Day 14 (2019-03-20)
Spring Data JPA

1. Entity 정의

```
@Getter
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Table(name = "MEMBER")
@Entity
@SequenceGenerator(name = "MEMBER_SEQ", sequenceName = "MEMBER_SEQ")
public class Member {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO, generator = "MEMBER_SEQ")
    @Column(name = "MEMBER_KEY")
    private Long memberKey;

    @Column(name = "MEMBER_ID", unique = true, nullable = false)
    private String memberId;

    @Column(name = "MEMBER_NAME", nullable = false)
    private String memberName;

    @Column(name = "PHONE")
    private String phone;
}
```

2. Repository 생성

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

3. CRUD 예제

(1) 기본 Package 구성

(2) Entity 정의 및 Repository 정의

(3) DTO 구성

(4) Service 작성

(5) Controller 작성 (ExceptionHandler)

(6) Testing

4. 페이징과 정렬 및 검색조건 적용

(1) 페이징

DBMS Vendor 마다 다른 Paging 및 Sorting 쿼리 문법 그리고 각 Vendor 별 최적화 방법 외에도 전체 Element 개수를 구하기 위한 쿼리가 부수적으로 필요함

1) Oracle

```
SELECT * FROM (
    SELECT ROWNUM AS RNUM, Z.* FROM (
        SELECT * FROM OP_SAMPLE ORDER BY ID DESC
    ) Z WHERE ROWNUM <= 1000000
)
WHERE RNUM >= 999991;
```

cubeCTMS의 페이징 쿼리

```
 SELECT PAGE_NO, PAGE_CNT, ROW_CNT, SITE_KEY, PROJ_KEY, INVT_KEY, TIRB_KEY, INVT_NAME,
                TPX_PROP_COL_CODE.GET_LABEL(1,643,77965,'TYPE',TYPE,'') TYPE, DOCUMENTS, SUBMIT_DT,
SPO_REVIEW_DT,
                IRB_REVIEW_DT, TPX_PROP_COL_CODE.GET_LABEL(1,643,77965,'REVIEW_RESULT',REVIEW_RESULT,'')
REVIEW_RESULT, IRB_APPR_DT, DEL_FLAG, COMMENTS
            FROM (
                SELECT CEIL(ROWNUM/10) PAGE_NO, CEIL((COUNT(*) OVER())/10) PAGE_CNT, COUNT(*) OVER() ROW_CNT,
SQ.*
                FROM (
                    SELECT MS.*
                    FROM (
                        SELECT SITE_KEY, PROJ_KEY, INVT_KEY, TIRB_KEY, INVT_NAME, TYPE, DOCUMENTS, SUBMIT_DT,
                                                                SPO_REVIEW_DT, IRB_REVIEW_DT, REVIEW_RESULT,
IRB_APPR_DT, DEL_FLAG, COMMENTS
                        FROM (
                            SELECT
                                 TAB.TIRB_KEY
                            ,    TAB.INVT_KEY
                            ,    '['||TAB.INVT_ID||'] '||TAB.INVT_NAME INVT_NAME
                            ,    TAB.INVT_ID
                            ,    TAB.TYPE
                            ,    TAB.TYPE_OTHER
                            ,    TAB.SUBMIT_DT
                            ,    TAB.SPO_REVIEW_DT
                            ,    TAB.IRB_REVIEW_DT
                            ,    TAB.COMMENTS
                            ,    TAB.REVIEW_RESULT
                            ,    TAB.IRB_APPR_DT
                            ,    TAB.INPUT_TIME
                            ,    TAB.DEL_FLAG
                            ,    TAB.PROJ_KEY
                            ,    TAB.SITE_KEY
                            ,    TAB.DOCUMENTS DOCUMENTS
                            FROM TAB
                        )
                        WHERE 1 = 1

                            ORDER BY INVT_NAME, INPUT_TIME DESC
                    ) MS
                ) SQ
                WHERE 1 = 1
                AND TAX_PREF_FILTER.CHK_FILTER('DEL_FLAG',DEL_FLAG, 'CHAR', 'LIKE')= 'Y'

            )
            WHERE PAGE_NO  = 1  OR 1 IS NULL
               OR(PAGE_CNT < 1 AND PAGE_CNT = PAGE_NO)
```

2) Mysql

```
SELECT column FROM table
LIMIT 10 OFFSET 10;
```

covering index를 사용한 최적화

```
SELECT    *
FROM
    events
WHERE
    (date,id) > ('2010-07-12T10:29:47-07:00',111866)
        AND event = 'editstart'
ORDER BY date, id
LIMIT 10
```

3) SQL Server

```
DECLARE @PageNumber AS INT, @RowspPage AS INT
SET @PageNumber = 2
SET @RowspPage = 10


SELECT ID_EXAMPLE, NM_EXAMPLE, DT_CREATE
FROM TB_EXAMPLE
ORDER BY ID_EXAMPLE
OFFSET ((@PageNumber - 1) * @RowspPage) ROWS
FETCH NEXT @RowspPage ROWS ONLY;
```

(2) Pageable Interface

JPA에는 기본적으로 방언(Dialect)이 등록 되어있으며 DBMS 마다 최적화된 쿼리 기법을 잘 적용할 수 있도록 고안되어있음.

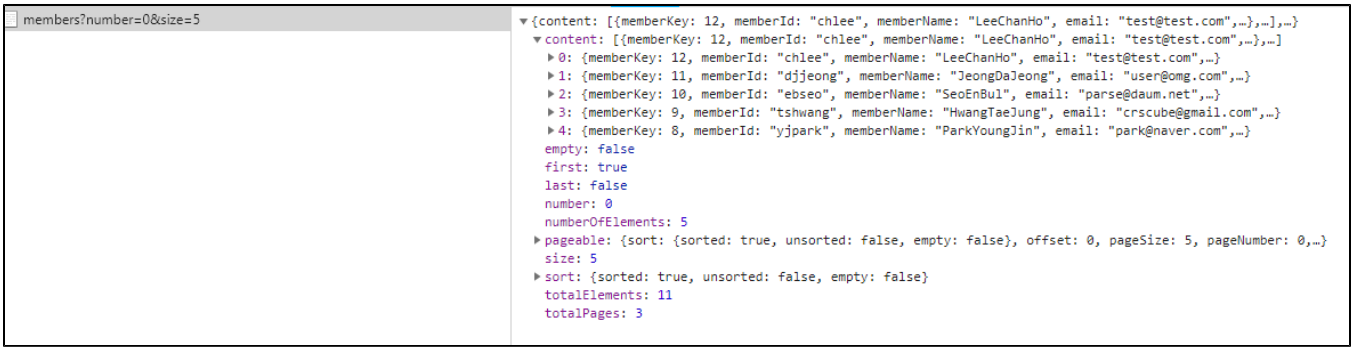
```
Query query = entityManager.createQuery("From Foo");
int pageNumber = 1;
int pageSize = 10;
query.setFirstResult((pageNumber-1) * pageSize);
query.setMaxResults(pageSize);
List <Foo> fooList = query.getResultList();
```

Spring Data JPA에서는 이것을 추상화하여 하나의 Interface와 그에 따른 구현체들로 정의하였고 Web application 영역 support를 위해 Command Object 형태로 사용할 수 있도록 함.

HandlerMethod에 Pageable 인터페이스만 선언하여 이것을 Repository의 findAll 메소드로 넘겨주기만 하면 귀찮은 페이징과 정렬 API 완료

```
/* MemberSearchController.java */
@GetMapping("/members")
public ResponseEntity<Page<MemberDto.MemberSearchResponse>> searchMembers(
        @PageableDefault(sort = {"memberKey"}, direction = Sort.Direction.DESC, value = 5) Pageable pageable,
        @RequestParam(name = "criteria", required = false, defaultValue = "") String searchOptionString,
        @RequestParam(name = "condition", required = false, defaultValue = "and") String condition) {
    return ResponseEntity.ok(this.memberSearchService.searchMembers(searchOptionString, condition, pageable));
}


/* MemberSearchService.java  */
public class MemberSearchService{
        public MemberSearchService(MemberRepository MemberRepository) {
            this.MemberRepository = MemberRepository;
        }

        public Page<MemberDSto.MemberSearchResponse> searchMembers(Pageable pageable) {
            return this.MemberRepository.findAll(predicate, pageable)
                    .map(MemberDto::asSearchResponse);
        }
}
```

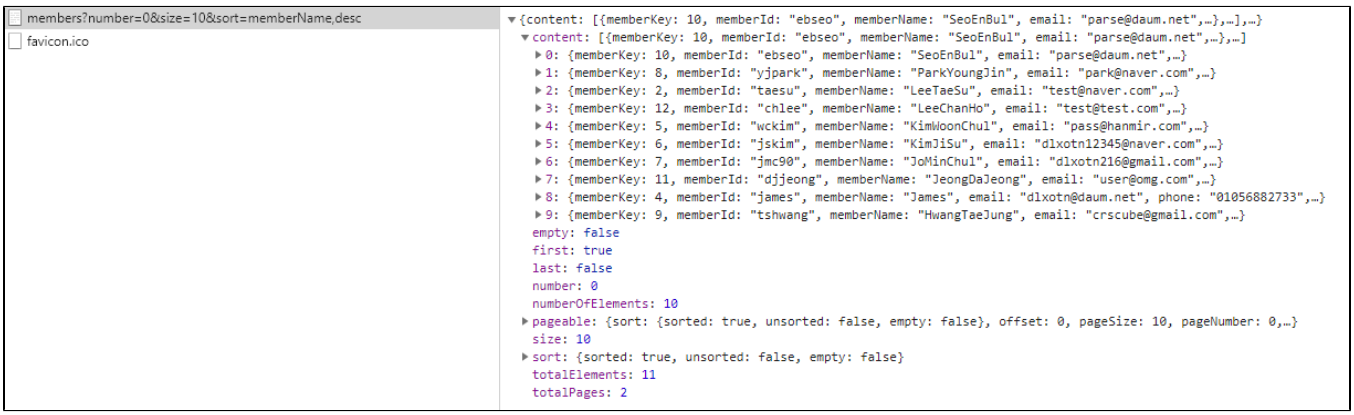
/members?number=0&size=5



발행된 쿼리(h2 dbms)

```
select member0_.member_key as member_k1_0_0_, team1_.team_key as team_key1_1_1_, member0_.email as email2_0_0_,
member0_.member_id as member_i3_0_0_, member0_.member_name as member_n4_0_0_,
         member0_.phone as phone5_0_0_, member0_.started_at as started_6_0_0_, member0_.team_key as
team_key7_0_0_, team1_.team_name as team_nam2_1_1_ from member member0_
inner join team team1_ on member0_.team_key=team1_.team_key
order by member0_.member_key desc limit ?
select count(member0_.member_key) as col_0_0_ from member member0_ inner join team team1_ on member0_.
team_key=team1_.team_key
```

/members?number=0&size=10&sort=memberName,desc



발행된 쿼리(h2 dbms)

```
select member0_.member_key as member_k1_0_0_, team1_.team_key as team_key1_1_1_, member0_.email as email2_0_0_,
member0_.member_id as member_i3_0_0_, member0_.member_name as member_n4_0_0_,
                member0_.phone as phone5_0_0_, member0_.started_at as started_6_0_0_, member0_.team_key as
team_key7_0_0_, team1_.team_name as team_nam2_1_1_ from member member0_
inner join team team1_ on member0_.team_key=team1_.team_key
order by member0_.member_name desc limit ?


select count(member0_.member_key) as col_0_0_ from member member0_ inner join team team1_ on member0_.
team_key=team1_.team_key
```

(3) 다양한 검색조건 지원

페이징과 정렬 외에도 필수 요소인 Filtering

동적인 검색 조건을 만들어 내는 방법으로 **QueryDSL**, Specification, Criteria를 사용할 수 있음

1) 정규표현식을 통해 Pattern matching

```
@GetMapping("Members")
public ResponseEntity<ApiResponse> searchMembers(Pageable pageable,
                                                 @RequestParam(value = "search", defaultValue = "", required =
false) String search,
                                                 @RequestParam(value = "condition", defaultValue = "and",
required = false) String condition) {
    MemberPredicateBuilder builder = new MemberPredicateBuilder();
    if (search != null) {
        Pattern pattern = Pattern.compile("(.+?)(=|!=|:|<|>|<=|>=)(*+?),", Pattern.UNICODE_CHARACTER_CLASS);
        Matcher matcher = pattern.matcher(SearchCriteria.replaceAllToCompilable(search) + ",");
        while (matcher.find()) {
            builder.with(matcher.group(1), matcher.group(2), matcher.group(3));
        }
    }
    BooleanExpression exp = builder.build(condition);

    return ResponseEntity.status(HttpStatus.OK)
            .contentType(MediaType.APPLICATION_JSON)
            .body(ApiResponse.fromSuccessResult(this.MemberSearchService.searchMembers(exp, pageable)));
}
```

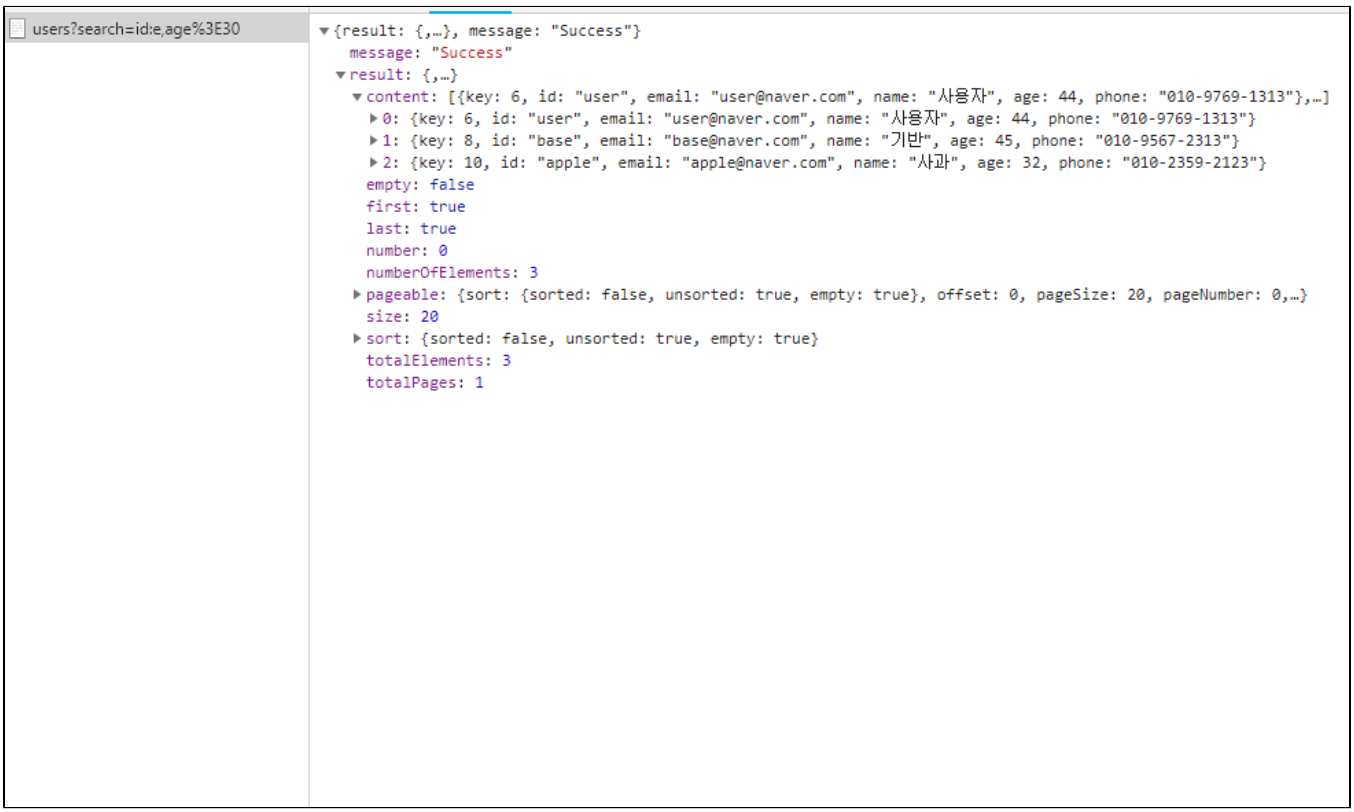
2) QueryDSL의 Predicate 이용

```
static class MemberPredicate {
    static BooleanExpression getPredicate(SearchCriteria criteria) {
        PathBuilder<Member> entityPath = new PathBuilder<>(Member.class, "Member");

        try {
            Double.parseDouble(criteria.getValue().toString());
            NumberPath<Integer> path = entityPath.getNumber(criteria.getKey(), Integer.class);
            int value = Integer.parseInt(criteria.getValue().toString());
            switch (criteria.getOperation()) {
                case "=":
                    return path.eq(value);
                case ">":
                    return path.gt(value);
                case ">=":
                    return path.goe(value);
                case "<":
                    return path.lt(value);
                case "<=":
                    return path.loe(value);
            }
        } catch (NumberFormatException e) {
            //Ignore
        }

        StringPath path = entityPath.getString(criteria.getKey());
        switch (criteria.getOperation()) {
            case "=":
                return path.equalsIgnoreCase(criteria.getValue().toString());
            case ":":
                return path.containsIgnoreCase(criteria.getValue().toString());
        }
        return null;
    }
}
```

3) 할 수 있는 것들

/members?criteria=id:e,age>30

→ id에 e가 포함되어있고 (id like 'e') age가 30 초과인 사용자 목록 조회



/members?criteria=memberId:a,phone:2723,startedAt<2019-03-20&condition=or&sort=memberName,desc

→ id에 a가 포함되거나 전화번호에 2723이 포함되거나 가입일이 2019년 3월 20일 이전인 사람에 대해 조회하며 이름에 대해 역순으로 정렬하여 조회

발행된 쿼리(h2 dbms)

```
select member0_.member_key as member_k1_0_0_, team1_.team_key as team_key1_1_1_, member0_.email as email2_0_0_,
member0_.member_id as member_i3_0_0_,
               member0_.member_name as member_n4_0_0_, member0_.phone as phone5_0_0_, member0_.started_at as
started_6_0_0_, member0_.team_key as team_key7_0_0_, team1_.team_name as team_nam2_1_1_ from member member0_
inner join team team1_ on member0_.team_key=team1_.team_key
where ?=? or lower(member0_.member_id) like ? escape '!' or lower(member0_.phone) like ? escape '!' or member0_.
started_at<?
order by member0_.member_name desc limit ?
2019-03-20 10:56:04.770 TRACE 11168 --- [nio-8080-exec-5] o.h.type.descriptor.sql.BasicBinder      : binding
parameter [1] as [BOOLEAN] - [false]
2019-03-20 10:56:04.770 TRACE 11168 --- [nio-8080-exec-5] o.h.type.descriptor.sql.BasicBinder      : binding
parameter [2] as [BOOLEAN] - [true]
2019-03-20 10:56:04.770 TRACE 11168 --- [nio-8080-exec-5] o.h.type.descriptor.sql.BasicBinder      : binding
parameter [3] as [VARCHAR] - [%a%]
2019-03-20 10:56:04.770 TRACE 11168 --- [nio-8080-exec-5] o.h.type.descriptor.sql.BasicBinder      : binding
parameter [4] as [VARCHAR] - [%2723%]
2019-03-20 10:56:04.770 TRACE 11168 --- [nio-8080-exec-5] o.h.type.descriptor.sql.BasicBinder      : binding
parameter [5] as [DATE] - [2019-03-20]

select count(member0_.member_key) as col_0_0_ from member member0_
inner join team team1_ on member0_.team_key=team1_.team_key
where ?=? or lower(member0_.member_id) like ? escape '!' or lower(member0_.phone) like ? escape '!' or member0_.
started_at<?
2019-03-20 10:56:04.773 TRACE 11168 --- [nio-8080-exec-5] o.h.type.descriptor.sql.BasicBinder      : binding
parameter [1] as [BOOLEAN] - [false]
2019-03-20 10:56:04.774 TRACE 11168 --- [nio-8080-exec-5] o.h.type.descriptor.sql.BasicBinder      : binding
parameter [2] as [BOOLEAN] - [true]
2019-03-20 10:56:04.774 TRACE 11168 --- [nio-8080-exec-5] o.h.type.descriptor.sql.BasicBinder      : binding
parameter [3] as [VARCHAR] - [%a%]
2019-03-20 10:56:04.774 TRACE 11168 --- [nio-8080-exec-5] o.h.type.descriptor.sql.BasicBinder      : binding
parameter [4] as [VARCHAR] - [%2723%]
2019-03-20 10:56:04.774 TRACE 11168 --- [nio-8080-exec-5] o.h.type.descriptor.sql.BasicBinder      : binding
parameter [5] as [DATE] - [2019-03-20]
```

사용자의 Search Filter를 유지하려면?

→ 특정 사용자가 조회 요청한 criteria와 sort 정보만 메뉴별로 저장하면 됨

5. N+1 Query 문제와 해결법

관계 Mapping에서 fetchType을 EAGER로 줄 경우 해당 Entity의 조회 시점에 연관된 Entity의 fetchOption을 보고 다시 쿼리를 발행
(ManyToOne 관계의 default fetchType은 EAGER)

```
@Getter
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Table(name = "MEMBER")
@Entity
@SequenceGenerator(name = "MEMBER_SEQ", sequenceName = "MEMBER_SEQ")
public class Member {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO, generator = "MEMBER_SEQ")
    @Column(name = "MEMBER_KEY")
    private Long memberKey;

    @Column(name = "MEMBER_ID", unique = true, nullable = false)
    private String memberId;

    @Column(name = "MEMBER_NAME", nullable = false)
    private String memberName;

    @Column(name = "PHONE")
    private String phone;


        @ManyToOne
        @JoinColumn(name = "TEAM_KEY")
        private Team team;

}
```

member의 findAll 메소드를 호출할 때 join을 걸어 조회하면 한번의 쿼리로 조회가 가능하지만 JPA는 이정도로 똑똑하지 못함.

따라서 Member 전체를 조회 후 **조회된 결과에 있는 Member가 속한 Team의 개수만큼 루프를 돌아 Team을 조회하는 쿼리를 날림**

```
select member0_.member_key as member_k1_0_, member0_.member_id as member_i2_0_, member0_.member_name as member_n3_0_, member0_.phone as phone4_0_, member0_.team_key as team_key5_0_ from member member0_ order by member0_.member_id asc limit ?
select team0_.team_key as team_key1_1_0_, team0_.team_name as team_nam2_1_0_ from team team0_ where team0_.team_key=?
binding parameter [1] as [BIGINT] - [5]
select team0_.team_key as team_key1_1_0_, team0_.team_name as team_nam2_1_0_ from team team0_ where team0_.team_key=?
binding parameter [1] as [BIGINT] - [4]
select team0_.team_key as team_key1_1_0_, team0_.team_name as team_nam2_1_0_ from team team0_ where team0_.team_key=?
binding parameter [1] as [BIGINT] - [3]
select team0_.team_key as team_key1_1_0_, team0_.team_name as team_nam2_1_0_ from team team0_ where team0_.team_key=?
binding parameter [1] as [BIGINT] - [6]
select team0_.team_key as team_key1_1_0_, team0_.team_name as team_nam2_1_0_ from team team0_ where team0_.team_key=?
binding parameter [1] as [BIGINT] - [2]
select team0_.team_key as team_key1_1_0_, team0_.team_name as team_nam2_1_0_ from team team0_ where team0_.team_key=?
binding parameter [1] as [BIGINT] - [1]
```

→ findAll을 통해 Member 전체를 조회했을 때 결과에 존재하는 Team이 1~6까지의 Team이므로 루프르 돌며 6번의 추가적인 쿼리가 실행 됨

따라서 관계 Mapping에서 fetchType은 무조건 LAZY로 설정 후 자주 사용되는 Entity에 대해서만 EAGER를 설정하는 방법을 권장함

ex) Board - BoardDetails 관계와 같이 1:1 관계이며 대부분 같이 붙어 다니는 Entity의 경우 fetchType을 EAGER로 선언 함

만약 사용자 조회 시 Team과 관련된 정보를 함께 조회가 필요하다면?

→ EAGER로 설정하지 않고 LAZY로 설정하여도 member.getTeam()을 호출 할 때마다 개별 쿼리가 발행되어 N+1 문제를 피할 수 없음

→ join fetch , Entity Graph 등의 대안이 있으며 Join을 수행하는 쿼리 한번으로 해결 가능

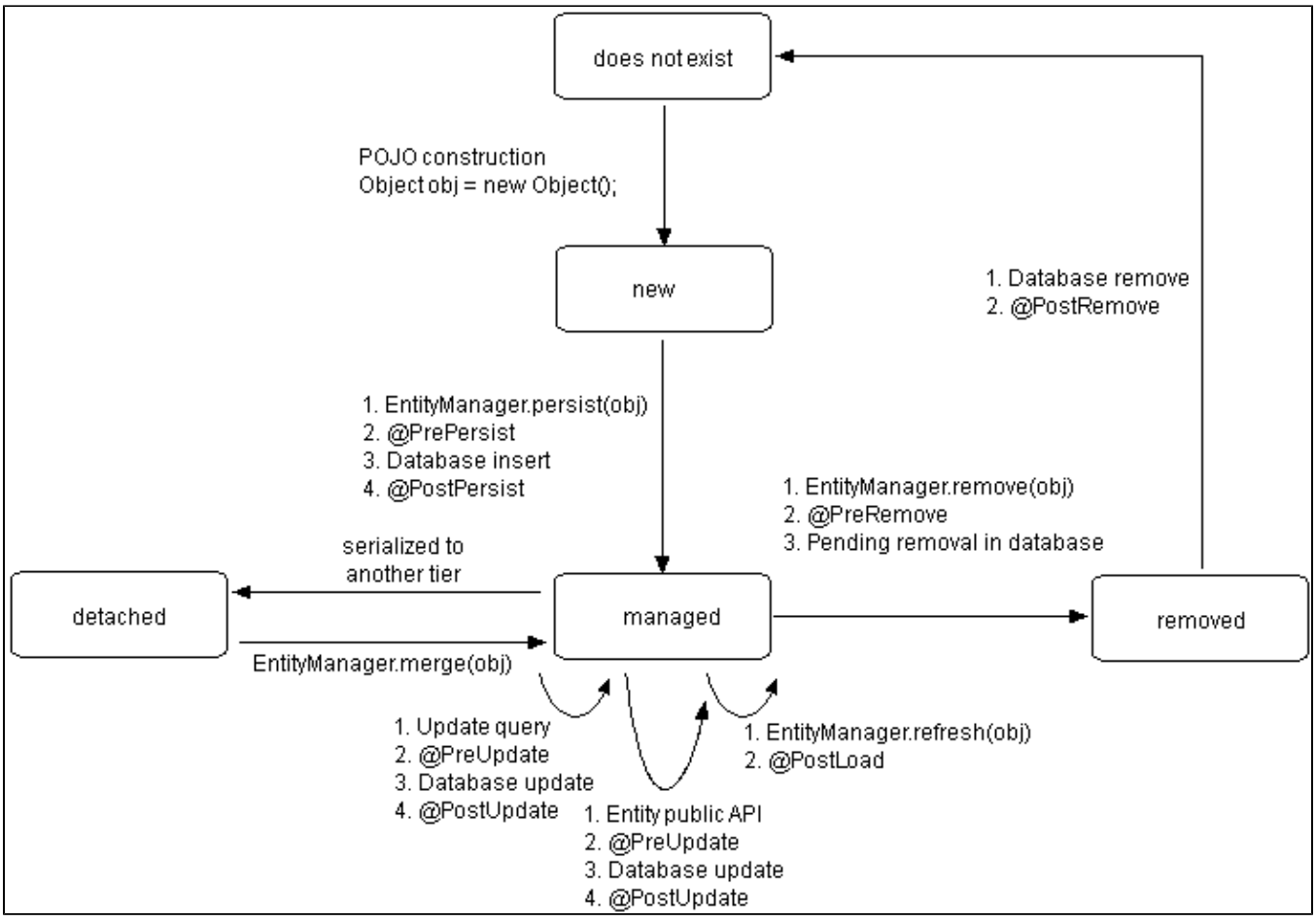
```
public interface MemberRepository extends JpaRepository<Member, Long> {
    @Query(value = "select a from Member a left join fetch a.team",
            countQuery = "select COUNT(a.memberKey) from Member a inner join a.team")
    Page<Member> findAllWithTeam(Pageable pageable);
}
```

발행된 쿼리(h2 dbms)

```
select member0_.member_key as member_k1_0_0_, team1_.team_key as team_key1_1_1_, member0_.member_id as
member_i2_0_0_, member0_.member_name as member_n3_0_0_,
                member0_.phone as phone4_0_0_, member0_.team_key as team_key5_0_0_, team1_.team_name as
team_nam2_1_1_ from member member0_
inner join team team1_ on member0_.team_key=team1_.team_key order by member0_.member_id asc limit ?
```

Spring Data JPA Auditing

1. Entity Listener



Callback Option	설명

@PrePersist	manager persist 의해 처음 호출될 때 실행됩니다.
@PostPersist	manager persist 에 의해 실행되고 불립니다. SQL INSERT 이후에 대응될 수 있습니다.
@PostLoad	로드 이후에 불립니다. SQL SELECT 이후에 대응될 수 있습니다.
@PreUpdate	SQL UPDATE 이전에 불립니다.
@PostUpdate	SQL UPDATE 이후에 불립니다.
@PreRemove	SQL DELETE 이전에 불립니다.
@PostRemove	SQL DELETE 이후에 불립니다

2. Spring Data Auditing

Spring Data 하위 프로젝트에선 언제 누가 수정 또는 변경했는지를 관리하는 Audit 기능이 제공 됨.

→ 아래와 같이 @EnableJpaAuditing을 선언하고 auditorAware Bean을 선언

```
@Configuration
@EnableJpaAuditing(auditorAwareRef = "auditorAware")
public class EnableAuditConfiguration {
    private MemberRepository memberRepository;

    public EnableAuditConfiguration(MemberRepository memberRepository) {
        this.memberRepository = memberRepository;
    }

    @Bean
    public AuditorAware<Member> auditorAware() {
        return () -> this.memberRepository.findByMemberId("admin");
    }
}
```

→ Entity의 Field에 아래와 같은 어노테이션을 선언만 해주면 JPA는 Entity가 저장되기 전에 알아서 값을 채워 줌

누구에 의해 Enntity가 생성되었는가? (@CreatedBy)

언제 Entity가 생성 되었는가? (@CreatedDate)

누구에 의해 Entity가 변경되었는가? (@LastModifiedBy)

언제 Entity가 변경 되었는가? (@LastModifiedDate)₩

※ 아래와 같이 BaseEntity를 선언하여 Auditing이 필요한 Entity가 이것을 상속하면 공통관리가 가능함

```
@Getter
@AllArgsConstructor
@NoArgsConstructor
@MappedSuperclass
@EntityListeners(value = {AuditingEntityListener.class})
public abstract class BaseEntity {

    @Column(name = "REASON", nullable = false)
    private String reason = "-";

    @Column(name = "DESCRIPTION", nullable = false)
    private String description = "-";

    @CreatedDate
    @Column(name = "CREATED_AT", updatable = false)
    private LocalDateTime createdAt;

    @LastModifiedDate
    @Column(name = "UPDATED_AT")
    private LocalDateTime updatedAt;

    public abstract Member getCreatedBy();

    public abstract Member getUpdatedBy();
}
```

3. Spring Data Envers

EnableJpaAuditing을 통해 언제, 누가에 대한 처리는 완료했으나 Entity에 대한 Revision 관리가 필요함(History table 등을 이용하여 Revision 관리)

→ Spring Data JPA의 확장 프로젝트인 Spring Data Envers를 이용하면 간단하게 처리할 수 있음

아래의 의존성을 추가

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-envers</artifactId>
</dependency>
```

아래의 옵션을 application.properties에 추가

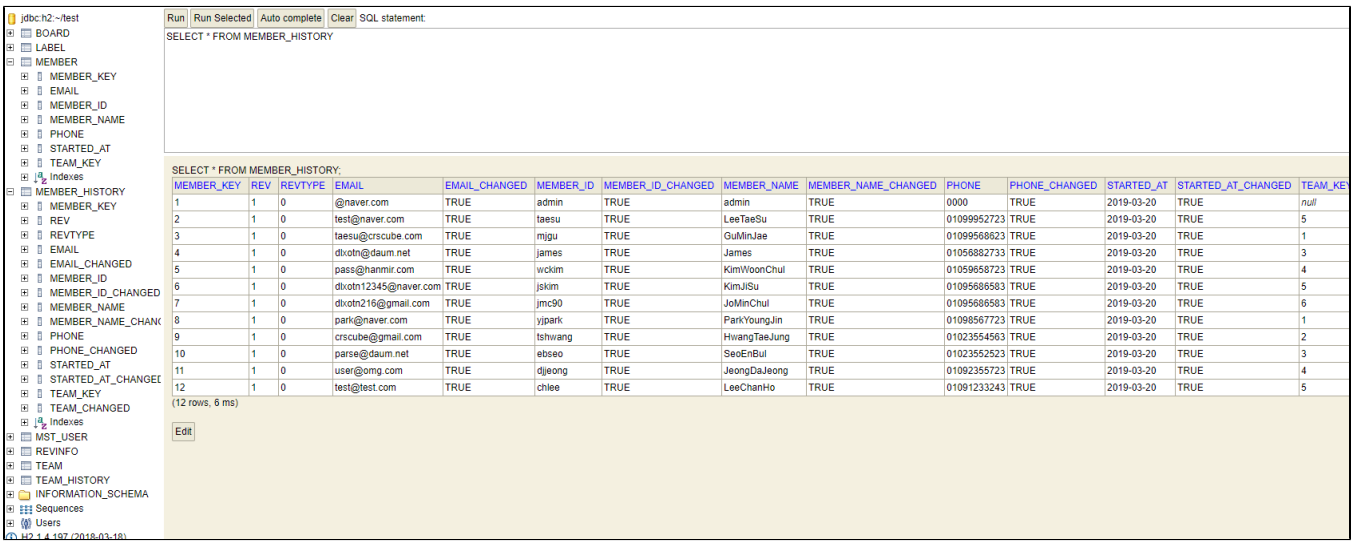
```
spring.jpa.properties.org.hibernate.envers.audit_table_suffix=_HISTORY
spring.jpa.properties.org.hibernate.envers.modified_flag_suffix=_CHANGED
```

프로퍼티	설명

`spring.jpa.properties.org.hibernate.envers.audit_table_suffix`	Revision table의 이름규칙으로 _HISTORY를 설정한 경우 MEMBER 테이블의 Revision 관리 테이블은 MEMBER_HISTORY로 생성됨 (default: _AUD)
`spring.jpa.properties.org.hibernate.envers.modified_flag_suffix`	테이블의 각 컬럼마다 변경된 여부를 저장하는 컬럼을 추가적으로 생성할 수 있음 _CHANGED로 지정한 경우 MEMBER_NAME 컬럼을 추적하는 Revision Column은 MEMBER_NAME_CHANGED로 생성 됨 (default: _MOD)

추적이 필요한 Entity에 @Audited 어노테이션 선언

```java
@Table(name = "MEMBER")
@Entity
@SequenceGenerator(name = "MEMBER_SEQ", sequenceName = "MEMBER_SEQ")
@Audited(withModifiedFlag = true)
public class Member {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO, generator = "MEMBER_SEQ")
    @Column(name = "MEMBER_KEY")
    private Long memberKey;

    @Column(name = "MEMBER_ID", unique = true, nullable = false)
    private String memberId;

    @Column(name = "MEMBER_NAME", nullable = false)
    private String memberName;

    @Column(name = "EMAIL", nullable = false)
    private String email;

    @Column(name = "PHONE")
    private String phone;

    @Column(name = "STARTED_AT")
    @Builder.Default
    private LocalDate startedAt = LocalDate.now();

    @ManyToOne
    @JoinColumn(name = "TEAM_KEY")
    private Team team;

    public Member updateMember(String memberName, String phone) {
        this.memberName = memberName;
        this.phone = phone;
        return this;
    }

    public void setTeam(Team team) {
        if (this.team != null) {
            this.team.removeMember(this);
        }

        this.team = team;
        this.team.addMember(this);
    }
}
```



각 Entity마다 생성된 _HISTORY 테이블을 확인할 수 있음

컬럼	설명
REV	Transaction ID로 특정 트랜잭션에서 변경된 모든 Entity의 Revision을 조회할 수 있도록 하기 위함
REVTYPE	Revision Type 0 생성 1 변경 2 삭제