

# 관계형 데이터베이스와 ORM

---

## 데이터베이스 (Database)

### 데이터베이스의 정의

- 데이터를 효율적으로 관리하기 위한 일종의 창고
- 특정 조직의 여러 사용자가 데이터를 공유하여 사용할 수 있도록 통합 저장된 데이터의 집합
- 행과 열로 구성된 시트에서 사용자가 정의한 형식으로 데이터를 관리하는 엑셀파일과 유사

## 관계형 데이터베이스 (Relational Database)

### 관계형 데이터베이스

- 1970년에 E. F. Codd 가 제안한 데이터 관계형 모델에 기초하는 디지털 데이터베이스

### 관계형 모델 (Relational Model)

- 데이터를 컬럼(column)과 로우(row)를 이루는 하나 이상의 테이블(또는 관계)로 정리
- 고유 키(Primary key)가 각 로우(row)를 식별 로우(row)는 레코드(record)나 튜플(tuple)로 부른다
- 관계(Relationship)는 서로 다른 테이블들 사이의 상호작용에 기반을 두고 형성된 논리적 연결이다.
- 관계(Relationship)는 테이블 간에 둘 다 존재한다.
  - 이 관계들은 일대일, 일대다, 다대다, 이렇게 세 가지 형태로 이루어진다.
  - 모두 다 관계형 데이터베이스

column, row, primary key, foreign key, relationship, transaction, SQL, MySQL, Oracle

## 하지만 내가 아는 프로그래밍 언어는

### Java

- 객체지향(Object-oriented) 프로그래밍 언어
- 패러다임의 불일치 발생
  - 관계형 데이터베이스 ≠ 객체 지향 프로그래밍 언어

## 데모

- IntelliJ에서 VCS에서 가져와 프로젝트 생성
- URL: <https://github.com/dongmyo/academy-spring-jpa>
- 복제(Clone)
- 실행 구성: Tomcat 서버, 로컬, 데모
- 전체 소스 코드 살펴보기:
  - **pom.xml**: 메이븐 프로젝트
  - 패키징: war
  - Spring MVC + Spring JDBC

- JdbcTemplate
  - H2 데이터베이스
  - 자바 기반
  - 오픈소스
  - 관계형 데이터베이스
- H2 데이터베이스 다운로드 및 실행:
    - Download > All Platforms: <https://www.h2database.com/html/main.html>
    - 압축 파일 해제
    - jar 파일 실행: `java -jar h2/bin/h2-2.1.214.jar`
  - 연습:
    - 이전 Demo 프로그램에서 `User` 클래스에 `age` 필드 추가하기
    - 하지만...
      - SQL 직접 수정
      - 텍스트 편집으로 오타가 있어도 런타임에서 확인 가능
      - 객체와의 매핑은 별개의 작업
      - 쿼리 수행 결과와 객체와의 매핑은 별도 수작업 필요
      - Repository의 CRUD 메서드와 SQL을 함께 변경
    - 추가적으로:
      - 상속 구조의 표현
      - 연관관계 참조
      - 객체 그래프 탐색 등

## ORM

- ORM (Object-Relational Mapping)
- ORM 프레임워크가 중간에서 객체와 관계형 데이터베이스를 매핑
- ORM을 사용하면 DBMS 벤더마다 다른 SQL에 대한 종속성을 줄이고 호환성을 향상시킬 수 있음

데이터베이스 벤더들마다 다른 쿼리를 작성해주라는 뜻

## JPA

- JPA (Java Persistence API)
  - 자바 ORM 기술 표준
  - 표준 명세:
    - JSR 338 - Java Persistence 2.2
- JPA (Jakarta Persistence API)
  - Jakarta Persistence 3.1
- JPA 구현
  - Hibernate, EclipseLink, DataNucleus
  - Hibernate가 사실상 표준 (de facto) JPA 구현체임.

- JPA는 스펙이다. [1]

## JPA를 사용해야 하는 이유

1. SQL 중심적인 개발 -> 객체 중심으로 개발
  - JPA를 사용하면 객체를 중심으로 개발하고, 지루하고 반복적인 CRUD용 SQL을 개발자가 직접 작성하지 않아도 된다.
2. 패러다임 불일치 해결
  - JPA는 객체와 관계형 데이터베이스 사이의 패러다임의 불일치로 인해 발생하는 문제(상속, 연관관계, 객체 그래프 탐색 등)를 해결한다.
3. 생산성
  - Spring Data JPA를 사용하면 interface 선언만으로도 쿼리 구현이 가능하기 때문에, 지루하고 반복적인 CRUD 쿼리를 손쉽게 대처할 수 있다.
4. 유지보수성
  - JPA를 사용하면 컬럼 추가/삭제 시 관련된 CRUD 쿼리를 모두 수정하는 대신, JPA가 관리하는 모델(Entity)을 수정하면 된다.
5. 데이터 접근 추상화와 벤더 독립성
  - 데이터베이스 벤더마다 미묘하게 다른 데이터 타입이나 SQL을 JPA를 이용하면 손쉽게 해결이 가능하다.

## Spring Framework과 JPA

- Spring Data: 다양한 데이터 저장소에 대한 접근을 추상화하기 위한 Spring 프로젝트 (JPA, JDBC, Redis, MongoDB, Elasticsearch 등을 지원한다)
- Spring Data JPA: repository 추상화를 통해 interface 선언만으로도 구현 가능하며, 메서드 이름으로 쿼리를 생성할 수 있다. 또한 Web Support(페이징, 정렬, 도메인 클래스 컨버터 기능)을 제공한다.

## Demo

---

이전 Demo 프로그램에서는 트랜잭션 적용, Spring + JPA 설정을 살펴보았습니다.

### Spring Framework의 트랜잭션 추상화

- **PlatformTransactionManager**: Spring Framework 트랜잭션 추상화의 핵심 인터페이스

```
public interface PlatformTransactionManager extends TransactionManager
{
    TransactionStatus getTransaction(TransactionDefinition definition)
    /*...*/;
    void commit(TransactionStatus status) throws TransactionException;
    void rollback(TransactionStatus status) throws
    TransactionException;
}
```

- 선언적 트랜잭션: **@Transactional**

Demo: Spring + JPA 설정을 살펴본다.

- 설정:
  - `pom.xml`: `dependencyManagement`에 `spring-data-bom` 추가
  - `application.properties`: 데이터베이스 연결 정보 설정, Hibernate 속성값 지정
  - `@Configuration`: JPA 설정 클래스 작성
  - `@EnableJpaRepositories`: Spring Data JPA를 사용하기 위한 설정
  - `@Transactional`: 트랜잭션 적용을 위한 어노테이션

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-bom</artifactId>
      <version>2021.2.0</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>

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

```
@Bean
public LocalContainerEntityManagerFactoryBean
entityManagerFactory(DataSource dataSource) {
    LocalContainerEntityManagerFactoryBean emf = new
    LocalContainerEntityManagerFactoryBean();
    emf.setDataSource(dataSource);
    emf.setPackagesToScan("com.nhnacademy.springjpa.entity");
    emf.setJpaVendorAdapter(jpaVendorAdapters());
    emf.setJpaProperties(jpaProperties());

    return emf;
}

private JpaVendorAdapter jpaVendorAdapters() {
    HibernateJpaVendorAdapter hibernateJpaVendorAdapter = new
    HibernateJpaVendorAdapter();
    hibernateJpaVendorAdapter.setDatabase(Database.H2);

    return hibernateJpaVendorAdapter;
}

private Properties jpaProperties() {
    Properties jpaProperties = new Properties();
    jpaProperties.setProperty("hibernate.show_sql", "true");
    jpaProperties.setProperty("hibernate.format_sql", "true");
}
```

```

        jpaProperties.setProperty("hibernate.use_sql_comments", "true");
        jpaProperties.setProperty("hibernate.globally_quoted_identifiers",
            "true");
        jpaProperties.setProperty("hibernate.temp.use_jdbc_metadata_defaults",
            "false");

        return jpaProperties;
    }

```

## Spring Data JPA 설정 예시

### 의존성 관리

- `dependencyManagement` 태그 내에서 `spring-data-bom`을 추가하여 Spring Data JPA에 필요한 라이브러리들의 버전을 일괄적으로 관리할 수 있다.
- `dependency` 태그 내에서 `spring-data-jpa` 라이브러리를 추가하여 Spring Data JPA를 사용할 수 있다.

### EntityManagerFactory 설정

- `LocalContainerEntityManagerFactoryBean`을 사용하여 `EntityManagerFactory`를 설정한다.
- `DataSource`를 주입받아 `emf.setDataSource(dataSource)` 메서드를 호출하여 `DataSource`를 설정한다.
- `emf.setPackagesToScan("com.nhnacademy.springjpa.entity")` 메서드를 호출하여 Entity 클래스들이 위치한 패키지를 설정한다.
- `jpaVendorAdapters()` 메서드를 호출하여 Hibernate JPA 구현체를 설정한다.
- `jpaProperties()` 메서드를 호출하여 Hibernate JPA 설정을 추가로 설정한다.
- 예제에서는 SQL 출력, SQL 포매팅, SQL 코멘트, Globally quoted identifiers 설정, JDBC metadata 설정 등을 추가로 설정하였다.## Bean Configuration

### Transaction Manager

Spring Framework은 트랜잭션을 추상화하여 다양한 방식으로 트랜잭션을 다룰 수 있게 해주는데, 그 중에서 `DataSourceTransactionManager`와 `JpaTransactionManager`를 살펴보겠습니다.

- `DataSourceTransactionManager`: JDBC의 `Connection`을 사용하여 트랜잭션을 다룹니다. 따라서 JDBC 기반의 프로그램에서 사용하기 적합합니다.
- `JpaTransactionManager`: JPA의 `EntityManager`를 사용하여 트랜잭션을 다룹니다. JPA 기반의 프로그램에서 사용하기 적합합니다.

`@Transactional` 어노테이션을 사용하여 선언적 트랜잭션을 사용할 수 있습니다. 이를 사용하면 메서드 단위로 트랜잭션을 관리할 수 있습니다.

```

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

```

```
        return transactionManager;
    }
```

## EntityManager

---

**EntityManager**은 엔티티의 저장, 수정, 삭제, 조회 등 엔티티와 관련된 모든 일을 처리하는 관리자입니다.

```
public interface EntityManager {
    public <T> T find(Class<T> entityClass, Object primaryKey);
    public <T> T find(Class<T> entityClass, Object primaryKey, Map<String,
Object> properties);
    public <T> T find(Class<T> entityClass, Object primaryKey,
LockModeType lockMode);
    public <T> T find(Class<T> entityClass, Object primaryKey,
LockModeType lockMode, Map<String, Object> properties);

    public void persist(Object entity);

    public <T> T merge(T entity);

    public void remove(Object entity);

    // ...
}
```

## EntityManagerFactory

---

**EntityManager**를 생성하는 팩토리

```
public interface EntityManagerFactory {
    public EntityManager createEntityManager();
    public EntityManager createEntityManager(Map map);
    public EntityManager createEntityManager(SynchronizationType
synchronizationType);
    public EntityManager createEntityManager(SynchronizationType
synchronizationType, Map map);

    // ...
}
```

## JPA/Hibernate Logging

---

### SQL

- JPA properties
  - `hibernate.show-sql=true`
  - `hibernate.format_sql=true`
- logback logger

```
<logger name="org.hibernate.SQL" level="debug" additivity="false">
  <appender-ref ref="console" />
</logger>
```

- binding parameters

```
<logger name="org.hibernate.type.descriptor.sql.BasicBinder" level="trace"
additivity="false">
  <appender-ref ref="console" />
</logger>
```

- 맵핑된 sql 파라미터도 보여주는 로거, 둘중 하나만 쓰면 됨
- cf.) `org.hibernate.type.descriptor.sql.BasicExtractor`

## Demo

- cf.) Spring 없이 JPA 사용하기
  - [예시](#)
- IntelliJ에서 Jakarta EE 프로젝트 시작
  - Project template: Library
  - Java EE 8 > Hibernate 선택
  - h2 DB 사용
  - View > Tool Windows > Persistence
  - persistence.xml 생성
  - User Entity 생성
    - 주의! `@Table(name = "Users")`
  - Main class 생성
  - EntityManagerFactort / EntityManager 를 이용해서 Entity 를 저장
  - h2 web console 에서 데이터 확인

## Entity 맵핑

---

### Entity / Entity 맵핑

- Entity란?
  - JPA를 이용해서 데이터베이스 테이블과 맵핑할 클래스
- Entity 맵핑
  - Entity 클래스에 데이터베이스 테이블과 컬럼, 기본 키, 외래 키 등을 설정하는 것

컬럼들은 클래스의 필드들과 매핑

- 어노테이션
  - `@Entity` : JPA가 관리할 객체임을 명시
  - `@Table` : 매핑할 DB 테이블 명 지정
  - `@Id` : 기본 키(PK) 매핑
  - `@Column` : 필드와 컬럼 매핑 (생략 가능)
- 예제

```
@Entity
//멤버스란 테이블이 자바의 멤버 클래스와 매핑
@Table(name = "Members")
public class Member {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @Column(name = "created_dt")
    private LocalDateTime createdAt;
}
~~~~~java
@Entity
@Table(name = "Members")
public class Member {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name; //컬럼어노테이션은 생략 가능

    @Column(name = "created_dt") //dbtable의 컬럼을 명시, 이름이 달라서
    private LocalDateTime createdAt;
}
```

## 필드와 컬럼 매핑

- `@Column` : 객체 필드를 컬럼에 매핑, 생략 가능
- `@Temporal` : 날짜 타입 매핑 (`DATE`, `TIME`, `TIMESTAMP`)
  - Java 8의 `date/time` (`LocalTime`, `LocalDate`, `ZonedDateTime`) 타입은 `@Temporal`을 붙이지 않는다.
- `@Transient` : 특정 필드를 컬럼에 매핑하지 않을 경우에 지정

## 도메인



![image](../image/스크린샷 2023-05-08 오후 1.28.29 1.png)

## 실습: Items 테이블에 대한 Entity 매핑

---

Items 테이블에 대한 Entity 매핑을 위해 Entity 클래스를 생성하고 컬럼 매핑을 해보세요.

`git checkout entity`

## 기본 키(Primary Key) 매핑 전략

---

### 자동 생성

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

### 직접 할당

- 애플리케이션에서 직접 식별자 값을 할당

### 예제

```
public class Item {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "item_id")
    private Long itemId;

    // ...
}

public @interface GeneratedValue {
    GenerationType strategy() default AUTO;
    String generator() default "";
}

public enum GenerationType {
    TABLE,
    SEQUENCE,
    IDENTITY,
    AUTO
}
```

### 실습

Orders 테이블에 대한 Entity 매핑

Orders 테이블에 대한 Entity 매핑을 위해 Entity 클래스를 생성하고 컬럼 매핑을 해봅시다.

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

    @Column(name = "order_date")
    private LocalDateTime orderDate;

    @Column(name = "status")
    private String status;

    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;

    // ...
}
```

### 복합 Key (Composite key)

복합 키란 둘 이상의 필드를 조합하여 기본 키(PK)를 생성하는 방식입니다. 복합 키를 사용하려면 `@IdClass` 어노테이션 또는 `@EmbeddedId`와 `@Embeddable` 어노테이션을 사용해야 합니다.

`@IdClass` 복합 키를 사용할 때는 `@IdClass` 어노테이션을 이용해 Entity class 레벨에서 지정해줍니다. `@Id` 어노테이션을 필드에 지정하며, 복합 키를 구성하는 모든 필드에 `@Id` 어노테이션을 붙여줍니다.

```
@Entity
@Table(name = "OrderItems")
@IdClass(OrderItem.Pk.class)
public class OrderItem {
    @Id
    @Column(name = "order_id")
    private Long orderId;

    @Id
    @Column(name = "line_number")
    private Integer lineNumber;

    // ...

    @NoArgsConstructor
    @AllArgsConstructor
    @EqualsAndHashCode
    public static class Pk implements Serializable {
        private Long orderId;
```

```

        private Integer lineNumber;
    }
}

```

@EmbeddedId / @Embeddable 복합 키를 사용할 때는 @EmbeddedId와 @Embeddable 어노테이션을 이용해 복합 키 식별자 클래스를 만들어줍니다. 복합 키 식별자 클래스에는 @Embeddable 어노테이션을 붙여주고, Entity 클래스의 필드에는 @EmbeddedId 어노테이션을 붙여줍니다.

```

@Entity
@Table(name = "OrderItems")
public class OrderItem {
    @EmbeddedId
    private Pk pk;

    // ...

    @Embeddable
    @NoArgsConstructor
    @AllArgsConstructor
    @EqualsAndHashCode
    public static class Pk implements Serializable {
        private Long orderId;
        private Integer lineNumber;
    }
}

```

복합키를 하나의 필드에서 사용할수 있다는 장점이 있다. 구조적으로 접근할수 있는 장점

```

@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode
@Embeddable
public static class Pk implements Serializable {
    @Column(name = "order_id")
    private Long orderId;

    @Column(name = "line_number")
    private Integer lineNumber;
}

```

### 복합 Key Class 제약조건

- PK 제약조건을 그대로 따름
- PK 제약 조건

- The primary key class must be public and must have a public no-arg constructor.
- The primary key class must be serializable.
- The primary key class must define equals and hashCode methods.

인스턴스같은지 확인해야해서 public이어야하고 기본 생성자가 반드시 존재해야한다. 그래야 key값을 jpa가 만들 수 있다.

저장되고 읽어야 하니까 e엔티티 매니저가 유일하게 구별해야해서 시리얼라이저블해야한다

서로 다른 인스턴스가 같은값인지 ( 동등한지 확인하기위해 내가 만든 유저인스턴스와 , 디비에서 가져온 것을 확인)확인하기위해 equals랑 hashCode 메소드가 구현되어야한다.

## 실습

OrderItems 테이블에 대한 Entity 매핑을 위해 Entity 클래스를 생성하고 컬럼 매핑을 해봅시다. 복합 Key 매핑을 위한 두 가지 방법을 모두 실습해봅시다.

- @IdClass
- @EmbeddedId / @Embeddable

## EntityManager / EntityManagerFactory

- EntityManagerFactory: EntityManager를 생성하는 팩토리
  - 데이터베이스를 하나만 사용하는 애플리케이션은 일반적으로 EntityManagerFactory를 하나만 사용
  - EntityManagerFactory를 만드는 비용이 매우 크기 때문에 하나만 만들어서 전체에서 공유 (thread-safe)
- EntityManager: Entity의 저장, 수정, 삭제, 조회 등 Entity와 관련된 모든 일을 처리하는 관리자
  - EntityManagerFactory가 생성 → 생성 비용이 크지 않다
  - EntityManager는 thread-safe하지 않음
  - 여러 thread 간에 절대 공유하면 안 됨
  - 각각의 요청마다 별도의 EntityManager를 생성해서 사용

데이터 베이스 클래스에서 나오는 인스턴스를 관리해주는게 @Entity

@Entity와 관련된 데이터와 관련된 쿼리를 자동으로 설정해준다.

엔티티매니저

반드시 싱글톤으로 써야할거같지만

트랜잭션 세이프하지 않다.

하나의 트랜잭션당 하나의 엔티티 매니저

여러개의 엔티티매니저가 돌아가기때문에 트랜잭션 매니저에서 jpa 엔티티 매니저를 써야 하는것이기 때문이다.

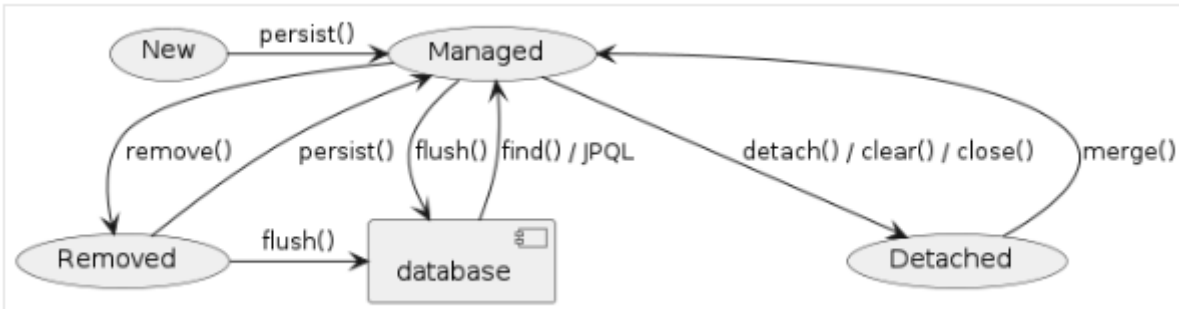
entity 매니저 팩토리는 어플리케이션 전체에 하나만 있으면 되기때문에 싱글톤으로 되었다

커넥션이 달라지면 공장이 하나 더있어야하고

커넥션이 같으면 공장 하나만 있으면 된다.

## 영속성 컨텍스트

- Entity를 영구 저장하는 환경
  - @PersistenceContext
- EntityManager가 관리하는 영역
- 영속성 컨텍스트에서 Entity의 생명주기
  - 비영속 (new/transient): 영속성 컨텍스트와 전혀 관계가 없는 상태
  - 영속 (managed): 영속성 컨텍스트에 저장된 상태
  - 준영속 (detached): 영속성 컨텍스트에 저장되었다가 분리된 상태
  - 삭제 (removed): 삭제하기 위해 표시한 상태



detach는 데이터베이스에 저장하지 않는다  
 remove 만으론 디비반영 x  
 flush를 하면 디비반영 o

- 영속성 컨텍스트에서 Entity의 생명주기
- 비영속 (new/transient): 영속성 컨텍스트와 전혀 관계가 없는 상태
- 영속 (managed): 영속성 컨텍스트에 저장된 상태
- 준영속 (detached): 영속성 컨텍스트에 저장되었다가 분리된 상태
- 삭제 (removed): 삭제하기 위해 표시한 상태

## 영속성 컨텍스트가 Entity를 관리하면 얻을 수 있는 이점

- 1차 캐시
- 동일성 보장
- 트랜잭션을 지원하는 쓰기 지연
- 변경 감지
- 지연 로딩

1. 엔티티 매니저랑 상관없이 new 로 생성된 객체는 비영속상태
2. persist() 메소드를 통하면 엔티티매니저가 관리하는 상태로 변환(db 에 들어간 상태는 아님) -> flush 후 db에 저장
3. 추상화된 리포지토리를 보면 entitymanager를 불일이 없다??? 스프링 레포지토리에는 save save delete find 엔티티매니저 => 스프링 레포지토리

```
persist,merge => save => saveAndFlush()  
remove => delete  
find = > find
```

## @PersistenceContext

- EntityManager를 주입받기 위한 어노테이션
- 주입받은 EntityManager는 트랜잭션 내에서 사용되며, 트랜잭션이 종료되면 자동으로 플러시되어 DB에 반영된 게시판 데이터베이스 테이블에 대해 Entity 매핑해보겠습니다.

게시판 테이블 정보:

- 테이블 이름: board
- 컬럼 정보:
  - id (PK, 자동 생성)
  - title (VARCHAR)
  - content (TEXT)
  - writer (VARCHAR)
  - created\_at (DATETIME)
  - updated\_at (DATETIME)

Entity 클래스는 다음과 같이 작성할 수 있습니다.

```
@Entity  
@Table(name = "board")  
public class Board {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    private String title;  
  
    @Lob  
    private String content;  
  
    private String writer;  
  
    @Column(name = "created_at")  
    private LocalDateTime createdAt;  
  
    @Column(name = "updated_at")  
    private LocalDateTime updatedAt;  
  
    // getters and setters  
}
```

위의 코드에서 @Lob 어노테이션은 콘텐츠가 긴 경우 TEXT 데이터 타입을 사용하기 위한 것입니다.

```
1. stereotype이 뭔가요
2. @Bean은 뭔가요
3.
dataSource.setInitialSize(10); // 초기 커넥션풀 갯수
dataSource.setMaxTotal(10); // 최대 커넥션 풀 갯수
dataSource.setMinIdle(10); // 놓고있는 커넥션의 최소 갯수
dataSource.setMaxIdle(10); // 놓고있는 커넥션풀의 최대 갯수
//베스트는 모두 갯수를 일치시키는게 베스트(보통 200으로 셋팅)
dataSource.setMaxWaitMillis(1000); // 커넥션풀이 바쁠때 대기시간
4. 커넥션 풀은 커넥션에 드는 비용이 많아서 사용
5.
dataSource.setTestOnBorrow(true); // 커넥션풀에서 커넥션을 가져올때 살아있는지 확인
dataSource.setTestOnReturn(true); // 못쓰는건지 확인? 하고 반환
dataSource.setTestWhileIdle(true); // 주기적으로 살아있는지 확인
//약간의 성능저하가 있을 수도 있다.
// 그래도 커넥션 비용보다 낫다
6. dataSource.setUrl("jdbc:h2:~/spring-jpa;DATABASE_TO_UPPER=false;"
    + "INIT=RUNSCRIPT FROM 'classpath:/script/schema.sql'");
6. 유닛테스트는 관심있는 부분만 테스트
7. 통합테스트는 외부시스템을 다 통합하고 테스트
```

<sup>1</sup>: JPA는 Java에서 ORM을 사용하기 위한 API 스펙입니다. JPA는 인터페이스와 애노테이션을 정의하며, 여러 구현체가 있습니다. 대표적인 구현체로는 Hibernate, EclipseLink, OpenJPA 등이 있습니다. ([돌아가기](#))