


인프런 - 실전! 스프링 부트와 JPA 활용1 - 웹 애플리케이션 개발

 2025-06-19 - 도메인 분석 설계

학습 주제

- 엔티티 설계 시 주의사항
- 테이블/컬럼명 자동 생성 전략

주요 개념 요약

항목	설명
엔티티에는 Setter 사용 지양	변경 포인트가 많아져 추적이 어렵고 유지보수성이 떨어짐
연관관계는 지연 로딩 (LAZY)	EAGER는 JPQL과 충돌 시 예측 불가능한 SQL + N+1 문제 발생 위험
컬렉션은 필드에서 초기화	Hibernate는 프록시로 대체하므로 NPE 방지 위해 초기화 필요 (new ArrayList<>())
Naming 전략	SpringPhysicalNamingStrategy → 카멜 케이스 → 스네이크 케이스 변환 등 자동 규칙 적용

실습 코드

```
@Entity
public class Member {

    @Id @GeneratedValue
    private Long id;

    private String name;

    @OneToMany(mappedBy = "member")
    private List<Order> orders = new ArrayList<>();

}

Member member = new Member();

// 영속화 전: 필드 초기화된 컬렉션
System.out.println(member.getOrders().getClass());
// → class java.util.ArrayList

em.persist(member);

// 영속화 후: Hibernate 프록시 컬렉션
System.out.println(member.getOrders().getClass());
// → class org.hibernate.collection.internal.PersistentBag
```

2025-06-21 - 웹 계층 개발

💡 학습 주제

- XToOne 관계는 fetch join으로 1회에 조회
- XToMany 관계는:
 - 1. @BatchSize or default_batch_fetch_size로 IN 절 최적화
 - 2. DTO를 루트/서브 쿼리 방식으로 나누어 조회
 - 3. Flat DTO로 join 후 메모리에서 groupBy 재조합

🧠 주요 개념 요약

항목	설명
변경 감지 (Dirty Checking)	영속성 컨텍스트에서 엔티티를 조회한 후 데이터를 수정하여 커밋 시점에 UPDATE SQL 실행
병합 (Merge)	준영속 상태의 엔티티를 영속 상태로 변경할 때 사용. 다만 모든 필드를 덮어쓰기 때문에 null 값이 반영될 위험 존재
변경 시 고려사항	Dirty Checking 방식이 부분 변경이 가능하고, 실수 가능성이 적으므로 권장됨

✍ 실습 코드

```
@Entity
public class Member {

    @Id @GeneratedValue
    private Long id;

    private String name;

    private String address;

}

//변경 감지 -> 이름 만 변경됨
Member member1 = em.find(Member.class, memberId);
member1.setName("AA");

//병합 사용 -> 새로 생성된 회원의 모든 필드로 변경 하므로 Address가 null로 변경된다.
Member member2 = new Member();
member2.setName("AA");
em.merge(member2);
```

📖 인프런 - 실전! 스프링 부트와 JPA 활용2 - API 개발과 성능 최적화

📅 2025-06-23 - API개발 고급 : ㄹ 지연로딩과 조회 성능 최적화

💡 학습 주제

- API개발시 성능 최적화
- xToOne(@ManyToOne, @OneToOne)의 관계에서의 조회 성능 최적화 방법

🧠 주요 개념 요약

항목	설명
InvalidDefinitionException	<code>com.fasterxml.jackson.databind.exc.InvalidDefinitionException: No serializer found for class org.hibernate.proxy.pojo.bytebuddy.ByteBuddyInterceptor</code> → LAZY 로딩으로 인해 Jackson이 Proxy 객체를 직렬화하려다 실패함
@JsonIgnore	Entity 반환 시 양방향 매핑이 있을 경우, 순환 참조로 인한 무한 루프 방지를 위해 필드에 <code>@JsonIgnore</code> 를 반드시 설정해야 함
N+1 문제 원인	예: Order 2건 조회 시, 각 Order의 연관된 Member/Delivery가 LAZY 로딩으로 각각 추가 쿼리를 유발함. EAGER 사용 시 더 많은 예측 불가능한 쿼리 발생 가능
N+1 해결 방법	대부분은 fetch join 으로 해결. 성능 병목 발생 시 Dto 조회 고려. 최후의 수단으로 Native SQL, <code>JdbcTemplate</code> 활용 가능

🖋️ 실습 코드

📌 1. 엔티티 설정

```
@Entity
public class Order {

    @Id @GeneratedValue
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "MEMBER_ID")
    private Member member;

    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "DELIVERY_ID")
    private Delivery delivery;
}

@Entity
public class Member {

    @Id @GeneratedValue
```

```
private Long id;

@OneToMany(mappedBy = "member")
private List<Order> orders = new ArrayList<>();
}
```

📌 2. Fetch Join 사용 예시

```
List<Order> orders = em.createQuery(
    "select o from Order o " +
    "join fetch o.member m " +
    "join fetch o.delivery d", Order.class)
.getResultList();
```

📌 3. DTO 직접 조회 예시 (필요한 필드만 조회)

```
List<OrderSimpleQueryDto> result = em.createQuery(
    "select new
    jpabook.jpashop.repository.ordersimpequery.OrderSimpleQueryDto(" +
    "o.id, m.name, o.orderDate, o.status, d.address) " +
    "from Order o " +
    "join o.member m " +
    "join o.delivery d", OrderSimpleQueryDto.class)
.getResultList();
```

📅 2025-06-29 - API개발 고급 : 컬렉션 조회 최적화

💡 학습 주제

- API 성능 최적화를 위한 엔티티 조회 전략
- @OneToMany 관계에서 발생하는 N+1 문제 해결 방법
- 컬렉션 페이징 최적화 전략 (@BatchSize, DTO 분할 조회 등)

🧠 주요 개념 요약

항목	설명
@OneToMany + LAZY	LAZY 설정 시 연관 엔티티를 순차적으로 조회하며 N+1 문제가 발생
@JsonIgnore 주의	양방향 참조로 인해 무한 루프 방지에 사용되나, Entity 직접 반환은 API 스펙이 불안정해지므로 DTO 사용을 권장
Fetch Join + distinct	SQL은 1번만 실행되지만 페이징이 불가능 (컬렉션 join이므로 중복 row 발생)

항목	설명
페이징 최적화 방법 #1	XToOne(@ManyToOne, @OneToOne)은 fetch join으로 조회하고, 컬렉션은 LAZY로 두고 @BatchSize나 hibernate.default_batch_fetch_size로 in절 처리
BatchSize 한계	DB의 in절 제한을 초과하면 오류 발생 (예: Oracle은 1000개 제한)
페이징 최적화 방법 #2	DTO로 루트 조회 후, 컬렉션을 개별 쿼리로 조회하여 매핑. 또는 flat DTO로 조회 후 메모리에서 그룹핑
실무 권장 순서	1) 엔티티 조회로 접근 → 2) BatchSize 최적화 → 3) DTO 분할 조회 → 4) Flat DTO → 5) Native SQL or JdbcTemplate

실습 코드

1. DTO 직접 조회 - N+1 방식

```

List<OrderQueryDto> result = findOrders();

result.forEach(o -> {
    List<OrderItemQueryDto> orderItems = findOrderItems(o.getOrderid());
    o.setOrderItems(orderItems);
});

private List<OrderQueryDto> findOrders() {
    return em.createQuery(
        "select new jpabook.jpashop.repository.order.query.OrderQueryDto(" +
        "o.id, m.name, o.orderDate, o.status, d.address)" +
        " from Order o" +
        " join o.member m" +
        " join o.delivery d", OrderQueryDto.class)
        .getResultList();
}

private List<OrderItemQueryDto> findOrderItems(Long orderId) {
    return em.createQuery(
        "select new jpabook.jpashop.repository.order.query.OrderItemQueryDto(" +
        "oi.order.id, i.name, oi.orderPrice, oi.count)" +
        " from OrderItem oi" +
        " join oi.item i" +
        " where oi.order.id = :orderId", OrderItemQueryDto.class)
        .setParameter("orderId", orderId)
        .getResultList();
}

```

2. DTO 직접 조회 - 컬렉션 일괄 조회 및 매핑

- 쿼리 수 최소화 (N+1 → 2회)
- 페이징 불가하지만 성능과 단순성은 우수

```

List<OrderQueryDto> result = findOrders();

Map<Long, List<OrderItemQueryDto>> orderItemMap =
    findOrderItemMap(toOrderIds(result));

result.forEach(o -> o.setOrderItems(orderItemMap.get(o.getOrderId())));

private List<Long> toOrderIds(List<OrderQueryDto> result) {
    return result.stream()
        .map(OrderQueryDto::getOrderId)
        .collect(Collectors.toList());
}

private Map<Long, List<OrderItemQueryDto>> findOrderItemMap(List<Long>
orderIds) {
    List<OrderItemQueryDto> orderItems = em.createQuery(
        "select new jpabook.jpashop.repository.order.query.OrderItemQueryDto("
+
        "oi.order.id, i.name, oi.orderPrice, oi.count)" +
        " from OrderItem oi" +
        " join oi.item i" +
        " where oi.order.id in :orderIds", OrderItemQueryDto.class)
        .setParameter("orderIds", orderIds)
        .getResultList();

    return orderItems.stream()
        .collect(Collectors.groupingBy(OrderItemQueryDto::getOrderId));
}

```

🔪 3. Flat DTO → 메모리 그룹핑 방식

- 중복 row로 인해 메모리 비용 증가
- 페이징 불가능 (모든 조인 결과를 메모리에 올린 뒤 재구성)
- 정렬과 필터링이 제한적

```

List<OrderFlatDto> flats = orderQueryRepository.findAllByDto_flat();

List<OrderQueryDto> result = flats.stream()
    .collect(Collectors.groupingBy(
        o -> new OrderQueryDto(o.getOrderId(), o.getName(), o.getOrderDate(),
            o.getOrderStatus(), o.getAddress()),
        Collectors.mapping(
            o -> new OrderItemQueryDto(o.getOrderId(), o.getItemName(),
                o.getOrderPrice(), o.getCount()),
            Collectors.toList()
        )
    ))
    .entrySet().stream()
    .map(e -> new OrderQueryDto(
        e.getKey().getOrderId(), e.getKey().getName(),
        e.getKey().getOrderDate(),
        e.getKey().getOrderStatus(), e.getKey().getAddress(), e.getValue()))
    .collect(Collectors.toList());

```

```
public List<OrderFlatDto> findAllByDto_flat() {
    return em.createQuery(
        "select new jpabook.jpashop.repository.order.query.OrderFlatDto(" +
        "o.id, m.name, o.orderDate, o.status, d.address, " +
        "i.name, oi.orderPrice, oi.count)" +
        " from Order o" +
        " join o.member m" +
        " join o.delivery d" +
        " join o.orderItems oi" +
        " join oi.item i", OrderFlatDto.class)
        .getResultList();
}
```

마무리

- JPA는 1:N 페치 조인 시 페이징이 불가하므로, 상황에 맞는 전략을 선택하는 것이 중요
- 실무에서는 다음의 우선순위를 따라야 함:

엔티티 조회 (Fetch Join XToOne) →
 @BatchSize 컬렉션 조회 →
 DTO 분할 조회 →
 Flat DTO →
 Native SQL / JdbcTemplate

2025-06-29 - API개발 고급 - 실무 필수 최적화


💡 학습 주제

- OSIV(Open Session In View) 개념 이해
- 실무 환경에서의 운영 전략 및 설정 방법

🧠 주요 개념 요약

항목	설명
Open Session In View (OSIV)	Hibernate의 OSIV와 JPA의 Open EntityManager In View는 사실상 동일한 개념으로, 트랜잭션 범위 외에도 영속성 컨텍스트(EntityManager)를 유지해 뷰 렌더링 시 LAZY 로딩을 허용함
spring.jpa.open-in-view 옵션	true (기본값): request ~ response 전체 사이클 동안 DB 커넥션과 영속성 컨텍스트 유지 false : 트랜잭션 종료 시점(@Transactional 종료)과 함께 DB 커넥션 반납
기본 경고 로그	open-in-view 옵션을 설정하지 않으면 스프링 부트 실행 시 기본값 true 로 동작하며 다음과 같은 경고 로그 출력됨: WARN ... spring.jpa.open-in-view is enabled by default.
Fetch Join + distinct의 부작용	컬렉션 fetch join 시 중복 row가 발생하여 페이징이 불가능해짐. 이는 OSIV와는 별개 이슈지만 자주 연계되므로 주의

항목	설명
Service 분리 전략 (Command-Query Responsibility)	핵심 도메인 로직은 XxxService, 화면 조회/쿼리 최적화는 XxxQueryService 등으로 분리하여 책임을 명확히 함 (CQRS-like 분리)

 실습 코드


 application.yml 설정

```
spring:
  jpa:
    hibernate:
      ddl-auto: create
    properties:
      hibernate:
        format_sql: true
        default_batch_fetch_size: 100
    open-in-view: false
```

✓ open-in-view: false 설정 시, 서비스 계층 내부에서 필요한 모든 데이터를 미리 조회(fetch)해야 함 ✓ 이후 컨트롤러/뷰 렌더링 시에는 더 이상 LAZY 로딩이 동작하지 않음


✓ 운영 전략 가이드

시스템 유형	OSIV 권장 설정	이유
B2C (고객-facing)	false	커넥션 자원 낭비 방지, 성능 이슈 방지
Admin/내부 백오피스	true 또는 선택적	데이터량 적고 빠른 개발이 필요할 경우 허용

 실무 팁


- open-in-view: false 설정 시 모든 쿼리는 서비스 계층 내부에서 명시적으로 처리되어야 함
- 컨트롤러에서 LAZY 로딩을 발생시키면 LazyInitializationException 예외 발생 가능
- 반드시 DTO 변환을 트랜잭션 내부에서 완료해야 함

인프런 - 실전! 스프링 데이터 JPA

 2025-06-30 - 공통 인터페이스 기능 : 공통인터페이스

 학습 주제

- JPA 공통 인터페이스 기능 확인
- @EnableJpaRepositories를 통한 위치 지정
- JpaRepository<T, ID> 분석

 주요 개념 요약

항목	설명
@EnableJpaRepositories	JavaConfig에서 JPA Repository의 스캔 경로 지정 가능. Spring Boot에서는 <code>@SpringBootApplication</code> 하위 패키지를 자동 인식하므로 생략 가능
JpaRepository	Spring Data JPA가 프록시 기반의 구현체를 런타임에 자동 생성 (<code>class jdk.proxy2.\$ProxyXXX</code>)
상속 구조 (v3.3.13 기준)	<p><code>JpaRepository</code> → <code>ListCrudRepository</code> → <code>CrudRepository</code> → <code>Repository</code></p> <p><code>JpaRepository</code> → <code>ListPagingAndSortingRepository</code> → <code>PagingAndSortingRepository</code> → <code>Repository</code></p>
주요 메서드	내부적으로 <code>EntityManager</code> 를 통해 <code>find()</code> , <code>save()</code> , <code>delete()</code> 등의 반복 로직을 추상화하여 제공함

실습 코드

1. @EnableJpaRepositories 사용 예시

```
@EnableJpaRepositories(basePackages = "jpabook.jpashop.repository")
```

2. JpaRepository 구조 (v3.3.13 기준)

- `ListCrudRepository`: CRUD 기능 제공
- `ListPagingAndSortingRepository`: 페이징 + 정렬 기능
- `QueryByExampleExecutor`: Example 객체 기반 동적 쿼리 생성 지원 (내부적 Criteria 생성)

```
public interface JpaRepository<T, ID>
    extends ListCrudRepository<T, ID>,
           ListPagingAndSortingRepository<T, ID>,
           QueryByExampleExecutor<T> {

    // ...

}
```

마무리

- `JpaRepository<T, ID>` 인터페이스를 상속하면 별도의 구현 없이 CRUD, 페이징, 정렬 기능 자동 제공
- Spring Data JPA는 런타임에 프록시 객체로 구현체를 자동 생성
- 기본 설정만으로도 생산성과 일관된 Repository 계층 구현 가능

17 2025-06-30 - 쿼리 메소드 기능 : 메소드이름 쿼리생성, JPA NamedQuery, @Query

학습 주제

- 메서드 이름 기반 쿼리 생성 규칙

- JPA NamedQuery 특징 및 사용법
- `@Query` 어노테이션의 특징과 활용

🧠 주요 개념 요약

항목	설명
메서드 이름 기반 쿼리 생성	<code>find</code> , <code>read</code> , <code>query</code> , <code>get</code> 등의 접두어 + <code>By</code> 를 조합하여 자동 쿼리 생성 예: <code>findByUsernameAndAgeGreaterThan()</code>
COUNT / EXISTS / DELETE	<code>countBy</code> , <code>existsBy</code> , <code>deleteBy</code> 등의 접두어 사용 가능
DISTINCT / LIMIT	<code>findDistinctBy</code> , <code>findTop3By</code> , <code>findFirst3By</code> 등의 키워드 사용 가능
JPA NamedQuery	엔티티 클래스에 <code>@NamedQuery</code> 를 선언하여 사용. 런타임 이전(앱 시작 시점)에 문법 오류 검출 가능
@Query	커스텀 JPQL을 정의할 수 있으며, DTO 조회도 가능. <code>@NamedQuery</code> 보다 유연하며 문법 오류도 앱 시작 시점에 확인 가능
파라미터 바인딩	이름 기반(<code>:name</code>) 또는 위치 기반(<code>?0</code>) 지원 <code>@Param("name")</code> 사용 권장 (가독성 및 안정성 향상)
IN 절 지원	<code>List<String></code> 과 같은 컬렉션 타입 파라미터로 <code>IN</code> 조건을 표현 가능
리턴 타입 처리	List 반환 시 결과가 없으면 빈 컬렉션 반환 도메인 객체 반환 시 null 반환 가능 → <code>Optional</code> 사용 권장
단건 조회 예외	조회 결과가 둘 이상일 경우 <code>NonUniqueResultException</code> 발생 가능성 있음 (스프링에서 <code>IncorrectResultSizeDataAccessException</code> 으로 추상화하여 내려줌)

🖋 실습 코드

📌 1. 메서드 이름 기반 쿼리

```
public interface MemberRepository extends JpaRepository<Member, Long> {
    // 두 쿼리는 동일하게 동작
    List<Member> findByUsernameAndAgeGreaterThan(String username, int age);
    List<Member> findMemberByUsernameAndAgeGreaterThan(String username, int
age);
}
```

📌 2. JPA NamedQuery 사용

- Spring Data JPA는 우선 NamedQuery를 먼저 탐색한 뒤 메서드 이름으로 생성한 쿼리 여부를 판단

```
// Entity
@Entity
```

```

@NamedQuery(
    name = "Member.findByUsername",
    query = "SELECT m FROM Member m WHERE m.username = :username"
)
public class Member {
    ...
}

// Repository
@Query(name = "Member.findByUsername")
List<Member> findByUsername(@Param("username") String username);

```

📌 3. @Query 기본 사용 예

```

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

```

📌 4. @Query DTO 조회

```

@Query("SELECT new study.datajpa.dto.MemberDto(m.id, m.username, t.name) " +
    "FROM Member m JOIN m.team t")
List<MemberDto> findMemberDto();

```

📌 4. 컬렉션 파라미터 (IN 절)

```

@Query("select m from Member m where m.username in :names")
List<Member> findByName(@Param("names") List<String> names);

```



마무리

- @Query는 NamedQuery의 장점을 포함하면서도 더 유연하므로 일반적으로 선호됨
- 메서드 이름 기반 쿼리는 간단한 조건 검색에 유용하지만, 복잡한 조건은 @Query 사용이 더 적합
- 단건 조회 시에는 반드시 Optional를 사용하여 null과 예외를 명확히 구분하는 것이 바람직함

17 2025-07-01 - 쿼리 메소드 기능 : JPA 페이징과 정렬



학습 주제

- JPA에서 페이징 처리 방법
- Page와 Slice의 차이점 및 사용처



주요 개념 요약

항목	설명
Pageable	페이징 및 정렬 정보를 담고 있는 객체. <code>PageRequest.of()</code> 를 통해 생성
Page	총 데이터 개수를 조회하는 추가 쿼리를 포함함. 전체 페이지 수 계산 가능
Slice	다음 페이지 존재 여부만 판단. 총 개수 조회 X → 무한 스크롤/더보기 UI에 적합
CountQuery 분리	복잡한 조인 쿼리에서 Count 성능이 저하될 경우 별도로 분리하여 성능 최적화
Hibernate 6의 Left Join 최적화	Spring Boot 3.x 이상에서 조건 없는 <code>left join</code> 은 제거됨 → 명시적으로 <code>fetch join</code> 사용 필요

실습 코드

1. PageRequest를 사용한 페이징 + 정렬

```
PageRequest pageRequest = PageRequest.of(0, 3, Sort.by(Direction.DESC, "username"));
```

2. P Page, Slice 인터페이스 구조

```
public interface Page<T> extends Slice<T> {}

public interface Slice<T> extends Streamable<T>{}
```

3. Count 쿼리 분리 예시

```
@Query(
    value = "select m from Member m",
    countQuery = "select count(m.username) from Member m"
)
Page<Member> findMemberAllCountBy(Pageable pageable);
```

4. DTO 변환 (Page 유지)

```
Page<Member> page = memberRepository.findByAge(10, pageRequest);
Page<MemberDto> dtoPage = page.map(m -> new MemberDto(m));
```

5. Hibernate 6의 Left Join 최적화 주의

```
@Query("select m from Member m left join m.team t")
Page<Member> findByAge(int age, Pageable pageable);
```

→ 조건이 없는 경우 Hibernate 6에서는 left join이 생략될 수 있음

```
select
  m1_0.member_id,
  m1_0.age,
  m1_0.team_id,
  m1_0.username
from
  member m1_0
```

📌 마무리

- Spring Boot 3.x 이상에서는 Hibernate 6의 쿼리 최적화 동작을 이해하고 있어야 함
- Page와 Slice의 사용 목적을 명확히 구분하여 사용
- 조인이 많은 경우 Count 쿼리 성능 병목이 생길 수 있으므로, 쿼리 분리 전략을 적용할 것

📅 2025-07-01 - 쿼리 메소드 기능 : 벌크성 수정 쿼리

💡 학습 주제

- JPA에서 벌크성 수정 쿼리 처리 방법
- 벌크성 수정 쿼리 사용 시 주의할 점

🧠 주요 개념 요약

항목	설명
벌크성 수정 쿼리	특정 조건에 맞는 데이터를 한 번에 일괄 수정/삭제 하는 쿼리
@Modifying	@Query와 함께 사용되어 DML 쿼리(UPDATE/DELETE)를 실행할 수 있게 해주는 어노테이션
@Modifying(clearAutomatically = true)	영속성 컨텍스트를 자동으로 clear 하여 DB와 캐시 불일치 문제를 방지 함
주의사항	벌크 쿼리는 DB에 직접 반영되고 영속성 컨텍스트에는 영향을 주지 않음 → flush() + clear() 또는 clearAutomatically 필수

✍ 실습 코드

📌 1. 스프링 데이터 JPA를 사용한 벌크성 수정 쿼리

```
@Modifying(clearAutomatically = true)
@Query("update Member m set m.age = m.age + 1 where m.age >= :age")
int bulkAgePlus(@Param("age") int age);
```

△ 예외 상황

- @Modifying 없이 DML 쿼리를 작성할 경우:

```
org.springframework.dao.InvalidDataAccessApiUsageException:
QueryExecutionRequestException: Not supported for DML operations
```

→ 반드시 @Modifying을 함께 명시할 것

마무리

- JPA의 벌크 쿼리는 영속성 컨텍스트와 동기화되지 않으므로 clear 작업이 필수
- @Modifying(clearAutomatically = true)는 flush/clear의 자동 대체 역할을 수행
- 벌크 연산 이후 조회 시, 캐시 불일치 문제를 방지하기 위해 컨텍스트 정리 작업을 명확히 할 것

2025-07-01 - 쿼리 메소드 기능 : @EntityGraph

💡 학습 주제

- @EntityGraph 사용 이유
- 사용 시 주의사항

🧠 주요 개념 요약

항목	설명
@EntityGraph	연관된 엔티티를 지연로딩이 아닌 fetch join 방식으로 즉시 로딩함. 기본적으로 LEFT OUTER JOIN을 사용
@NamedEntityGraph	엔티티에 정의된 이름 기반의 EntityGraph를 재사용할 수 있도록 설정
LAZY Loading	지연로딩으로 인해 발생하는 N+1 문제를 해결하는 방법 중 하나로 fetch join을 간편하게 적용
주의사항	단순 연관관계에는 유용하나, 복잡한 조인 조건이나 여러 단계 조인에는 직접 JPQL 작성 권장

🖋 실습 코드

📌 1. EntityGraph 기본 사용

```
// 기본 메서드 재정의 시 사용
@Override
@EntityGraph(attributePaths = {"team"})
List<Member> findAll();

// JPQL과 함께 사용하는 경우
@EntityGraph(attributePaths = {"team"})
@Query("select m from Member m")
List<Member> findMemberEntityGraph();

// 메서드 쿼리 이름 기반으로 간편하게 사용하는 경우
```

```
@EntityGraph(attributePaths = {"team"})
List<Member> findByUsername(String username);
```

📌 2. NamedEntityGraph 정의 및 사용

```
// 엔티티 내부에 정의
@NamedEntityGraph(
    name = "Member.all",
    attributeNodes = @NamedAttributeNode("team")
)
@Entity
public class Member {
    // ...
}

// NamedEntityGraph 사용
@EntityGraph("Member.all")
@Query("select m from Member m")
List<Member> findMemberEntityGraph();
```

📌 마무리

- @EntityGraph는 fetch join을 어노테이션으로 간편하게 지정할 수 있는 기능이며, 기본적으로 LEFT OUTER JOIN으로 동작함
- N+1 문제를 해결하는 용도로 적합하며, 단순한 연관관계에 한해 사용하는 것이 바람직함
- 복잡한 연관 조회 및 조건이 필요한 경우에는 직접 JPQL과 fetch join을 명시하는 것이 안전하고 명확함

📅 2025-07-01 - 쿼리 메소드 기능 : JPA Hint & Lock

💡 학습 주제

- JPA에서 Query Hint 및 Lock 기능 학습
- Hibernate 기반 힌트와 비관적 락의 사용법

🧠 주요 개념 요약

항목	설명
@QueryHints	SQL 힌트가 아닌 **JPA 구현체(Hibernate 등)**에 전달되는 힌트로, readOnly 설정 등을 통해 성능을 최적화할 수 있음
@Lock	트랜잭션 내에서 비관적 락 (예: SELECT FOR UPDATE) 또는 낙관적 락 을 설정할 수 있으며, LockModeType 에 따라 동작이 달라짐 (PESSIMISTIC_WRITE , OPTIMISTIC 등)
forCounting 옵션	Page 타입 반환 시, 내부적으로 실행되는 count 쿼리에도 동일한 힌트 를 적용하도록 설정하는 옵션

📌 실습 코드

📌 1. @QueryHints - 단건 조회 (readOnly)

```
@QueryHints(
    value = @QueryHint(name = "org.hibernate.readOnly", value = "true")
)
Member findReadOnlyByUsername(String username);
```

📌 2. @QueryHints - 페이징 + forCounting 옵션

```
@QueryHints(
    value = @QueryHint(name = "org.hibernate.readOnly", value = "true"),
    forCounting = true
)
Page<Member> findByUsername(String name, Pageable pageable);
```

📌 3. @Lock - 비관적 락 사용

```
@Lock(LockModeType.PESSIMISTIC_WRITE)
List<Member> findByUsername(String name);
```

→ 실행 시 다음 쿼리와 유사한 SELECT ... FOR UPDATE가 수행됨

📌 마무리

- @QueryHints는 주로 읽기 전용(readOnly) 트랜잭션 최적화 시 사용되며, Hibernate에 내부적으로 전달됨
- @Lock은 동시성 문제를 피하기 위해 DB 락을 트랜잭션 레벨에서 설정할 수 있음
- 페이지네이션 시 count 쿼리에도 힌트를 적용하려면 forCounting = true를 반드시 명시
- 락이나 힌트는 기능은 간단하지만, 실제 적용 시 트랜잭션, 락 전략, 구현체 동작을 정확히 이해하고 있어야 함

📅 2025-07-05 - 확장 기능 : 사용자 정의 리포지토리 구현

💡 학습 주제

- Spring Data JPA에서 사용자 정의 리포지토리(Custom Repository) 구현 방식 이해
- QueryDSL 적용 시 활용되는 패턴 학습

🧠 주요 개념 요약

항목	설명
명명 규칙	기본적으로 리포지토리 인터페이스명 + Impl을 구현 클래스명으로 인식하여 자동 연결
Interface 구성	사용자 정의 기능을 위해 별도의 인터페이스를 정의하고, 이를 리포지토리 인터페이스에 상속

항목	설명
Impl 대신 다른 이름 사용	<code>@EnableJpaRepositories</code> 또는 XML 설정을 통해 구현 클래스의 접미어(postfix)를 변경 가능

실습 코드

1. Impl 대신 다른이름으로 변경

```
<repositories base-package="study.datajpa.repository"
repository-impl-postfix="Impl" />
```

```
@EnableJpaRepositories(basePackages = "study.datajpa.repository",
repositoryImplementationPostfix = "Impl")
```

2. 사용자 정의 리포지토리 예제

1. 사용자 정의 인터페이스 생성

```
public interface MemberRepositoryCustom {
    List<Member> findMemberCustom();
}
```

2. 구현 클래스 작성

```
@RequiredArgsConstructor
public class MemberRepositoryImpl implements MemberRepositoryCustom {

    private final EntityManager em;

    @Override
    public List<Member> findMemberCustom() {
        return em.createQuery("select m from Member m", Member.class)
            .getResultList();
    }
}
```

3. 기존 리포지토리에 통합

```
public interface MemberRepository
    extends JpaRepository<Member, Long>,
        MemberRepositoryCustom {
}
```

📌 마무리

- 커스텀 쿼리(QueryDSL 등)를 적용할 때 유용하게 사용하는 패턴
- Spring Boot 2.x 이상에서는 Custom + Impl 방식 외에도 @EnableJpaRepositories 설정을 통해 유연하게 클래스명 지정이 가능함
- 실무에서는 복잡한 조회 로직이나 성능 최적화가 필요한 쿼리를 사용자 정의 리포지토리로 분리하면 유지보수와 테스트가 용이함

📅 2025-07-05 - 확장 기능 : Auditing

💡 학습 주제

- 테이블에 공통으로 정의되는 필드 처리 방법
- 등록일, 등록자, 수정일, 수정자 자동 처리

🧠 주요 개념 요약

항목	설명
@EnableJpaAuditing	Config 클래스에 설정하여 JPA Auditing 기능을 활성화
@EntityListeners	엔티티에 AuditingEntityListener를 등록하여 이벤트 기반 처리 활성화
@MappedSuperclass	엔티티가 공통 필드를 상속받도록 선언. 자식 엔티티에서 필드를 인식하게 함
AuditorAware	현재 사용자를 반환하는 컴포넌트. 생성자(createdBy), 수정자(modifiedBy) 자동 주입에 필요
BaseEntity vs BaseTimeEntity	날짜만 관리하는 BaseTimeEntity와 사용자 정보도 포함하는 BaseEntity로 역할을 분리하여 유연한 상속 가능

✍️ 실습 코드

📌 1. Auditing 예제

1. BaseTimeEntity 생성

```
@Getter
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public class BaseTimeEntity {
    @CreatedDate
    @Column(updatable = false)
    private LocalDateTime createdDate;
    @LastModifiedDate
    private LocalDateTime lastModifiedDate;
}
```

2. BaseEntity 생성

```

@Getter
@MappedSuperclass
public class BaseEntity extends BaseTimeEntity {
    @CreatedBy
    @Column(updatable = false)
    private String createdBy;
    @LastModifiedBy
    private String lastModifiedBy;
}

```

3. AuditorAware 및 @EnableJpaAuditing 설정

```

@EnableJpaAuditing
@SpringBootApplication
public class DataJpaApplication {

    public static void main(String[] args) {
        SpringApplication.run(DataJpaApplication.class, args);
    }

    @Bean
    public AuditorAware<String> auditorProvider() {
        // 실제 운영에서는 Spring Security 기반 사용자 정보 반환
        return () -> Optional.of(UUID.randomUUID().toString());
    }
}

```

마무리

- 공통 필드(Audit) 관리는 코드 중복 제거 및 유지보수에 효과적임
- BaseTimeEntity와 BaseEntity를 분리함으로써 유연한 설계가 가능
- 실무에서는 @CreatedBy, @LastModifiedBy에 Spring Security의 사용자 정보를 연동하여 실 사용자 기반의 감사 로그를 남기는 것이 일반적임

2025-07-05 - 확장 기능 : Web 확장

학습 주제

- 도메인 클래스 컨버터의 이해와 실무 적용 방향
- 웹 요청에서 Pageable을 통한 페이징 및 정렬 처리 방식 학습

주요 개념 요약

항목	설명
도메인 클래스 컨버터	HTTP 파라미터로 엔티티 객체를 직접 매핑. 실무에서는 엔티티 노출 대신 DTO 사용 권장
Pageable	컨트롤러 파라미터로 페이징 정보 자동 주입 (page, size, sort 등)

항목	설명
페이징 요청 파라미터	<code>page</code> 는 0부터 시작, <code>size</code> 는 한 페이지에 노출할 개수, <code>sort</code> 는 정렬 필드와 방향 지정 가능
글로벌 설정	<code>application.yml</code> 또는 <code>application.properties</code> 로 <code>default-page-size</code> , <code>max-page-size</code> 등을 설정
@PageableDefault	메서드 단위에서 개별 <code>Pageable</code> 기본값 설정 가능
@Qualifier	여러 개의 페이징 정보를 사용할 때 접두사로 구분 (예: <code>member_page</code> , <code>order_page</code>)
1부터 시작하는 페이지 처리	<code>one-indexed-parameters</code> 옵션은 응답값 정확도 문제로 비권장. 직접 처리 클래스를 구현하는 것이 좋음

실습 코드

1. Pageable 예제

1. Pageable 요청파라미터 예제

```
/members?page=0&size=3&sort=id,desc&sort=username,desc
```

2. Pageable 글로벌 설정

```
spring.data.web.pageable.default-page-size=20    # 기본 페이지 크기
spring.data.web.pageable.max-page-size=2000      # 최대 페이지 크기
```

3. Pageable 개별 설정

```
@GetMapping("/members_page")
public String list(
    @PageableDefault(size = 12, sort = "username",
                     direction = Sort.Direction.DESC)
    Pageable pageable
) {
    // pageable.getPageNumber() 등 활용
}
```

4. 다수 페이징 정보일 경우 sample URL

```
/members?member_page=0&order_page=1
```

```
@GetMapping("/members")
public String list(
    @Qualifier("member") Pageable memberPageable,
    @Qualifier("order") Pageable orderPageable
)
```

```

    } {
        // 각각의 페이징 처리
    }

```

마무리

- Spring MVC에서 Pageable을 활용하면 별도 파라미터 매핑 없이 간단하게 페이징 처리 가능
- 글로벌 설정과 @PageableDefault를 병행하면 일관성과 유연성을 모두 확보할 수 있음
- 실무에서는 도메인 직접 노출보다는 DTO를 통해 계층 간 분리를 유지하는 것이 권장됨
- 1부터 시작하는 페이지 처리는 별도의 커스텀 Resolver로 직접 구현하는 방식이 정확함

2025-07-05 - 스프링 데이터 JPA 분석 : 구현체 분석

💡 학습 주제

- 스프링 데이터 JPA의 구현체 내부 동작 방식 이해
- 새로운 엔티티 판단 방식과 성능 관련 고려 사항 정리

🧠 주요 개념 요약

항목	설명
SimpleJpaRepository	Spring Data JPA에서 제공하는 기본 공통 리포지토리 구현체 (save , findAll , delete 등 처리)
@Transactional	서비스 계층에서 트랜잭션이 없으면 Repository가 트랜잭션을 시작하며, 존재하면 전파 받아 실행
@Transactional(readOnly = true)	읽기 전용 트랜잭션 설정. 플러시 무시, 2차 캐시에 저장하지 않으며 읽기 성능 향상
새로운 엔티티 판단	id == null 또는 id == 0 (Primitive 타입 시 고려)인 경우 새 엔티티로 간주
merge() 단점	병합 시 DB를 먼저 조회하여 성능 저하 가능성이 있으며, 값이 없으면 새 엔티티로 인식
@CreateDate + Persistable	Persistable<T> 을 구현하면 isNew() 메서드를 통해 명확히 새 엔티티 여부 판단 가능

📝 실습 코드

📌 1. SimpleJpaRepository save 구현체 분석

```

@Repository
@Transactional(readOnly = true)
public class SimpleJpaRepository<T, ID> {

    @Transactional
    public <S extends T> S save(S entity) {
        if (entityInformation.isNew(entity)) {
            em.persist(entity);
        }
    }
}

```

```

        return entity;
    } else {
        return em.merge(entity);
    }
}
}

```

📌 2. Persistable을 통한 새로운 엔티티 판단

```

@Entity
@EntityListeners(AuditingEntityListener.class)
@NoArgsConstructor(access = AccessLevel.PROTECTED)
@Getter
public class Item implements Persistable<String> {

    @Id
    private String id;

    @CreatedDate
    private LocalDateTime createdAt;

    public Item(String id) {
        this.id = id;
    }

    @Override
    public boolean isNew() {
        return createdAt == null;
    }
}

```

📋 마무리

- Spring Data JPA는 내부적으로 persist/merge 여부를 판단하여 save 처리
- Persistable과 @CreatedDate 조합은 비표준 ID 생성 전략에서도 유용
- @Transactional(readOnly = true)는 쓰기 방지 및 성능 개선 효과가 있으므로 읽기 전용 API에 적극 활용
- merge보다는 persist 기반 설계가 성능적으로 바람직하므로 ID 생성 전략을 고려한 판단 로직 구현이 중요함

📅 2025-07-05 - 나머지 기능들: Specifications (명세), Query By Example, Projections, 네이티브 쿼리

💡 학습 주제

- Spring Data JPA에서 제공하는 동적 쿼리 기능 학습
- 각 기능의 사용법 및 실무 적합성 분석

🧠 주요 개념 요약

항목	설명
Specifications	JPA Criteria 기반 명세 객체 조합. 가독성이 낮고 복잡하여 실무에서는 QueryDSL 권장
Query by Example	Example 객체와 Matcher로 동적 쿼리 구성. INNER JOIN만 가능하며 = 연산만 지원
Projections	인터페이스 기반 필드 선택 조회. Root 엔티티가 아니면 최적화 어려움. Left Join으로 처리됨
동적 Projections	파라미터로 Projection 타입을 받아 동적으로 조회 가능
네이티브 쿼리	<code>@Query(nativeQuery = true)</code> 사용. 동적 쿼리 불가능하며 문법 오류 검증이 불가능함

실습 코드

1. Specifications 예제

```
// Probe 객체 구성
Member member = new Member("m1");
Team team = new Team("teamA");
member.setTeam(team);

Specification<Member> spec =
    MemberSpec.username("m1").and(MemberSpec.teamName("teamA"));

List<Member> result = memberRepository.findAll(spec);

// 명세 정의
public static Specification<Member> username(String username) {
    return (root, query, builder) ->
        builder.equal(root.get("username"), username);
}
```

2. Query By Example 예제

```
ExampleMatcher matcher = ExampleMatcher.matching()
    .withIgnorePaths("age");

Example<Member> example = Example.of(member, matcher);
List<Member> result = memberRepository.findAll(example);
```

3. Projections 예제

- 조회할 엔티티의 필드를 getter 형식으로 지정하면 해당 필드만 선택해서 조회(Projection)

```
public interface UsernameOnly {
    String getUsername();
}

public interface MemberRepository extends JpaRepository<Member, Long> {
    List<UsernameOnly> findProjectionsByUsername(String username);
}
```

}

📌 4. 동적 Projections 예제

```
public interface MemberRepository extends JpaRepository<Member, Long> {
    <T> List<T> findProjectionsByUsername(String username, Class<T> type);
}

// 사용
List<UsernameOnly> result =
    memberRepository.findProjectionsByUsername("m1", UsernameOnly.class);
```

📌 5. 네이티브 쿼리 예제

```
public interface MemberRepository extends JpaRepository<Member, Long> {

    @Query(value = "select * from member where username = ?", nativeQuery =
true)
    Member findByNativeQuery(String username);
}
}
```

📌 마무리

- Specifications, Query by Example, Projections, 네이티브 쿼리는 공식적으로 지원되지만 실무에서는 대부분 QueryDSL이 대체
- QueryDSL은 타입 안정성과 가독성, 복잡한 동적 조건 조합 등에서 유리함
- Projections는 최적화 제약이 있으며, 네이티브 쿼리는 유지보수 및 동적 처리에 어려움이 많음
- 기능 자체는 학습 가치가 있으나 실무에서는 QueryDSL 중심의 구현을 기본 전략으로 고려하는 것이 바람직