**이다.

JDBC는 Java 코드에서 **SQL 실행** → **결과 조회** → **DB 연결 관리**를 할 수 있도록 해주는 **Java SE 표준 라이브러리**이며, **DB 벤더는 JDBC 드라이버를 제공**해야 한다.

### 2. 주요 구성 요소

구성요소	설명
DriverManager	DB 연결을 위한 <b>드라이버 관리</b> 및 <b>커넥션 생성기</b>
Connection	DB와의 <b>연결 객체</b>
Statement, PreparedStatement	SQL을 실행하는 객체 (쿼리, 파라미터 바인딩)
ResultSet	SELECT 쿼리 결과를 순회하며 조회하는 객체

### 3. JDBC 연결 절차 (5단계)

- 1

2

3

4

5
1. JDBC 드라이버 로드

2. DB와 연결 (Connection)

3. SQL 생성 및 실행 (Statement)

4. 결과 처리 (ResultSet)

5. 연결 자원 해제 (close)

### 4. 실전 코드 예시 (MySQL 기준)

예시: `users` 테이블에서 모든 사용자 조회

```
1 import java.sql.*;
2
3 public class JdbcExample {
4
5     public static void main(String[] args) {
6         String url = "jdbc:mysql://localhost:3306/testdb"; // DB URL
7         String username = "testuser";
8         String password = "testpass";
9
10        String sql = "SELECT id, name, email FROM users";
11    }
```

```

12     try (
13         Connection conn = DriverManager.getConnection(url, username, password);
14         PreparedStatement pstmt = conn.prepareStatement(sql);
15         ResultSet rs = pstmt.executeQuery();
16     ) {
17         while (rs.next()) {
18             long id = rs.getLong("id");
19             String name = rs.getString("name");
20             String email = rs.getString("email");
21
22             System.out.printf("ID: %d, Name: %s, Email: %s\n", id, name, email);
23         }
24     } catch (SQLException e) {
25         e.printStackTrace(); // 예외 로그
26     }
27 }
28 }

```

## ⚠ 5. JDBC 자원 정리 (try-with-resources 권장)

- `Connection`, `Statement`, `ResultSet` 은 모두 **AutoCloseable**
- `try-with-resources` 로 자원 자동 반환을 보장해야 함
- 예전 방식 ( `finally` 에서 수동 close )는 오류 발생 위험 높음

```

1 try (
2     Connection conn = ...
3     PreparedStatement stmt = ...
4     ResultSet rs = ...
5 ) {
6     ...
7 }

```

## 🔒 6. PreparedStatement vs Statement

항목	Statement	PreparedStatement
SQL 실행 방식	문자열 직접 전달	SQL 구조 + 바인딩
SQL Injection 방지	✗	✓
파라미터 바인딩	불가능	가능 ( ? 자리 바인딩 )
성능	상대적 느림	SQL 캐싱으로 반복 성능 향상

## INSERT 예시

```
1 String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
2 PreparedStatement stmt = conn.prepareStatement(sql);
3 stmt.setString(1, "Alice");
4 stmt.setString(2, "alice@example.com");
5 int rowsInserted = stmt.executeUpdate();
```

## 7. JDBC URL 예시

DB 종류	URL 형식
MySQL	<code>jdbc:mysql://host:port/dbname</code>
PostgreSQL	<code>jdbc:postgresql://host:port/dbname</code>
H2	<code>jdbc:h2:mem:testdb</code>
Oracle	<code>jdbc:oracle:thin:@host:port:SID</code>

## 마무리 요약

항목	설명
역할	Java에서 DB에 SQL 보내고 결과 받는 API
구성	DriverManager, Connection, Statement, ResultSet
바인딩	PreparedStatement + ? + setXXX 메서드
자원 정리	try-with-resources로 안전하게 close
위험	직접 문자열 조작 시 SQL Injection 발생 가능

## JdbcTemplate

### 1. JdbcTemplate이란?

Spring Framework가 제공하는 JDBC API 간소화 도구

#### 목적

- 반복적인 JDBC 보일러플레이트 코드 제거
- 자동 자원 해제 (`Connection`, `PreparedStatement`, `ResultSet`)
- SQL 바인딩을 명시적으로, 안전하게 지원
- Exception → `DataAccessException` 으로 통일된 예외 처리

## 2. 핵심 특징 요약

특징	설명
✓ SQL 실행 자동화	SQL 실행, 바인딩, 자원 해제까지 한 줄
✓ 자원 자동 해제	try-catch-finally 생략 가능
✓ PreparedStatement 자동 사용	SQL Injection 방지
✓ ResultSet → 객체 매핑	RowMapper 를 통해 결과를 객체로 변환
✓ 예외 추상화	SQLException → DataAccessException 변환

## 3. JdbcTemplate 설정 (Spring Boot 기준)

```
1 @Configuration
2 public class JdbcConfig {
3
4     @Bean
5     public JdbcTemplate jdbcTemplate(DataSource dataSource) {
6         return new JdbcTemplate(dataSource);
7     }
8 }
```

✦ `spring-boot-starter-jdbc` 또는 `spring-boot-starter-data-jpa` 를 의존성에 추가하면 `DataSource` 는 자동 주입됨.

## 4. 주요 사용 메서드

메서드	설명
<code>query()</code>	SELECT 쿼리 수행, RowMapper 사용
<code>queryForObject()</code>	단일 행/단일 값 조회
<code>update()</code>	INSERT / UPDATE / DELETE
<code>batchUpdate()</code>	대량 처리
<code>execute()</code>	일반 쿼리 실행 (DDL 등)

## 5. 실전 예제

### ✓ 테이블

```
1 CREATE TABLE users (  
2     id BIGINT AUTO_INCREMENT PRIMARY KEY,  
3     name VARCHAR(50),  
4     email VARCHAR(100)  
5 );
```

### ✓ User 모델

```
1 public class User {  
2     private Long id;  
3     private String name;  
4     private String email;  
5  
6     // 생성자, getter/setter  
7 }
```

### ✓ RowMapper 구현

```
1 public class UserRowMapper implements RowMapper<User> {  
2     @Override  
3     public User mapRow(ResultSet rs, int rowNum) throws SQLException {  
4         return new User(  
5             rs.getLong("id"),  
6             rs.getString("name"),  
7             rs.getString("email")  
8         );  
9     }  
10 }
```

### ✓ Repository 클래스

```
1 @Repository  
2 public class UserRepository {  
3  
4     private final JdbcTemplate jdbc;  
5  
6     public UserRepository(JdbcTemplate jdbc) {  
7         this.jdbc = jdbc;  
8     }  
9  
10    public List<User> findAll() {  
11        String sql = "SELECT * FROM users";  
12        return jdbc.query(sql, new UserRowMapper());  
13    }
```

```

13     }
14
15     public User findById(Long id) {
16         String sql = "SELECT * FROM users WHERE id = ?";
17         return jdbc.queryForObject(sql, new UserRowMapper(), id);
18     }
19
20     public void save(User user) {
21         String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
22         jdbc.update(sql, user.getName(), user.getEmail());
23     }
24
25     public void update(User user) {
26         String sql = "UPDATE users SET name = ?, email = ? WHERE id = ?";
27         jdbc.update(sql, user.getName(), user.getEmail(), user.getId());
28     }
29
30     public void delete(Long id) {
31         jdbc.update("DELETE FROM users WHERE id = ?", id);
32     }
33 }

```

## 6. 트랜잭션 연동

JdbcTemplate은 Spring의 트랜잭션 관리(`@Transactional`)와 완전히 통합됨.

```

1  @Service
2  public class UserService {
3
4      private final UserRepository userRepository;
5
6      @Transactional
7      public void register(User user) {
8          userRepository.save(user);
9          // 예외 발생 시 자동 롤백
10     }
11 }

```

## 7. 실전 주의사항

주의	설명
SQL은 명시적으로 작성해야 함	ORM처럼 자동 생성 없음
RowMapper는 재사용 가능	하나의 RowMapper로 여러 곳에서 사용 가능
<code>queryForObject()</code> 는 0건이면 예외	<code>EmptyResultDataAccessException</code> 주의
Enum, LocalDate 등 매핑은 수동 처리	객체 매핑시 커스텀 RowMapper 필요

✅ 마무리 요약

항목	설명
목적	JDBC 코드 간결화 및 안전한 SQL 실행
주요 메서드	<code>query()</code> , <code>queryForObject()</code> , <code>update()</code>
SQL 바인딩	? 사용, 타입 안전한 <code>PreparedStatement</code>
결과 매핑	<code>RowMapper</code> 를 사용해 객체 변환
트랜잭션	Spring <code>@Transactional</code> 과 완전 연동

MyBatis 연동

🔗 1. MyBatis란?

Java 객체와 SQL 사이의 매핑(Mapping)을 수동으로 명확히 기술할 수 있도록 도와주는 반자동 ORM 프레임워크

✔ 특징 요약

항목	설명
🔍 SQL 중심	SQL을 직접 작성함 (JPA와 달리 자동 생성 없음)
📄 XML or 어노테이션	XML 또는 Java 어노테이션으로 SQL 매핑
🔄 객체 ↔ DB 매핑	쿼리 결과를 Java 객체로 바인딩
Spring과 통합 용이	<code>@Mapper</code> , <code>SqlSession</code> , <code>MappersScan</code> 사용 가능

🏗 2. MyBatis vs JPA 비교

항목	MyBatis	JPA (Hibernate)
SQL 작성	수동, 직접 작성	자동 생성 기반 (JPQL)
제어 방식	SQL 중심	객체 중심
복잡한 쿼리	매우 유리	불리함
러닝커브	낮음 (SQL 기준)	높음 (객체/캐시 중심)
성능 튜닝	SQL 직접 제어 가능	자동 캐시/패치 전략 필요

### ⚙️ 3. MyBatis 기본 구성 요소

구성요소	설명
<code>SqlSessionFactory</code>	DB 연결과 매퍼 객체 생성을 담당
<code>Mapper Interface</code>	SQL 실행 메서드 정의
<code>Mapper XML</code>	SQL 정의 ( <code>&lt;select&gt;</code> , <code>&lt;insert&gt;</code> 등)
<code>TypeAlias</code> , <code>TypeHandler</code>	객체와 SQL 타입 간 매핑 지원 도구

### 🔧 4. Spring Boot + MyBatis 연동 예제

#### 📦 ① 의존성 설정 (Maven)

```
1 <dependency>
2   <groupId>org.mybatis.spring.boot</groupId>
3   <artifactId>mybatis-spring-boot-starter</artifactId>
4   <version>3.0.2</version> <!-- 최신 버전 확인 필요 -->
5 </dependency>
6
7 <dependency>
8   <groupId>mysql</groupId>
9   <artifactId>mysql-connector-j</artifactId>
10 </dependency>
```

#### ⚙️ ② `application.yml`

```
1 spring:
2   datasource:
3     url: jdbc:mysql://localhost:3306/testdb
4     username: testuser
5     password: testpass
6     driver-class-name: com.mysql.cj.jdbc.Driver
7
8   mybatis:
9     mapper-locations: classpath:/mappers/**/*.xml
10    type-aliases-package: com.example.domain
```



### ③ 도메인 객체

```
1 public class User {
2     private Long id;
3     private String name;
4     private String email;
5
6     // Getter, Setter
7 }
```

### ④ Mapper 인터페이스

```
1 @Mapper
2 public interface UserMapper {
3     User findById(Long id);
4     List<User> findAll();
5     void insert(User user);
6     void update(User user);
7     void delete(Long id);
8 }
```

 `@Mapper` 가 붙은 인터페이스는 MyBatis가 구현체를 자동 생성함

### ⑤ Mapper XML ( `resources/mappers/UserMapper.xml` )

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
4
5 <mapper namespace="com.example.mapper.UserMapper">
6
7     <select id="findById" parameterType="long" resultType="User">
8         SELECT * FROM users WHERE id = #{id}
9     </select>
10
11     <select id="findAll" resultType="User">
12         SELECT * FROM users
13     </select>
14
15     <insert id="insert" parameterType="User">
16         INSERT INTO users (name, email)
17         VALUES (#{name}, #{email})
18     </insert>
19
20     <update id="update" parameterType="User">
21         UPDATE users
22         SET name = #{name}, email = #{email}
23         WHERE id = #{id}
24     </update>
```

```

25
26     <delete id="delete" parameterType="long">
27         DELETE FROM users WHERE id = #{id}
28     </delete>
29
30 </mapper>

```

## 5. 서비스 및 컨트롤러 연동

```

1  @Service
2  public class UserService {
3
4      private final UserMapper userMapper;
5
6      public UserService(UserMapper userMapper) {
7          this.userMapper = userMapper;
8      }
9
10     public List<User> getUsers() {
11         return userMapper.findAll();
12     }
13 }

```

```

1  @RestController
2  @RequestMapping("/api/users")
3  public class UserController {
4
5      private final UserService userService;
6
7      public UserController(UserService userService) {
8          this.userService = userService;
9      }
10
11     @GetMapping
12     public List<User> getAll() {
13         return userService.getUsers();
14     }
15 }

```

## 6. 트랜잭션 처리

Spring Boot와 연동하면 `@Transactional` 그대로 사용 가능.

```

1  @Service
2  public class OrderService {
3
4      private final OrderMapper orderMapper;
5      private final InventoryMapper inventoryMapper;
6
7      @Transactional
8      public void placeOrder(Order order) {
9          orderMapper.insert(order);
10         inventoryMapper.decrease(order.getItemId(), order.getQuantity());
11     }
12 }

```

## ✅ 마무리 요약

항목	설명
핵심 개념	SQL 매핑 중심의 반자동 ORM
설정 요소	Mapper Interface + XML Mapper
장점	복잡한 SQL 유리, 명시적 쿼리 제어
Spring 통합	@Mapper, @MapperScan, @Transactional
비교	SQL이 명확해야 하면 → MyBatis / 자동 매핑이 좋다면 → JPA

## ORM 개념

### ✖ 1. ORM이란?

객체(Object)와 관계형 데이터베이스(Relational DB) 사이의  
패러다임 차이를 자동으로 매핑해주는 기술

즉,

- Java에서는 객체,
- DB에서는 테이블로 데이터를 다루는 이 구조 차이를  
ORM이 자동으로 중간에서 변환해주는 것이다.

### 📌 2. 핵심 동작 원리

자바 세계	DB 세계
User 클래스	users 테이블
user.getName()	SELECT name FROM users WHERE id=?
new User(...)	INSERT INTO users (...) VALUES (...)

자바 세계	DB 세계
<code>user.setName("kim") → save()</code>	<code>UPDATE users SET name="kim"</code>

➡ ORM은 위 과정을 **SQL 없이 객체 단위로 처리**할 수 있도록 해준다.

### 🧠 3. 왜 ORM을 쓰는가?

목적	설명
🚩 생산성	SQL 없이 객체만으로 CRUD 가능
📦 유지보수성	DB 테이블 구조와 객체 모델 간 동기화 용이
🧪 테스트성	DB 없이 메모리 기반 테스트 가능 (H2 등)
🔄 이식성	DBMS 변경 시 영향 최소화
💎 코드 깔끔함	DAO/SQL 코드가 줄어듦

### ⚙️ 4. 대표적인 ORM 프레임워크

언어	ORM 기술
Java	Hibernate (JPA), MyBatis (반자동 ORM)
Python	SQLAlchemy, Django ORM
C#	Entity Framework
Ruby	ActiveRecord
JavaScript	Sequelize, Prisma

🔴 Java에서 "ORM"이라 하면 보통 **JPA(Hibernate)**를 의미한다.

### 🧪 5. 간단한 JPA 예시

```

1 @Entity
2 public class User {
3     @Id @GeneratedValue
4     private Long id;
5
6     private String name;
7     private String email;
8 }
```

```
1 User user = new User("Alice", "alice@example.com");
2 entityManager.persist(user); // → INSERT INTO users ...
```





```
1 User u = entityManager.find(User.class, 1L); // → SELECT * FROM users WHERE id=1
```

```
1 u.setName("Bob");
2 // 변경 감지 (dirty checking) → 트랜잭션 커밋 시 UPDATE 자동 실행
```

## 6. ORM vs SQL 직접 작성 (JDBC/MyBatis)

항목	ORM (JPA)	SQL 직접 작성 (JDBC/MyBatis)
SQL 작성량	거의 없음	명시적으로 SQL 작성
속도 제어	자동 캐시, Lazy Loading	직접 튜닝 필요
복잡 쿼리	불편함 (JPQL)	매우 자유로움
유지보수	Entity와 DB 동기화 쉬움	SQL 변경 추적 어려움
러닝커브	높음 (Hibernate 문법, 영속성 컨텍스트 등)	낮음 (SQL 친숙자에 유리)

## 7. ORM 사용 시 주의사항

항목	설명
 N+1 문제	Lazy Loading 남발 시 쿼리 폭발 위험
 과도한 추상화	SQL 튜닝 어려움 → 성능 민감한 부분은 SQL로 따로 작성 필요
 쿼리 추적 어려움	어떤 SQL이 실행되는지 로그 확인 필요 (show_sql, p6spy 등)
 객체 생명주기 관리 필수	persist, detach, merge, remove 등 JPA 상태 이해 필요

## 마무리 요약

항목	설명
정의	객체 ↔ 관계형 데이터 간 자동 매핑 기술
역할	SQL 없이도 객체로 DB 연동 가능
주요 기술	Hibernate (JPA), MyBatis
장점	생산성, 유지보수성, 코드 일관성
단점	성능 튜닝 난이도, 러닝커브 있음

# JPA와 Hibernate

## 🌱 1. JPA란?

### Java Persistence API

Java 진영의 공식 ORM 표준 명세

→ 자바 객체를 관계형 DB에 매핑하기 위한 인터페이스 기반 API

- 자바 표준 (javax.persistence.\*)
- SQL 작성 없이 객체를 DB와 매핑
- 인터페이스 규격만 정의, 실제 동작은 구현체가 담당

## 🏠 2. Hibernate란?

### JPA의 대표적인 구현체

→ JPA가 정의한 인터페이스를 실제로 동작하도록 만든 라이브러리

- JPA 이전부터 존재한 독립 ORM 프레임워크
- JPA 기능 + 고급 기능 포함 (캐시, 배치, 통계 등)

## 🎯 3. JPA vs Hibernate 비교

항목	JPA	Hibernate
정체성	표준 인터페이스	JPA 구현체 (라이브러리)
위치	javax.persistence.*	org.hibernate.*
개발 방식	인터페이스 기반	구현체 기반
목적	여러 구현체 지원	성능과 기능 강화
사용 방법	EntityManager	Session, JpaTransactionManager
기능 확장성	제한적	풍부함 (2차 캐시, Envers, Statistics 등)

📌 정리하면:

- JPA는 "규칙서",
- Hibernate는 "규칙을 따라 만든 실제 프로그램"이다.

## 4. 아키텍처 흐름

```
1 [ 애플리케이션 코드 ]
2   ↓ (JPA API)
3 [ EntityManager, @Entity, JPQL ]
4   ↓
5 [ Hibernate (JPA 구현체) ]
6   ↓
7 [ JDBC ]
8   ↓
9 [ MySQL, Oracle, PostgreSQL 등 DB ]
```

## 5. 실전 예제

### JPA 인터페이스 기반

```
1 @Entity
2 public class User {
3     @Id @GeneratedValue
4     private Long id;
5     private String name;
6 }
```

```
1 User user = new User("Alice");
2 entityManager.persist(user); // JPA 표준 API
```

### Hibernate 고급 기능 사용 예

```
1 Session session = entityManager.unwrap(Session.class);
2 session.setDefaultReadOnly(true); // Hibernate 전용 기능
```

## 6. Hibernate의 대표 확장 기능들

기능	설명
2차 캐시	EhCache, Infinispan 등 통합 지원
Envers	엔티티 수정 이력 자동 기록
Hibernate Validator	Bean Validation 연동
고급 Fetch 전략	Join, FetchProfile
Batch insert/update	성능 향상 가능 (hibernate.jdbc.batch_size)

## ⚠ 7. 주의사항 및 실전 팁

주의사항	설명
영속성 컨텍스트 이해 필수	<code>persist</code> , <code>merge</code> , <code>detach</code> , <code>flush</code> 정확히 구분 필요
N+1 문제	Lazy Loading 남발 → 반드시 Fetch Join, EntityGraph로 해결
Flush 타이밍	쿼리 실행은 즉시 X → commit 전 자동 Flush
DDL 자동 생성	실무에서는 <code>ddl-auto: none</code> 권장 (자동 테이블 생성 위험)
JPA 표준 준수 vs Hibernate 독자 확장	특정 기능 사용 시 이식성 손해 있음

## ✅ 마무리 요약표

항목	JPA	Hibernate
역할	ORM 표준 명세 (API)	구현체 (라이브러리)
관계	인터페이스	구현 클래스
사용 방법	EntityManager	Session
예외 타입	<code>javax.persistence.*</code>	<code>org.hibernate.*</code>
표준성	✅	❌ (전용 기능 많음)
학습 난이도	보통	약간 더 높음 (기능 다양)

## 🏗 실무에서는?

✓ **Spring Data JPA** = JPA + Hibernate + Spring 추상화

→ 대부분의 프로젝트는

**JPA를 코드로 사용하고, Hibernate를 내부 구현체로 사용함**

```
1 spring:
2   jpa:
3     hibernate:
4       ddl-auto: update
5       show-sql: true
```



# Entity 매핑

- `@Entity`, `@Table`, `@Id`, `@GeneratedValue`, `@Column`

## 1 @Entity

### ◆ 정의

JPA에서 이 클래스가 **영속성 관리 대상(엔티티)**임을 나타낸다.

### ◆ 역할

- 이 클래스는 **DB 테이블과 매핑될** 자바 객체임을 선언
- 반드시 **기본 생성자(public 또는 protected)**가 있어야 함

### ◆ 조건

- 기본 생성자 필요
- final 클래스 ❌, 인터페이스 ❌
- 필드는 프록시를 위해 private 또는 protected 권장

### ✓ 예시

```
1 @Entity
2 public class User {
3     ...
4 }
```

## 2 @Table

### ◆ 정의

이 엔티티가 **매핑될 테이블 이름** 및 기타 설정을 정의함.

### ◆ 기본값

- 클래스 이름을 그대로 테이블 이름으로 사용 (ex: `User` → `user`)

### ◆ 속성

속성	설명
<code>name</code>	매핑할 테이블 이름 지정
<code>schema</code>	스키마 이름
<code>uniqueConstraints</code>	유니크 제약조건 지정

## ✓ 예시

```
1 @Entity
2 @Table(name = "users")
3 public class User { ... }
```

## 3 @Id

### ◆ 정의

이 필드를 기본 키(Primary Key)로 지정함.

### ◆ 특징

- 하나 이상의 필수 식별자 필드에 지정
- 영속성 컨텍스트는 이 ID로 엔티티를 구분함

## ✓ 예시

```
1 @Id
2 private Long id;
```

## 4 @GeneratedValue

### ◆ 정의

@Id 필드의 값을 자동 생성하는 전략을 지정

### ◆ 속성 (전략)

전략	설명
AUTO (기본값)	DB 방언에 따라 자동 선택
IDENTITY	DB의 auto_increment 사용 (MySQL 등)
SEQUENCE	시퀀스 객체 사용 (Oracle, PostgreSQL 등)
TABLE	키 생성용 테이블 사용 (비추천)

## ✓ 예시

```
1 @Id
2 @GeneratedValue(strategy = GenerationType.IDENTITY)
3 private Long id;
```

## 5 @Column

### ◆ 정의

엔티티 필드를 **DB 컬럼과 매핑**하는 설정

### ◆ 생략 가능

→ 생략 시 필드명을 그대로 컬럼 이름으로 사용

### ◆ 주요 속성

속성	설명
<code>name</code>	DB 컬럼 이름 지정
<code>nullable</code>	NULL 허용 여부 (true 기본)
<code>unique</code>	유일값 제약 조건
<code>length</code>	문자열 길이 (기본 255)
<code>columnDefinition</code>	DB에 생성될 컬럼의 DDL 정의

### ✓ 예시

```
1 @Column(name = "user_name", nullable = false, length = 100)
2 private String name;
```

### ✓ 종합 예제

```
1 @Entity
2 @Table(name = "users")
3 public class User {
4
5     @Id
6     @GeneratedValue(strategy = GenerationType.IDENTITY)
7     private Long id;
8
9     @Column(name = "user_name", nullable = false, length = 100)
10    private String name;
11
12    @Column(unique = true, nullable = false)
13    private String email;
14
15    protected User() {} // 기본 생성자 (JPA 필수)
16
17    public User(String name, String email) {
18        this.name = name;
19        this.email = email;
20    }
```

## ! 실전 팁

항목	권장 사항
기본 생성자	반드시 <code>protected</code> 이상으로 선언
ID 전략	MySQL은 <code>IDENTITY</code> , Oracle은 <code>SEQUENCE</code> 권장
<code>@Column</code> 생략	기본값으로 잘 작동하되, 명시적 설정이 가독성 높음
이름 규칙	컬럼/테이블 이름은 snake_case, 필드는 camelCase 많이 사용
DDL 자동 생성	실무에서는 <code>spring.jpa.hibernate.ddl-auto=none</code> 설정하고 SQL 수동 작성 권장

## ✓ 마무리 요약

어노테이션	역할
<code>@Entity</code>	이 클래스가 JPA 엔티티임을 선언
<code>@Table</code>	매핑되는 테이블 이름 지정
<code>@Id</code>	기본 키 필드 지정
<code>@GeneratedValue</code>	기본 키 자동 생성 전략
<code>@Column</code>	컬럼 매핑 상세 설정 (이름, 제약 조건 등)

## Spring Data JPA

### • `JpaRepository`, `CrudRepository`

#### ✖ 1. 공통 개념: Spring Data Repository

Spring Data JPA는 **Repository** 인터페이스 계층 구조를 통해 JPA 엔티티에 대한 **CRUD**, **페이징**, **정렬**, **쿼리 메서드** 기능을 자동으로 제공한다.

```

1 Repository (최상위)
2   └─ CrudRepository<T, ID>
3     └─ PagingAndSortingRepository<T, ID>
4       └─ JpaRepository<T, ID>

```

## 2. CrudRepository<T, ID>

### ✓ 정의

가장 기본적인 CRUD 기능만 제공하는 인터페이스

```
1 public interface CrudRepository<T, ID> extends Repository<T, ID> {  
2     <S extends T> S save(S entity);  
3     Optional<T> findById(ID id);  
4     Iterable<T> findAll();  
5     void delete(T entity);  
6     long count();  
7     boolean existsById(ID id);  
8 }
```

### ✓ 특징

- 기본 CRUD만 필요할 때 충분
- 정렬, 페이징 기능은 없음

---

## 3. JpaRepository<T, ID>

### ✓ 정의

`CrudRepository` + `PagingAndSortingRepository` + JPA 특화 기능을 모두 포함하는 가장 완성된 리포지토리

```
1 public interface JpaRepository<T, ID> extends PagingAndSortingRepository<T, ID> {  
2     List<T> findAll();  
3     List<T> findAllById(Iterable<ID> ids);  
4     void flush();  
5     <S extends T> S saveAndFlush(S entity);  
6     void deleteInBatch(Iterable<T> entities);  
7 }
```

### ✓ 특징

- 페이징, 정렬 지원 ( `Pageable` )
  - JPA 내부 캐시, 배치 기능, `flush` 제어 등 고급 기능 제공
  - 실무에서는 보통 이걸 기본 `Repository`로 사용
-

## 4. 사용 예시

### 엔티티

```
1 @Entity
2 public class User {
3     @Id @GeneratedValue
4     private Long id;
5
6     private String name;
7     private String email;
8 }
```

### ✓ JpaRepository 구현

```
1 public interface UserRepository extends JpaRepository<User, Long> {
2     Optional<User> findByEmail(String email);
3 }
```

```
1 List<User> users = userRepository.findAll(); // 전체 조회
2 Page<User> page = userRepository.findAll(PageRequest.of(0, 10)); // 페이지 조회
```

### ✓ CrudRepository 구현 (비추천이지만 가능)

```
1 public interface UserRepository extends CrudRepository<User, Long> {
2     Optional<User> findByEmail(String email);
3 }
```

```
1 Iterable<User> users = userRepository.findAll();
```

## 5. 기능 차이 비교표

기능	CrudRepository	JpaRepository
기본 CRUD	✓	✓
페이징/정렬	✗	✓ (Pageable, Sort)
배치 삭제	✗	✓ (deleteAllInBatch())
flush 제어	✗	✓ (saveAndFlush(), flush())
findAll() 리턴 타입	Iterable<T>	List<T>
실무 적합도	낮음	높음 (추천)

## 💡 6. 언제 뭘 써야 할까?

상황	선택
정말 단순한 CRUD만 필요할 때	<code>CrudRepository</code> 가능 (테스트/학습용)
페이징, 정렬, 고급 쿼리 기능 필요	✅ <code>JpaRepository</code> 추천
Spring Data JPA 공식/커뮤니티 예시 대부분	<code>JpaRepository</code> 기준
Querydsl 등 통합 시 확장성 고려	<code>JpaRepository</code> 기반으로 설계하는 게 유리

### ✅ 마무리 요약

항목	<code>CrudRepository</code>	<code>JpaRepository</code>
정의	기본 CRUD 전용	CRUD + 페이징/정렬 + 고급 기능
상속 계층	Repository → Crud	→ Paging → Jpa
반환 타입	Iterable 중심	List, Page 중심
실무 사용	거의 안 씀	✅ 대부분 이걸 씀
추천 여부	❌ 제한적	✅ 표준처럼 사용됨

## • @Query, Query Method

### 🔗 1. Query Method란?

메서드 이름만으로 자동으로 쿼리를 생성하는 Spring Data JPA의 기능

```
1 Optional<User> findByEmail(String email);
2 List<User> findByAgeGreaterThanAndStatus(String age, Status status);
```

➡ 이름을 분석해 Spring이 자동으로 JPQL을 만들어 실행

### ✅ 문법 구성

```
1 findBy + 필드명 + 조건 키워드
```

키워드	의미
<code>And</code> , <code>Or</code>	논리 연산
<code>Between</code>	범위
<code>LessThan</code> , <code>GreaterThan</code>	비교
<code>Like</code> , <code>Containing</code> , <code>Startswith</code>	문자열 검색

키워드	의미
<code>OrderBy</code>	정렬 조건
<code>Top3</code> , <code>First</code>	결과 개수 제한

## ■ 예제

```
1 List<User> findByAgeGreaterThanAndStatusOrderByCreateDateDesc(int age, Status
  status);
```

→ JPQL:

```
1 SELECT u FROM User u
2 WHERE u.age > :age AND u.status = :status
3 ORDER BY u.createdDate DESC
```

## ✂ 2. `@Query` 어노테이션

메서드 이름으로 표현하기 어려운 쿼리를 직접 JPQL 또는 Native SQL로 작성할 수 있게 해주는 어노테이션

### ■ 기본 JPQL 사용

```
1 @Query("SELECT u FROM User u WHERE u.email = :email")
2 Optional<User> findByEmail(@Param("email") String email);
```

### ■ Native SQL 사용

```
1 @Query(value = "SELECT * FROM users WHERE email = :email", nativeQuery = true)
2 Optional<User> findNativeByEmail(@Param("email") String email);
```

### ■ 조건/컬럼 제한

```
1 @Query("SELECT u.name FROM User u WHERE u.age > :age")
2 List<String> findNamesByAge(@Param("age") int age);
```

### ■ DML 쿼리 (insert/update/delete)

```
1 @Modifying
2 @Query("UPDATE User u SET u.status = :status WHERE u.id = :id")
3 void updateUserStatus(@Param("id") Long id, @Param("status") Status status);
```

✂ DML에는 반드시 `@Modifying` 붙여야 함



### 3. Query Method vs @Query 비교표

항목	Query Method	@Query
정의 방식	메서드 이름으로 정의	JPQL/SQL 직접 작성
단순 조건	✅ 매우 간편	가능하지만 비효율적
복잡한 조인/조건	❌ 불가 또는 매우 길어짐	✅ 자유롭게 작성 가능
결과 컬럼 제한	❌ 불편함	✅ SELECT name 등 자유
native SQL	❌	✅ <code>nativeQuery = true</code>
유지보수성	이름 길어지면 어려움	SQL 가독성 유지 가능
런타임 오류	메서드명 오류 시 컴파일 실패	쿼리 문법 오류는 런타임에 발생

#### 실무 설계 팁

상황	방법
단순 조회 (id, email, status 등)	✅ Query Method
복잡한 조건, 조인, 동적 쿼리	✅ @Query or QueryDSL
정렬, 페이징	<code>findByStatusOrderByCreatedDateDesc(Pageable pageable)</code>
네이티브 성능 최적화	<code>@Query(nativeQuery = true)</code> 또는 별도 Repository

#### 실전 예시 종합

```
1 public interface UserRepository extends JpaRepository<User, Long> {
2
3     // Query Method
4     Optional<User> findByEmail(String email);
5
6     List<User> findByStatusAndAgeGreaterThan(Status status, int age);
7
8     // JPQL 직접 작성
9     @Query("SELECT u FROM User u WHERE u.status = :status ORDER BY u.createdDate
10 DESC")
11     List<User> findRecentUsers(@Param("status") Status status);
12
13     // Native SQL
14     @Query(value = "SELECT * FROM users WHERE name LIKE %:name%", nativeQuery =
15 true)
16     List<User> searchByName(@Param("name") String name);
17 }
```

```

16 // 업데이트 쿼리
17 @Modifying
18 @Query("UPDATE User u SET u.status = 'DELETED' WHERE u.lastLogin < :cutoff")
19 void softDeleteInactive(@Param("cutoff") LocalDateTime cutoff);
20 }

```

## ✅ 마무리 요약

항목	설명
Query Method	간단한 조건 쿼리는 메서드 이름만으로 처리
@Query	복잡한 조건, 다중 조인, native SQL이 필요할 때 사용
선택 기준	단순 = Query Method, 복잡 = @Query or QueryDSL
정렬/페이징	Pageable 함께 사용 가능
성능 최적화	반드시 실행 SQL을 로그로 확인 (spring.jpa.show-sql: true)

## 페이징과 정렬

### 🧩 1. 페이징(Paging)이란?

대용량 데이터를 한 번에 가져오지 않고,  
클라이언트가 요청한 특정 페이지(범위)만큼만 조회하는 기법

### 📦 2. 정렬(Sorting)이란?

데이터를 지정한 기준(필드)에 따라 오름차순 / 내림차순 정렬하여 가져오는 기능

### 📐 3. 핵심 인터페이스

#### ✅ Pageable

- 요청 정보를 담는 객체
- 페이지 번호, 페이지 크기, 정렬 정보를 포함

```
1 PageRequest.of(페이지번호, 페이지크기, Sort)
```

- 페이지 번호는 0부터 시작!

#### ✅ Page<T>

- 조회 결과를 담는 객체
- 총 페이지 수, 현재 페이지, 전체 데이터 개수 등의 메타 정보 포함

## 💡 4. 리포지토리 사용 예시

```
1 public interface UserRepository extends JpaRepository<User, Long> {
2
3     Page<User> findByStatus(Status status, Pageable pageable);
4 }
```

## ■ 5. 컨트롤러 코드 예시

```
1 @GetMapping("/users")
2 public Page<User> getUsers(
3     @RequestParam(defaultValue = "0") int page,
4     @RequestParam(defaultValue = "10") int size,
5     @RequestParam(defaultValue = "createdAt,desc") String[] sort
6 ) {
7     Pageable pageable = PageRequest.of(page, size, Sort.by(sortDirection(sort),
8         sortField(sort)));
9     return userRepository.findByStatus(Status.ACTIVE, pageable);
10 }
11 private Sort.Direction sortDirection(String[] sort) {
12     return sort.length > 1 && sort[1].equalsIgnoreCase("asc") ? Sort.Direction.ASC :
13     Sort.Direction.DESC;
14 }
15 private String sortField(String[] sort) {
16     return sort[0];
17 }
```

## 🔧 6. 정적 페이징 예시

```
1 Pageable pageable = PageRequest.of(0, 20, Sort.by("name").ascending());
2 Page<User> page = userRepository.findByStatus(Status.ACTIVE, pageable);
3
4 List<User> content = page.getContent(); // 현재 페이지 데이터
5 int totalPages = page.getTotalPages(); // 총 페이지 수
6 long totalElements = page.getTotalElements(); // 전체 데이터 수
7 boolean isFirst = page.isFirst(); // 첫 페이지 여부
```

## 📦 7. 정렬 예시

```
1 Sort.by("name").ascending()
2 Sort.by("createdAt").descending()
3 Sort.by(Sort.Order.asc("age"), Sort.Order.desc("createdAt"))
```

## 8. 커스텀 정렬과 DTO 매핑

```
1 @Query("SELECT new com.example.dto.UserDto(u.name, u.email) FROM User u WHERE u.status = :status")
2 Page<UserDto> findDtoByStatus(@Param("status") Status status, Pageable pageable);
```

`Page<T>`는 DTO에도 적용 가능. 단, `SELECT new ...` 구문 사용

## 9. 주의사항

항목	주의할 점
페이지 번호	0부터 시작 (1 아님!)
정렬 필드명 오타	런타임 예외 발생 (주의)
<code>Page&lt;User&gt;</code> vs <code>List&lt;User&gt;</code>	<code>Page</code> 는 페이징 메타 포함, <code>List</code> 는 단순 데이터
Count 쿼리	<code>Page</code> 반환 시 <b>자동으로 count 쿼리 실행됨</b> (성능 고려)

## 마무리 요약

항목	설명
<code>Pageable</code>	요청 정보 (page, size, sort)
<code>Page&lt;T&gt;</code>	응답 결과 + 메타 데이터
정렬 방법	<code>Sort.by("field").ascending()</code>
실무 추천	<code>Page&lt;T&gt;</code> + DTO projection + 정렬 필드 검증
JPA 조인 시	반드시 <code>DISTINCT</code> , <code>countQuery</code> 별도 지정 필요 (페이징 깨질 수 있음)

## Fetch 전략

### • Lazy vs Eager

#### 1. 개념 요약

로딩 방식	설명
Lazy(지연 로딩)	관계된 엔티티를 실제 사용할 때까지 로딩을 미룸
Eager(즉시 로딩)	엔티티가 로딩될 때 즉시 관계 객체도 함께 로딩

## 2. 언제 사용되는가?

JPA에서 @OneToMany, @ManyToOne, @OneToOne, @ManyToMany 관계를 정의할 때  
연관 엔티티를 어떤 시점에 로딩할지를 정하는 전략

## 3. 기본 설정

관계	기본 로딩 방식
@ManyToOne, @OneToOne	<b>EAGER</b> (즉시 로딩) ← 비추천
@OneToMany, @ManyToMany	<b>LAZY</b> (지연 로딩) ← 기본값이 좋음

📌 실무에선 모든 연관관계를 **LAZY로 명시적 설정**하는 것이 강력 추천

```
1 @ManyToOne(fetch = FetchType.LAZY)
2 private Team team;
```

## 4. 실전 예시

### 엔티티

```
1 @Entity
2 public class Member {
3     @Id @GeneratedValue
4     private Long id;
5     private String name;
6
7     @ManyToOne(fetch = FetchType.LAZY) // ← 여기!
8     private Team team;
9 }
```

```
1 @Entity
2 public class Team {
3     @Id @GeneratedValue
4     private Long id;
5     private String name;
6 }
```

### ✓ LAZY: 지연 로딩

```
1 Member member = entityManager.find(Member.class, 1L); // 쿼리 1번 (member만)
2 Team team = member.getTeam(); // 여기서 쿼리 1번 추가 실행됨
```

총 2번 쿼리 실행:

```
1 SELECT * FROM member WHERE id = 1;
2 SELECT * FROM team WHERE id = ?; -- 실제 사용할 때 실행됨
```

## ✓ EAGER: 즉시 로딩

```
1 Member member = entityManager.find(Member.class, 1L);
```

→ 자동으로 조인 쿼리 실행:

```
1 SELECT m.*, t.* FROM member m
2 LEFT JOIN team t ON m.team_id = t.id
3 WHERE m.id = 1;
```

## 🔥 5. N+1 문제 (Lazy 로딩 부작용 예)

```
1 List<Member> members = memberRepository.findAll(); // member 10개 조회
2
3 for (Member m : members) {
4     System.out.println(m.getTeam().getName()); // team 10번 조회!
5 }
```

→ 쿼리 11번 실행 (1 + 10)

➡ 이것이 **N+1 문제**

## 🔧 해결 방법

방법	설명
<code>JOIN FETCH</code>	JPQL에서 Fetch Join 사용
<code>@EntityGraph</code>	스프링 데이터 JPA에서 관계 필드 명시적 로딩
DTO Projection	필요한 필드만 조회하는 DTO 쿼리 사용
Batch Size 설정	LAZY 로딩 시 일괄 조회 ( <code>hibernate.default_batch_fetch_size</code> )

## 🧠 실무 설계 전략

설계 기준	권장 방식
모든 연관 관계	기본은 <code>LAZY</code> 로 명시 설정
단일 쿼리로 조인 조회	<code>fetch join</code> , <code>@EntityGraph</code>
조회 전용 DTO API	직접 JPQL, QueryDSL, native 쿼리로 필드만 조회

설계 기준	권장 방식
양방향 관계	무조건 <code>LAZY</code> , 순환 참조 주의 ( <code>toString()</code> , <code>JSON</code> 등)

## ✅ 마무리 요약표

항목	Lazy	Eager
정의	실제 접근 시 로딩	즉시 로딩
기본 적용	<code>@OneToMany</code> 등	<code>@ManyToOne</code> 등 (주의)
장점	성능 최적화, 선택적 로딩	편리함 (초보자용)
단점	N+1 문제 가능	불필요한 조인 과다
실무 권장	✅ 기본으로 사용	❌ 의도적으로만 사용

## 영속성 컨텍스트

### 🧩 1. 정의

엔티티를 영구 저장(`persist`)하는 환경이며,  
1차 캐시, 변경 감지, 트랜잭션 동기화 등을 담당하는 **JPA 내부 메모리 공간**

#### 📌 키워드:

- 메모리상의 DB 유사 구조
- 엔티티의 상태를 추적하고 캐시함
- 하나의 `EntityManager` 에 1개의 영속성 컨텍스트가 할당됨

### 🔄 2. 엔티티의 생명주기와 상태

상태	설명
비영속 ( <code>new</code> )	객체만 생성했지만 JPA가 관리하지 않음
영속 ( <code>managed</code> )	<code>persist()</code> 호출 후 JPA가 상태 추적
준영속 ( <code>detached</code> )	영속 상태였지만 더 이상 관리되지 않음 ( <code>detach</code> , <code>clear</code> , <code>close</code> )
삭제 ( <code>removed</code> )	<code>remove()</code> 로 삭제 예정 상태

### ⚙️ 3. 주요 기능 정리

기능	설명
✅ 1차 캐시	DB에서 불러온 엔티티를 메모리에 저장하여 재조회 시 재사용
✅ 동일성 보장	같은 트랜잭션 내에서 같은 ID의 엔티티는 항상 <b>같은 객체(==)</b>
✅ 변경 감지 (Dirty Checking)	엔티티 필드가 변경되면 자동으로 <b>UPDATE</b> 실행
✅ 지연쓰기 (Flush)	트랜잭션 종료 직전까지 SQL 실행 지연, 모아서 DB 반영
✅ 쓰기 지연 저장소	<code>persist()</code> 해도 바로 INSERT되지 않고, 커밋 시 일괄 처리됨

### 💡 4. 동작 흐름 예제

```
1 @Transactional
2 public void updateUser(Long id) {
3     User user = entityManager.find(User.class, id); // 1. 조회 → 1차 캐시에 저장
4     user.setName("변경된 이름"); // 2. 상태 변경 감지
5     // 3. 트랜잭션 커밋 시 flush → update 쿼리 자동 실행
6 }
```

➡ 개발자는 update 쿼리를 직접 작성하지 않아도,  
JPA가 **변경된 필드를 비교해서 SQL을 자동 생성**해줌.

### 💻 5. 1차 캐시 실전 예

```
1 User u1 = em.find(User.class, 1L); // DB 조회 → 1차 캐시에 저장
2 User u2 = em.find(User.class, 1L); // 1차 캐시에서 조회, 쿼리 X
3
4 System.out.println(u1 == u2); // true (동일 객체)
```

### 🔥 6. flush(플러시)

영속성 컨텍스트의 변경 사항을 DB에 반영하는 시점

시점	설명
트랜잭션 커밋 시	자동 호출 (가장 일반적)
JPQL 실행 전	쿼리 실행 전에 자동 flush
명시적 호출	<code>em.flush()</code> 직접 호출 가능

➡ flush는 DB에 **쓰기만 실행**하고, 트랜잭션은 여전히 **열려 있음**



## 7. detach, clear, close

메서드	기능
<code>em.detach(entity)</code>	특정 엔티티만 준영속화
<code>em.clear()</code>	모든 영속성 컨텍스트 비우기
<code>em.close()</code>	영속성 컨텍스트 종료 (EntityManager 소멸)

## 8. 실무에서 주의할 점

항목	설명
<code>em.find()</code> vs <code>em.getReference()</code>	전자는 즉시 로딩, 후자는 프록시 (Lazy)
변경 감지는 영속 상태에서에만	준영속 객체는 추적 불가
동일성 보장 = 성능 보장	같은 객체는 재조회 없이 바로 사용 가능
영속성 컨텍스트는 트랜잭션 단위로 관리	Spring에서 <code>@Transactional</code> 은 영속성 컨텍스트 단위 의미도 포함
<code>flush()</code> 는 쓰지만, <code>commit</code> 이 진짜 반영	단순 flush만으로 트랜잭션 종료되지 않음

## 마무리 요약표

개념	설명
영속성 컨텍스트	엔티티를 추적하고 저장하는 JPA 내부 메모리
1차 캐시	같은 엔티티 ID 조회 시 메모리에서 반환
변경 감지	Setter로 값 바꾸면 flush 시 자동 update
flush	변경 내용 DB 반영 (SQL 실행), 트랜잭션 유지됨
동일성 보장	같은 트랜잭션 안에서 <code>user == user</code> 항상 true
생명주기	new → persist() → managed → detach/remove

# 트랜잭션 처리

## • @Transactional

### ✖ 1. 정의

Spring Framework에서 제공하는 **선언적 트랜잭션 처리 어노테이션**  
메서드나 클래스에 선언하면, 해당 범위를 **하나의 트랜잭션 단위**로 실행함

즉,

- 메서드 진입 → 트랜잭션 시작
- 정상 종료 → `commit()`
- 예외 발생 → `rollback()`  
을 자동으로 처리해줌.

### 🎯 2. 동작 범위

선언 위치	적용 대상
클래스	클래스의 모든 <b>public 메서드</b>
메서드	해당 메서드만 트랜잭션 적용

```
1 @Transactional
2 public class OrderService {
3     public void placeOrder() {} // 트랜잭션 o
4 }
```

### ⚙ 3. 주요 속성

속성	기본값	설명
<code>propagation</code>	REQUIRED	전파 방식: 기존 트랜잭션이 있으면 참여
<code>isolation</code>	DEFAULT	DB 격리 수준
<code>readOnly</code>	false	읽기 전용 최적화
<code>rollbackFor</code>	RuntimeException	어떤 예외에 대해 롤백할지 지정
<code>timeout</code>	-1 (무제한)	제한 시간 초과 시 롤백

## 4. 트랜잭션 전파(propagation)

값	설명
REQUIRED	현재 트랜잭션 존재 시 참여, 없으면 새로 생성 (기본값)
REQUIRES_NEW	기존 트랜잭션 중단 후 새로운 트랜잭션 생성
MANDATORY	반드시 트랜잭션 있어야 함, 없으면 예외
NESTED	SavePoint로 하위 트랜잭션 생성 (JDBC 필요)
NEVER	트랜잭션 있으면 예외
SUPPORTS	트랜잭션 있으면 참여, 없으면 그냥 실행
NOT_SUPPORTED	트랜잭션 없이 실행 (중단 후 비트랜잭션 실행)

## 5. 예제

```
1 @Transactional
2 public void transferMoney(Account from, Account to, int amount) {
3     from.decrease(amount); // 변경 감지
4     to.increase(amount);    // 변경 감지
5     // 트랜잭션 커밋 시 update 자동 발생
6 }
```

## 6. 예외에 따른 롤백 규칙

예외 타입	롤백 여부 (기본)
RuntimeException	✅ 롤백됨
Error	✅ 롤백됨
Checked Exception	❌ 커밋됨

🔴 Checked Exception도 롤백하려면 `rollbackFor` 명시해야 함:

```
1 @Transactional(rollbackFor = Exception.class)
```

## 7. 내부 동작 구조 (프록시 기반 AOP)

- Spring은 `@Transactional` 이 붙은 객체를 프록시로 감싸고, 트랜잭션 시작/종료/롤백을 자동 삽입함.
- 이때, 같은 클래스 내 메서드 호출은 트랜잭션 적용되지 않음 (프록시 우회)

```
1 // 트랜잭션 적용 ❌  
2 this.innerMethod();
```

✅ 해결 방법: 메서드를 다른 Bean으로 분리 or 자기 자신을 주입

## ⚠ 8. 실무 주의사항 요약

항목	설명
private 메서드 ❌	프록시 적용 안 됨
self-invocation ❌	같은 클래스 안에서 내부 호출하면 트랜잭션 무시됨
readOnly=true	SELECT 전용 메서드에만 지정 → flush 최적화
@Transactional 은 Service Layer 중심에 선언	Controller/Repository에 선언 지양

## ✅ 마무리 요약

항목	설명
정의	선언적 트랜잭션 경계 지정
동작 시점	메서드 진입 시 트랜잭션 시작, 종료 시 commit/rollback
롤백 대상	기본은 RuntimeException, 명시적으로 변경 가능
readOnly	SELECT 전용 메서드에 flush 최소화 적용
내부 호출 ❌	프록시 구조로 인해 우회되므로 주의
권장 위치	Service 계층에서 사용, 핵심 유스케이스 단위로 나누기

## N+1 문제와 해결 전략

### • Fetch Join

#### ✖ 1. Fetch Join이란?

JPA에서 연관된 엔티티를 한 번의 쿼리로 함께 로딩하기 위한 JPQL의 조인 방식 중 하나

🔴 단순한 JOIN과는 다르게,  
연관 엔티티까지 한 번에 가져오며 영속성 컨텍스트에 함께 저장함

## 2. 기본 JOIN vs FETCH JOIN

구분	JOIN	FETCH JOIN
목적	조인 조건에만 사용 (조회 대상은 루트 엔티티)	연관 엔티티를 함께 조회하고 영속화
연관 객체 로딩	✗	✓
1차 캐시에 등록	✗ (루트만)	✓ (루트 + 연관 모두)
N+1 해결	✗	✓

## 3. 예시 엔티티

```
1 @Entity
2 public class Member {
3     @Id @GeneratedValue
4     private Long id;
5
6     private String name;
7
8     @ManyToOne(fetch = FetchType.LAZY)
9     private Team team;
10 }
```

## 4. Fetch Join 예제

```
1 @Query("SELECT m FROM Member m JOIN FETCH m.team")
2 List<Member> findAllWithTeam();
```

→ 실행되는 JPQL → SQL 변환 예시:

```
1 SELECT m.*, t.*
2 FROM member m
3 JOIN team t ON m.team_id = t.id;
```

✓ 결과:

- Member 와 Team 모두 한 번에 로딩
- member.getTeam() 호출 시 추가 쿼리 없음

## ❌ 5. Fetch Join이 없을 때 (N+1 문제 발생)

```
1 List<Member> members = memberRepository.findAll(); // 쿼리 1번
2
3 for (Member m : members) {
4     System.out.println(m.getTeam().getName()); // 쿼리 N번 추가 발생!
5 }
```

➡ 총 1 + N 번의 쿼리 실행

➡ 성능 저하 심각 → 반드시 **Fetch Join** 또는 **DTO 조회**로 해결해야 함

## 🔍 6. 다대일 / 일대다 Fetch Join

✅ 다대일 (@ManyToOne) → OK

```
1 @Query("SELECT m FROM Member m JOIN FETCH m.team")
2 List<Member> findAllWithTeam();
```

→ 다수의 Member에 대해 각각 1개의 Team만 연결되므로 중복 없음

⚠ 일대다 (@OneToMany) → 주의!

```
1 @Query("SELECT t FROM Team t JOIN FETCH t.members")
2 List<Team> findAllWithMembers();
```

→ Team이 여러 Member와 매핑될 경우, Team 객체가 중복 조회됨

➡ 해결 방법:

- DISTINCT 키워드 사용
- 페이징 불가능 (DB에서 join 후 메모리에서 페이징됨)

## ⚠ 7. 페이징과 Fetch Join

상황	처리 방법
단일 엔티티 페이징	문제 없음
@OneToMany + Fetch Join	❌ 페이징 깨짐 (in-memory 처리됨)
해결책	Batch Size 설정, DTO 직접 조회, 다단계 조회 분리

## ✓ 8. 실무 정리

용도	Fetch Join 여부
연관 엔티티를 자주 보여줄 때	✓ Fetch Join으로 한 번에 조회
단순 ID 조회	✗ Lazy 그대로 두기
페이징 + OneToMany	✗ Fetch Join 대신 DTO or Batch Size
readOnly API	DTO 프로젝션 or QueryDSL preferred

## 🔥 9. Fetch Join vs EntityGraph vs Lazy 비교

항목	Fetch Join	@EntityGraph	Lazy (기본)
제어 위치	JPQL	Repository 메서드 선언	엔티티 필드
조인 쿼리 직접 작성	✓	✗	✗
N+1 해결	✓	✓	✗
페이징 호환성	✗ (OneToMany)	✗	✓
가장 강력한 제어	✓	중간	약함

## ✓ 마무리 요약표

항목	설명
정의	연관 엔티티를 함께 로딩하는 JPQL 조인 방식
장점	N+1 문제 해결, 성능 최적화
단점	페이징 불가, 중복 데이터 가능성
사용 시점	Lazy + Fetch Join으로 필요한 경우만 즉시 조회
실무 팁	OneToMany는 Fetch Join 지양 → DTO or Batch Size로 대체

## • EntityGraph

### ✖ 1. 정의

JPA에서 연관된 엔티티를 로딩할 때 **Fetch** 전략을 정적으로 또는 동적으로 지정할 수 있는 어노테이션

즉, JPQL을 직접 작성하지 않아도

연관 필드를 지연로딩 → 즉시로딩(Fetch Join처럼) 바꿔주는 기능.

## 2. 왜 사용하는가?

문제	해결
<code>LAZY</code> 로 설정한 필드에서 N+1 문제 발생	<code>@EntityGraph</code> 로 필요한 연관 필드만 Fetch
Fetch Join은 JPQL에서만 사용 가능	EntityGraph는 <b>Repository</b> 메서드에 선언적 적용 가능
페이징 시 Fetch Join은 깨짐	EntityGraph는 <b>페이징 지원됨</b> (단, 다대일만)

## 3. 사용 방법

### ① 기본형: 속성 기반

```
1 @Entity
2 public class Member {
3     @Id @GeneratedValue
4     private Long id;
5
6     private String name;
7
8     @ManyToOne(fetch = FetchType.LAZY)
9     private Team team;
10 }
```

```
1 @Repository
2 public interface MemberRepository extends JpaRepository<Member, Long> {
3
4     @EntityGraph(attributePaths = {"team"})
5     List<Member> findAll(); // team도 함께 로딩됨
6 }
```

- `attributePaths`는 **LAZY** 필드 이름을 문자열로 명시
- 이 메서드를 호출하면 **Member + Team**을 Join Fetch처럼 가져옴

### ② JPQL + EntityGraph

```
1 @Query("SELECT m FROM Member m WHERE m.status = :status")
2 @EntityGraph(attributePaths = {"team"})
3 List<Member> findWithTeamByStatus(@Param("status") Status status);
```



### ✓ ③ Named EntityGraph 선언형 (복잡한 경우)

```
1 @Entity
2 @NamedEntityGraph(
3     name = "Member.withTeam",
4     attributeNodes = @NamedAttributeNode("team")
5 )
6 public class Member { ... }
```

```
1 @EntityGraph(value = "Member.withTeam")
2 List<Member> findByStatus(Status status);
```

## 💡 4. Fetch Join vs EntityGraph 비교

항목	Fetch Join	EntityGraph
사용 위치	JPQL 쿼리 내에서 직접	Repository 메서드 선언부
동적 제어	✗	✓ (attributePaths)
페이징 호환성	✗ (OneToMany 시 깨짐)	✓ (OneToOne, ManyToOne 기준)
유지보수성	JPQL에 의존	선언적이고 깔끔함
실무 적합성	복잡 조인, 최적 쿼리 직접 제어할 때	페이징 + 간단 연관 로딩할 때

## 🔍 5. EntityGraph 동작 예시

```
1 // 내부적으로 실행되는 SQL (Fetch Join과 유사)
2 SELECT m.*, t.*
3 FROM member m
4 LEFT JOIN team t ON m.team_id = t.id;
```

→ 하지만 JPQL 없이, 오직 메서드 선언부만으로 이 동작이 가능

## ⚠ 6. 실전 주의사항

주의	설명
attributePaths 오타 주의	필드 이름 문자열이기 때문에 컴파일 타임 검증 ✗
컬렉션 (@OneToMany)은 사용 제한	페이징과 함께 쓰면 안 됨 (Fetch Join과 동일)
복잡 조인에는 한계 있음	이때 JPQL + Fetch Join + DTO 조립 추천
동적 쿼리에선 JpaSpecificationExecutor와 통합 가능	QueryDSL 또는 Specification으로 조건 분기 가능

## ✅ 마무리 요약

항목	설명
정의	연관 엔티티를 선언적으로 Fetch Join처럼 로딩
목적	N+1 문제 해결, JPQL 없는 Fetch 전략 지정
사용 위치	Repository 인터페이스 메서드에 붙임
성능	Fetch Join 수준으로 빠름 (프록시 X)
페이징 지원	✅ (단일 연관 관계에 한함)
추천 사용	조회용 API, 페이징 처리 시 연관 엔티티 로딩

## • Batch Size

### 🔗 1. 정의

JPA에서 @ManyToOne, @OneToMany, @ElementCollection 등 Lazy 로딩 연관 컬렉션을 다수 조회할 때, N+1 문제를 1 + 1 쿼리로 줄이기 위해 사용하는 쿼리 튜닝 기법

### 🎯 2. 왜 사용하는가?

문제	설명
N+1 문제	루트 엔티티 N개 조회 → 연관 엔티티 N번 추가 조회
Lazy 로딩	@ManyToOne, @OneToMany 를 기본적으로 LAZY로 설정 시 발생
해결책	fetch join, EntityGraph, 또는 ➡ Batch Size

### 💡 3. 작동 원리

```
1 | List<Member> members = memberRepository.findAll(); // 10명
```

각 Member에 연관된 Team이 Lazy일 경우, 다음과 같이 SQL이 실행됨:

```
1 | SELECT * FROM member; -- 1회
2 | SELECT * FROM team WHERE id = 1; -- 10회
3 | ...
```

➡ 총 11회 쿼리 (1 + N)

✅ Batch Size 설정 시, 다음처럼 최적화됨:

```
1 SELECT * FROM team WHERE id IN (1, 2, 3, 4, 5, ..., 10); -- 딱 1회
```

Lazy 로딩이지만 JPA가 내부적으로 컬렉션을 모아서 IN 쿼리로 처리함

## ⚙️ 4. 설정 방법

### ✅ ① 전역 설정 (application.yml)

```
1 spring:
2   jpa:
3     properties:
4       hibernate.default_batch_fetch_size: 100
```

### ✅ ② 어노테이션 기반 (JPA 표준은 아님)

```
1 @OneToMany(mappedBy = "member")
2 @BatchSize(size = 50)
3 private List<Order> orders;
```

```
1 @ManyToOne(fetch = FetchType.LAZY)
2 @BatchSize(size = 20)
3 private Team team;
```

🔴 어노테이션보다 전역 설정이 실무에서 더 많이 사용됨

## 📊 5. 실전 예시

```
1 List<Member> members = memberRepository.findAll(); // 10명
2 for (Member m : members) {
3   System.out.println(m.getTeam().getName()); // Lazy Team 조회
4 }
```

- team\_id 10개를 보고
- `SELECT * FROM team WHERE id IN (?, ?, ..., ?)` 한 번만 실행됨

## 🔍 6. EntityGraph vs Batch Size

항목	EntityGraph	Batch Size
접근 방식	명시적 조인	지연 로딩 최적화
작동 시점	쿼리 실행 시	Lazy 필드 접근 시
페이징	✅ 사용 가능	✅ 사용 가능
DTO 조회	❌	❌

항목	EntityGraph	Batch Size
제어력	정적	런타임 동적
용도	자주 조회되는 필드 미리 로딩	Lazy 필드가 많을 때 N+1 줄이기

## ⚠ 7. 주의사항

항목	설명
컬렉션 크기가 크면 비효율	IN 절이 커져서 성능 저하 우려 있음
자동 batching은 Lazy 필드에만	Eager엔 적용 ❌
Hibernate 구현체에 의존	JPA 표준 아님 (Hibernate만 지원)
내부적으로 <code>IN (...)</code> 이므로 DB IN 절 제한 확인 필요	

## ✅ 마무리 요약

항목	설명
정의	Lazy 연관 필드를 IN 쿼리로 일괄 로딩
대상	<code>@OneToMany</code> , <code>@ManyToOne</code> , <code>@ElementCollection</code>
설정 위치	<code>hibernate.default_batch_fetch_size</code>
효과	$N+1 \rightarrow 1 + 1$ 쿼리로 최적화
실무 팁	Lazy 기본 + EntityGraph or Batch Size 혼합 사용 권장
권장 크기	일반적으로 50~200 사이 튜닝 필요