스프링 JPA

스프링 JPA 한눈에 요약

- **JPA(Java Persistence API)**: 자바 객체(엔티티)를 관계형 DB 테이블에 매핑하기 위한 표준 API(명세). 직접 SQL을 쓰지 않고도 객체를 저장/조회/갱신/삭제할 수 있다.
- **Hibernate** 등 구현체(Provider)가 실제 동작을 처리한다.
- **Spring Data JPA**: JPA를 더 쉽게(Repository 패턴, 쿼리 메서드) 쓰도록 도와주는 스프링 프로젝트.

핵심 개념과 용어 (기본)

- **Entity**: DB 테이블에 매핑되는 자바 클래스. `@Entity`로 표시.
- **PK (Primary Key)**: `@Id` — 엔티티를 구분하는 유일한 값.
- **Persistence Context (영속성 컨텍스트)**: `EntityManager`가 관리하는 1차 캐시. 같은 트랜잭션(또는 같은 EntityManager) 내에서 동일 PK의 엔티티는 같은 인스턴스로 관리된다.
- **영속 상태 (persistent)**: DB와 동기화 가능한 상태. `em.persist()` 또는 JPA가 조회한 엔티티는 영속 상태.
- **준영속(detached)**: 영속성 컨텍스트에서 분리된 엔티티.
- **Transient**: JPA가 모르는 일반 객체(저장 전).
- **Dirty Checking**: 트랜잭션 커밋 시점에 변경된 필드를 자동으로 감지해 UPDATE 쿼리를 만든다.
- **EntityManager**: JPA 핵심 API. 스프링에서는 보통 `@PersistenceContext`로 주입되지만 대부분 `JpaRepository` / `@Transactional`로 추상화해 사용.

주요 어노테이션 (엔티티 레벨)

```
@Entity
@Table(name = "users")          // 생략 가능
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) // AUTO, SEQUENCE, TABLE 등
    private Long id;

    @Column(nullable = false, length = 100)
    private String name;

    @Column(unique = true)
    private String email;
}
```

- `@Table`, `@Column`으로 세부 옵션 지정(이름, 길이, null, unique 등).
- `@GeneratedValue` 전략: `IDENTITY`, `SEQUENCE`, `TABLE`, `AUTO` (DB/환경에 따라 선택).

연관관계 매핑 (관계형 DB 매핑의 핵심)

관계: **OneToMany, ManyToOne, OneToOne, ManyToMany**

중요: **소유(owning) 쪽**(foreign key를 관리하는 쪽)과 **mappedBy(주인이 아님)** 개념

단방향/양방향 예시

예: Member(다) - Team(1)

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

    private String name;

    @OneToMany(mappedBy = "team") // 연관관계의 주인은 Member.team
    private List<Member> members = new ArrayList<>();
}
```

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

    private String username;

    @ManyToOne(fetch = FetchType.LAZY) // 기본은 EAGER for ManyToOne?
실제 JPA 기본은 EAGER for ManyToOne? (주의: ManyToOne의 기본은 EAGER)
    @JoinColumn(name = "team_id")
    private Team team;
}
```

- **주인(owner)**: foreign key를 가진 쪽 (`Member`의 `team` 필드).
- `mappedBy`는 주인이 아닌 쪽에 적는다(여기선 `Team.members`).
- 연관관계 변경 시 **양쪽 모두 값을 맞춰줘야 함** (`member.setTeam(team)` 후 `team.getMembers().add(member)`).

> 팁: 연관관계 편의 메서드 작성

```
public class Member {
    public void changeTeam(Team team) {
        this.team = team;
        team.getMembers().add(this);
    }
}
```

Fetch 전략: LAZY vs EAGER

- `FetchType.LAZY`: 연관 엔티티를 실제로 사용할 때 조회(지연 로딩). 성능상 권장.
- `FetchType.EAGER`: 엔티티 조회 시 즉시 함께 조회(즉시 로딩). 복잡한 연관관계에선 N+1 문제 유발.
- 권장: 대부분 연관관계는 **LAZY**로 설정하고, 필요한 경우 JPQL의 `fetch join`이나 `@EntityGraph`로 해결.

중요한 속성: cascade, orphanRemoval

- `cascade = CascadeType.ALL` 등은 부모의 작업(예: persist, remove)을 자식에게 전파.
- `orphanRemoval = true`: 부모에서 참조가 끊긴 자식 엔티티는 자동 삭제.
- 주의: cascade를 남발하면 의도치 않은 삭제/수정이 발생할 수 있음.

Repository (Spring Data JPA)

- 인터페이스 상속으로 CRUD 제공

```
public interface MemberRepository extends JpaRepository<Member, Long
> {
   List<Member> findByUsername(String username);
   @Query("select m from Member m where m.username = :name")
   List<Member> findByName(@Param("name") String name);
}
```

- **쿼리 메서드**: 메서드 이름으로 쿼리 자동 생성 (`findByXAndYOrderByZDesc`).
- `@Query`로 JPQL 또는 nativeQuery=true로 네이티브 SQL 사용 가능.
- 페이징/정렬: `Pageable`, `Page<T>` 지원.

JPQL / Criteria / Native Query

- **JPQL**: 엔티티 객체를 대상으로 하는 쿼리문. 예: `select m from Member m where m.username = :name`
- **Criteria API**: 타입 안전 쿼리 생성(복잡해서 실무에선 잘 안씀, QueryDSL 대체 사용 흔함).
- **Native Query**: 실제 SQL을 직접 사용.

트랜잭션과 영속성 컨텍스트

- `@Transactional` (보통 Service 레이어에): 메서드가 실행되는 동안 트랜잭션 활성화 → 영속성 컨텍스트 유지.

- 트랜잭션 커밋 시점에 `flush`(영속성 컨텍스트 → DB) 되고, Dirty Checking으로 변경 사항 적용.
- `Transactional(readOnly = true)`는 조회 전용으로 최적화(쓰기 금지, 일부 구현체에서 성능 개선).

흔한 문제·해결 (실무팁)

1. **LazyInitializationException**
   - 원인: 영속성 컨텍스트가 닫힌(트랜잭션이 끝난) 상태에서 LAZY 연관관계 접근.
   - 해결: 서비스 레이어에서 필요한 필드를 미리 조회(fetch join), DTO로 변환, `@Transactional` 범위 확장, 또는 `OpenSessionInView` (권장X).
2. **N+1 문제**
   - 예: 회원 100명을 조회 → 각 회원의 팀을 LAZY로 접근하면 팀을 추가 100번 조회.
   - 해결: `JOIN FETCH` 사용한 JPQL, `@EntityGraph`, 또는 배치 사이즈 (`hibernate.default_batch_fetch_size`) 설정.

```
@Query("select m from Member m join fetch m.team")
List<Member> findAllWithTeam();
```

3. **무분별한 cascade 사용** → 의도치 않은 삭제 발생. cascade는 생명주기를 함께할 때만 사용.
4. **양방향 연관관계에서 mappedBy 오류** → 연관 업데이트는 주인 쪽에서만 관리해야 함.
5. **대용량 업데이트/삭제**: JPQL bulk update/delete는 영속성 컨텍스트를 무시(반드시 `em.clear()` 또는 적절히 동기화 필요).

DTO 사용 (엔티티를 API로 그대로 노출하면 안 되는 이유)

- 엔티티는 DB와 밀접한 객체(지연 로딩·변경 추적 로직 포함). 외부로 직렬화하면 보안·성능 문제.
- 서비스 레이어에서 **DTO로 변환**해 반환.

```
public class MemberDto {
    private Long id;
    private String username;
    private String teamName;
    // 생성자, getter
}
```

- JPQL에서 `new` 키워드로 DTO 생성 가능:

```
select new com.example.MemberDto(m.id, m.username, t.name) from Mem
ber m join m.team t
```

예제: 간단한 CRUD 흐름 (Controller → Service → Repository)

```
@RestController
@RequiredArgsConstructor
public class MemberController {
    private final MemberService memberService;

    @PostMapping("/members")
    public ResponseEntity<Long> create(@RequestBody MemberCreateReq
uest req) {
        Long id = memberService.join(req.getUsername(), req.getTeamId());
        return ResponseEntity.ok(id);
    }
}

@Service
@RequiredArgsConstructor
public class MemberService {
    private final MemberRepository memberRepository;
    private final TeamRepository teamRepository;

    @Transactional
```

```
    public Long join(String username, Long teamId) {
        Team team = teamRepository.findById(teamId).orElseThrow();
        Member member = new Member(username, team);
        memberRepository.save(member); // 영속화
        return member.getId();
    }
}
```

성능 최적화 체크리스트

- 필요한 필드만 조회 (프로젝션/DTO)
- N+1 문제 감지 → fetch join / @EntityGraph / batch fetch
- 인덱스(자주 검색하는 컬럼에 인덱스 추가)
- 대용량 배치 처리 시 `saveAll()` + flush/clear 전략
- `@Transactional(readOnly = true)`로 조회 성능 개선
- 쿼리 실행계획(EXPLAIN) 확인 (네이티브 쿼리일 경우)

테스트와 마이그레이션

- **통합 테스트**: `@DataJpaTest`로 Repository 테스트.
- **DB 마이그레이션**: Flyway 또는 Liquibase 사용 권장(스키마 관리).

잠금(락) — 동시성 제어

- **Optimistic Locking**: `@Version` 필드 사용. 충돌 시 예외 발생 → 재시도 로직 필요.
- **Pessimistic Locking**: `@Lock(LockModeType.PESSIMISTIC_WRITE)` 또는 `entityManager.lock(...)` 사용.

실무에서의 권장 패턴 / Best Practices

- 엔티티는 **테이블 매핑과 비즈니스 메서드**(연관관계 변경 편의 메서드)만 포함. 비지니스 로직은 서비스에.
- Controller → Service → Repository 층 분리.
- 엔티티를 그대로 API로 반환하지 말고 DTO 사용.
- 연관관계는 LAZY로 두고, 필요한 경우 명시적으로 패치.
- 테스트 케이스를 작성해 N+1, Lazy Init 문제 확인.
- 쿼리는 먼저 JPQL/QueryDSL로 표현하고, 성능 이슈가 있을 때 네이티브 SQL 고도화.

자주 쓰는 코드 스니펫 모음

- 페이징:

```
Page<Member> page = memberRepository.findAll(PageRequest.of(0, 10, Sort.by("username").descending()));
```

- 부분 업데이트 (dirty checking 이용):

```
@Transactional
public void updateName(Long id, String name) {
    Member m = repository.findById(id).orElseThrow();
    m.setUsername(name); // 커밋 시점에 UPDATE 자동 발생
}
```

- bulk update 주의:

```
@Modifying
@Query("update Member m set m.status = :status where m.lastLogin < :cutoff")
int bulkUpdate(@Param("status") String status, @Param("cutoff") LocalDate cutoff);
```

이런 bulk 쿼리 후에는 영속성 컨텍스트와 DB가 불일치하므로 `@Modifying(clearAutomatically = true)` 또는 수동 `em.clear()` 필요.

결론 — 핵심 포인트 요약

- JPA는 객체와 DB를 연결해주는 강력한 도구지만 **영속성 컨텍스트**, **트랜잭션**, **LAZY 로딩**, **연관관계의 주인** 같은 개념을 반드시 이해해야 안전하고 성능 좋은 코드를 작성할 수 있다.
- 실무 팁: 엔티티는 가능한 단순하게, 서비스에서 비즈니스 로직 처리, DTO 사용, N+1 방지에 신경쓰기.