5. 데이터 처리 및 JPA 연동

Spring Data JPA 기초

Spring Data JPA는 **Spring과 JPA(Java Persistence API)**를 쉽게 연동하여
데이터베이스 접근을 자동화하고, 반복적인 CRUD 코드를 제거해주는 **추상화 계층 프레임워크**다.
즉, **JPA를 더 쉽고 빠르게 사용하는 Spring의 표준 방식**이다.

---

1. Spring Data JPA란?

> **Spring + JPA + Repository 패턴**을 결합한 ORM 데이터 접근 기술

- 인터페이스만 정의하면 Spring이 구현체를 자동 생성
- 반복적인 SQL 작성 없이도 CRUD, 페이징, 정렬, 쿼리 메서드 가능
- `Hibernate`, `EclipseLink` 등의 JPA 구현체 위에서 동작

---

2. 기본 구성 요소

구성 요소	설명
`Entity`	테이블과 매핑되는 클래스 (`@Entity`)
`Repository`	데이터 접근 계층 인터페이스 (`extends JpaRepository`)
`Service`	비즈니스 로직 계층, 트랜잭션 관리
`@Transactional`	데이터 변경 시 트랜잭션 적용
`application.yml`	DB 연결 설정, JPA 설정

---

3. 의존성 설정

Gradle

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    runtimeOnly 'com.h2database:h2' // or MySQL, PostgreSQL
}
```

---

4. application.yml 설정 예시

```yaml
spring:
  datasource:
    url: jdbc:h2:mem:testdb
    driver-class-name: org.h2.Driver
    username: sa
    password:
  jpa:
    hibernate:
      ddl-auto: update  # create, validate, none, update 등
    show-sql: true
    properties:
      hibernate:
        format_sql: true
```

---

5. Entity 클래스 정의

```java
@Entity
public class Member {

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

    private String name;
    private String email;

    // 생성자, Getter/Setter 생략
}
```

- `@Entity` : 테이블과 매핑
- `@Id` : 기본 키
- `@GeneratedValue` : 자동 증가 전략

---

6. Repository 인터페이스 정의

```java
public interface MemberRepository extends JpaRepository<Member, Long> {
    // 기본 제공: save, findById, findAll, delete 등
}
```

- Spring Data JPA는 이 인터페이스의 **구현체를 자동 생성**해준다
- `JpaRepository<T, ID>`를 상속하면 CRUD 기본 메서드를 모두 상속받음

---

7. 사용 예시

1) 서비스 계층

```
@Service
@RequiredArgsConstructor
public class MemberService {

    private final MemberRepository memberRepository;

    public Member save(String name, String email) {
        return memberRepository.save(new Member(name, email));
    }

    public List<Member> findAll() {
        return memberRepository.findAll();
    }
}
```

---

8. 쿼리 메서드 (Method 이름 기반 자동 쿼리 생성)

```
public interface MemberRepository extends JpaRepository<Member, Long> {
    List<Member> findByName(String name);
    boolean existsByEmail(String email);
    List<Member> findByNameContaining(String keyword);
}
```

Spring은 메서드 이름을 분석하여 JPQL을 자동 생성한다.

---

9. 페이징 & 정렬

```
Page<Member> findAll(Pageable pageable);
```

사용 예:

```
Pageable pageable = PageRequest.of(0, 10, Sort.by("name").descending());
Page<Member> members = memberRepository.findAll(pageable);
```

---

10. 트랜잭션 처리

- 읽기 전용: `@Transactional(readOnly = true)`
- 데이터 변경: `@Transactional`

```java
@Transactional
public void registerMember(MemberDto dto) {
    Member member = new Member(dto.getName(), dto.getEmail());
    memberRepository.save(member);
}
```

---

11. 정리

항목	설명
`@Entity`	테이블과 매핑되는 클래스
`@Id`	기본 키 필드 지정
`@GeneratedValue`	자동 증가 전략
`JpaRepository`	CRUD + 쿼리 메서드 제공
쿼리 메서드	메서드명으로 자동 JPQL 생성
트랜잭션	`@Transactional`로 보장
설정	`application.yml`로 DB + JPA 구성

결론

Spring Data JPA는 JPA를 실무에서 **더 쉽게 사용하게 해주는 고수준 프레임워크**이다.
기본 CRUD는 물론, 쿼리 메서드, 페이징, 정렬, 동적 쿼리(QueryDSL) 등도 폭넓게 지원한다.
Entity, Repository, Service 계층 구조를 정확히 설계하고, 필요한 쿼리는 메서드명 또는 JPQL로 선언함으로써 **SQL을 직접 작성하지 않아도 강력한 데이터 접근이 가능하다.**

Entity, Repository, Service 설계

Spring Boot에서의 데이터 처리 계층은 다음과 같은 **3단 구조(Entity → Repository → Service)**로 구성된다.
이 구조는 **비즈니스 로직, 데이터 저장소, 도메인 객체의 책임을 분리**함으로써 유지보수성과 확장성을 극대화한다.

---

1. 전체 설계 구조

```
[Client 요청]
   ↓
@Controller
   ↓
@Service        ← 비즈니스 로직, 트랜잭션 처리
   ↓
@Repository     ← 데이터베이스 연동 (JPA)
   ↓
@Entity         ← 테이블과 매핑되는 도메인 객체
```

---

2. Entity 설계

역할: DB 테이블과 1:1 매핑되는 객체
규칙: 기본 생성자 필수, `@Id` 필요, `equals/hashCode` 주의

```
@Entity
public class Member {

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

    @Column(nullable = false)
    private String name;

    private String email;

    protected Member() {}  // JPA 기본 생성자

    public Member(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // Getter/Setter 생략 가능 (롬복 권장)
}
```

---

3. Repository 설계

역할: 데이터 CRUD를 처리하는 인터페이스
기반: `JpaRepository<Entity, ID>`

```
public interface MemberRepository extends JpaRepository<Member, Long> {
    Optional<Member> findByEmail(String email);
    boolean existsByName(String name);
}
```

- Spring Data JPA가 자동으로 구현체를 생성
- `findByXxx`, `existsByXxx`, `countByXxx` 등 다양한 메서드 지원

---

4. Service 설계

역할: 비즈니스 로직 처리, 트랜잭션 관리, 여러 리포지토리 조합

```
@Service
@RequiredArgsConstructor
public class MemberService {

    private final MemberRepository memberRepository;
```

```
    @Transactional
    public Member register(String name, String email) {
        if (memberRepository.existsByName(name)) {
            throw new IllegalStateException("이미 존재하는 회원입니다.");
        }
        Member member = new Member(name, email);
        return memberRepository.save(member);
    }

    @Transactional(readOnly = true)
    public List<Member> findAll() {
        return memberRepository.findAll();
    }

    @Transactional(readOnly = true)
    public Member findById(Long id) {
        return memberRepository.findById(id)
                .orElseThrow(() -> new EntityNotFoundException("회원 없음"));
    }
}
```

- `@Transactional`을 통해 **트랜잭션 경계 설정**
- 읽기 전용 트랜잭션에는 `readOnly = true`로 성능 최적화

---

5. DTO와 응답 구조

```
public class MemberDto {
    private String name;
    private String email;

    public static MemberDto from(Member member) {
        return new MemberDto(member.getName(), member.getEmail());
    }
}
```

→ Service → Controller 간 데이터 전달 시 Entity 대신 DTO 사용 권장

---

6. Controller에서 사용 예시

```
@RestController
@RequestMapping("/api/members")
@RequiredArgsConstructor
public class MemberController {

    private final MemberService memberService;

    @PostMapping
    public ResponseEntity<MemberDto> register(@RequestBody MemberDto dto) {
```

```
        Member saved = memberService.register(dto.getName(), dto.getEmail());
        return ResponseEntity.status(HttpStatus.CREATED).body(MemberDto.from(saved));
    }

    @GetMapping("/{id}")
    public ResponseEntity<MemberDto> get(@PathVariable Long id) {
        return ResponseEntity.ok(MemberDto.from(memberService.findById(id)));
    }
}
```

---

7. 계층 분리의 이점

계층	책임	특징
Entity	도메인 모델	DB 구조를 반영
Repository	데이터 접근	인터페이스로 선언만
Service	비즈니스 처리	로직, 트랜잭션 담당
Controller	HTTP 처리	요청 매핑, DTO 변환 담당

- **관심사 분리:** 한 계층의 수정이 다른 계층에 영향을 최소화
- **테스트 용이성:** 단위 테스트와 Mock 주입에 유리
- **확장성:** 계층 간 연결을 유지하며 기능 확장 가능

---

8. 패키지 구조 예시

```
com.example.app
├── domain/
|   └── Member.java
├── repository/
|   └── MemberRepository.java
├── service/
|   └── MemberService.java
├── controller/
|   └── MemberController.java
├── dto/
|   └── MemberDto.java
```

결론

Spring Boot에서의 Entity, Repository, Service 설계는 **클린 아키텍처를 기반으로 한 책임 분리 구조**다.
데이터 구조(Entity), 저장소 인터페이스(Repository), 비즈니스 로직(Service)를 명확히 분리하면 **유지보수성, 확장성, 테스트 효율성**을 극대화할 수 있다.

CRUD 및 페이징 처리

Spring Data JPA를 이용한 **CRUD 및 페이징 처리**는 매우 간결하고 강력한 기능 중 하나다.
복잡한 SQL 작성 없이도 기본적인 데이터 저장, 조회, 수정, 삭제는 물론이고,
페이징, 정렬도 손쉽게 구현할 수 있다.

---

1. 기본 CRUD (Create / Read / Update / Delete)

`JpaRepository<Entity, ID>`를 상속하면 다음과 같은 기본 메서드를 자동으로 사용할 수 있다.

메서드	설명
`save(entity)`	저장 또는 수정 (id 값 존재 여부에 따라)
`findById(id)`	ID로 단건 조회
`findAll()`	전체 목록 조회
`delete(entity)`	엔티티 삭제
`deleteById(id)`	ID로 삭제
`count()`	총 개수 반환
`existsById(id)`	존재 여부 확인

예시

```
Member member = new Member("kim", "kim@example.com");
memberRepository.save(member); // CREATE

Optional<Member> findOne = memberRepository.findById(1L); // READ

member.setName("kim2");
memberRepository.save(member); // UPDATE

memberRepository.deleteById(1L); // DELETE
```

---

2. 페이징 처리

Spring Data JPA는 `PagingAndSortingRepository`를 상속한 `JpaRepository`를 통해 **자동 페이징 처리 기능**을 제공한다.

2.1 메서드 시그니처

```
Page<T> findAll(Pageable pageable);
```

2.2 Pageable 객체 생성

```
Pageable pageable = PageRequest.of(page, size, Sort.by("name").descending());
```

- `page`: 0부터 시작
- `size`: 페이지당 개수
- `Sort`: 정렬 필드 및 방향

2.3 Controller 예시

```
@GetMapping("/members")
public Page<MemberDto> list(@RequestParam(defaultValue = "0") int page,
                            @RequestParam(defaultValue = "10") int size) {
    Pageable pageable = PageRequest.of(page, size, Sort.by("id").descending());
    Page<Member> result = memberRepository.findAll(pageable);
    return result.map(MemberDto::from);
}
```

---

3. 응답 구조 (Page 객체)

`Page<T>`는 다음과 같은 정보를 함께 포함한다:

- `getContent()` → 현재 페이지 데이터 리스트
- `getTotalElements()` → 전체 항목 수
- `getTotalPages()` → 총 페이지 수
- `getNumber()` → 현재 페이지 번호
- `isFirst()`, `isLast()` → 시작/끝 여부

→ API 응답 시, 다음과 같은 구조가 자동 생성됨:

```
{
  "content": [...],
  "pageable": {...},
  "totalPages": 3,
  "totalElements": 27,
  "last": false,
  "first": true,
  ...
}
```

---

4. 정렬 처리

정렬은 `PageRequest` 생성 시 `Sort` 객체로 함께 전달:

```
Sort sort = Sort.by("createdAt").descending().and(Sort.by("name").ascending());
Pageable pageable = PageRequest.of(0, 20, sort);
```

---

5. 페이징 + 조건 검색

쿼리 메서드 또는 `@Query`를 사용할 수 있다.

```
Page<Member> findByNameContaining(String name, Pageable pageable);
```

→ `/members?name=kim&page=0&size=10`

---

6. 응답 DTO 매핑

`Page.map()`을 활용하여 Entity를 직접 노출하지 않고 DTO로 변환할 수 있다.

```
Page<MemberDto> dtoPage = entityPage.map(MemberDto::from);
```

---

7. 정리된 Controller 예시

```
@GetMapping("/api/members")
public ResponseEntity<Page<MemberDto>> getMembers(
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "10") int size) {

    Pageable pageable = PageRequest.of(page, size, Sort.by("id").descending());
    Page<MemberDto> result = memberRepository.findAll(pageable).map(MemberDto::from);

    return ResponseEntity.ok(result);
}
```

---

8. 결론

Spring Data JPA의 CRUD + 페이징 기능은 다음과 같은 장점을 가진다:

항목	장점
기본 메서드 제공	저장, 수정, 삭제, 조회 모두 구현 없이 사용 가능
페이징 지원	`Pageable` + `Page<T>` 객체로 구조화된 응답 가능
정렬 지원	`Sort` 객체 조합으로 다중 필드 정렬

항목	장점
DTO 매핑	`map()` 을 통한 안전한 데이터 변환
쿼리 메서드 통합	검색 조건 + 페이징 조합도 자연스럽게 처리 가능

JPA 쿼리 메소드 정의

Spring Data JPA에서는 **SQL이나 JPQL을 직접 작성하지 않고도**,
메서드 이름만으로 쿼리를 자동 생성하는 기능을 제공합니다.
이를 **쿼리 메서드(Query Method)**라 하며, `JpaRepository` 에서 아주 강력하게 활용됩니다.

---

1. 기본 구조

```
반환타입 findBy필드명조건(조건값)
```

> 예: `findByName(String name)`
> → JPQL: `SELECT m FROM Member m WHERE m.name = :name`

---

2. 쿼리 메서드의 규칙

키워드	설명	예시
`findBy`	조회 시작	`findByName`
`existsBy`	존재 여부	`existsByEmail`
`countBy`	개수 반환	`countByStatus`
`deleteBy`	삭제 수행	`deleteByCreatedBefore`
`readBy` / `queryBy`	`findBy` 와 동일	`readByName`

---

3. 비교 연산자

조건	예시 메서드	생성 JPQL
Equals	`findByEmail(String email)`	`email = :email`
Not Equals	`findByNameNot(String name)`	`name <> :name`
Greater Than	`findByAgeGreaterThan(int age)`	`age > :age`
Less Than	`findByAgeLessThan(int age)`	`age < :age`
Between	`findByAgeBetween(int min, int max)`	`age BETWEEN :min AND :max`
Like	`findByNameLike(String name)`	`name LIKE :name`

조건	예시 메서드	생성 JPQL
Contains	`findByNameContaining(String keyword)`	`name LIKE %:keyword%`
StartsWith	`findByNameStartingWith(String prefix)`	`name LIKE :prefix%`
EndsWith	`findByNameEndingWith(String suffix)`	`name LIKE %:suffix`
IsNull / IsNotNull	`findByDeletedAtIsNull()`	`deletedAt IS NULL`

---

4. 논리 연산자

연산자	설명	예시
`And`	모든 조건을 만족	`findByEmailAndName(String email, String name)`
`Or`	하나만 만족해도 가능	`findByEmailOrName(String email, String name)`

---

5. 정렬

정렬은 `Sort` 객체로 제어하는 것이 일반적이지만, 메서드 이름에도 포함 가능

```
List<Member> findByNameOrderByAgeDesc();
List<Member> findByStatusOrderByCreatedAtDescIdAsc();
```

---

6. 페이징 지원

```
Page<Member> findByName(String name, Pageable pageable);
List<Member> findByAgeGreaterThan(int age, Pageable pageable);
```

> `Pageable`을 파라미터로 넘기면 Spring Data JPA가 자동으로 페이징 쿼리를 생성함

---

7. 반환 타입

반환 타입	설명
`List<T>`	여러 개 반환 (기본)
`Optional<T>`	단일 객체 + null-safe
`T`	단일 객체 (nullable)
`Page<T>`	페이징된 결과
`Slice<T>`	다음 페이지 존재 여부만 판단 (총 개수 없음)

반환 타입	설명
`Stream<T>`	Java Stream 처리용 (주의: 트랜잭션 내에서 사용해야 함)

---

8. 예시 모음

```
List<Member> findByName(String name);
Optional<Member> findByEmail(String email);
List<Member> findByAgeBetween(int min, int max);
List<Member> findByNameContainingAndAgeGreaterThan(String keyword, int age);
Page<Member> findByStatus(String status, Pageable pageable);
boolean existsByEmail(String email);
long countByStatus(String status);
```

---

9. 조건 키워드 정리표

조건 키워드	의미
`Is`, `Equals`	=
`IsNot`, `Not`	!=
`GreaterThan`, `Gt`	>
`LessThan`, `Lt`	<
`Between`	BETWEEN
`Like`, `Containing`	LIKE '%...%'
`StartingWith`	LIKE '...%'
`EndingWith`	LIKE '%...'
`In`	IN (...)
`IsNull`, `Null`	IS NULL
`IsNotNull`, `NotNull`	IS NOT NULL

---

10. 주의사항

- 필드 이름은 정확히 Entity의 **속성명**과 일치해야 함
- **복잡한 조건**은 `@Query` 또는 **QueryDSL** 사용 권장
- 메서드명이 너무 길어지면 → **명시적 쿼리로 전환**

---

11. 예외 및 확장

1) 잘못된 필드명

```
findByNames() // 오류: Entity에 names 없음 → 컴파일 타임에서 감지 X
```

→ 테스트 코드로 반드시 검증해야 함

2) 커스텀 쿼리 혼합

```
@Query("SELECT m FROM Member m WHERE m.name = :name AND m.age >= :age")
List<Member> search(@Param("name") String name, @Param("age") int age);
```

결론

Spring Data JPA의 **쿼리 메서드 기능**은 SQL 없이도 복잡한 조건 검색을 **명확하고 선언적으로 표현**할 수 있게 한다.
메서드 이름만으로도 쿼리를 완성할 수 있다는 점에서 생산성과 가독성이 매우 뛰어나며,
조건이 복잡해질 경우 `@Query` 또는 QueryDSL로 이관하는 전략을 병행하면 안정적인 아키텍처 구성이 가능하다.

JPQL, Native Query

Spring Data JPA에서는 기본 쿼리 메서드 외에도 **복잡한 쿼리**가 필요한 경우,
JPQL(Java Persistence Query Language) 또는 **Native Query(SQL)**를 사용할 수 있다.
이를 통해 **조인, 서브쿼리, 집계, 복잡한 조건 검색** 등을 유연하게 처리할 수 있다.

---

1. JPQL이란?

JPQL은 **객체(Entity)를 대상으로 작성하는 SQL 유사 문법**이다.

특징	설명
테이블 기반이 아님	`SELECT * FROM member` → `SELECT m FROM Member m`
대소문자 구분	Entity 이름과 필드는 대소문자 구분
자동 매핑	결과가 자동으로 Entity에 매핑됨
DB 종속성 없음	Hibernate, EclipseLink 등과 독립적 작동

---

2. JPQL 기본 예시

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

```
@Query("SELECT m FROM Member m WHERE m.age > :age ORDER BY m.name DESC")
List<Member> findByAgeGreaterThan(@Param("age") int age);
```

3. @Query 애노테이션

Spring Data JPA에서는 `@Query`로 JPQL 또는 Native SQL을 직접 작성할 수 있다.

문법 구조

```
@Query("SELECT e FROM Entity e WHERE e.prop = :param")
List<Entity> methodName(@Param("param") 타입 값);
```

---

4. Native Query

Native Query는 **DB에 직접 SQL을 작성하여 실행**하는 방식이다.
→ JPQL로 표현하기 어려운 쿼리, 복잡한 SQL 문법 사용 가능

```
@Query(value = "SELECT * FROM member WHERE email = :email", nativeQuery = true)
Member findByEmailNative(@Param("email") String email);
```

- `nativeQuery = true` 필수
- 결과 매핑은 Entity 기준으로 이루어짐
- Join + Alias 등 **DB 종속적 표현 가능**

---

5. 반환 타입

반환 타입	설명
Entity	JPQL/Native 모두 가능
DTO	JPQL에서 `new 패키지명.생성자()` 사용
`Object[], List<Object[]>`	Native Query 다중 열 대응
Interface 기반 Projection	JPQL에서는 불가, Native는 일부 지원
`Page<T>`	페이징 대응 가능 (`countQuery` 추가 권장)

---

6. DTO 직접 조회 (JPQL)

```
@Query("SELECT new com.example.dto.MemberDto(m.name, m.age) FROM Member m")
List<MemberDto> findAllDto();
```

- `MemberDto(String name, int age)` 생성자 필요
- 전체 경로 패키지명 반드시 포함

---

7. 페이징 + JPQL

```
@Query("SELECT m FROM Member m WHERE m.age > :age")
Page<Member> findByAge(@Param("age") int age, Pageable pageable);
```

→ 자동으로 count 쿼리가 추가됨
→ 필요 시 `countQuery`를 명시해 성능 최적화 가능

```
@Query(
  value = "SELECT m FROM Member m WHERE m.status = :status",
  countQuery = "SELECT COUNT(m) FROM Member m WHERE m.status = :status"
)
Page<Member> findByStatus(@Param("status") String status, Pageable pageable);
```

---

8. Native Query + DTO 매핑

Entity가 아닌 DTO로 직접 매핑하려면 **ResultSetMapping**, Projection, 또는 interface 기반 Projection 사용이 필요하며 복잡하고 유지보수 어려워서 보통은 **QueryDSL 또는 직접 매핑**을 선택함

예시 (interface 기반):

```
public interface MemberSummary {
    String getName();
    int getAge();
}
```

```
@Query(value = "SELECT name, age FROM member", nativeQuery = true)
List<MemberSummary> findAllSummary();
```

---

9. 주의 사항

항목	주의점
Native Query	DB 종속성 있음 (MySQL vs PostgreSQL 등)
DTO 매핑	생성자 방식은 필드명 오류가 컴파일 타임에 안 잡힘
`*` 사용 금지	JPQL에서는 `SELECT *` 사용 불가
join 시 alias 필수	`SELECT m FROM Member m JOIN m.team t` 식으로 작성

---

10. 정리 비교

항목	JPQL	Native Query
대상	Entity 기준	테이블 기준
포터블	O	X (DB 종속적)
자동 매핑	Entity 또는 DTO 생성자	Entity 또는 Projection
사용 난이도	쉽고 선언적	강력하지만 복잡함
성능	Hibernate 최적화 대상	직접 튜닝 가능
페이징 지원	O	O (countQuery 수동 작성 권장)

결론

Spring Data JPA에서는

- **간단한 조건 검색 → 쿼리 메서드**
- **중간 복잡도 → @Query(JPQL)**
- **복잡한 쿼리 또는 성능 튜닝 → Native Query 또는 QueryDSL**
  이라는 흐름으로 쿼리 전략을 선택하는 것이 가장 합리적이다.

지연 로딩 vs 즉시 로딩

JPA에서 엔티티 간 연관 관계를 맺을 때,
연관된 엔티티를 언제 로딩할지를 결정하는 전략이 바로 `지연 로딩(LAZY)`과 `즉시 로딩(EAGER)`이다.

이 전략은 **퍼포먼스, 쿼리 효율성, 설계 일관성**에 큰 영향을 미치기 때문에
정확한 이해와 명확한 기준이 매우 중요하다.

---

1. 기본 개념

전략	설명
지연 로딩 (LAZY)	연관 객체를 실제로 사용할 때 DB에서 조회함
즉시 로딩 (EAGER)	주 엔티티가 로딩될 때, 연관 객체도 함께 즉시 조회함

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

```
@ManyToOne(fetch = FetchType.EAGER)
private Team team;
```

---

2. 동작 예시

예: Member → Team 관계

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

    private String name;

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

```
Member member = memberRepository.findById(1L).get();
Team team = member.getTeam(); // 여기서 SELECT team 쿼리 발생 (LAZY)
```

→ LAZY는 `getTeam()` **호출 시점**에 추가 쿼리 발생
→ EAGER는 `findById()` **실행 시점에 Join 쿼리로 함께 조회**

---

3. 쿼리 발생 비교

1) 지연 로딩 (LAZY)

```
-- Member만 조회
SELECT * FROM member WHERE id = 1;

-- team 접근 시
SELECT * FROM team WHERE id = ?;
```

2) 즉시 로딩 (EAGER)

```
-- 즉시 조인 발생
SELECT * FROM member m
JOIN team t ON m.team_id = t.id
WHERE m.id = 1;
```

---

4. 디폴트 값

관계 종류	기본 FetchType
`@OneToOne`	EAGER
`@ManyToOne`	EAGER
`@OneToMany`, `@ManyToMany`	LAZY

> 그러나 실무에서는 **모든 연관 관계를 LAZY로 설정하고, 필요한 곳에서 명시적으로 Fetch Join 또는 EntityGraph를 사용하는 전략**이 가장 안전하고 성능 최적화에 유리함

---

5. 문제점: EAGER의 치명적인 단점

- **의도하지 않은 Join 발생**
- **N+1 문제 유발 위험 증가**
- 재사용성이 떨어짐 (Repository 메서드마다 쿼리 구조가 달라짐)

```
List<Member> members = memberRepository.findAll(); // EAGER이면 회원 수만큼 팀 조회 발생 가능성
```

---

6. 성능 최적화 전략: Fetch Join

지연 로딩(LAZY)을 기본으로 하되,
JPQL에서 필요한 경우만 Fetch Join을 통해 함께 조회한다:

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

→ 1개의 SQL로 Member + Team 함께 조회

---

7. 성능 비교 요약

항목	LAZY	EAGER
쿼리 발생 시점	연관 객체 접근 시	즉시 (조회 시점)
쿼리 개수	필요할 때만 발생	항상 함께 발생
성능 유연성	높음 (최적화 가능)	낮음 (의도하지 않은 쿼리 발생)
권장 여부	✅ 권장 (기본 전략)	❌ 비권장 (특히 OneToMany에서 치명적)

---

8. 결론 및 실무 가이드

전략	실무 기준
기본 설정	모든 연관관계 `FetchType.LAZY` 설정 권장
즉시 조회 필요 시	`@Query` + `JOIN FETCH`, 또는 `EntityGraph` 사용
한번에 여러 연관 조회	DTO로 직접 조회 or QueryDSL
N+1 문제 감지	Hibernate 쿼리 로그 + 테스트 데이터로 반드시 확인

결론

- **지연 로딩(LAZY)**은 성능, 설계 일관성, 유연성 면에서 **사실상의 표준**이다.
- **즉시 로딩(EAGER)**은 편리해 보이지만, **예상치 못한 쿼리 폭주, Join, N+1 문제를 일으킬 가능성이 매우 높다.**

N+1 문제와 해결 방법

N+1 문제는 JPA에서 **지연 로딩(LAZY)** 설정과 연관된 **성능 문제 중 가장 빈번하고 치명적인 문제**다.
초기에는 눈에 띄지 않지만, 데이터가 많아질수록 **쿼리 수가 기하급수적으로 증가**하여 시스템 성능을 심각하게 저하시킨다.

---

1. N+1 문제란?

> **1개의 쿼리(N=1)**로 기본 데이터를 조회한 후,
> 각 데이터의 **연관 객체를 조회하기 위해 추가로 N개의 쿼리**가 발생하는 문제

예시: Member → Team (ManyToOne)

```
@Entity
public class Member {
    @ManyToOne(fetch = FetchType.LAZY)
    private Team team;
}

```

```
List<Member> members = memberRepository.findAll(); // 팀 정보도 사용함
for (Member m : members) {
    System.out.println(m.getTeam().getName());
}
```

발생 쿼리 흐름:

```
1. SELECT * FROM member      -- 기본 조회 (1회)
2. SELECT * FROM team WHERE id = ?   -- 각 멤버마다 팀 조회 (N회)
3. ...
```

→ 총 **1 + N 쿼리 발생**

---

2. 왜 발생하는가?

- `@ManyToOne(fetch = LAZY)`로 설정 → 지연 로딩
- 엔티티를 루프 돌면서 연관 객체에 접근하면, **그 시점마다 SQL 추가 실행**
- 개발자는 코드 상 문제 없어 보이지만, **DB에서는 엄청난 부하 발생**

---

3. 실무에서의 피해

상황	결과
회원 1,000명 조회 → 각 팀 조회	**총 1,001개의 쿼리 발생**
단순 관리자 조회 화면	페이지 느려짐, 트래픽 급증, 비용 폭증
캐시 없음	DB 부하 급증
지연 인식	QA 이후 또는 실제 운영에서 문제 드러남

---

4. 해결 방법

✅ 1) Fetch Join (JPQL 기반 최적화)

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

→ `JOIN FETCH`를 사용하면 **연관 객체를 한 번에 함께 조회**
→ 쿼리: 단 **1회만 실행**

```
SELECT m.*, t.* FROM member m
JOIN team t ON m.team_id = t.id
```

✅ 2) EntityGraph (선언형 Fetch Join)

```
@EntityGraph(attributePaths = {"team"})
@Query("SELECT m FROM Member m")
List<Member> findAllWithTeam();
```

→ 복잡한 Join 없이도, **Fetch Join과 같은 효과**

✅ 3) DTO로 직접 조회 (가장 권장)

```
@Query("SELECT new com.example.MemberDto(m.id, m.name, t.name) FROM Member m JOIN m.team t")
List<MemberDto> findAllMemberDto();
```

→ **불필요한 Entity 객체 생성 없이 바로 필요한 필드만 조회**
→ **대규모 페이지, 리스트 뷰에 매우 적합**

✅ 4) BatchSize 설정 (컬렉션 해결용)

N+1 문제는 OneToMany에서도 발생
이를 위한 해결 방법:

```
@OneToMany(mappedBy = "member")
@BatchSize(size = 100)
private List<Order> orders;
```

또는 전체 설정:

```
spring.jpa.properties.hibernate.default_batch_fetch_size=100
```

→ 지정된 개수만큼 `IN (...)` 쿼리로 묶어서 조회

---

5. 정리 비교

방법	장점	단점
Fetch Join	한 번의 쿼리로 연관 객체 함께 조회	다중 컬렉션 join 어려움
EntityGraph	선언적 방식, 코드 가독성 높음	복잡한 조합 시 한계
DTO 직접 조회	가장 성능 최적화	정형화된 쿼리 작성 필요
BatchSize	컬렉션 N+1 대응	IN 쿼리이므로 결과가 메모리에 집중됨

---

6. 실무 권장 전략

상황	전략
단건 조회	LAZY + 필요 시 Fetch Join
리스트/테이블 조회	DTO 직접 조회
@OneToMany 리스트 반복 조회	LAZY + BatchSize or QueryDSL
Admin 페이지	DTO + `JOIN FETCH` or `EntityGraph`
페이징 + 컬렉션 Join	절대 Fetch Join ❌ → DTO 분리 조회

---

7. 탐지 방법

- **Hibernate SQL 로그 확인**
- 테스트 시 `@Transactional(readOnly = true)`로 조회하고 **쿼리 수를 로그로 추적**
- `p6spy`, `datasource-proxy` 활용 추천

결론

- **N+1 문제는 무조건 피해야 할 성능 리스크**
- 모든 연관관계는 **기본 LAZY** + **Fetch Join 또는 DTO 직접 조회** 전략이 안전하다
- **페이징 + Fetch Join**은 JPA에서 지원하지 않으며 **결과 왜곡**이 발생할 수 있음

DTO 패턴, Projection

Spring Data JPA에서 **DTO 패턴**과 **Projection**은 엔티티를 그대로 노출하지 않고,
외부에 필요한 데이터만 추려서 반환하거나 성능을 최적화하는 데 매우 중요한 기법이다.
특히, API 응답 설계나 쿼리 최적화에서 핵심적인 역할을 한다.

---

1. DTO (Data Transfer Object) 패턴이란?

> Entity를 외부에 노출하지 않고, 필요한 필드만 추려서 전달하는 **전용 응답/요청용 객체**

사용 목적

- 보안성 (민감 정보 보호)
- 유연성 (뷰나 API에 따라 유동적 설계)
- 유지보수성 (API와 DB 구조 분리)
- 성능 최적화 (불필요한 필드 제외)

---

2. DTO 패턴 구조 예시

Entity

```
@Entity
public class Member {
    @Id @GeneratedValue
    private Long id;
    private String name;
    private String email;
    private String password;
}
```

DTO

```
@Getter
@AllArgsConstructor
public class MemberDto {
    private Long id;
    private String name;
    private String email;

    public static MemberDto from(Member m) {
        return new MemberDto(m.getId(), m.getName(), m.getEmail());
    }
}
```

사용 예

```
@GetMapping("/members")
public List<MemberDto> getAll() {
    return memberRepository.findAll().stream()
        .map(MemberDto::from)
        .toList();
}
```

---

3. DTO 직접 조회 (JPQL 기반)

```
@Query("SELECT new com.example.dto.MemberDto(m.id, m.name, m.email) FROM Member m")
List<MemberDto> findAllDto();
```

- `new 패키지명.클래스명(...)` 형태
- DTO에 **생성자 필수**

---

4. Interface 기반 Projection

> JPA가 쿼리 결과를 Interface 구현체로 **자동 생성하여 매핑**

인터페이스 정의

```
public interface MemberProjection {
    String getName();
    String getEmail();
}
```

Repository 쿼리

```
List<MemberProjection> findBy();
```

→ JPA가 내부적으로 Proxy를 생성해서 인터페이스 구현체를 반환

---

5. Class 기반 Projection (Open Projection)

```
public class MemberDto {
    private String name;
    private String email;

    public MemberDto(String name, String email) {
        this.name = name;
        this.email = email;
    }
}
```

```
@Query("SELECT new com.example.MemberDto(m.name, m.email) FROM Member m")
List<MemberDto> findAllBy();
```

---

6. 동적 Projection

```
<T> List<T> findBy(Class<T> type);
```

사용 예:

```
memberRepository.findBy(MemberProjection.class);
memberRepository.findBy(MemberDto.class);
```

---

7. Native Query + Projection

Native Query에서도 Interface 기반 Projection 사용 가능:

```
@Query(value = "SELECT name, email FROM member", nativeQuery = true)
List<MemberProjection> findAllProjection();
```

⚠️ 단, 필드명과 인터페이스 메서드명이 정확히 일치해야 함

---

8. 정리: DTO vs Projection

항목	DTO 패턴	Projection
선언 방식	별도 클래스	인터페이스 / 클래스
데이터 가공	로직 가능	불가능 (Getter만 존재)
복잡한 쿼리	유리	복잡도 증가 시 부적합
Query 직접 매핑	필요 (`@Query`)	자동 추론 가능
성능	효율적 (필드 최소)	매우 효율적
유연성	높음	제한적

---

9. 실무 가이드

용도	권장 방법
단순 조회	Interface Projection
계산, 가공 필요	DTO 객체 패턴
API 응답 전용	DTO 패턴
복잡한 구조	DTO + JPQL or QueryDSL
Native SQL 사용	DTO 생성자 또는 Projection

결론

- **DTO 패턴은 API 응답이나 View 모델에 유연하게 대응**할 수 있는 **표준 설계 전략**
- **Projection은 쿼리 성능 최적화에 매우 유리**하지만, 구조가 고정되어야 하며 복잡한 연산에는 한계가 있다
- 실무에서는 **Entity 직접 노출을 지양**하고, 항상 **DTO 또는 Projection 기반 응답 설계**를 지향하는 것이 안정적이고 유지보수에 유리하다

QueryDSL, Jooq 개요

Spring 환경에서 복잡한 쿼리를 타입 안전하고 유연하게 작성할 수 있도록 도와주는 대표적인 도구는 **QueryDSL**과 **jOOQ**입니다.
두 도구 모두 SQL을 자바 코드로 추상화해주지만, **철학과 사용 방식이 크게 다르며**
프로젝트 특성에 따라 적절한 선택이 중요합니다.

✅ 1. QueryDSL 개요

🔹 무엇인가?

- **JPQL을 타입 안전한 방식으로 작성할 수 있게 해주는 DSL (도메인 특화 언어)**
- **JPA 위에서 동작**하며, JPQL을 대신하여 사용
- 코드 자동 생성 도구를 통해 엔티티 전용 Q클래스를 만들어 쿼리 빌딩

🔹 특징

항목	설명
기반	JPA 엔티티
언어	Java 코드 기반 DSL
장점	컴파일 시점 타입 체크, IDE 자동완성
용도	동적 검색, 조건부 조합, 복잡한 조회 쿼리
대표 클래스	`JPAQueryFactory`, `QMember`, `BooleanBuilder` 등

🔹 예시 코드

```
QMember m = QMember.member;

List<Member> result = queryFactory
    .selectFrom(m)
    .where(m.age.gt(18).and(m.name.contains("kim")))
    .fetch();
```

→ 위 코드는 다음 JPQL을 대체:

```
SELECT m FROM Member m WHERE m.age > 18 AND m.name LIKE '%kim%'
```

✅ 2. jOOQ 개요

🔹 무엇인가?

- **SQL 자체를 자바 코드로 완전히 표현**할 수 있도록 만든 DSL
- JPA가 아닌 **SQL 중심** 개발을 지향
- JDBC 기반으로 직접 SQL 실행 (즉, **ORM 아님**)

🔹 특징

항목	설명
기반	테이블/뷰 기반 (DB 스키마에서 자동 코드 생성)

항목	설명
언어	SQL 스타일 DSL (`.select()` , `.where()`)
장점	SQL 기능 100% 활용, DB 벤더별 쿼리 최적화
용도	고성능 DB 연산, 복잡한 조인/집계, 프로시저 호출 등
단점	JPA와 연동 어려움, 객체 중심 도메인 모델 아님

🔹 예시 코드

```
DSLContext create = DSL.using(connection, SQLDialect.POSTGRES);

List<Record> result = create.select()
    .from(MEMBER)
    .where(MEMBER.AGE.gt(18))
    .fetch();
```

✅ 3. QueryDSL vs jOOQ 비교

항목	QueryDSL	jOOQ
중심 철학	객체 중심 (Entity)	SQL 중심 (테이블)
동작 기반	JPA, Hibernate 등 ORM	JDBC 직접 실행
타입 안전	O	O
복잡한 쿼리	O (좋음)	O (매우 강력)
DB 종속성	X (JPQL 기반)	O (DB Dialect 필요)
페이징	Pageable 지원 (Spring Data 통합)	수동 처리
코드 생성	엔티티 기반 Q클래스	DB 스키마 기반 POJO
실무 활용	복잡한 조회/조건 검색	고성능 집계, 네이티브 쿼리 최적화
Spring 통합	강력 (Spring Data QueryDSL 지원)	미약 (통합 수작업 필요)

✅ 4. 선택 기준

조건	추천
JPA 기반 도메인 모델 중심 아키텍처	✅ **QueryDSL**
고성능 SQL, 복잡한 집계, 프로시저 사용	✅ **jOOQ**
API 서버 + 동적 검색 조건	✅ QueryDSL

조건	추천
Data Warehouse, OLAP, ETL	✅ jOOQ
SQL이 익숙하고 RDBMS 기능 극대화 원함	✅ jOOQ
Spring Data JPA 연동 편의성 중시	✅ QueryDSL

✅ 5. 결론

- `**QueryDSL**`은 JPA와의 통합성, 도메인 중심 개발에 최적화된 도구이며, **실무에서 가장 널리 사용됨**
- `**jOOQ**`은 SQL의 모든 기능을 Java 코드로 활용하려는 **SQL 중심 개발자에게 최고의 도구**
- 둘 다 강력하지만 철학이 다르므로, **아키텍처 방향과 팀의 역량**에 맞춰 선택해야 한다.