

JPA

JPA를 사용해야 하는 이유

NHN FORWARD ►►

생산성

- JPA를 사용하면 지루하고 반복적인 CRUD용 SQL을 개발자가 직접 작성하지 않아도 된다
- Spring Data JPA를 사용하면 interface 선언만으로도 쿼리 구현이 가능하기 때문에 관리 도구 등에서 가볍게 사용할 수 있는 CRUD 쿼리를 손쉽게 대처할 수 있다

구분	Order	Item	OrderItem
insert	<code>insert into Orders (...)</code>	<code>insert into Item (...)</code>	<code>insert into OrderItems (...)</code>
update	<code>update Orders set ...</code>	<code>update Item set ...</code>	<code>update OrderItems set ...</code>
delete	<code>delete from Orders ...</code>	<code>delete from Items ...</code>	<code>delete from OrderItems ...</code>
...



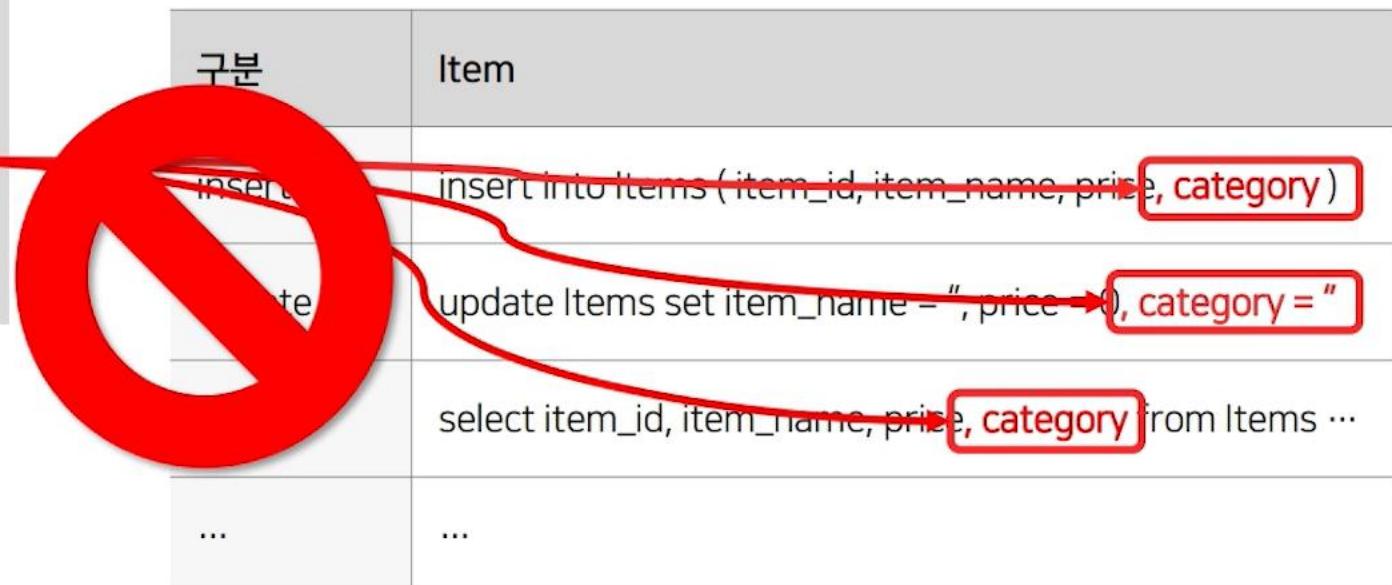
JPA를 사용해야 하는 이유

NHN FORWARD ►

유지보수

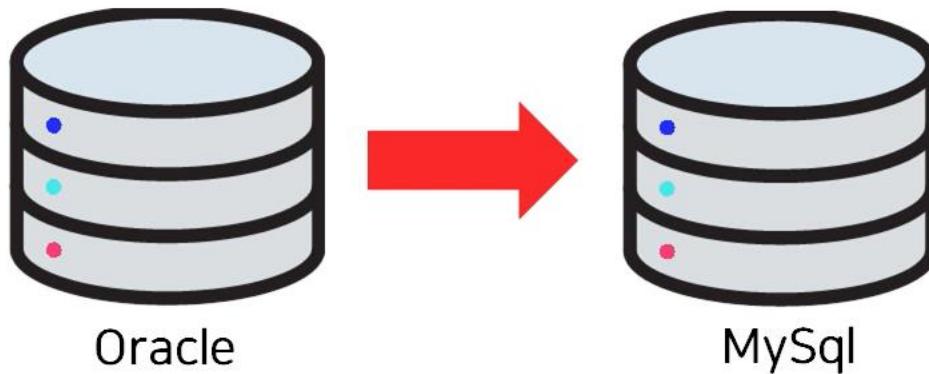
- 칼럼 추가/삭제 시 직접 관련된 CRUD 쿼리를 모두 수정하는 대신 JPA가 관리하는 모델(Entity)을 수정하면 된다

```
public class Item {  
    private Long itemId;  
    private String itemName;  
    private Long price;  
    private String category;  
}
```



데이터 접근 추상화와 벤더 독립성

- 데이터베이스 벤더마다 미묘하게 다른 데이터 타입이나 SQL을 JPA를 이용하면 손쉽게 해결할 수 있다



SQL 중심적인 개발에서 객체 중심으로 개발

패러다임 불일치 해결

- JPA는 객체와 관계형 데이터베이스 사이의 패러다임의 불일치로 발생하는 문제를 해결
- (상속, 연관 관계, 객체 그래프 탐색 등)

Java Persistence API

- 자바 진영의 ORM 기술 표준

ORM(Object-Relational Mapping)

- 데이터베이스 객체를 자바 객체로 매팅하여 객체 간의 관계를 바탕으로 SQL을 자동으로 생성

JPA

- 표준 명세
 - JSR 338 – Java Persistence 2.2

Hibernate

- JPA 실제 구현체 : Hibernate, EclipseLink, DataNuclues
- Hibernate가 사실상 표준(de facto)

Spring Data

- 다양한 데이터 저장소에 대한 접근을 추상화하기 위한 Spring 프로젝트
- JPA, JDBC, Redis, MongoDB, Elasticsearch 등

Spring Data JPA

- Repository 추상화를 통해 interface 선언만으로도 구현 가능
- 메서드 이름으로 쿼리 생성
- Web Support(페이징, 정렬, 도메인 클래스 컨버터 등)

의존성 라이브러리 추가

- Spring Data JPA, Hibernate 사용

```
org.springframework.data:spring-data-jpa  
org.hibernate:hibernate-entitymanager
```

빈 설정 - EntityManagerFactory

- LocalContainerEntityManagerFactoryBean 빈 등록
 - SqlSessionFactoryBean vs EntityManagerFactoryBean

```
@Bean LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource) {  
    LocalContainerEntityManagerFactoryBean emf  
        = new LocalContainerEntityManagerFactoryBean();  
    emf.setDataSource(dataSource);  
    emf.setPackagesToScan("com.nhnent.forward.mybatisjpa.entity");  
    emf.setJpaVendorAdapter(jpaVendorAdapters());  
    emf.setJpaProperties(jpaProperties());  
  
    return emf;  
}
```

빈 설정 - TransactionManager

- Transaction Manager 빈 등록
 - DataSourceTransactionManager vs JpaTransactionManager

```
@Bean  
public PlatformTransactionManager transactionManager(EntityManagerFactory emf) {  
    JpaTransactionManager transactionManager = new JpaTransactionManager();  
    transactionManager.setEntityManagerFactory(emf);  
  
    return transactionManager;  
}
```

EntityManager

- 엔티티(Entity)의 저장, 수정, 삭제, 조회 등 엔티티와 관련된 모든 일을 처리하는 관리자

```
public interface EntityManager {  
    public <T> T find(Class<T> entityClass, Object primaryKey);  
  
    public <T> T find(Class<T> entityClass, Object primaryKey, LockModeType lockMode);  
  
    public void persist(Object entity);  
  
    public <T> T merge(T entity);  
  
    public void remove(Object entity);  
  
    // ...  
}
```

EntityManagerFactory

- EntityManager를 생성하는 팩토리
- EntityManagerFactory는 애플리케이션 전체에서 하나만 생성해서 공유



Entity

- JPA를 이용해서 데이터베이스 테이블과 매핑할 클래스

어노테이션

- `@Entity`: JPA가 관리할 객체임을 명시
- `@Table`: 매핑할 데이터베이스 테이블 이름을 명시
- `@Id`: 기본 키(PK) 매핑
- `@Column`: 필드와 칼럼을 매핑

예) Entity 설정

- ItemEntity 클래스

```
@Entity          -.-> JPA가 관리할 객체
@Table(name = "Items") -.-> 매핑할 테이블 이름
public class ItemEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) -.-> 기본 키 매핑
    @Column(name = "item_id")
    private Long itemId;

    @Column(name = "item_name") -.-> 필드와 칼럼 매핑
    private String itemName;

    private Long price;
}
```

자동 생성

- TABLE 전략: 채번 테이블을 사용
- SEQUENCE 전략: 데이터베이스 시퀀스를 사용해서 기본 키를 할당 e.g.) Oracle
- IDENTITY 전략: 기본 키 생성을 데이터베이스에 위임 e.g.) MySQL
- AUTO 전략: 선택한 데이터베이스 방언(dialect)에 따라 기본 키 매핑 전략을 자동으로 선택

직접 할당

- 애플리케이션에서 직접 식별자 값을 할당

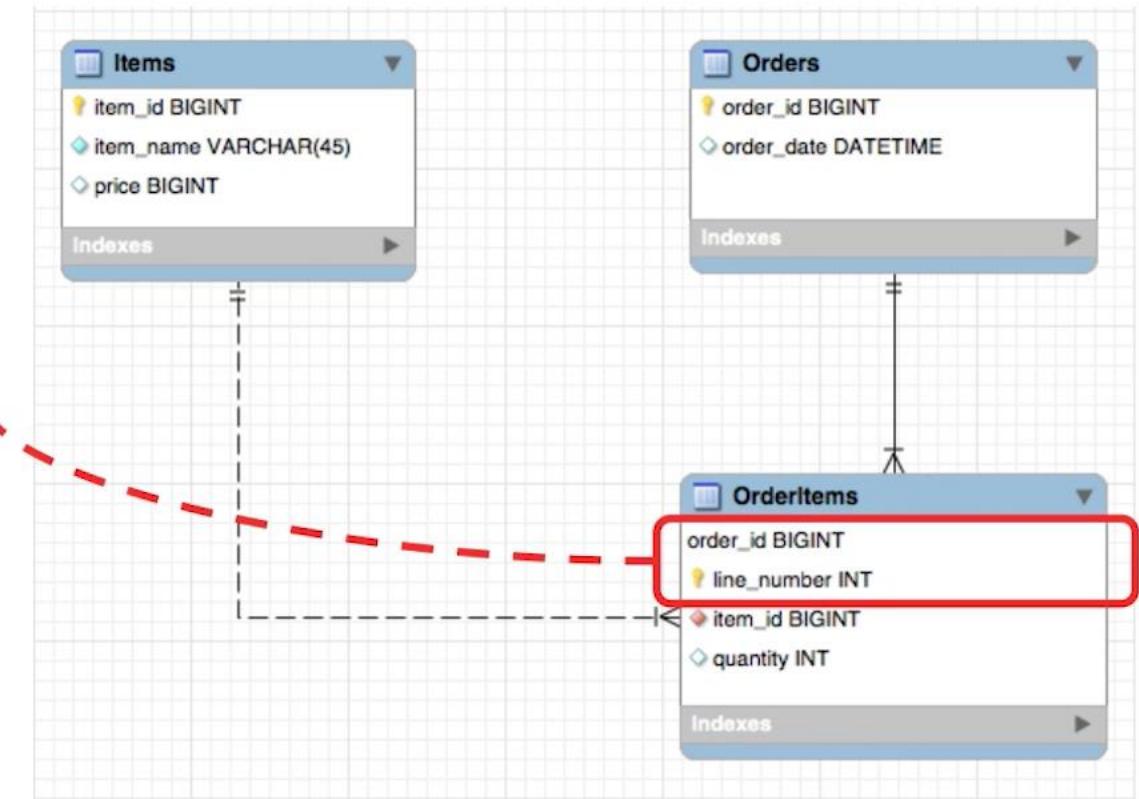
@GeneratedValue

```
public @interface GeneratedValue {  
    GenerationType strategy() default AUTO;  
    String generator() default "";  
}  
  
public enum GenerationType {  
    TABLE,  
    SEQUENCE,  
    IDENTITY,  
    AUTO  
}
```

예) Entity 설정

- OrderItemEntity 클래스

```
@Entity  
@Table(name = "OrderItems")  
public class OrderItemEntity {  
    @EmbeddedId  
    private Pk pk; // ← Red dashed arrow from code to diagram  
  
    @Column  
    private Integer quantity;  
  
    // ...  
}
```



@EmbeddedId , @Embeddable

- @EmbeddedId: Entity 클래스의 필드에 지정
- @Embeddable: 복합 키 식별자 클래스에 지정

```
@Entity  
@Table(name = "OrderItems")  
public class OrderItemEntity {  
    @EmbeddedId  
    private Pk pk;  
    // ...  
}  
  
@NoArgsConstructor @AllArgsConstructor  
@EqualsAndHashCode  
@Embeddable  
public static class Pk implements Serializable {  
    @Column(name = "order_id")  
    private Long orderId;  
  
    @Column(name = "line_number")  
    private Integer lineNumber;  
}
```

예) Entity 설정

- OrderEntity 클래스

```
@Entity
@Table(name = "Orders")
public class OrderEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "order_id")
    private Long orderId;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "order_date")
    private Date orderDate;
}
```

@Column: 객체 필드를 칼럼에 매핑

@Temporal: 날짜 타입 매핑

@Transient: 특정 필드를 칼럼에 매핑하지 않을 경우에 지정

```
@Temporal(TemporalType.TIMESTAMP)
@Column(name = "order_date")
private Date orderDate;

public enum TemporalType {
    DATE,
    TIME,
    TIMESTAMP
}
```

연관 관계(association)

- 데이터베이스 테이블 간의 관계(relationship)를 엔티티 클래스의 속성(attribute)으로 모델링
- 데이터베이스 테이블은 외래 키(FK)로 JOIN을 이용해서 관계 테이블을 참조
→ 객체는 참조를 사용해서 연관된 객체를 참조

외래 키(FK) 매팅

- @JoinColumn

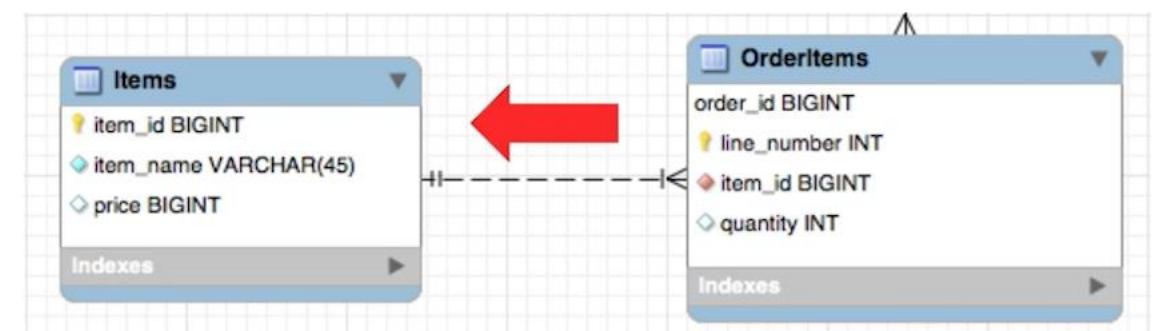
다중성(Multiplicity)

- @OneToOne
- @OneToMany
- @ManyToOne
- @ManyToMany

예) 연관 관계 설정

OrderItemEntity – ItemEntity 연관 관계 설정

```
public class OrderItemEntity {  
    // ...  
  
    @ManyToOne  
    @JoinColumn(name = "item_id")  
    private ItemEntity item;  
}
```



방향성

- 단방향(unidirectional)
- 양방향(bidirectional)

양방향 연관 관계

- 관계의 주인(owner)
 - 연관 관계의 주인은 외래 키(FK)가 있는 곳
 - 연관 관계의 주인이 아닌 경우, `mappedBy` 속성으로 연관 관계의 주인을 지정

영속성 전이

- CascadeType.ALL
- CascadeType.PERSIST
- CascadeType.MERGE
- CascadeType.REMOVE
- CascadeType.REFRESH
- CascadeType.DETACH

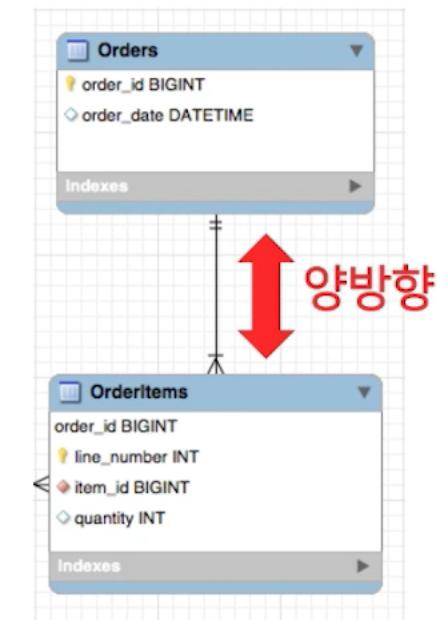
페치 전략(Fetch Strategy)

- FetchType.EAGER
- FetchType.LAZY

예) 연관 관계 설정

OrderEntity – OrderItemEntity 연관 관계 설정

```
public class OrderEntity {  
    @OneToMany(  
        mappedBy = "order",  
        cascade = { CascadeType.ALL }) 영속성 전이  
    List<OrderItemEntity> orderItems = new ArrayList<>();  
  
    public class OrderItemEntity {  
        @JoinColumn(name = "order_id")  
        @ManyToOne(fetch = FetchType.LAZY)  
        private OrderEntity order; 관계의 주인  
    }  
}
```



Spring Data Repository

- data access layer 구현을 위해 반복해서 작성했던, 유사한 코드를 줄일 수 있는 추상화 제공

```
// EntityManager를 통해 entity를 저장, 수정, 삭제, 조회
ItemEntity entity1 = new ItemEntity();
entity1.setItemName("peach");
entity1.setPrice(135L);
entityManager.persist(entity1);

ItemEntity entity2 = entityManager.find(ItemEntity.class, entity1.getItemId());
entity2.setPrice(235L);
entityManager.merge(entity2);

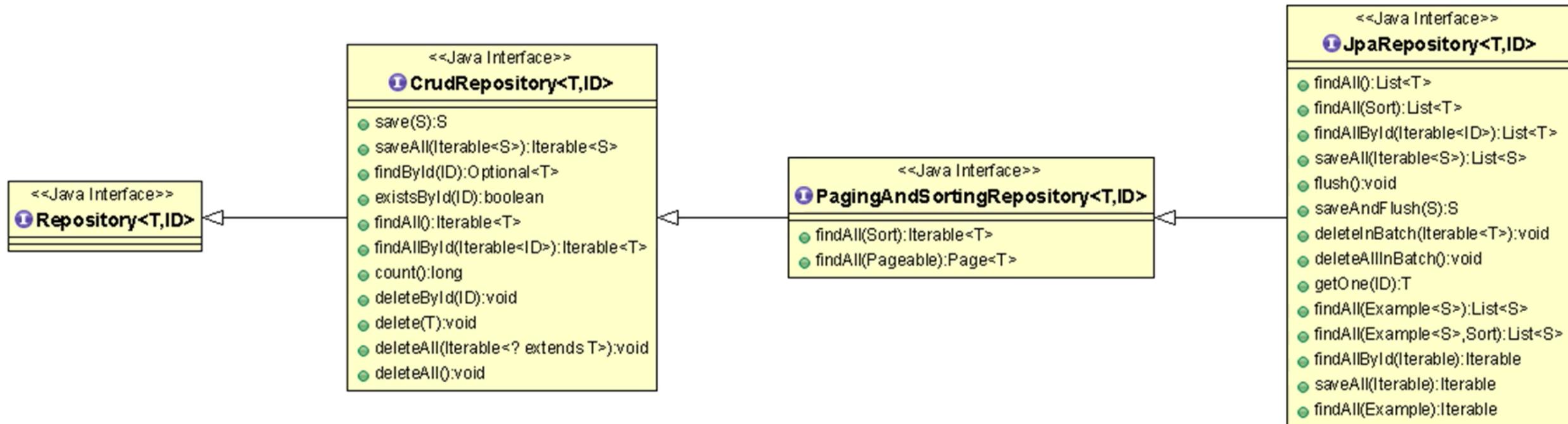
// JPQL, Criteria API를 이용해서 복잡한 쿼리 수행
String jpql = "select item from ItemEntity item where item.itemName like '%peach%'";
List<ItemEntity> entites = entityManager.createQuery(jpql, ItemEntity.class)
    .getResultList();
```

예) Repository 설정

- ItemRepository interface
 - JpaRepository interface 상속

```
public interface ItemRepository extends JpaRepository<ItemEntity, Long> {  
}
```

웬만한 CRUD, Pagination, Sorting 관련 메서드 제공



JpaRepository가 제공하는 메서드들이 실제 수행하는 쿼리

```
// insert / update  
<S extends T> S save(S entity);  
  
// select * from Items where item_id = {id}  
T findOne(ID id);  
  
// select count(*) from Items;  
long count();  
  
// delete from Items where item_id = {id}  
void delete(ID id);  
  
// ...
```

Spring Data JPA에서 제공하는 기능으로 이름 규칙에 맞춰 interface에 선언하면 쿼리 생성

cf.) <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repository-query-keywords>

```
public interface ItemRepository ... {  
    // select * from Items where item_name like '%{itemName}%'  
    ItemEntity findByItemNameLike(String itemName);  
  
    // select item_id from Items where item_name = '{itemName}'  
    // and price = {price} limit 1  
    boolean existsByItemNameAndPrice(String itemName, Long price);  
  
    // select count(*) from Items where item_name like '%{itemName}%'  
    int countByItemNameLike(String itemName);  
  
    // delete from Items where price between {price1} and {price2}  
    void deleteByPriceBetween(long price1, long price2);  
}
```

JPA에서 제공하는 객체 지향 쿼리

- JPQL: 엔티티 객체를 조회하는 객체 지향 쿼리
- Criteria API: JPQL을 생성하는 빌더 클래스

third party library를 이용하는 방법

- Querydsl
- jOOQ
- ...

ItemService – insert

```
public class ItemService {  
    @Autowired  
    private ItemRepository itemRepository;  
  
    @Transactional  
    public Item createItem(Item item) {  
        ItemEntity itemEntity = new ItemEntity();  
        itemEntity.setItemName(item.getItemName());  
        itemEntity.setPrice(item.getPrice());  
  
        ItemEntity newItem = itemRepository.save(itemEntity);  
  
        return newItem.toItemDto();  
    }  
}
```

ItemService – update

```
public class ItemService {  
    @Transactional  
    public Item updateItem(Item item) {  
        ItemEntity itemEntity = new ItemEntity();  
        itemEntity.setItemId(item.getItemId());  
        itemEntity.setItemName(item.getItemName());  
        itemEntity.setPrice(item.getPrice());  
  
        ItemEntity newItemEntity = itemRepository.save(itemEntity);  
  
        return newItemEntity.toItemDto();  
    }  
}
```

ItemService - delete

```
public class ItemService {  
    @Transactional  
    public boolean deleteItem(Long itemId) {  
        itemRepository.delete(itemId);  
        return true;  
    }  
}
```

ItemService - pagination 구현

- Spring Data JPA에서 제공하는 Pageable 사용

Spring Data 프로젝트에서 제공하는 web support 기능 중 하나

- Pageable: pagination 정보를 추상화한 인터페이스

```
public interface Pageable {  
    int getPageNumber();  
    int getPageSize();  
    int getOffset();  
  
    Sort getSort();  
  
    Pageable next();  
    Pageable previousOrFirst();  
    Pageable first();  
  
    boolean hasPrevious();  
}
```

ItemController – Pageable 적용

- Spring Data에서 page, size 파라미터값을 Controller의 Pageable 인자로 변환해서 전달

```
@RestController  
@RequestMapping("/items")  
public class ItemController {  
    @Autowired  
    ItemService itemService;  
  
    @GetMapping("")  
    public List<Item> getItems(Pageable pageable) {  
        return itemService.getItems(pageable);  
    }  
}
```

GET /items?page=0&size=30



ItemService - pagination 구현

- JpaRepository.findAll(Pageable pageable) 메서드로 Pageable 객체를 전달

```
public class ItemService {  
    public List<Item> getItems(Pageable pageable) {  
        Page<ItemEntity> itemPage = itemRepository.findAll(pageable);  
  
        return itemPage.getContent()  
            .stream()  
            .map(ItemEntity::toItemDto)  
            .collect(Collectors.toList());  
    }  
}
```

OrderService - insert

- 영속성 전이를 이용해서 OrderEntity를 저장할 때 OrderItemEntity도 함께 저장

```
public class OrderService {  
    @Transactional  
    public Order createOrder(Order order) {  
        OrderEntity orderEntity = new OrderEntity();  
        orderEntity.setOrderDate(new Date());  
  
        order.getOrderItems()  
            .forEach(orderItem -> {  
                ItemEntity itemEntity = new ItemEntity();  
                itemEntity.setItemId(orderItem.getItemId());  
  
                OrderItemEntity orderItemEntity = new OrderItemEntity();  
                orderItemEntity.setOrder(orderEntity);  
                orderItemEntity.getPk().setLineNumber(orderItem.getLineNumber());  
                orderItemEntity.setItem(itemEntity);  
                orderItemEntity.setQuantity(orderItem.getQuantity());  
  
                orderEntity.getOrderItems().add(orderItemEntity);  
            });  
  
        OrderEntity newEntity = orderRepository.save(orderEntity);  
        return newEntity.toOrderDto();  
    }  
}
```

- JPA 설정
 - spring-data-jpa, hibernate-entitymanager 의존성 라이브러리 추가
 - EntityManagerFactoryBean, JpaTransactionManager 빈 등록
- Entity 설정
 - @Entity, @Table, @Id, @Column, @Temporal, @GeneratedValue, @EmbeddedId, ...
- 연관 관계 설정
 - @JoinColumn, @OneToMany, @ManyToOne, mappedBy, FetchType.LAZY, ...
- Repository interface 생성 - JpaRepository 상속
- 애플리케이션에서의 사용 - Entity, Repository, Pageable, ...

- 상대적으로 높은 학습 곡선
 - But,
 - 기술적인 진입 장벽을 낮춰줄 여러 솔루션들이 존재(e.g. Spring Data JPA, QueryDSL 등)
 - 2015년 이후로 JPA 관련 다양한 국내 서적이 출간된 상태
- JPQL에는 분명 한계가 있다
 - But,
 - Native SQL
 - 다른 Data Access 기술과 혼용 및 분리(CQRS) 가능
- 그럼에도 불구하고 ...