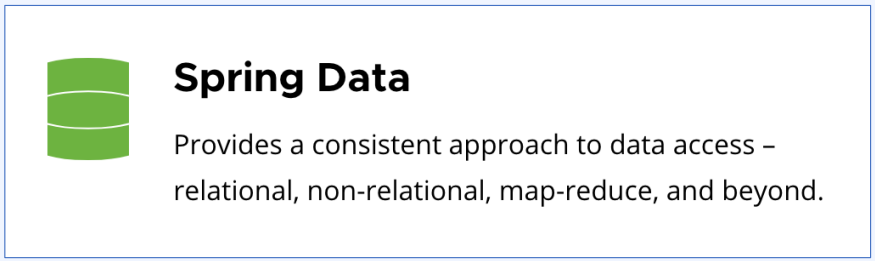
JPA로 게시판 만들기

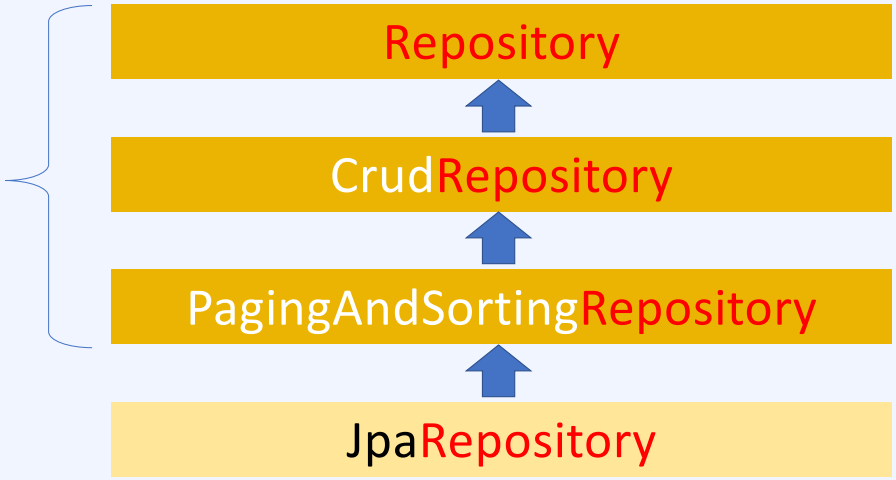
4 Spring Data JPA

Spring Data – 저장소 종류가 달라도 일관된 데이터 처리 방법을 제공

Spring Data JPA – JPA를 위한 저장소(JpaRepository) 와 관련 기능을 제공

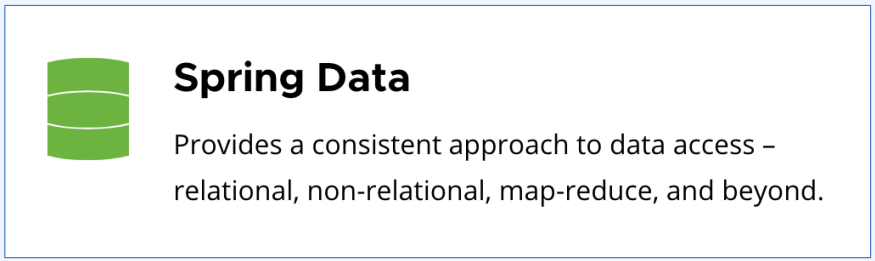


[참고] https://spring.io/projects/

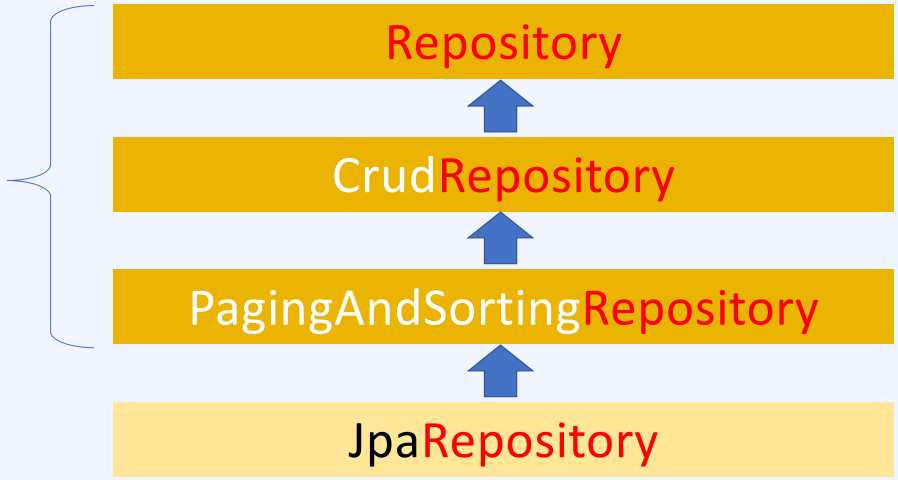


Spring Data – 저장소 종류가 달라도 일관된 데이터 처리 방법을 제공

Spring Data JPA – JPA를 위한 저장소(JpaRepository) 와 관련 기능을 제공



[참고] https://spring.io/projects/




```
public interface BoardRepository extends CrudRepository<Board, Long> {}
```

가장 많이 사용하는 Repository. CRUD를 위한 메서드를 제공.

분류	메서드 선언부 (T: 엔티티 타입, ID: 키 타입)	설 명

조회	long count();
	boolean existsById(ID id);
	Optional<T> findById(ID id);
	Iterable<T> findAllById(Iterable<ID> ids);
	Iterable<T> findAll();
		entity의 개수를 반환
		지정된 id의 entity가 존재하는지 확인
		지정된 id와 일치하는 entity 반환
		지정된 id들과 일치하는 entity 반환
		모든 entity를 반환

변경 저장	<S extends T> S save(S entity); <S extends T> Iterable<S> saveAll(Iterable<S> entities);	지정된 entity 저장 지정된 entitiy들 저장

삭제	void deleteById(ID id);
	void delete(T entity);
	void deleteAllById(Iterable<? extends ID> ids);
	void deleteAll(Iterable<? extends T> entities);
	void deleteAll();
		지정된 id와 일치하는 entity 삭제
		지정된 entity 삭제
		지정된 id들에 해당하는 entity 삭제
		지정된 entity들에 해당하는 entity 삭제
		모든 entity 삭제

페이징과 정렬 기능이 추가된 Repository. CrudRepository의 자손

```
// T: entity의 타입, ID: 키의 타입
public interface PagingAndSortingRepository<T, ID> extends CrudRepository<T, ID> {
    Iterable<T>  findAll(Sort sort);                  // 지정된 정렬 조건으로 정렬해서 반환
    Page<T>      findAll(Pageable pageable);    // 지정된 페이징 조건에 해당하는 페이지 반환
}
```

쿼리 메서드
1. JPA가 지원하는 쿼리

JPA는 쿼리를 작성하는 다양한 방법을 지원

1. JPQL : DB테이블이 아닌 entity를 대상으로 쿼리를 작성. SQL과 유사

```
SELECT b FROM Board b WHERE b.title = ?1
```

2. 쿼리 메서드(Query Method) : 메서드 이름으로 JPQL을 자동 생성

```
List<Board> list = boardRepo.findByTitleAndWriter("title1", "writer1");
```

3. JPA Criteria : JPQL을 메서드의 조합으로 작성.

```
cq.select(b).where(cb.equal(b.get("title"),"title1"));
```

4. Querydsl : JPQL을 메서드의 조합으로 작성. Criteria보다 간결. 오픈 소스

```
List<Board> list = queryFactory.selectFrom(board) .where(board.title.eq("title1")) .fetch();
```

5. Native SQL : JPQL대신 SQL을 직접 작성. 복잡한 SQL 작성 가능 - @Query

Spring Data에서 제공하는 기능. 메서드 이름으로 JPQL을 자동 생성

Repository에 메서드 이름을 규칙에 맞게 추가. 예) find + (entity명) + By + 속성명

```java
public interface BoardRepository extends CrudRepository<Board, Long> {
    // SELECT COUNT(*) FROM BOARD WHERE WRITER = writer
    int countAllByWriter(String writer);

    // SELECT * FROM BOARD WHERE WRITER = writer
    List<Board> findByWriter(String writer);

    // SELECT * FROM BOARD WHERE TITLE = title AND WRITER = writer
    List<Board> findByTitleAndWriter(String title, String writer);

    @Transactional // DELETE FROM BOARD WHERE WRITER = writer
    int deleteByWriter(String writer);
```

[참고] https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods

쿼리 메서드
3. 키워드와 작성 예시

분류	키워드	쿼리 메서드	조건절
논리	And, Or	findByTitleAndWriter(String title, String writer)	WHERE TITLE = title AND WRITER = writer
등가 비교	(생략), Is, Equals, IsNot, Not Null, IsNull NotNull, IsNotNull	findByTitle(String title) 또는 findByTitleIs(String title) findByTitleNot(String title) findByTitleIsNull() findByTitleIsNotNull()	WHERE TITLE = title WHERE TITLE <> title WHERE TITLE IS NULL WHERE TITLE IS NOT NULL
대소 비교	LessThan LessThanEqual GreaterThan GreaterThanEqual	findByBnoLessThan(Long bno) findByBnoLessThanEqual(Long bno) findByBnoGreaterThan(Long bno) findByBnoGreaterThanEqual(Long bno)	WHERE BNO < bno WHERE BNO <= bno WHERE BNO > bno WHERE BNO >= bno
날짜 비교	Before, After	findByInDateBefore(Date date) findByInDateAfter(Date date)	WHERE IN_DATE < date WHERE IN_DATE > date
와일드 카드	Like, NotLike StartingWith EndingWith Containing	findByTitleLike(String title) findByTitleStartingWith(String title) findByTitleEndingWith(String title) findByTitleContaining(String title)	WHERE TITLE LIKE CONCAT(title, '%') WHERE TITLE LIKE CONCAT(title, '%') WHERE TITLE LIKE CONCAT('%' , title) WHERE TITLE LIKE CONCAT('%' , title, '%')
정렬	OrderBy	findByWriterOrderByTitleAscViewCntDesc(String writer)	WHERE WRITER = writer ORDER BY TITLE ASC, VIEW_CNT DESC

[참고] https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repository-query-keywords

DB테이블이 아닌 entity를 대상으로 쿼리를 작성. SQL과 유사

자동 생성 : 쿼리 메서드(Spring Data), JPA Criteria(JPA 표준), Querydsl(오픈 소스)

수동 생성 : em.createQuery(), @Query

항목	JPQL	SQL
DB 독립적	O	X
조회 대상	Entity	DB 테이블
대소문자 구분	O	X
구문 자동 체크	O, X	X

직접 JPQL을 작성하는 방법은 2가지

1. EntityManager의 createQuery()로 작성

```
String query = "SELECT b FROM Board b";  // JPQL
TypedQuery<Board> tQuery = em.createQuery(query, Board.class); // 엔티티 지정
List<Board> list = tQuery.getResultList(); // 실행결과를 List<Board>로 반환
```

2. @Query로 작성 – 메서드 이름은 쿼리와 상관 없음

```
public interface BoardRepository extends CrudRepository<Board, Long> {
    @Query("SELECT b FROM Board b")  // org.springframework.data.jpa.repository.Query
    List<Board> findAllBoard();      // List<Board> list = boardRepo.findAllBoard();
```

JPQL의 매개변수를 지정하는 방법은 2가지

1. 매개변수 순서(default) - ?1은 첫 번째, ?2는 두 번째, ...

```
@Query("SELECT b FROM Board b WHERE b.title=?1 AND b.writer=?2")
List<Board> findByTitleAndWriter(String title, String writer);
```

2. 매개변수 이름 - @Param("이름")으로 바인딩. 생략하면 매개변수 이름으로 바인딩

```
@Query("SELECT b FROM Board b WHERE b.title = :title AND b.writer = :writer")
List<Board> findByTitleAndWriter(@Param("title")  String title,
                                 @Param("writer") String writer);
```

페이징 - @Query붙은 메서드에 Pageable타입의 매개변수 추가

```
@Query(value="SELECT * FROM BOARD", nativeQuery=true)
List<Board> findTitleAndWriter(Pageable pageable);
```

정렬 - 오름 차순(Sort.Direction.DESC), 내림 차순(Sort.Direction.ASC)

```
// 조회수(view_cnt)가 제일 높은 게시물 5개를 조회
// Pageable pageable = PageRequest.of(0, 5, Sort.Direction.DESC, "viewCnt"); // JPQ일 때는 viewCnt
Pageable pageable = PageRequest.of( page: 0,  size: 5, Sort.Direction.DESC,  ...properties: "view_cnt");
List<Board> list = boardRepo.findTitleAndWriter(pageable);
```

```
List<Sort.Order> sorts = new ArrayList<>();  // 정렬기준이 둘 이상일 때
sorts.add(new Sort.Order(Sort.Direction.DESC,  property: "view_cnt"));
sorts.add(new Sort.Order(Sort.Direction.ASC,  property: "up_date"));

Pageable pageable = PageRequest.of( page: 0,  size: 5, Sort.by(sorts));
```

JPQL과 SQL은 문자열이므로 타입이나 구문 체크가 어려움

쿼리를 타입에 안전하게 빌드해주는 JPA Criteria API 등장. 길고 읽기 어려움

JPA Criteria API의 단점을 보완하기 위해 Querydsl이 등장.

[JPA Criteria API]

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Board> cq = cb.createQuery(Board.class);
Root<Board> board = cq.from(Board.class);
cq.select(board).where(cb.equal(board.get("title"), o: "title1"));

TypedQuery<Board> tq = em.createQuery(cq);
List<Board> list = tq.getResultList();
```

[Querydsl]

```
QBoard board = QBoard.board;
JPAQueryFactory qf = new JPAQueryFactory(em);
List<Board> list = qf.selectFrom(board)
        .where(board.title.eq( right: "title1"))
        .fetch();
```

Q Type : Entity를 기반으로 자동 생성된 클래스.

JPQLQuery : JPQL쿼리를 위한 인터페이스

JPAQuery : JPLQuery의 구현체. 직접 생성하거나 JPAQueryFactory를 통해 생성

JPAQueryFactory : JPAQuery를 생성해 주는 클래스.

```
QBoard board = QBoard.board; // 참조를 간단히. static import로 대신 가능
JPAQueryFactory qf = new JPAQueryFactory(em); // 1. em에서 JPAQueryFactory를 생성

JPAQuery<Board> query = qf.selectFrom(board)  // 2. JPAQueryFactory로 JPAQuery생성
        .where(board.title.eq( right: "title1"));

List<Board> list = query.fetch();             // 3. JPAQuery를 실행해서 조회결과 얻기
```

쿼리 작성에 JPAQueryFactory, Q타입과 Q타입 필드의 메서드를 이용

일부 필드만 조회할 때는 JPAQuery<Tuple>, 하나는 JPAQuery<T>에 타입 지정

```
// Board의 모든 필드를 가져올 때는 JPAQuery<Board>, 일부만 가져올 때는 JPAQuery<Tuple>
JPAQuery<Tuple> query = qf.select(board.writer, board.viewCnt.sum()).from(board)
.where(board.writer.eq( right: "writer1").or(board.title.notLike( str: "title1%")))
.where(board.content.contains("content"))
.where(board.content.isNotNull())
.groupBy(board.writer)
.having(board.viewCnt.sum().gt( right: 1000))
.orderBy(board.writer.asc())
.orderBy(board.viewCnt.sum().desc());

List<Tuple> list = query.fetch();              // JPAQuery를 실행해서 조회결과 얻기
```

```
select board0_.writer, sum(board0_.view_cnt) from board board0_
where (board0_.writer=? or board0_.title not like ? escape '!')
and ( board0_.content like ? escape '!' )
and ( board0_.content is not null )
group by board0_.writer
having sum(board0_.view_cnt)>?
order by board0_.writer asc, sum(board0_.view_cnt) desc
```

BooleanBuilder를 이용하면 조건에 따라 쿼리가 달라지게 할 수 있다.

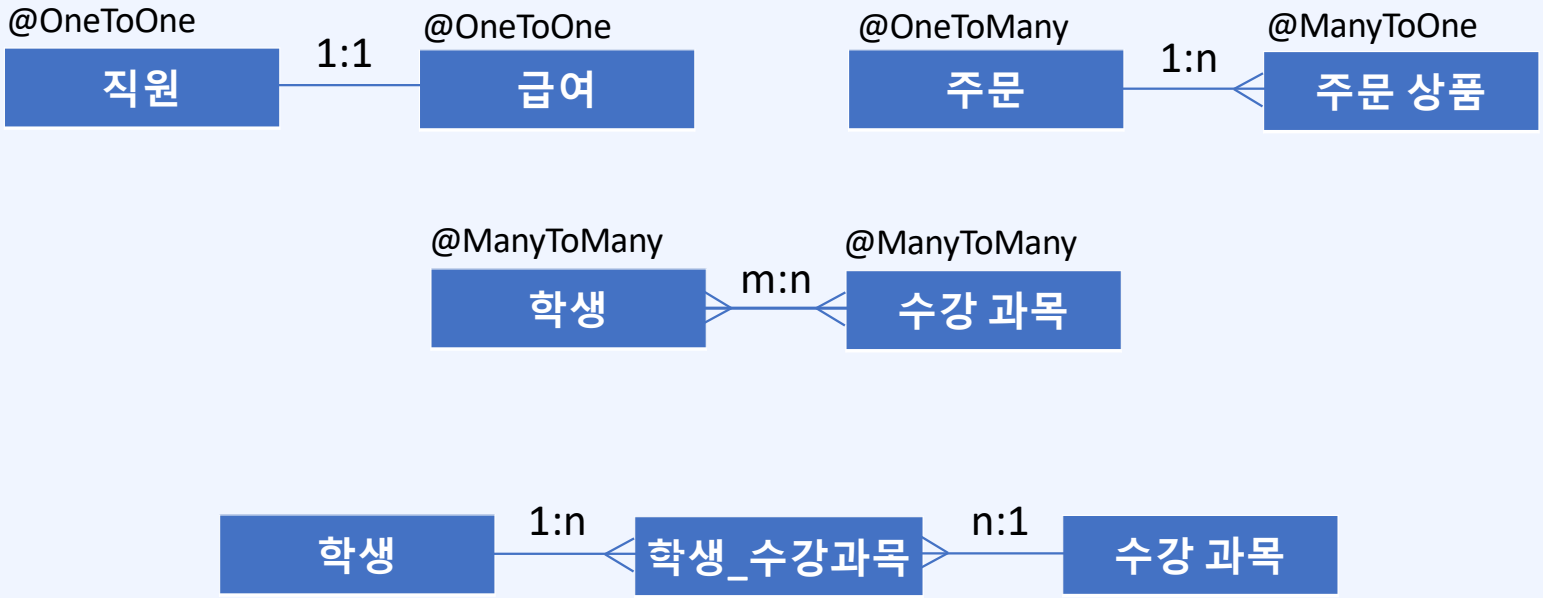
```
String searchBy = "TC"; // T: title, C: content, TC: title or content
String keyword  = "1";
keyword = "%" + keyword+ "%";
// ...
BooleanBuilder builder = new BooleanBuilder();

// 검색 조건에 따라 동적으로 쿼리가 달라지게 한다.
if(searchBy.equalsIgnoreCase( anotherString: "T"))
    builder.and(board.title.like(keyword));
else if(searchBy.equalsIgnoreCase( anotherString: "C"))
    builder.and(board.content.like(keyword));
else if(searchBy.equalsIgnoreCase( anotherString: "TC"))
    builder.and(board.title.like(keyword).or(board.content.like(keyword)));

JPAQuery<Board> query = qf.selectFrom(board)
        .where(builder)
        .orderBy(board.upDate.desc());
```

연관 관계 맵핑하기
1. 연관 관계의 종류

관계수(cardinality) : 두 엔티티 간의 대응되는 행의 개수

관계수의 종류 : 일대일(1:1), 일대다(1:n), 다대일(n:1), 다대다(m:n)

관계의 방향 : 관계형 모델(양방향), 객체 모델(단방향)



연관 관계 맵핑하기
2. 일대일 관계 - 단방향

@JoinColumn으로 연결할 엔티티의 키를 지정. cart.getMember()로 조회 가능

```java
@Entity
public class Cart {
    @Id
    @Column(name="cart_id")
    private Long id;

    @OneToOne
    @JoinColumn(name="member_id")
    private Member member;

    @Override
    public String toString() {
        return "Cart{" +
                "id=" + id +
                ", member=" + member +
                '}';
    }
    // getters & setters
}
```

```java
@Entity
public class Member {
    @Id
    @Column(name="member_id")
    private Long id;
    private String password;
    private String name;

    @Override
    public String toString() {
        return "Member{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", password='" + password + '\'' +
                '}';
    }
    // getters & setters
}
```

연관 관계 맵핑하기
3. 일대일 관계 - 양방향

반대쪽 엔티티에도 @OneToOne을 적용. toString()의 상호 참조 주의

```java
@Entity
public class Cart {
    @Id
    @Column(name="cart_id")
    private Long id;

    @OneToOne  // FK가 만들어짐
    @JoinColumn(name="member_id", nullable = false)
    private Member member;

    @Override
    public String toString() {
        return "Cart{" +
                "id=" + id +
                ", member=" + member +
                '}';
    }
// getters & setters
}
```

```java
@Entity
public class Member {
    @Id
    @Column(name="member_id")
    private Long id;
    private String password;
    private String name;

    @OneToOne(mappedBy="member") // FK가 안만들어짐
    @JoinColumn(name="cart_id")
    private Cart cart;

    @Override
    public String toString() {
        return "Member{" +
                "id=" + id +
                ", password='" + password + '\'' +
                ", name='" + name + '\'' +
//              ", cart=" + cart +
                '}';
    }
    // getters & setters
}
```

연관 관계 맵핑하기
4. 다대일 관계 - 단방향

@JoinColumn으로 연결할 엔티티의 키를 지정. board.getUser()로 조회 가능

```java
@Entity
public class Board {
    @Id
    private Long bno; // 게시물 번호
    private String title;
//    private String writer;
    private String content;
    private Long viewCnt;
    @ManyToOne // FK가 만들어짐
    @JoinColumn(name="user_id", nullable = false)
    private User user;
    @Temporal(value= TemporalType.TIMESTAMP)
    private Date inDate;
    @Temporal(value= TemporalType.TIMESTAMP)
    private Date upDate;
```

```java
@Entity
public class User {
    @Id  // PK로 지정
    @Column(name="user_id")
    private String id;
    private String password;
    private String name;
    private String email;
    private Date inDate; // 입력일
    private Date upDate; // 변경일
```

연관 관계 맵핑하기
5. 다대일 관계 - 양방향

두 엔티티에 @OneToMany, @ManyToOne과 @JoinColumn을 각각 적용

FetchType.EAGER는 join으로 한번에 조회, Lazy는 getBoardList()를 호출할 때 따로 조회

```java
@Entity
public class Board {
    @Id
    private Long bno; // 게시물 번호
    private String title;
//    private String writer;
    private String content;
    private Long viewCnt;
    @ManyToOne // FK가 만들어짐
    @JoinColumn(name="user_id", nullable = false)
    private User user;
    @Temporal(value= TemporalType.TIMESTAMP)
    private Date inDate;
    @Temporal(value= TemporalType.TIMESTAMP)
    private Date upDate;
```

```java
@Entity
public class User {
    @Id  // PK로 지정
    @Column(name="user_id")
    private String id;
    private String password;
    private String name;
    private String email;
    private Date inDate; // 입력일
    private Date upDate; // 변경일

    @OneToMany(mappedBy="user", fetch=FetchType.EAGER)
    List<Board> list = new ArrayList<>();
```