

chpt5

Mybatis 사용해보기

Mybatis 소개

설명출처블로그

자바에서 제공하는 JDBC를 보다 편리하게 사용할 수 있도록 해주는 프레임워크

- SQL 쿼리문을 코드 내에서 쓰지 않고 Mapper 파일에서 관리함으로써 코드와 SQL 쿼리를 분리
- SQL 쿼리 수정 시 코드를 직접 수정하지 않아도 돼 유지보수 시 안정성을 높임
- 코드를 간소화하여 가시성을 높여줌
=>간단히 자바와 스프링에서 DB를 다루는 JDBC를 좀 더 편하고 깔끔하게 다루기 위한 DB 연동 프레임워크 또는 라이브러리
특히 스프링에서는 Mybatis를 사용하기 위한 모듈을 제공하므로 더욱 편하게 사용할 수 있습니다. DAO를 대신할 mapper를 만들 때 매우 편리합니다.

출처블로그

ORM(Object Relation Mapping) : 객체와 관계형 데이터 베이스 간의 매핑을 지원하는 것

개발자가 지정한 SQL, 저장프로시저, 몇가지 고급 매핑을 지원하는 프레임워크

기존 JDBC를 이용하여 프로그래밍을 하면 소스안에 SQL문을 작성했지만, Mybatis에서는 SQL을 XML 파일에 작성하기 때문에 SQL의 변환이 자유롭고 가독성이 좋다는 장점이 있다.

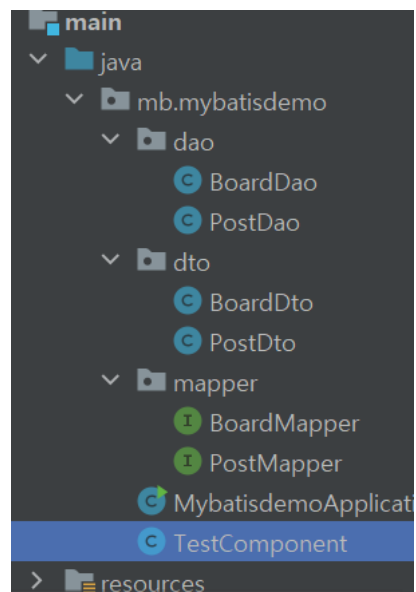
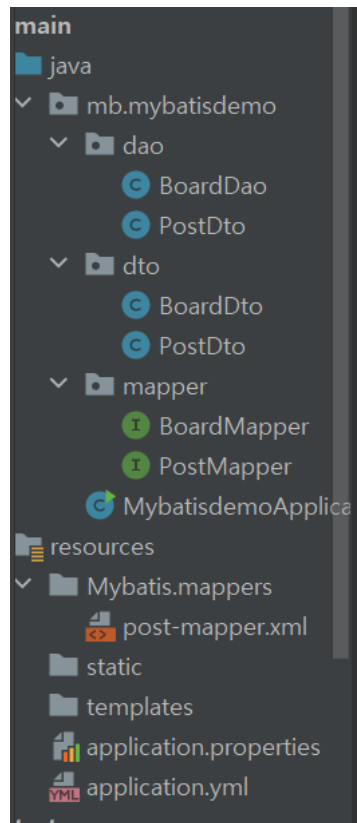
출처블로그

1. 이클립스와 연동하기 위한 DB작성(스키마+user와board테이블)
2. DB를 객체화 하기 위한 dto클래스와 mapper작성(mapper클래스+mapper.xml)
3. mapper를 mybatis가 알아들을 수 있게 서로 연동해주는 MybatisConfig클래스 작성

Mybatis로 Database 사용해보기

- spring initializer에서 spring web, mySql framework, mybatis 선택 후 생성
- yml 파일 작성해주고 폴더 구조 다음 사진과 같이

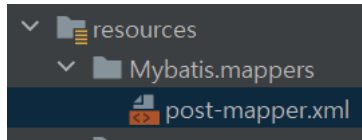
(위에는 BEFORE 이건 AFTER
)



```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://127.0.0.1:3306/demo_schema
    username: {mysql 아이디}
    password: {mysql 비밀번호}

mybatisdemo :
  mapper-locations: "classpath:mybatis/mappers/*.xml"
  configuration:
    map-underscore-to-camel-case : true
```

- resource에 mybatis-mapper & xml 파일 생성 & xml 채우기



메이븐 "pom.xml"에 DB연동에 필요한 라이브러리 의존 설정 추가

- MySQL Connector
- Mybatis
- Mybatis-spring (스프링에서 Mybatis 연동을 위한 모듈)
- spring-jdbc (기본 자바 JDBC가 아닌 스프링의 JDBC)
- common-dbcp2 (톰캣에서 커넥션풀을 이용할 수 있도록 아파치에서 제공하는 라이브러리)
- spring-test (스프링에 Mybatis가 정상적으로 연동되었는지 확인 용도)

```
<?xml version="1.0" encoding="UTF-8"?> <!--xml 버전 명시-->
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="mb.mybatisdemo.mapper.PostMapper">
</mapper>
```

- **dao** : data access object
=> 데이터를 주고 받게 해주는 스프링의 레포지토리 기능, 객체 클래스를 불러오는 것
=> 스프링의 다른 애플리케이션과 소통하기 위한 클래스
- **dto** : 데이터를 실제로 담기위한 것
- **mapper** : mapper.xml에서 작성한 애들을 postmapper, boardmapper의 interface 역할
=> mapper 폴더 안 정의된 애들은 우리가 xml에서 정의해준 것에 따라서 작동이 되는 방식

출처Mybatis XML을 사용해서 코딩 순서

테이블 생성 및 설정

도코메인 객체의 설계 및 클래스 작성

DAO 인터페이스 / 실행 기능 인터페이스 정의

XML Mapper 생성 및 SQL문 작성

XML 작성

MyBatis에 작성한 XML Mapper 인식 설정

DAO 인터페이스 구현한 클래스 작성

스프링에 DAO 등록

아래 설명 출처

DAO(Data Access Object)

- 데이터베이스의 data에 접근하기 위한 객체이며 데이터베이스 접근을 하기 위한 로직과 비즈니스 로직을 분리하기 위해 사용한다.
- 사용자는 자신이 필요한 Interface를 DAO에게 던지고 DAO는 이 Interface를 구현한 객체를 사용자에게 편리하게 사용할 수 있도록 반환한다.
DAO는 데이터베이스와 연결할 Connection까지 설정되어 있는 경우가 많다.
- 그래서 현재 쓰이는 MyBatis 등을 사용할 경우 커넥션풀까지 제공되고 있기 때문에 DAO를 별도로 만드는 경우는 드물다.

DTO(Data Transfer Object)

- VO라고도 표현하며 계층 간 데이터 교환을 위한 자바 빈즈(Java Beans)이다.
- 데이터베이스 레코드의 데이터를 매핑하기 위한 데이터 객체를 말한다. DTO는 보통 로직을 가지고 있지 않고 data와 그 data에 접근을 위한 getter, setter만 가지고 있다.

- 정리하면 DTO는 Database에서 Data를 얻어 Service나 Controller 등으로 보낼 때 사용하는 객체를 말한다.

```
public class PersonDTO {
    private String name;
    private int age;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

- 위의 클래스를 보면 getter/setter가 존재한다. 여기서 중요한 건 Property(프로퍼티) 개념인데 자바는 Property가 문법적으로 제공되지 않는다.
- 자바에서 Property라는 개념을 사용하기 위해 지켜야 할 약속이 있다.
- setter/getter에서 set과 get 이후에 나오는 단어가 Property라고 약속하는 것이다.
- 그래서 위 클래스에서 프로퍼티는 name과 age이다.
- 중요한 것은 프로퍼티가 멤버 변수 name, age로 결정되는 것이 아닌 getter/setter에서의 name과 age 임을 명심해야 한다.
- 즉 멤버 변수는 아무렇게 지어도 영향이 없고 getter/setter로 프로퍼티(데이터)를 표현한다는 것이다.
- 자바는 다양한 프레임워크에서 데이터 자동화 처리를 위해 리플렉션 기법을 사용하는데, 데이터 자동화 처리에서 제일 중요한 것은 표준 규격이다. 예를 들어 위 클래스 DTO에서 프로퍼티가 name, age라면 name, age의 키값으로 들어온 데이터는 리플렉션 기법으로 setter를 실행시켜 데이터를 넣을 수 있다.
- 중요한 것은, 우리가 setter를 요청하는 것이 아닌 프레임워크 내부에서 setter가 실행된다는 점이다.
- 그래서 layer 간(특히 서버 => 뷰로 이동 등)에 데이터를 넘길 때 DTO를 쓰면 편하다는 것이 이런 이유 때문이다. 뷰에 있는 form에서 name 필드 값을 프로퍼티에 맞춰 넘겼을 때 받아야 하는 곳에서 일일이 처리하는 것이 아니라 name 속성의 이름과 매칭되는 프로퍼티에 자동적으로 DTO가 인스턴스화되어 PersonDTO를 자료형으로 값을 받을 수 있다.

DTO(Data Transfer Object) 란 ? - 풀 네임에서 유추할 수 있듯이 데이터를 객체로 만들어주는 클래스이다.

- 우리가 MySQL에 테이블의 형태로 만든 데이터들을
- import com.example.demo.dto.BoardDto로 import하고
- BoardDto board 이렇게 객체를 선언해주면
- board.setTitle("제목") 이런식으로 객체화하여 사용할 수 있게 된다.
- DAO(Data Access Object) 란 ? - 데이터 베이스에 접속해서 데이터 추가, 삭제, 수정 등의 작업을 하는 클래스
- 일반적인 JSP 혹은 Servlet 페이지내에 위의 로직을 함께 기술할 수 도 있지만, 유지보수 및 코드의 모듈화를 위해 별도의 DAO클래스를 만들어 사용 한다. 실제로 DB에 insert,update,delete등의 작업을 해주는 클래스라고 보면 될듯하다.
- Mybatis나 JDBC나 모두 웹 프로그램을 DB와 접속시켜주는 기능을 하는데 JDBC의 복잡성을 Mapping으로 개선하여 쉽게 사용하게 만들어주는게 Mybatis의 기능이다.

Mybatis 제어문

(1) postmapper & postdto 부분 (인터페이스 부분 완성하는 것)

1. postmapper.xml

```
<?xml version="1.0" encoding="UTF-8"?> <!--xml 버전 명시-->
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="mb.mybatisdemo.mapper.PostMapper">
    <insert id="createPost" parameterType="mb.mybatisdemo.dto.PostDto">
        insert into POST(title, content, title, writer,board)
        values (title, content, writer, board)
```

```

    </insert>
</mapper>

-> PostMapper에서 id createPost에 이 SQL문을 적용해라
-> Parameter type은 PostMapper안에서 createPost가 받아들일 + 사용할 parameter 값을 지정해주는 것

```

- SQL 문 문법으로 써주기
- **중요한 건 ID 값 & mapper**
- 위의 두가지를 가지고 어떤 함수에 xml에 작성된 SQL문을 적용해줄지 결정해줄거라서 ID와 위에 선언된 NAMESPACE(MAPPER INTERDACE) 잘 작성이 중요함 =>

1. 1 바탕으로 PostMapper작성

```

package mb.mybatisdemo.mapper;

import mb.mybatisdemo.dto.PostDto;

public interface PostMapper {
    int createPost(PostDto dto);
}

```

- > xml에서 아이디값 일치 / 인터페이스 가리키는 것 일치/ param 일치 -> xml 적용하게 됨

1. PostDto 만들어주기

```

package mb.mybatisdemo.dto;

/*
id int
title varchar
content varchar
writer varchar
board varchar
*/
public class PostDto {
    private int id;
    private String title;
    private String content;
    private String writer;
    private String board;

    public PostDto() {
    }

    public PostDto(int id, String title, String content, String writer, String board) {
        this.id = id;
        this.title = title;
        this.content = content;
        this.writer = writer;
        this.board = board;
    }

    public int getId() {
        return id;
    }

    public String getContent() {
        return content;
    }

    public String getTitle() {
        return title;
    }

    public String getWriter() {
        return writer;
    }

    public String getBoard() {
        return board;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}

```

```

    public void setContent(String content) {
        this.content = content;
    }

    public void setWriter(String writer) {
        this.writer = writer;
    }

    public void setBoard(String board) {
        this.board = board;
    }

    @Override
    public String toString() {
        return "PostDto{" +
            "id=" + id +
            ", title='" + title + '\'' +
            ", content='" + content + '\'' +
            ", writer='" + writer + '\'' +
            ", board='" + board + '\'' +
            '}';
    }
}

```

- 이제 postdto 있으니 mapper와 PostDtoMapper에서 애네를 사용해줄 때
근데 이때 xml 파일에서

```

insert into POST(title, content, title, writer,board)
values (title, content, writer, board)

```

이렇게 되어있는데 이렇게 되어있으면 postdto의 변수가 가는게 아니라 title이라는 문자열 그자체가 전달이 되는 것이다 이를 변수로 인식
되게 하려면 아래처럼 코드 수정 needed

=> 이런 식으로 이제 CRUD 작성 (XML & MAPPER)

xml

```

<?xml version="1.0" encoding="UTF-8"?> <!--xml 버전 명시-->
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="mb.mybatisdemo.mapper.PostMapper">

    <insert id="createPost" parameterType="mb.mybatisdemo.dto.PostDto">

        insert into POST(title, content, title, writer,board)
        values (#{title}, #{content}, #{writer}, ${board})
    <!--문자열의 경우는 #{dto의변수명} 이거는 따옴표를 추가해줌-->
    <!--board는 조인 테이블을 위한 id 값 , 애는 따옴표 추가 안함-${}->
    </insert>

    <select id="readPost"
        parameterType="int"
        resultType="mb.mybatisdemo.dto.PostDto">
        select * from post where id = ${id}
    </select><!--하나만 가져오는 경우-->

    <select id="readPostAll"
        resultType="mb.mybatisdemo.dto.PostDto">
        select * from post <!--list의 구현체로 반환하게 됨-->
    </select>

    <update id="updatePost"
        parameterType="mb.mybatisdemo.dto.PostDto">
        update post set
            title=#{title},
            content=#{content},
            writer=#{writer},
            board=#{board}
        where id=${id}
    </update>

    <delete id="deletePost" parameterType="int">
        delete from post where id=${id}
    </delete>
</mapper>

```

postmapper

```
package mb.mybatisdemo.mapper;

import mb.mybatisdemo.dto.PostDto;

import java.util.List;

public interface PostMapper {
    int createPost(PostDto dto);
    PostDto readPost(int id);
    List<PostDto> readPostAll();
    int updatePOST(PostDto dto);
    int deletePOST(PostDto dto);
}
```

=> mapper에서 insert, delete, update return 값으로 int 써주기, 결과로 몇개의 row가 영향 받았는지 알려주기 때문에

(2) postdao 다루기 - 실제로 mapper 사용해서 통신하는 클래스 부분

postdao

```
package mb.mybatisdemo.dao;

import org.apache.ibatis.session.SqlSessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

@Repository
public class PostDao {
    private final SqlSessionFactory sessionFactory;

    public PostDao(
        @Autowired SqlSessionFactory sessionFactory
    ){
        this.sessionFactory=sessionFactory;
    }
}
```

=> 이때 sessionFactory 사용 위해서는 spring ioc 관리하에 있어야 하니까 따라서 @Component 중에서도 데이터 주고받는 클래스, 컴포넌트라는 것을 명시해줄 @Repository 사용

- 이제 createPost부터 만들기

```
PostDao {
    private final SqlSessionFactory sessionFactory;

    public PostDao(
        @Autowired SqlSessionFactory sessionFactory
    ){
        this.sessionFactory=sessionFactory;
    }

    public int createPost(PostDto dto){
        SqlSession session = sessionFactory.openSession();
        //이제 xml과 mapping된 PostMapper 넣어주기
        PostMapper mapper = session.getMapper(PostMapper.class);
        //PostMapper 할당해줄 건데, 애플 세션에서
        // PostMapper랑 동일한 객체, 구현체를 달라하면
        // PostMapper 구현한 구현체를 주게됨
        int rowAffected= mapper.createPost(dto);
        //DB유지하기 위해선 세션 유지, 이 세션 유지 안하고
        //한번 통신하고 닫아주는 것이 아래
        session.close(); //이렇게 해주면 다음에 또 이 닫은 세션 열어서 활용 가능
        return rowAffected;
    }
}
```

위와 같은 createPost를 아래와 같이 변경하기도 가능

```
public int createPost(PostDto dto){
    try (SqlSession session = sessionFactory.openSession()){
        PostMapper mapper=session.getMapper(PostMapper.class);
        return mapper.createPost(dto);
    }
}
```

=> open, close 알아서 해줌 (close가 IOException임)

```

public PostDto readPost(int id){
    try (SqlSession session = sessionFactory.openSession()){
        PostMapper mapper=session.getMapper(PostMapper.class);
        //왜 굳이 sessionFactory에서 세션을 열고 닫지?
        //강 매퍼만 바로 사용하면 안됨?이라는 의문들기 가능
        //mapperinstance는 threadsafe하지 않음
        return mapper.readPost(id);
    }
}

public List<PostDto> readPostAll(){
    try (SqlSession session = sessionFactory.openSession()){
        PostMapper mapper=session.getMapper(PostMapper.class);
        return mapper.readPostAll();
    }
}

public int updatePost(PostDto dto) {
    try (SqlSession session = sessionFactory.openSession()) {
        PostMapper mapper = session.getMapper(PostMapper.class);
        return mapper.updatePost(dto);
    }
}

public int deletePost(int id) {
    try (SqlSession session = sessionFactory.openSession()) {
        PostMapper mapper = session.getMapper(PostMapper.class);
        return mapper.deletePost(id);
    }
}
}

```

- 이를 TEST 해주기 위해서 MAIN 함수 밑에 TESTCOMPONENT 생성 후 코드 작성

```

package mb.mybatisdemo;

import mb.mybatisdemo.dao.PostDao;
import mb.mybatisdemo.dto.PostDto;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.util.List;

@Component
public class TestComponent {
    private final PostDao postdao;
    public TestComponent(
        @Autowired PostDao postdao) {
        this.postdao=postdao;
        PostDto newPost = new PostDto();
        newPost.setTitle("From mybatis");
        newPost.setContent("Hello batis");
        newPost.setWriter("shucream");
        newPost.setBoard(0);
        this.postdao.createPost(newPost);

        List<PostDto> postDtoList=this.postdao.readPostAll();
        System.out.println(postDtoList.size()-1);

        PostDto firstPost = postDtoList.get(0);
        firstPost.setContent("content updated by batis");
        postdao.updatePost(firstPost);

        System.out.println(this.postdao.readPost(firstPost.getId()));
    }
}

```

=> 근데 자꾸 내가 아래 명시해 둔 에러 뜨며 작동이 안되네 일단 파일 삭제시켜두고 보류..

(+) xml에 sql 문 하나더 추가

```

<select id="readPostQuery"
        parameterType="mb.mybatisdemo.dto.PostDto"
        resultType="mb.mybatisdemo.dto.PostDto">

    select * from post
    where title = #{title}
    <if test="writer!=null">
        and writer = #{writer}
    </if>
</select>

```


- xml에 추가했으면 PostMapper에도 추가

```
public interface PostMapper{
    int createPost(PostDto dto);
    PostDto readPost(int id);
    List<PostDto> readPostAll();
    PostDto readPostQuery(PostDto dto);//추가
    int updatePost(PostDto dto);
    int deletePost(int id);
}
```

- 이번에는 createall도 추가

```
<insert id="createPostAll"
        parameterType="mb.mybatisdemo.dto.PostDto">

    insert into POST(title, content, title, writer,board)
    values
    <foreach collection="list" item="item" separator=",">
        <!--separator 기준 : , 기준으로 나누기
        & 하나의 컬렉션에서 , 으로 나뉜 각각 애들을 item으로 부르겠삼-->
        (#{item.title}, #{item.content}, #{item.writer}, ${item.board})
    </foreach>
</insert>
```

- 인터페이스에도 추가

```
public interface PostMapper{
    int createPostAll(List<PostDto> dtoList);
    int createPost(PostDto dto);
    PostDto readPost(int id);
    List<PostDto> readPostAll();
    PostDto readPostQuery(PostDto dto);//추가
    int updatePost(PostDto dto);
    int deletePost(int id);
}
```

(3) BoardMapper xml & 인터페이스 만들기

boarddto

```
package mb.mybatisdemo.dto;

public class BoardDto {
    private int id;
    private String name;

    public BoardDto(){

    }
    public BoardDto(int id, String name){
        this.id=id;
        this.name=name;
    }

    public int getId() {
        return id;``
    }

    public String getName() {
        return name;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "BoardDto{" +
            "id=" + id +
            ", name=" + name + '\'' +

```

```
}
    '});
```

board.xml

```
<?xml version="1.0" encoding="UTF-8"?> <!--xml 버전 명시-->
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="mb.mybatisdemo.mapper.BoardMapper">
    <insert id="createBoard"
        useGeneratedKeys="true"
        keyProperty="id"
        parameterType="mb.mybatisdemo.dto.BoardDto"
        >

        insert into board(name) values ({name})

    </insert>
```

BoardMapper

```
package mb.mybatisdemo.mapper;

import mb.mybatisdemo.dto.BoardDto;

public interface BoardMapper {
    int createBoard(BoardDto dto);
}
```

boarddao

```
package mb.mybatisdemo.dao;

import mb.mybatisdemo.dto.BoardDto;
import mb.mybatisdemo.mapper.BoardMapper;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.springframework.beans.factory.annotation.Autowired;

public class BoardDao {
    private final SqlSessionFactory sessionFactory;
    public BoardDao(@Autowired SqlSessionFactory sessionFactory)
    {
        this.sessionFactory=sessionFactory;
    }

    public int createBoard(BoardDto dto){
        try(SqlSession session = sessionFactory.openSession()){
            BoardMapper mapper = session.getMapper(BoardMapper.class);
            return mapper.createBoard(dto);
        }
    }
}
```

- 이를 테스트하기 위한 testcomponent

코드를 입력하세요

db에서는 fk가 하나의 pk를 바라본다
=> fk, pk에 대한 개념

Auto Generated Keys

board.xml에서

```
<insert id="createBoard"
    useGeneratedKeys="true"
    keyProperty="id"
    parameterType="mb.mybatisdemo.dto.BoardDto"
    >
    <insert id="createBoard"
        useGeneratedKeys="true"
```

```
keyProperty="id"
parameterType="mb.mybatisdemo.dto.BoardDto"
>
```

- > id property를 가진 키를 useGeneratedKeys를 통해 생성하라는 것
(+)mybatis useGeneratedKeys를 이용해서 auto_increment 값을 얻기
출처블로그
- Mapper xml 파일을 보니까, 이 부분이 insert를 하는 부분으로 보여요. 이 부분을 바꿔볼 건데요. 문서에, useGeneratedKeys라는 것이 있어요. 이것은, JDBC의 getGeneratedKeys를 이용한다는 옵션입니다. 그러면 이 메서드는 또 뭔지 볼까요? 문서를 보시면, 자동 생성 키값들을 사용하기 위해서 사용된다는 것을 알 수 있어요. 문서를 다시 보시면, useGeneratedKeys를 설명하는 부분에 갑자기 eg. 가 보입니다.
- 테이블의 어느 속성들을 받을 것인지 따로 정할 수도 있는데요. 이것은 keyColumn으로 설정

에러

Caused by: org.apache.ibatis.binding.BindingException: Type interface mb.mybatisdemo.mapper.PostMapper is not known to the MapperRegistry.

- testcomponent를 만들고 돌려보니 이런 에러 발생
=> 보통은 xml에서 지정한 인터페이스 함수와 내가 만든 인터페이스 함수명이 일치하지 않을 때 발생
=> 하지만 나는 이름이 아주 똑같았다.
- 이번엔 이 에러

Execution failed for task ':MybatisdemoApplication.main()'.
Process 'command 'C:/Users/DONGYUN/.jdk/corretto-11.0.14/bin/java.exe' finished with non-zero exit value 1

```
package mb.mybatisdemo;

import mb.mybatisdemo.dao.BoardDao;
import mb.mybatisdemo.dao.PostDao;
import mb.mybatisdemo.dto.BoardDto;
import mb.mybatisdemo.dto.PostDto;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.util.List;

@Component
public class TestComponent {
    private final PostDao postdao;
    //private final BoardDao boardDao;

    public TestComponent(
        @Autowired PostDao postdao, BoardDao boardDao) {
        this.postdao = postdao;
        //this.boardDao = boardDao;
    }

    //
    //      BoardDto boarddto = new BoardDto();
    //      boarddto.setName("new board");
    //      this.boardDao.createBoard(boarddto);
    //      System.out.println(boarddto.getId());

    PostDto newPost = new PostDto();
    newPost.setTitle("From mybatis");
    newPost.setContent("Hello batis");
    newPost.setWriter("shucream");
    newPost.setBoard(0);
    this.postdao.createPost(newPost);

    List<PostDto> postDtoList=this.postdao.readPostAll();
    System.out.println(postDtoList.size()-1);

    PostDto firstPost = postDtoList.get(0);
    firstPost.setContent("content updated by batis");
    postdao.updatePost(firstPost);

    System.out.println(this.postdao.readPost(firstPost.getId()));
```

```
}  
}
```

=> 저녁먹고 반드시 에러원인찾아낸다

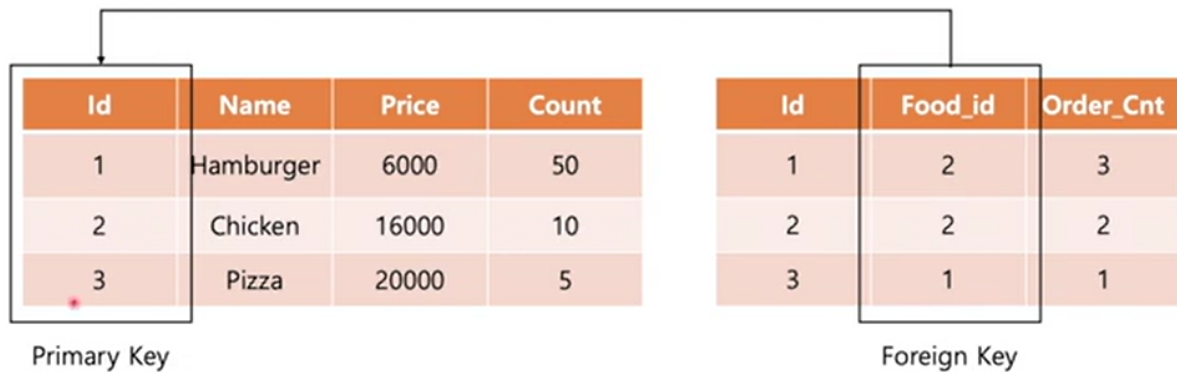
Caused by: org.springframework.beans.BeanInstantiationException: Failed to instantiate [mb.mybatisdemo.TestComponent]: Constructor threw exception; nested exception is org.apache.ibatis.binding.BindingException: Type interface mb.mybatisdemo.mapper.PostMapper is not known to the MapperRegistry.

- 이번엔 빈 주입에러

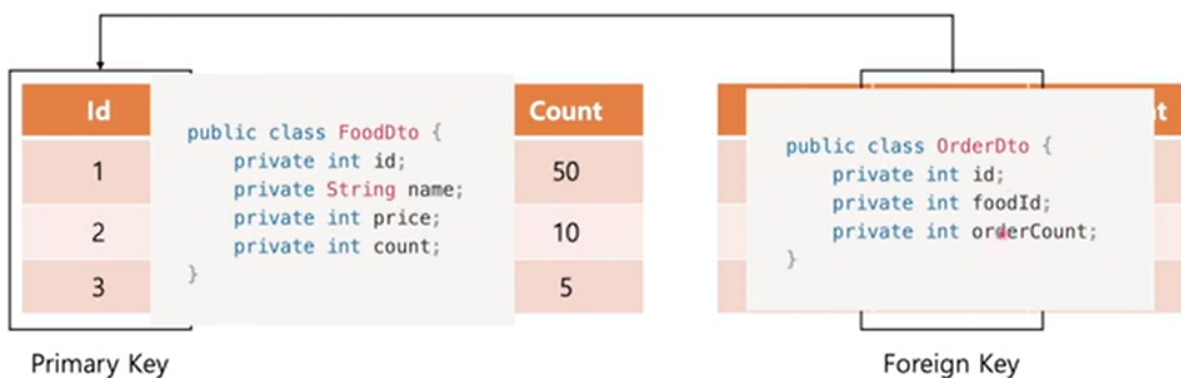
```
package mb.mybatisdemo;  
  
import mb.mybatisdemo.dao.BoardDao;  
import mb.mybatisdemo.dao.PostDao;  
import mb.mybatisdemo.dto.BoardDto;  
import mb.mybatisdemo.dto.PostDto;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
  
import java.util.List;  
  
@Component  
public class TestComponent {  
    private final PostDao postdao;  
    //private final BoardDao boardDao;  
  
    public TestComponent(  
        @Autowired PostDao postdao, BoardDao boardDao) {  
        this.postdao = postdao;  
        //this.boardDao = boardDao;  
  
        //  
        //        BoardDto boarddto = new BoardDto();  
        //        boarddto.setName("new board");  
        //        this.boardDao.createBoard(boarddto);  
        //        System.out.println(boarddto.getId());  
  
        PostDto newPost = new PostDto();  
        newPost.setTitle("From mybatis");  
        newPost.setContent("Hello batis");  
        newPost.setWriter("shucream");  
        newPost.setBoard(0);  
        this.postdao.createPost(newPost);  
  
        List<PostDto> postDtoList=this.postdao.readPostAll();  
        System.out.println(postDtoList.size()-1);  
  
        PostDto firstPost = postDtoList.get(0);  
        firstPost.setContent("content updated by batis");  
        postdao.updatePost(firstPost);  
  
        System.out.println(this.postdao.readPost(firstPost.getId()));  
    }  
}
```

ORM

관계형 데이터베이스의 한계



자바 코드로 표현한 버전



```

public class Food {
    private String name;
    private int price;
    private int count;
}
    
```

```

public class Order {
    private Food orderFood;
    private int orderCount;
}
    
```

- 주문은 음식을 fk로 갖고 있어야 하므로 Food를 자신의 클래스 안에 가져야만 한다

(+) 설명 출처 블로그

- JPA(Java Persistence API)가 무엇인지 알아보려고한다. JPA는 자바 진영에서 ORM(Object-Relational Mapping) 기술 표준으로 사용되는 인터페이스의 모음이다. 그 말은 즉, 실제로 구현된것이 아니라 구현된 클래스와 매핑을 해주기 위해 사용되는 프레임워크이다. JPA를 구현한 대표적인 오픈소스로는 Hibernate가 있다.
- ORM(Object-Relational Mapping)
우리가 일반적으로 알고 있는 애플리케이션 Class와 RDB(Relational DataBase)의 테이블을 매핑(연결)한다는 뜻이며, 기술적으로는 어플리케이션의 객체를 RDB 테이블에 자동으로 영속화 해주는 것이라고 보면된다.

장점

SQL문이 아닌 Method를 통해 DB를 조작할 수 있어, 개발자는 객체 모델을 이용하여 비즈니스 로직을 구성하는데만 집중할 수 있음.

(내부적으로는 쿼리를 생성하여 DB를 조작함. 하지만 개발자가 이를 신경 쓰지 않아도됨)

Query와 같이 필요한 선언문, 할당 등의 부수적인 코드가 줄어들어, 각종 객체에 대한 코드를 별도로 작성하여 코드의 가독성을 높임

객체지향적인 코드 작성이 가능하다. 오직 객체지향적 접근만 고려하면 되기때문에 생산성 증가

매핑하는 정보가 Class로 명시 되었기 때문에 ERD를 보는 의존도를 낮출 수 있고 유지보수 및 리팩토링에 유리
 예를들어 기존 방식에서 MySQL 데이터베이스를 사용하다가 PostgreSQL로 변환한다고 가정해보면, 새로 쿼리를 짜야하는 경우가 생김. 이런 경우에 ORM을 사용한다면 쿼리를 수정할 필요가 없음

단점

프로젝트의 규모가 크고 복잡하여 설계가 잘못된 경우, 속도 저하 및 일관성을 무너뜨리는 문제점이 생길 수 있음
 복잡하고 무거운 Query는 속도를 위해 별도의 튜닝이 필요하기 때문에 결국 SQL문을 써야할 수도 있음
 학습비용이 비쌈

Object Relational Mapping

=> 근데 관계형 데이터베이스에서 사용하는 자료 형태는 객체 지향 관점에서 맞지 않음
 ---> 이 때문에 ORM(Object Relational Mapping) 등장 => 관계형 데이터를 객체로 표현하는 프로그래밍 기법

Id	Name	Price	Count
1	Hamburger	6000	50
2	Chicken	16000	10
3	Pizza	20000	5



```
public class Food {
    private String name;
    private int price;
    private int count;
}
```

Id	Food_id	Order_Cnt
1	2	3
2	2	2
3	1	1



```
public class Order {
    private Food orderFood;
    private int orderCount;
}
```

=> 이를 위해 등장한 것 : JPA

- 이는 직접 ORM을 구현해주는 것은 아님
 : 이미 존재하는 자바의 객체들에 대해 데이터 상의 테이블 내에는 어떻게 표현이 될 지 정해주는 ANNOTATION

```
@Entity
@DynamicUpdate
public class FoodEntity extends BaseEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private Long price;
    private Long count;
}
```

- 위의 네개가 JPA라는 API가 제공해주는 어노테이션
- JPA 자체는 관계형 데이터를 객체로 표시해주는 기능

(+) JPA 추가 설명 [설명 출처 블로그](#)

- JPA는 기술 명세이다
- JPA는 Java Persistence API의 약자로, 자바 어플리케이션에서 관계형 데이터베이스를 사용하는 방식을 정의한 인터페이스이다. 여기서 중요하게 여겨야 할 부분은, JPA는 말 그대로 인터페이스라는 점이다. JPA는 특정 기능을 하는 라이브러리가 아니다. 마치 일반적인 백엔드 API가 클라이언트가 어떻게 서버를 사용해야 하는지를 정의한 것처럼, JPA 역시 자바 어플리케이션에서 관계형 데이터베이스를 어떻게 사용해야 하는지를 정의하는 한 방법일 뿐이다.
- JPA는 단순히 명세이기 때문에 구현이 없다. JPA를 정의한 javax.persistence 패키지의 대부분은 interface, enum, Exception, 그리고 각종 Annotation으로 이루어져 있다. 예를 들어, JPA의 핵심이 되는 EntityManager는 아래와 같이 javax.persistence.EntityManager 라는 파일에 interface로 정의되어 있다.

JPA Hibernate

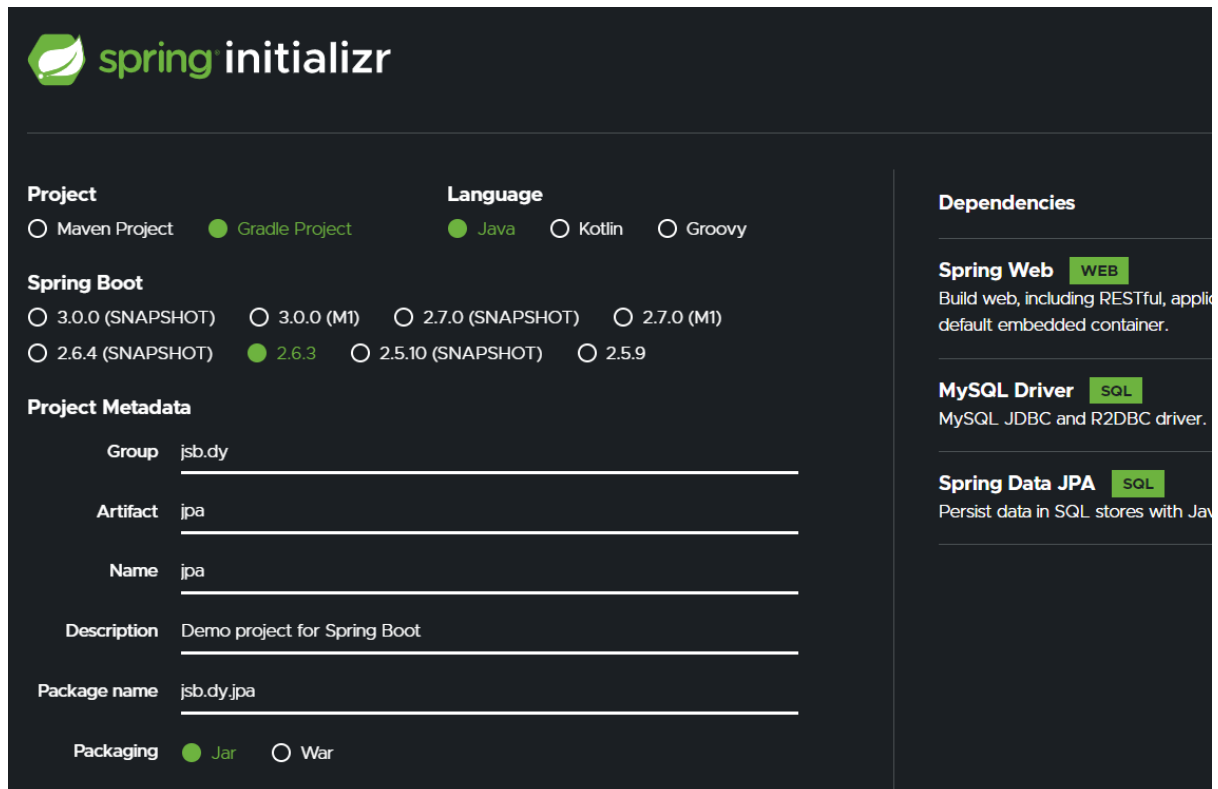
- 실제로 사용하는 것은 하이버네이트
- 하이버네이트가 JPA의 API를 활용해서 관계형 데이터베이스를 다루어주는 역할을 하는 것
=> JPA에 대한 이해가 충분 시 직접 ORM 프레임워크 만들기 가능

(+) 하이버네이트 설명 출처 블로그

- Hibernate는 JPA의 구현체이다
- Hibernate는 JPA라는 명세의 구현체이다. 즉, 위에서 언급한 javax.persistence.EntityManager와 같은 인터페이스를 직접 구현한 라이브러리이다. JPA와 Hibernate는 마치 자바의 interface와 해당 interface를 구현한 class와 같은 관계이다.
- “Hibernate는 JPA의 구현체이다”로부터 도출되는 중요한 결론 중 하나는 JPA를 사용하기 위해서 반드시 Hibernate를 사용할 필요가 없다는 것이다. Hibernate의 작동 방식이 마음에 들지 않는다면 언제든지 DataNucleus, EclipseLink 등 다른 JPA 구현체를 사용해도 되고, 심지어 본인이 직접 JPA를 구현해서 사용할 수도 있다. 다만 그렇게 하지 않는 이유는 단지 Hibernate가 굉장히 성숙한 라이브러리이기 때문일 뿐

JPA 활용하기

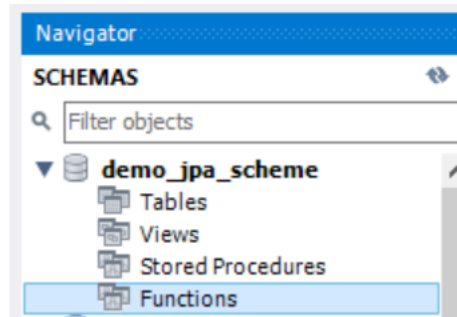
프로젝트 생성, 디펜던시 저 세개 추가하기



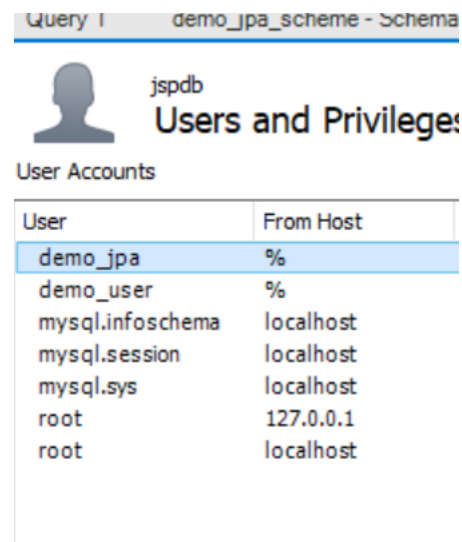
The image shows the Spring Initializr web interface for configuring a new project. The interface is divided into several sections:

- Project:** Includes radio buttons for Maven Project, Gradle Project (selected), and Language options: Java (selected), Kotlin, and Groovy.
- Spring Boot:** Includes radio buttons for various versions: 3.0.0 (SNAPSHOT), 3.0.0 (M1), 2.7.0 (SNAPSHOT), 2.7.0 (M1), 2.6.4 (SNAPSHOT), 2.6.3 (selected), 2.5.10 (SNAPSHOT), and 2.5.9.
- Project Metadata:** Includes input fields for Group (jsb.dy), Artifact (jpa), Name (jpa), Description (Demo project for Spring Boot), and Package name (jsb.dy.jpa). There is also a Packaging section with radio buttons for Jar (selected) and War.
- Dependencies:** A list of dependencies to be added to the project, each with a checkbox and a label:
 - Spring Web** (WEB): Build web, including RESTful, application default embedded container.
 - MySQL Driver** (SQL): MySQL JDBC and R2DBC driver.
 - Spring Data JPA** (SQL): Persist data in SQL stores with Java Persistence API.

root계정에서 스키마 새로 만들어주기 => 이거 localhost에서 진행하기



아이디 ; demo_jpa / 비번 ; 늘쓰던애..



- 특권 설정 들어가서 특권 all 선택

Setup New Connection

Connection Name: 이름 알아서

Connection Method: Standard (TCP/IP)

Parameters SSL Advanced

Hostname: 127.0.0.1 Port: 3306

Username: demo_jpa

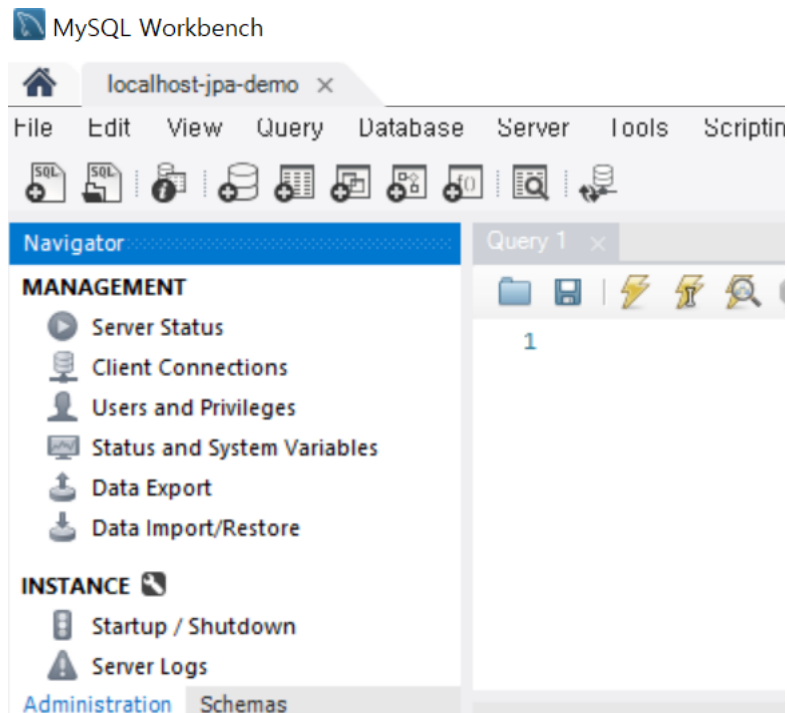
Password:

Default Schema: 아까 mysql에서 만들어놨던 스키마 내용

- 커넥션 만들기

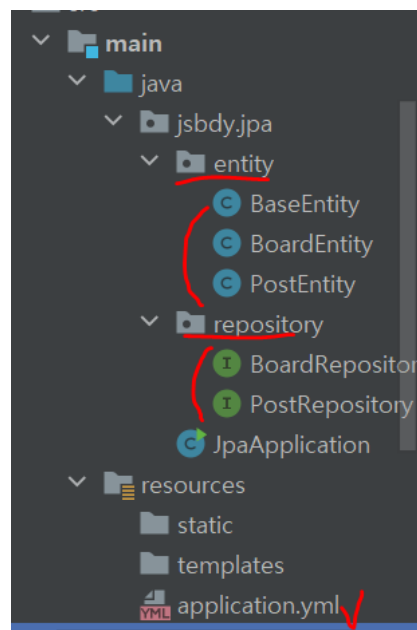


- 그리고 이와 같은 새로운 mysql 아이 만들기



=> mybatis 사용할 때는 테이블 값 미리 지정해줬어야 하는데 jpa는 알아서 작성o

Entity 작성 & JPA Relationships



- 위와 같이 spring 파일 만들어주기
- `application.yml` 파일 작성

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://127.0.0.1:3306/demo_jpa_scheme
    username : demo_jpa
    password : qpqpqp0614@
  jpa :
    hibernate :
      ddl-auto : create
      show-sql : false
    properties :
      hibernate :
        dialect: org.hibernate.dialect.MYSQL8Dialect
```

위에서 잘못됐고 아래처럼 jpa앞에 indentation 있었어야 함 ^^..

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://127.0.0.1:3306/demo_jpa_scheme?serverTimezone=UT
    username : demo_jpa
    password : qpqpqp0614
  jpa :
    hibernate :
      ddl-auto : create
      show-sql : false
    properties :
      hibernate :
        dialect: org.hibernate.dialect.MYSQL8Dialect
```

=> 복습 설명 : jpa는 annotation들의 라이브러리, 이것 사용하는 건 하이버네이트

- ddl auto : 테이블 생성, 삭제 자동으로 해주는 아이
- but 보통 상용환경에서는 create로 사용 x , 높게 써봐야 update, 일반 상황은 none
- show sql : jpa가 실제로 작동하면서 사용할 sql문이 있는데 이를 보여줄 지 말지 여부
- dialect는 hibernate에게 우리가 사용할 mysql 이거야~라고 참고하라 알려주는 것
 - db별로 문법이 꼬매씩 다들텐데 배티스를 쓴다면 직접 해야 하지만 하이버네이트는 대신해주기 때문에 우리가 어떤 db를 사용할 것인지는 알려주어야 한다.

1) `PostEntity` 클래스 작성

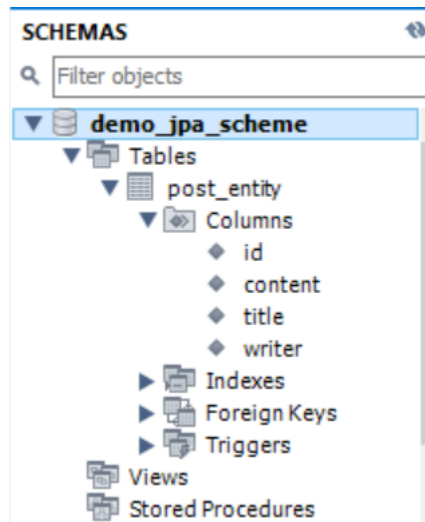
```
@Entity
public class PostEntity {
    @Id //jpa에게 아래의 아이디 pk라는 것임을 알려주는 것
    private Long id;
    //jpa 사용하면 프리미티브 타입 말고 클래스 기반 오브젝트 사용
    private String title;
    private String content;
    private String writer;
}
```

- `@Entity` 를 `PostEntity` 클래스 위에 붙여주기
- `@Id` 를 붙여서 pk가 무엇인지 알려주기

그리고

밑에 작성한 수많은 예러들을 거친 채^^

run을 해주니



이렇게 잘 생기게 된다!

- 이렇듯 jpa는 yml같은 설정에 명시된 것에 따라서 mysql을 만들어주는 역할을 수행한다음

2) BoardEntity 작성

```
package jsbdy.jpa.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

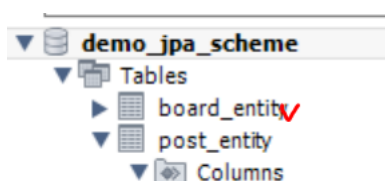
/**
 * id int
 * name varchar
 */
@Entity
public class BoardEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
}
```

3) Entity 생성자, getter, setter, toString() 작성

4) PostEntity는 BoardEntity에 속해져있는 관계임 설정

```
/*boardentity와의 관계 명시를 위해 추가*/
@ManyToOne(
    /*어떠한 관계를 대상으로 관계했나?*/
    targetEntity = BoardEntity.class,
    fetch = FetchType.LAZY
) /*다대일관계*/
private BoardEntity boardentity;
/* 여러개의 post, 게시글은 한개의 board, 게시판에 소속돼있는 것*/
```

- 다만 지금 돌리면 db랑 mysql이랑 다르면 에러 발생하기도 한다
- 그래도 일단은 괜찮으니 진행해보자, 그리고 postentity 그리고 foreignkey 까지 workbench에 성공적으로 생겼음을 알 수 있음



5) 하나의 BoardEntity 여러개의 PostEntity를 가지고 있음 알리기

- 그리고 추가적으로 좀 바꿨으니깐 getter, setter, constructor도 수정수정

```
@OneToMany(  
    targetEntity = PostEntity.class,  
    fetch = FetchType.LAZY,  
    mappedBy = "boardentity"  
    /*PostEntity에 정의된 BoardEntity의 이름*/  
)  
private List<PostEntity> postentitylist = new ArrayList<>();
```

- 여기까지가 테이블에 등록시키는 단계, 이제 이를 직접 활용하기 위해서 Repository 이요해줄 것임

6) BoardRepository (CRUD 레포지토리 확장)

```
package jsbdy.jpa.repository;  
  
import jsbdy.jpa.entity.BoardEntity;  
import org.springframework.data.repository.CrudRepository;  
  
/*  
T : 이 레포지토리가 어떤 엔티티 위한 것인지  
ID : 아이디가 어떤 타입으로 작성이 되는지  
*/  
  
public interface BoardRepository extends CrudRepository<BoardEntity, Long> {  
}
```

(+) CrudRepository

```
@RepositoryBean  
public interface CrudRepository<T, ID> extends Repository<T, ID> {  
  
    Saves a given entity. Use the returned instance for further operations as the save changed the entity instance completely.  
    Params: entity – must not be null.  
    Returns: the saved entity; will never be null.  
    Throws: IllegalArgumentException – in case the given entity is null.  
  
    <S extends T> S save(S entity);  
}
```

7) TestComponent 작성

방금 만든 아이가 잘 생성됐음을 잘 알 수 있음

```
@Component  
public class TestComponent {  
    public TestComponent(  
        @Autowired BoardRepository boardrepository  
    ){  
        BoardEntity boardentity = new BoardEntity();  
        boardentity.setName("new board");  
        BoardEntity newboardentity = boardrepository.save(boardentity);  
        System.out.println(newboardentity.getName());  
    }  
}
```

```
1 • select * from board_entity;
```

Result Grid		Filter Rows:
	id	name
▶	1	new board
•	HULL	HULL

- 이때 jpa는 눈에 보이는 코드만 전부라고 생각하면 안됨
- 따라서 바로 생성한 엔티티에 있는 속성을 프린트하는 것이 아닌 그것을 save하고 난 후의 엔티티의 속성을 출력하는 것이 더 좋다

8) PostEntity 생성

```
package jsbdy.jpa.repository;
import jsbdy.jpa.entity.PostEntity;
import org.springframework.data.repository.CrudRepository;
public interface PostRepository extends CrudRepository<PostEntity, Long> {
}
```

=> 이제 postentity 사용 가능

9) TestComponent 재작성

```
@Component
public class TestComponent {
    public TestComponent(
        @Autowired BoardRepository boardrepository,
        PostRepository postrepository
    ){
        BoardEntity boardentity = new BoardEntity();
        boardentity.setName("new board");
        BoardEntity newboardentity = boardrepository.save(boardentity);

        PostEntity postentity = new PostEntity();
        postentity.setWriter("hello ORM");
        postentity.setContent("created by Hibernate");
        postentity.setWriter("jsbdy");
        postentity.setBoardentity(newboardentity);//save되고 난 후의 entity 지정
        PostEntity newpostentity = postrepository.save(postentity);
    }
}
```

```
2 • select * from post_entity;
```

Result Grid					Filter Rows:	Edit:
id	content	title	writer	boardentity_id		
1	created by Hibernate	HULL	jsbdy	1		

10) findById 말고 다른 것을 찾아볼까?

일단 디폴드로 주어지는 것으로는 findByld가 있어 근데 작성자로 찾는 것은 안됨?

PostRepository 가서 아래와 같이 리스트

```
public interface PostRepository extends CrudRepository<PostEntity, Long> {  
    List<PostEntity> findAllByWriter(String writer);  
}
```

TestComponent 가서 아래와 같이 추가

```
System.out.println(postrepository.findAllByWriter("jsbdy").size());
```

- where wrtier = 'jsbdy' 와 똑같은 의미 따라서 1 이라고 잘 프린트되는 것을 볼 수 있음

```
EntityManagerFactory for persistence unit 'default'  
1✓
```

```
List <PostEntity> findAllByWriterAndBoardEntity(String writer, BoardEntity entity);  
List <PostEntity> findAllByWriterContaining(String writer);
```

11) BaseEntity

- 모든 엔티티들이 기본적으로 어떠한 속성 가지고 있게 하고 싶으면 지정해줌

```
@MappedSuperclass  
@EntityListeners(AuditingEntityListener.class)  
public abstract class BaseEntity{ //추상 클래스로 만들기  
    @CreatedDate  
    @Column(updatable=false)  
    private Instant createdAt;  
  
    @LastModifiedDate  
    @Column(updatable = true)  
    private Instant updatedAt;
```

- 추상클래스로 지정 & 물론 getter setter도
- 그리고 아까 만든 PostEntity, BoardEntity에 extends
- 따로 추가적으로 추상메소드는 만들어주지 않아서 엔티티들도 따로 선언해줄 애들 존재 x
- 근데 Auditing이란 아이는 따로 스프링에서 지정된 애가 아니라서 application에 추가적으로 지정해주기 (@EnableJpaAuditing) & run

Table: board_entity

Columns:

id	bigint AI PK
created_at	datetime(6)
updated_at	datetime(6)
name	varchar(255)

- createdat, updatedat 잘 생성됨

Table, Column, JoinColumn, JoinTable

PostEntity

- @Entity말에 아래의 것 추가

```
@Table(name="post")
```

- ```
@ManyToOne(
 /*어떠한 관계를 대상으로 관계했나?*/
 targetEntity = BoardEntity.class,
 fetch = FetchType.LAZY
) /*다대일관계*/
@JoinColumn(name="board_id")
```

```
- @Entity말에 추가
@Table(name="board")

- @Column(name="board_name") 아래에 추가
private String name;
```

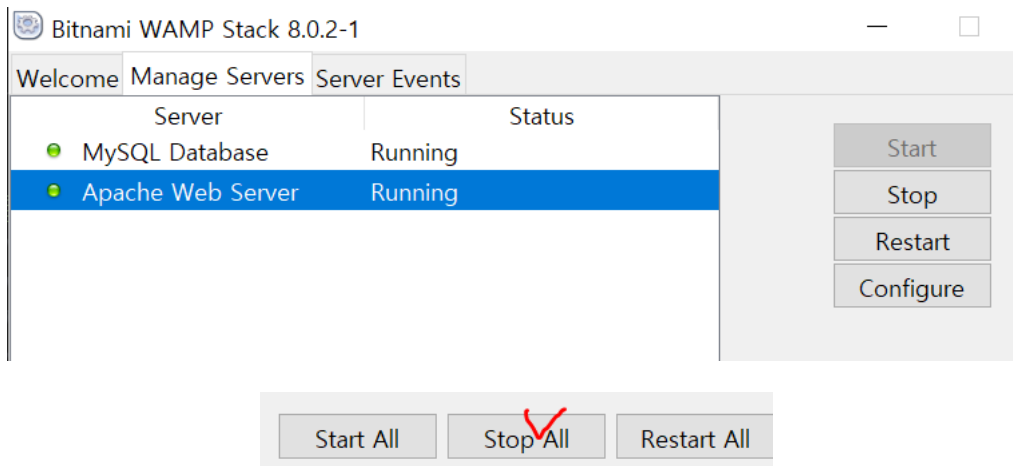
|   | id   | created_at | updated_at | content | title | writer | boardentity_id |
|---|------|------------|------------|---------|-------|--------|----------------|
| * | NULL | NULL       | NULL       | NULL    | NULL  | NULL   | NULL           |

|   | id   | created_at | updated_at | name |
|---|------|------------|------------|------|
| * | NULL | NULL       | NULL       | NULL |

mysql 컬럼에 잘 추가되고 반영됨 ○○

```
at
org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator.initiateService(JdbcEnvironmentIniti
~[hibernate-core-5.4.33.jar:5.4.33]
at
org.hibernate.boot.registry.internal.StandardServiceRegistryImpl.initiateService(StandardServiceRe
~[hibernate-core-5.4.33.jar:5.4.33]
at
org.hibernate.service.internal.AbstractServiceRegistryImpl.createService(AbstractServiceRegistryIn
~[hibernate-core-5.4.33.jar:5.4.33]
... 33 common frames omitted
Process finished with exit code 1
```



(2) 해결참고블로그

/example?serverTimezone=UTC&characterEncoding=UTF-8 이걸 뒤에 추가

Error creating bean with name 'entityManagerFactory' defined in class path resource [org/springframework/boot/autoconfigure/orm/jpa/HibernateJpaConfiguration.class]: Invocation of init method failed; nested exception is org.hibernate.service.spi.ServiceException: Unable to create requested service

Error creating bean with name 'entityManagerFactory' defined in class path resource [org/springframework/boot/autoconfigure/orm/jpa/HibernateJpaConfiguration.class]: Invocation of init method failed; nested exception is org.hibernate.service.spi.ServiceException: Unable to create requested service [org.hibernate.engine.jdbc.env.spi.JdbcEnvironment] at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.initializeBean(Spring Boot [Spring Boot] Hibernate 사용 시 Error creating bean with name 'entityManagerFactory' 해결 블로그

(3)

Unable to load class <https://www.leawy.com/thread/hibernate-core-and-annotations/7792/java-lang-classnotfoundexception-could-not-load-requested-class-org-hibernate-mysql8dialect.htm>

(4)

Access denied for user 'demo\_jpa'@'localhost' (using password: NO) at com.mysql.cj.jdbc.exceptions.SQLException.createSQLException(

<https://stackoverflow.com/questions/2995054/access-denied-for-user-rootlocalhost-using-passwordno>



### Explanation

In the above example, the issue can be identified by the below mysql database error.

Caused by: java.sql.SQLException: Access denied for user 'root'@'localhost' to database 'testdb'

This spring boot exception is due to database access permission issue. The below code will resolve this database error

```
CREATE USER 'root'@'localhost' IDENTIFIED BY 'root';
GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' WITH GRANT OPTION;
FLUSH PRIVILEGES;
```

<https://www.yawintutor.com/error-5068-unable-to-open-jdbc-connection-for-ddl/>

- root로 mysql에 로그인하고

```
mysql> use mysql;
```

Database changed

```
mysql> select * from user;
```

아래처럼 쳐서 demo\_jpa라는 아이를 살펴보았더니 N 천국 ^^ 권한없음 천국~~

[illegible]

```
mysql> create user 'demo_jpa'@'localhost';
```

Query OK, 0 rows affected (0.04 sec)

```
mysql> grant all privileges on . to ''demo_jpa'@'localhost';
```

Query OK, 0 rows affected (0.04 sec)

flush privileges;

Query OK, 0 rows affected (0.03 sec)

- 일케 해주고 돌리니깐 드디어 됐다  
mysql 권한 문제였나봄가

(5) jpa find~함수 작성 시 엔티티 명, 레포지토리 명 (소대문자 구별하는 것) 아주 중요!

```
List <PostEntity> findAllBywriterAndBoardEntity(String writer, BoardEntity entity);
```

라고 처음에 작성했는데, 나같은 경우에는 PostEntity에서 BoardEntity를 정의할 때

```
private BoardEntity boardentity;
```

이렇게 정의했었음 근데 jpa에서

```
findAllByWriterAndBoardEntity
```

이렇게 썼더니여러가 남 당연함

저렇게 쓰면 jpa는 boardEntity를 찾을텐데 나는 boardentity라고 정의해놨으니.. 따라서 아래처럼 수정하면 잘 됨ㅋ

```
List <PostEntity> findAllByWriterAndBoardentity
```

## Service, Repository를 이용하여 CRUD 데이터 다루기

- 이전에 하던 파일에 Controller 추가

### 1) **PostController** 생성 & 작성

```
@RestController
public class PostController {
 private static final Logger logger = LoggerFactory.getLogger(PostController.class);
}
```

### 2) **PostService** 생성 & 작성

```
@Service
public class PostService {
 private static final Logger logger = LoggerFactory.getLogger(PostService.class);
}
```

- service : 비즈니스 로직, repository : 데이터 액세스, 컨트롤러 : 프론트 게이트웨이

### 3) **Post레포지토리 - PostDao** 명으로 생성 & 작성

```
@Repository
public class PostDao {
 private static final Logger logger = LoggerFactory.getLogger(PostDao.class);
}
```

### 4) 이제 애네를 controller 에 등록해줘야지 & 기본 crud

```
@RestController
@RequestMapping("post")
public class PostController {
 private static final Logger logger = LoggerFactory.getLogger(PostController.class);
 private final PostService postservice;

 public PostController(
 @Autowired PostService postservice
){
 this.postservice = postservice;
 }
 @PostMapping()
 public void createPost(){

 }

 @GetMapping("{id}")
 public void readPost(@PathVariable("id")int id){

 }
 @GetMapping("")
 public void readPostAll(){

 }

 @PutMapping("{id}")
 public void updatePost(@PathVariable("id")int id){

 }
 @DeleteMapping("{id}")
 public void deletePost(@PathVariable("id") int id){

 }
}
```

```
}
}
```

- 여기까지 했는데 우리가 entity는 만들어놨는데 데이터를 주고받을 객체를 아직 생성하지 않았음 (이전에 진행했던 dto 의 존재)
- 우리가 이전에 했던 crud랑 가장 큰 차이점이라고 할 수 있음
- entity사용하면 안되나 라는 의문이 들 지도 모르는데 entity는 단순한 데이터의 표현만을 나타내는 존재임, 단순한 crud에서 entity를 dto로 사용하는 것은 바람직하지 않음
- => 그래서 postdto를 하나 더 추가적으로 만들어보자

## 5) Postdto (getter,setter,toString은 생략..)

```
public class PostDto {
 private int id;
 private String title;
 private String content;
 private String writer;
 private int boardId;
}
```

=> dao 까지 이렇게 정의한 dto 오브젝트가 들어가게 될 것이다.

- dao에서 엔티티를 조정하고 이를 돌려주게 할 예정~
- 그럼 이제 PostCotroller가 다룰 데이터가 바로 PostDto라고 지정해주러 가야한다~

## 6) PostController에 dto 지정 & @ResponseStatus(HttpStatus.알맞은 요청)추가

```
@PostMapping()
@ResponseStatus(HttpStatus.CREATED)
public void createPost(
 @RequestBody PostDto postdto
){

}

@GetMapping("/{id}")
public PostDto readPost(
 @PathVariable("id")int id
){
}

@GetMapping("")
public List<PostDto > readPostAll(){
}

@PutMapping("/{id}")
@ResponseStatus(HttpStatus.ACCEPTED)
public PostDto updatePost(
 @PathVariable("id")int id,
 @RequestBody PostDto postdto
){
}

@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.ACCEPTED)
public PostDto deletePost(
 @PathVariable("id") int id
){
}
}
```

## 7) postservice 추가

```
public class PostService {
 private static final Logger logger = LoggerFactory.getLogger(PostService.class);

 public void createPost(PostDto postdto){

 }

 public PostDto readPost(int id){
 }
}
```

```

 }

 public List<PostDto> readPostAll(){

 }

 public void updatePost(int id, PostDto postdto){

 }

```

## 8) postdao 추가 작성

```

@Repository
public class PostDao {
 private static final Logger logger = LoggerFactory.getLogger(PostDao.class);
 private final PostRepository postrepository;
 //postrepository 지정해주기
 public PostDao(
 @Autowired PostRepository postrepository
){
 this.postrepository=postrepository;
 }
 public void createPost(PostDto postdto){
 PostEntity postentity = new PostEntity();
 postentity.setTitle(postdto.getTitle());
 postentity.setContent(postdto.getContent());
 postentity.setWriter(postdto.getWriter());
 postentity.setBoardentity(null);
 this.postrepository.save(postentity);
 }

 public PostEntity readPost(int id){
 Optional<PostEntity> postentity = this.postrepository.findById((long) id);
 if(postentity.isEmpty()){
 throw new RuntimeException(HttpStatus.NOT_FOUND);
 }
 return postentity.get();
 }

 public Iterator<PostEntity> readPostAll(){
 return this.postrepository.findAll().iterator();
 }

 public void updatePost(int id, PostDto postdto){
 PostEntity postentity = new PostEntity();
 postentity.setTitle(postdto.getTitle()==null?postentity.getTitle() : postdto.getTitle());
 postentity.setContent(postdto.getContent()==null?postentity.getContent() : postdto.getContent());
 postentity.setWriter(postdto.getWriter()==null?postentity.getWriter() : postdto.getWriter());
 postentity.setBoardentity(null);
 this.postrepository.save(postentity);
 }
 public void deletePost(int id){
 Optional<PostEntity> targetentity = this.postrepository.findById((long) id);
 if(targetentity.isEmpty()){
 throw new RuntimeException(HttpStatus.NOT_FOUND);
 }
 this.postrepository.delete(targetentity.get());
 }
}

```

## 9) postdao 추가 작성

```

@Service
public class PostService {
 private static final Logger logger = LoggerFactory.getLogger(PostService.class);
 private final PostDao postdao; //postdao 추가 선언 (활용 위해)

 public PostService(@Autowired PostDao postdao, PostDao postdao1){
 this.postdao = postdao; //postdao 초기화
 }

 public void createPost(PostDto postdto){
 this.postdao.createPost(postdto);
 }

 public PostDto readPost(int id){
 PostEntity postentity = this.postdao.readPost(id);
 }
}

```

```

 }

 public List<PostDto> readPostAll(){
 Iterator<PostEntity> iterator=this.postdao.readPostAll();
 }

 public void updatePost(int id, PostDto postdto){
 this.postdao.updatePost(id,postdto);
 }

 public void deletePost(int id){
 this.postdao.deletePost(id);
 }
}

```

- 이제 내부의 entity를 postdto 로 변환해주는 과정이 필요

```

@Service
public class PostService {
 private static final Logger logger = LoggerFactory.getLogger(PostService.class);
 private final PostDao postdao; //postdao 추가 선언 (활용 위해)

 public PostService(@Autowired PostDao postdao, PostDao postdao1){
 this.postdao = postdao; //postdao 초기화
 }

 public void createPost(PostDto postdto){
 this.postdao.createPost(postdto);
 }

 public PostDto readPost(int id){
 PostEntity postentity = this.postdao.readPost(id);
 return new PostDto(
 Math.toIntExact(postentity.getId()),
 postentity.getTitle(),
 postentity.getContent(),
 postentity.getWriter(),
 postentity.getBoardentity()==null
 ?0:Math.toIntExact(postentity.getBoardentity().getId())
);
 }

 public List<PostDto> readPostAll(){
 Iterator<PostEntity> iterator=this.postdao.readPostAll();
 List<PostDto> postDtoList = new ArrayList<>();

 while(iterator.hasNext()){
 PostEntity postentity = iterator.next();
 postDtoList.add(new PostDto(
 Math.toIntExact(postentity.getId()),
 postentity.getTitle(),
 postentity.getContent(),
 postentity.getWriter(),
 postentity.getBoardentity()==null
 ?0:Math.toIntExact(postentity.getBoardentity().getId())
));
 }
 return postDtoList;
 }

 public void updatePost(int id, PostDto postdto){
 this.postdao.updatePost(id,postdto);
 }

 public void deletePost(int id){
 this.postdao.deletePost(id);
 }
}

```

## 10 ) PostController

```

package jsbdy.jp;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;
import java.util.List;

```

```

@RestController
@RequestMapping("post")
public class PostController {
 private static final Logger logger = LoggerFactory.getLogger(PostController.class);
 private final PostService postservice;

 public PostController(
 @Autowired PostService postservice
){
 this.postservice = postservice;
 }
 @PostMapping()
 @ResponseStatus(HttpStatus.CREATED)
 public void createPost(
 @RequestBody PostDto postdto
){
 this.postservice.createPost(postdto);
 }

 @GetMapping("/{id}")
 public PostDto readPost(
 @PathVariable("id")int id
){
 return this.postservice.readPost(id);
 }

 @GetMapping("")
 public List<PostDto > readPostAll(){
 return this.postservice.readPostAll();
 }

 @PutMapping("/{id}")
 @ResponseStatus(HttpStatus.ACCEPTED)
 public void updatePost(
 @PathVariable("id")int id,
 @RequestBody PostDto postdto
){
 this.postservice.updatePost(id, postdto);
 }

 @DeleteMapping("/{id}")
 @ResponseStatus(HttpStatus.ACCEPTED)
 public void deletePost(
 @PathVariable("id") int id
){
 this.postservice.deletePost(id);
 }
}

```

=>

잘 된당 ㅎㅎ

The screenshot shows a REST client interface with the following details:

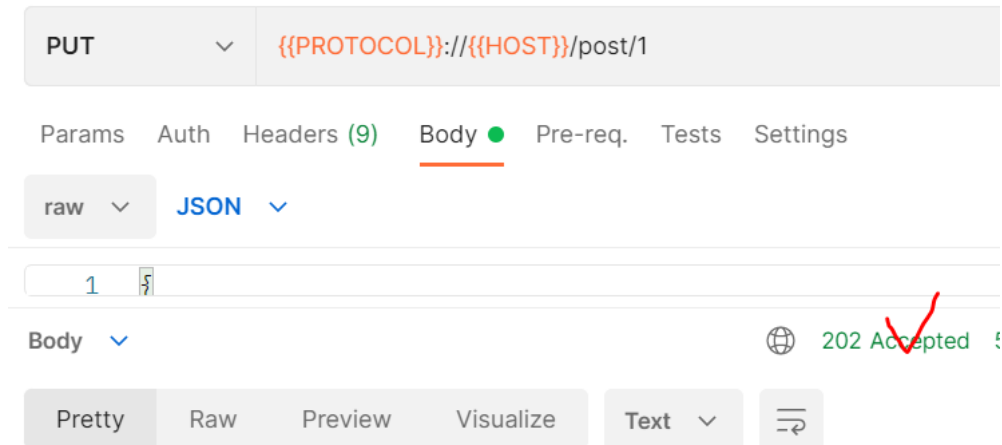
- URL:** {{PROTOCOL}}://{{HOST}}/post
- Method:** POST
- Body:**

```

1 {
2 ... "title": "Rest-spring CRUD",
3 ... "content": "test posting",
4 ... "writer": "shucream"

```
- Response:** 201 Created, 195 ms, 128 B
- Status:** Success (indicated by a red checkmark)
- Buttons:** Send, Beautify, Save Response

- 아냐 근데 put에서 에러 생김;;



- 일케해서 날리면 id2가 수정이 되는게 아니라 저 update된 애로 새로운 id3가 더 생성이 돼버림;;허거덩
- > 코드 수정

```
public void updatePost(int id, PostDto postdto){
 PostEntity postentity = new PostEntity();
```

- postdao의 updatePost부분에서 `PostEntity postentity = new PostEntity();` 로 계속 새로운 엔티티 생성하고 있었삼;;ㅋㅋ

```
public void updatePost(int id, PostDto postdto){
 Optional<PostEntity> targetentity = this.postrepository.findById(Long.valueOf(id));
 if (targetentity.isEmpty()){
 throw new RuntimeException(HttpStatus.NOT_FOUND);
 }
 PostEntity postentity=targetentity.get();

 postentity.setTitle(postdto.getTitle()==null?postentity.getTitle() : postdto.getTitle());
 postentity.setContent(postdto.getContent()==null?postentity.getContent() : postdto.getContent());
 postentity.setWriter(postdto.getWriter()==null?postentity.getWriter() : postdto.getWriter());
 postentity.setBoardentity(null);
 this.postrepository.save(postentity);
}
```

## 추가

### 1. RequestParam과 비교

#### 5.1. RequestParam

<http://192.168.0.1:8080?aaa=bbb&ccc=ddd>

#### 5.2. PathVariable

<http://192.168.0.1:8080/bbb/ddd>

Type 1 => <http://127.0.0.1?index=1&page=2>

- 파라미터의 값과 이름을 함께 전달하는 방식으로 게시판 등에서 페이지 및 검색 정보를 함께 전달하는 방식을 사용할 때 많이 사용됩니다.

Type 2 => <http://127.0.0.1/index/1>

- Rest api에서 값을 호출할 때 주로 많이 사용합니다.
- @RequestParam 사용하기  
Type 1의 URL을 처리할 때 @RequestParam을 사용하게 됩니다.  
아래의 예제와 같이 Controller 단에서 사용합니다.

```
@GetMapping("read")
public ModelAndView getFactoryRead(int factoryId, SearchCriteria criteria)
{ }
```

위의 경우 /read?no=1와 같이 url이 전달될 때 no 파라미터를 받아오게 됩니다.

@RequestParam 어노테이션의 괄호 안의 경우 전달인자 이름(실제 값을 표시)입니다.

이렇게 @RequestParam의 경우 url 뒤에 붙는 파라미터의 값을 가져올 때 사용을 합니다.

- 

#### @PathVariable의 경우

url에서 각 구분자에 들어오는 값을 처리해야 할 때 사용합니다.

```
@PostMapping("delete/{idx}")
@ResponseBody
public JsonResultVo postDeleteFactory(@PathVariable("idx") int factoryIdx) {
 return factoryService.deleteFacotryData(factoryIdx);
}
```