LIKE LION
POSSIBILITY TO
REALITY

LIKELION UNIV.

봄이 왔다 Spring

개발자는 왜 why? 를 계속 물어야 한다

Array vs List

Web history Spring Boot MVC pattern

🦁 Web history

Hypertext Link

- 웹의 초기 버전에서는 Hipertext라는 형태로 저장되어서 Link를 통해 접근
- 각 텍스트에 다른 링크로 갈 수 있게끔 연결이 되어있는 형태

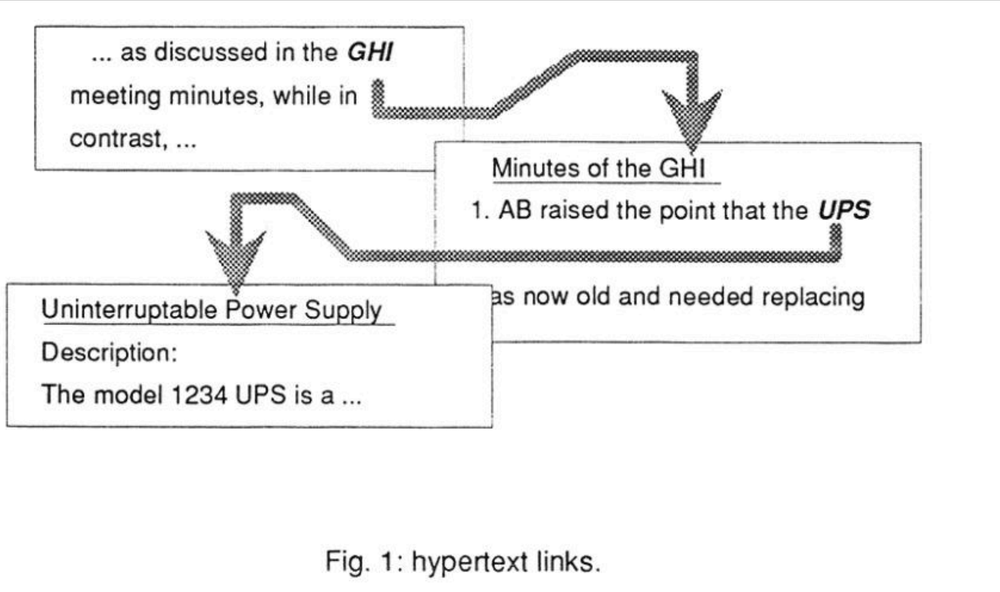


Fig. 1: hypertext links.

🦁 Web history

Hypertext Link

- 웹의 초기 버전에서는 Hipertext라는 형태로 저장되어서 Link를 통해 접근
- 각 텍스트에 다른 링크로 갈 수 있게끔 연결이 되어있는 형태

==========> 정적인 구조를 가짐

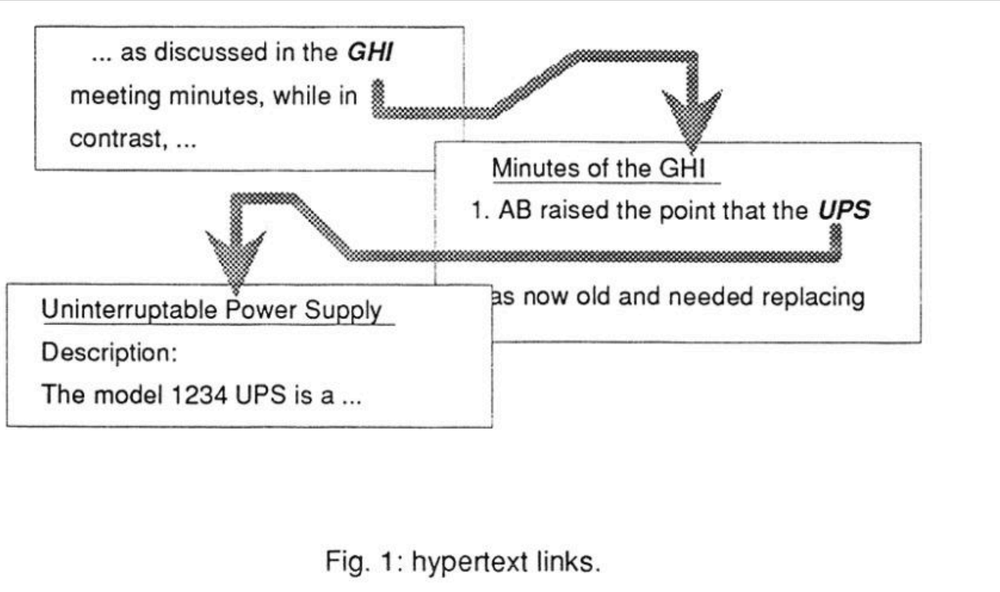


Fig. 1: hypertext links.

🦁 Web history

Client-Server Architecture

- 클라이언트에서 네트워크를 통해 서버로 문서를 요청
- 서버에서는 네트워크를 통해 Hipertext 문서를 reponse로 전달

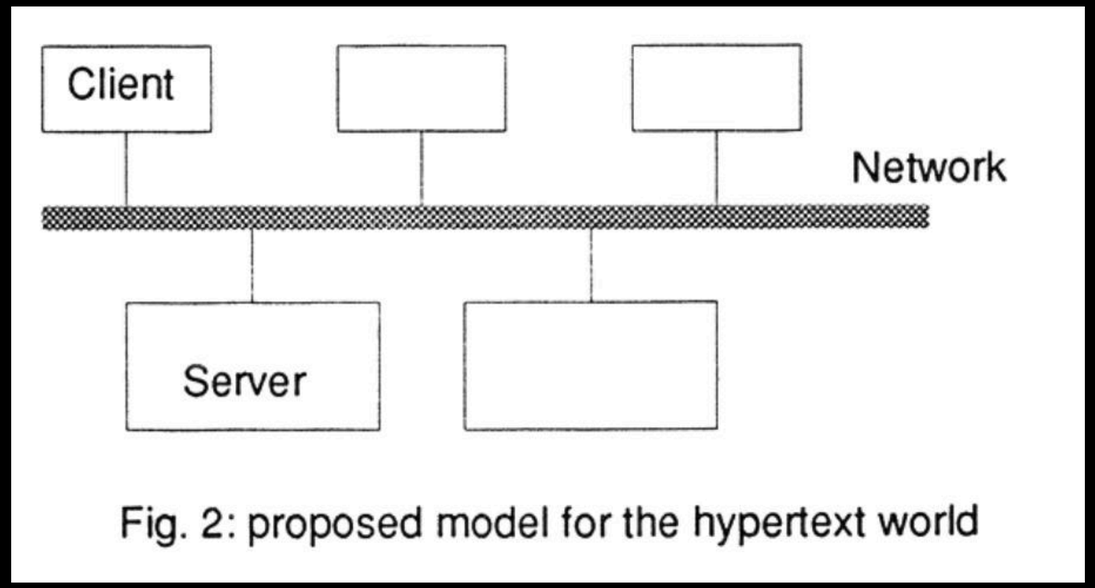


Fig. 2: proposed model for the hypertext world

🦁 Web history

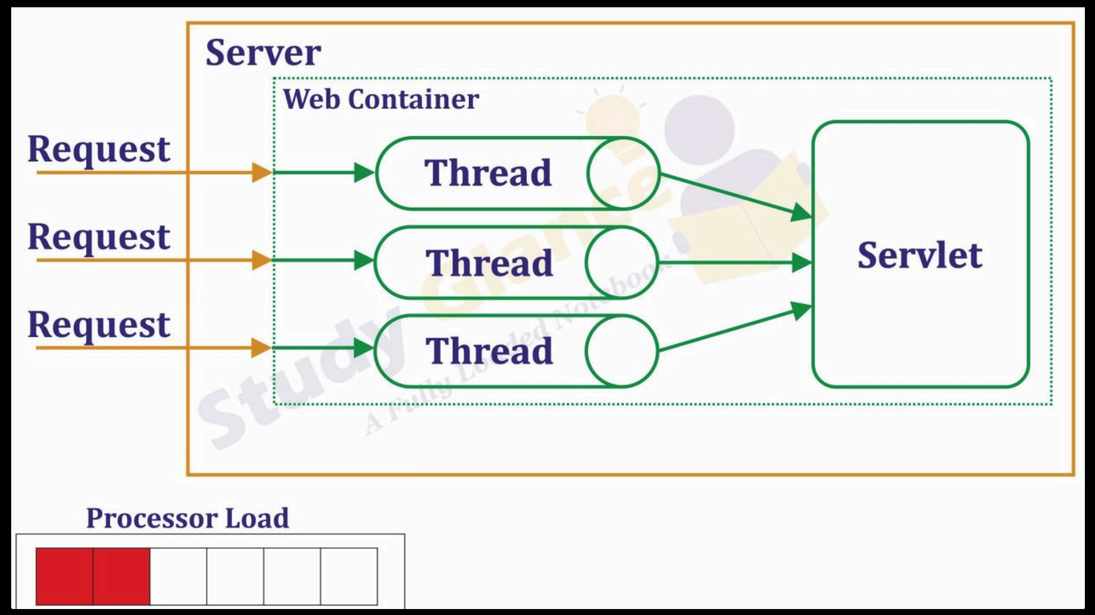
Dynamic Web Service

- 이미 만들어 놓은 웹 페이지를 보여주는 것이 아니라, 사용자의 응답에 따라 변하는 웹 페이지를 만들려면 어떻게 해야할까?

🦁 Web history

Servlet

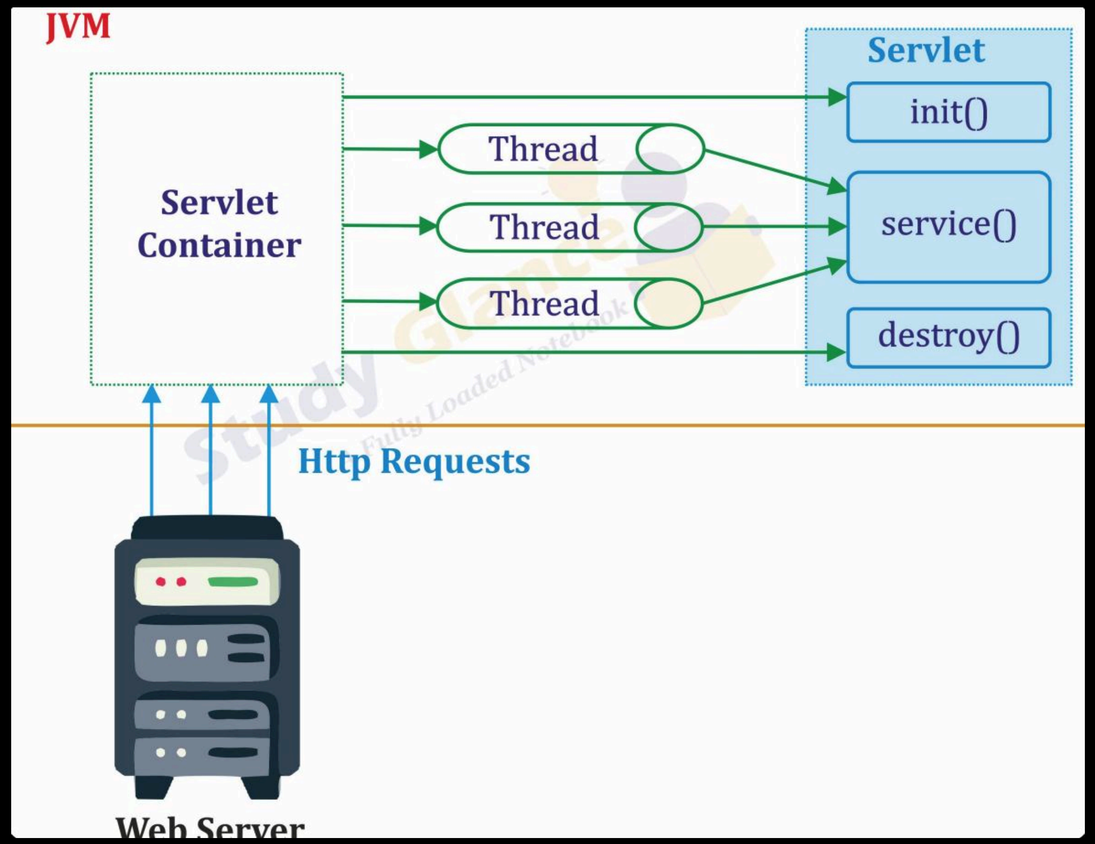
- Java EE에 포함된 기술
- Servlet은 스레드 단위로 클라이언트의 요청을 처리함 -> 효율적
- Process vs Thread
  - Process는 하나의 큰 덩어리
  - Thread는 하나의 큰 덩어리인 Process를 여러 개로 쪼갠 것



🦁 Web history

Web Application Server (WAS)

- 웹 서버는 HTTP 요청을 Servlet Container에 전달
- Servlet Container는 Servlet을 실행시켜 동적인 컨텐츠를 생성함
  - 대표적인 예시가 톰캣(Tomcat)



Servlet의 단점

- 자바 코드를 통해서 HTML 문서를 직접 작성해야 한다는 한계
  - 프론트와 백의 코드가 하나의 자바 파일 안에 들어가게 됨
  - 하나의 코드가 여러 기능을 수행하게 됨

```java
public class AdditionServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        int num1 = Integer.parseInt(request.getParameter("a"));
        int num2 = Integer.parseInt(request.getParameter("b"));
        int sum = num1 + num2;

        out.println("<html><head><title>Result</title></head><body>");
        out.println("<h3>The sum is: " + sum + "</h3>");
        out.println("</body></html>");
        out.close();
        }
    }
```

Web history Spring Boot MVC pattern

🦁 Spring Boot

Spring?

- Spring은 자바를 기반으로 만들어진 웹 어플리케이션 프레임워크
  - 프레임워크는 뼈대나 근간을 이루는 코드들의 묶음을 의미함

- 프레임워크를 이용해서 프로그램의 기본 흐름을 정의
  - 팀원들이 자신의 코드를 해당 프레임워크 위에 추가로 개발하는 방식으로 진행됨

- 자바의 Spring은 경량 프레임워크이며, 간단하게 jar 파일을 이용해서 개발이 가능하게 만들어짐

Spring?

- 기존에는 EJB(Enterprise JavaBeans)와 같은 복잡한 방식으로 로직을 구현
- POJO(Plain Old Java Object)의 방식을 도입해서 보다 쉽게 구현
  - EJB의 겨울 뒤에 Spring이라는 봄이 온다!

- POJO
  - 평범한 옛날 방식의 자바 객체
  - 자바의 평범한 클래스를 사용하는 것처럼 로직을 작성
  - 간단한 속성과 메소드로만 구현

🦁 Spring Boot

Spring Boot?

- Spring Boot는 기존 Spring보다 더 간편한 프로젝트
- 기존에는 Spring 프로젝트를 생성할 때마다 반복적으로 모든 세팅을 다 적용 해주어야 했음
- Spring Boot를 사용하면 이 설정을 대신 해줌

- 내장 서버 지원
  - Spring은 WAS를 별도로 설치하고 외부에서 관리해야 함
  - Spring Boot는 WAS의 예시인 Tomcat을 기본적으로 내장하고 있어서 배포가 간단함

- Bean 설정
  - Spring은 각 설정 파일에서 Bean을 명시적으로 등록해야 함
  - 일정한 규칙에 따라서 Spring Boot는 자동으로 빈을 등록해줌

Spring vs Spring Boot

- Spring -> Spring Boot의 순서로 배우는 것이 더 효과적일 것 같지만, 시간의 한계로 인해 Spring Boot만 다룰 예정
- 크게 다른 점은 Bean과 Tomcat, 그리고 프로젝트 생성시 설정 세팅

=====> 기본적인 문법은 동일함!!

Inversion of Control

- IoC(Inversion of Control)
  - Spring container가 설정 정보를 통해 객체를 자동으로 관리함
  - 객체의 생성과 소멸에 대한 책임이 개발자에서 프레임워크(Spring)으로 역전(Inversion)

Inversion of Control

- 기존 코드를 구현할 때는 아래 사진처럼 new를 사용해서 생성

```
public class Service {
    2 usages
    private PostService postService;
    public Service() {
        postService = new PostService();
    }
    no usages
    public void createService() {
        postService.getPostResponseWithUserId("parameter");
    }
}
```

Inversion of Control

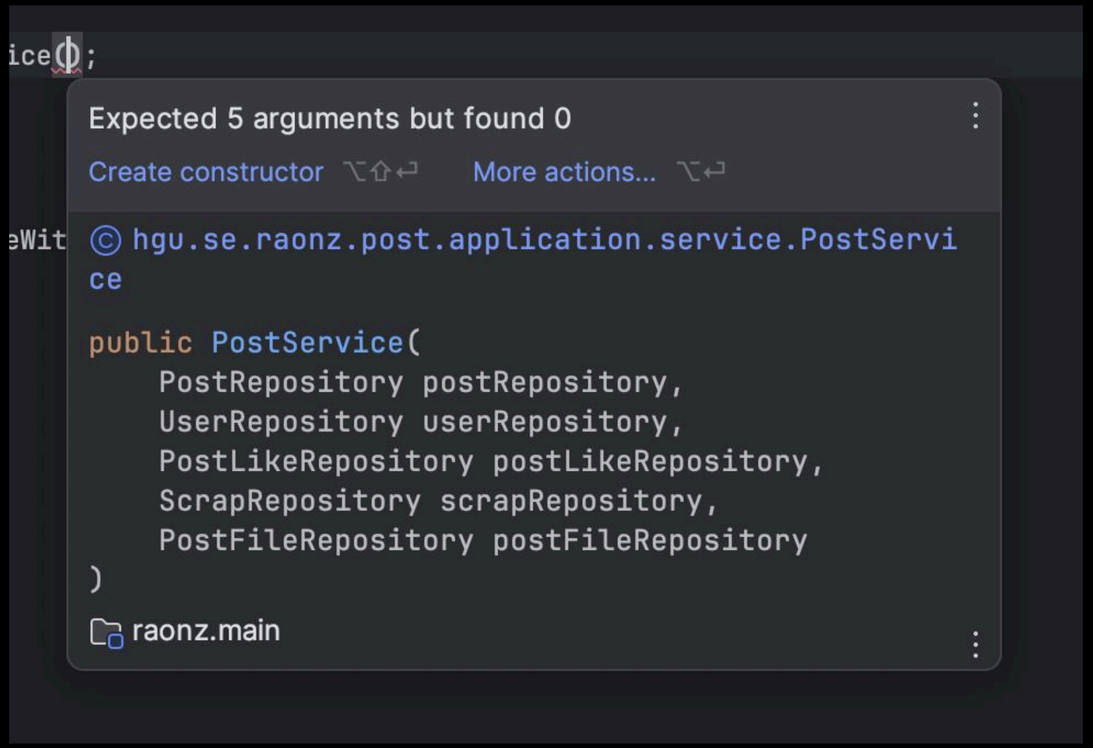
- 기존 코드를 구현할 때는 아래 사진처럼 new를 사용해서 생성
- 생성자가 필요로 하는 parameter를 전부 다 Instance로 만들어서 넣어 주어야 함

```
public class Service {
    2 usages
    private PostService postService;
    public Service() {
        postService = new PostService();
    }
    no usages
    public void createService() {
        postService.getPostResponseWithUserId("parameter");
    }
}
```

Inversion of Control

- 기존 코드를 구현할 때는 아래 사진처럼 new를 사용해서 생성
- 생성자가 필요로 하는 parameter를 전부 다 Instance로 만들어서 넣어 주어야 함

```
public class Service {
    2 usages
    private PostService postService;
    public Service() {
        postService = new PostService();
    }
    no usages
    public void createService() {
        postService.getPostResponseWithUserId("parameter");
    }
}
```



Inversion of Control

- IoC를 적용하게 되면 아래 사진처럼 argument로 받아오는 값에 대한 Instance 생성을 개발자가 하지 않음
- Spring Container에서 자동으로 생성과 소멸을 진행함

```
public class Service {
    2 usages
    private final PostService postService;
    public Service(PostService postService) {
        this.postService = postService;
    }
    no usages
    public void createService() {
        postService.getPostResponseWithUserId("parameter");
    }
}
```

Inversion of Control 예시

- HttpServletRequest request라는 Instance를 생성하지 않아도 사용이 가능함

```
choijh4161@gmail.com <21900764@handong.ac.kr>
@GetMapping("/rank/like")
public ResponseEntity<List<PostResponse>> getRankStar(HttpServletRequest request) {
    String token = jwtProvider.resolveToken(request);
    String user_id = jwtProvider.getAccount(token);
    List<PostResponse> postResponseList = postService.getRankLikePostResponseList(user_id);

    return ResponseEntity.ok(postResponseList);
}
```

```
32 implementations
17 public interface HttpServletRequest extends ServletRequest {
       no usages
18     String BASIC_AUTH = "BASIC";
       no usages
19     String FORM_AUTH = "FORM";
       no usages
20     String CLIENT_CERT_AUTH = "CLIENT_CERT";
       no usages
21     String DIGEST_AUTH = "DIGEST";
22
```

Dependency Injection(DI)

- DI는 의존성 주입이라는 뜻을 가짐
- IoC의 개념을 구체적으로 구현한 대표적인 방법
- 객체 간의 의존 관계를 직접 코드를 통해 생성해서 연결하는 것이 아니라, 외부에서 객체를 대신 생성하고 주입
- Instance를 우리가 만들 필요가 없이, 프레임워크에서 만들어줌

Dependency Injection(DI)

- DI를 사용하게 된 배경
  - A가 B를 직접 new로 선언해서 사용할 경우, B가 바뀌면 A도 같이 수정을 진행해야 함
  - 예시
    - 자동차 엔진을 고쳤는데 현재 내 차에 맞지 않게 수정이 되었을 경우, 차를 수정해야 할 필요가 생김

  - 테스트가 어려움
    - JUnit을 썼을 때의 경우처럼, A를 테스트 하기 위해 B와 C 등등을 다 Instance로 선언해야 함

  - 객체 생성 자체가 하드코딩 됨
    - 다른 경우의 수를 커버하는 것이 어려움

Dependency Injection(DI)

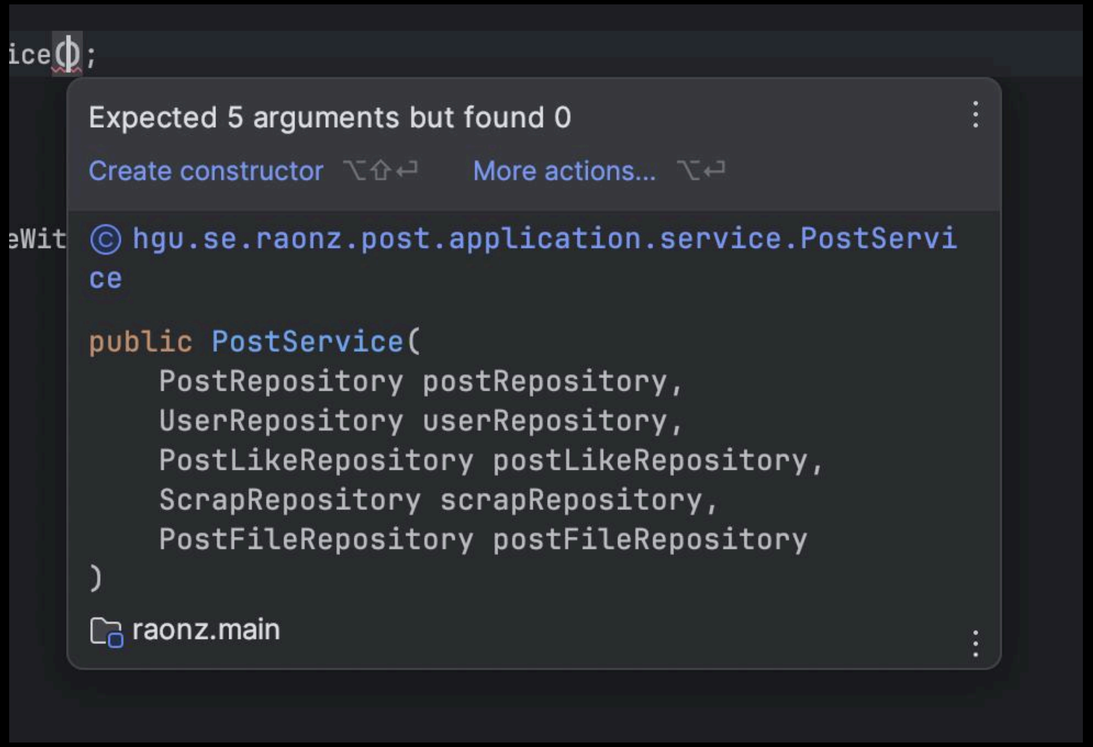
- 의존하는 객체를 외부에서 주입
  - 객체를 사용하는 쪽에서 직접 new로 생성하지 않음
  - 주입 받을 객체를 외부 컨테이너가 생성하고 관리하여 필요시 주입

- IoC의 구현 예시
  - 객체의 제어 권한을 개발자에서 컨테이너로 넘겨서 실질적으로 DI가 이를 구현함

Dependency Injection(DI)

- PostService라는 Instance를 만들기 위해서는 원래 5개의 arguments가 필요함

```
public class Service {
    2 usages
    private PostService postService;
    public Service() {
        postService = new PostService();
    }
    no usages
    public void createService() {
        postService.getPostResponseWithUserId("parameter");
    }
}
```



Dependency Injection(DI)

- 그 대신, 주입 받을 argument를 선언하면, 자동으로 필요한 Instance를 생성함
  - PostService가 필요로 하는 5개의 argument의 Instance를 Spring이라는 컨테이너가 생성해서 주입

```
public class Service {
    2 usages
    private final PostService postService;
    public Service(PostService postService) {
        this.postService = postService;
    }
    no usages
    public void createService() {
        postService.getPostResponseWithUserId("parameter");
    }
}
```

Dependency Injection(DI)

- 그 대신, 주입 받을 argument를 선언하면, 자동으로 필요한 Instance를 생성함
  - PostService가 필요로 하는 5개의 argument의 Instance를 Spring이라는 컨테이너가 생성해서 주입

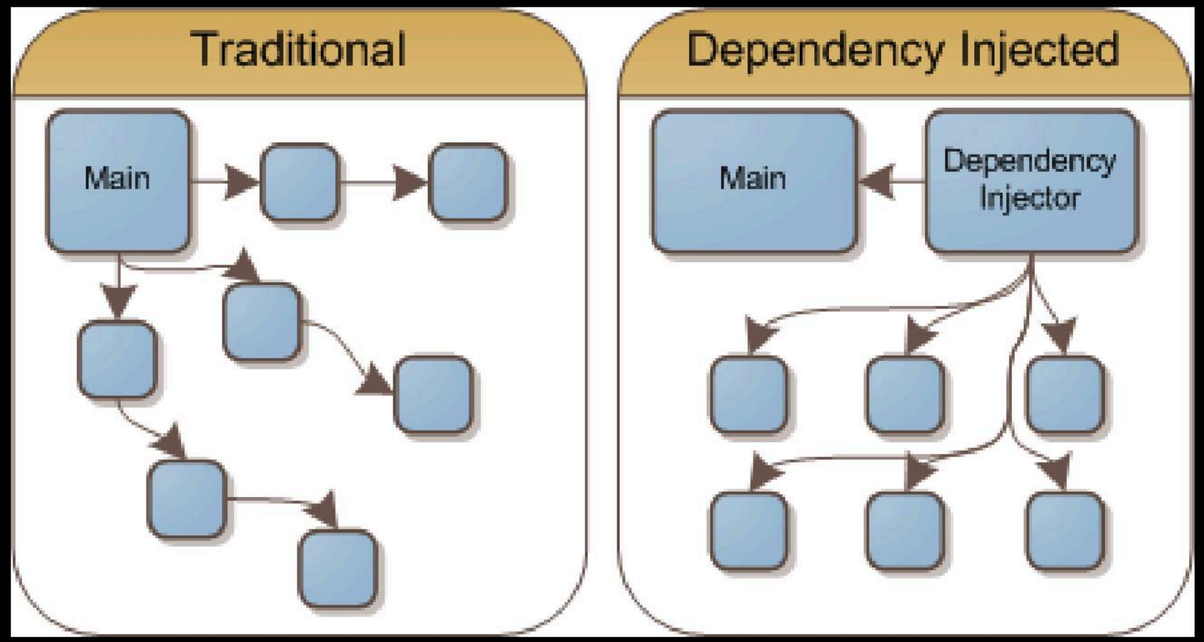
- 해당 argument들이 의존하는 다른 Class도 자동으로 Instance로 만들어서 사용
  - 혁명! 진짜 편함!

```
public class Service {
    2 usages
    private final PostService postService;
    public Service(PostService postService) {
        this.postService = postService;
    }
    no usages
    public void createService() {
        postService.getPostResponseWithUserId("parameter");
    }
}
```

Dependency Injection(DI)

- 기존에는 main에서 의존하는 모든 Instance를 직접 new를 사용해서 생성함
- DI를 적용할 경우, Spring container가 모든 Instance를 관리하고, 이를 main에 주입하기만 함
  - main에서 개발자가 직접 관리할 필요가 줄어듦

- 어노테이션!!!



Web history Spring Boot MVC pattern

MVC Pattern

- 기존 Servlet / JSP -> 비즈니스 로직과 화면이 같은 코드 안에 존재함

```
<%@ page import="java.io.*,java.util.*" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Result</title>
</head>
<body>
<%
    int num1 = Integer.parseInt(request.getParameter("a"));
    int num2 = Integer.parseInt(request.getParameter("b"));
    int sum = num1 + num2;

    out.println("<h3>The sum is: " + sum + "</h3>");
%>
</body>
</html>
```

MVC Pattern

- MVC 에서는 데이터의 처리와 데이터의 표시를 분리함

- Model
  - 어플리케이션 데이터와 관련된 로직을 처리함
- View
  - 사용자에게 데이터를 시각적으로 표시함
- Controller
  - 사용자의 입력을 적절히 분배하고, Model과 View를 관리함

MVC Pattern

- MVC 에서는 데이터의 처리와 데이터의 표시를 분리함

- Model
  - 어플리케이션 데이터와 관련된 로직을 처리함
- View
  - 사용자에게 데이터를 시각적으로 표시함
- Controller
  - 사용자의 입력을 적절히 분배하고, Model과 View를 관리함

MVC Pattern

- 백엔드에서는 Model과 Controller를 처리함
  - Model: 웹 서비스에서 사용하는 로직
  - Controller: request / response를 적당히 잘 분배

- 백엔드에서는 Model을 다시 2가지로 나누어서 사용함
  - Service
    - 실제 비즈니스 로직을 관리하고 구현
  - Repository
    - 데이터베이스와의 연결을 담당함

MVC Pattern

- 백엔드에서는 Model과 Controller를 처리함
  - Model: 웹 서비스에서 사용하는 로직
  - Controller: request / response를 적당히 잘 분배

- 백엔드에서는 Model을 다시 2가지로 나누어서 사용함
  - Service
    - 실제 비즈니스 로직을 관리하고 구현
  - Repository
    - 데이터베이스와의 연결을 담당함

- Repository를 사용하는 방법에는 크게 2가지가 존재함
  - MyBatis
  - JPA Repository

MVC Pattern

- 백엔드에서는 Model과 Controller를 처리함
  - Model: 웹 서비스에서 사용하는 로직
  - Controller: request / response를 적당히 잘 분배

- 백엔드에서는 Model을 다시 2가지로 나누어서 사용함
  - Service
    - 실제 비즈니스 로직을 관리하고 구현
  - Repository
    - 데이터베이스와의 연결을 담당함

- Repository를 사용하는 방법에는 크게 2가지가 존재함
  - MyBatis
  - JPA Repository

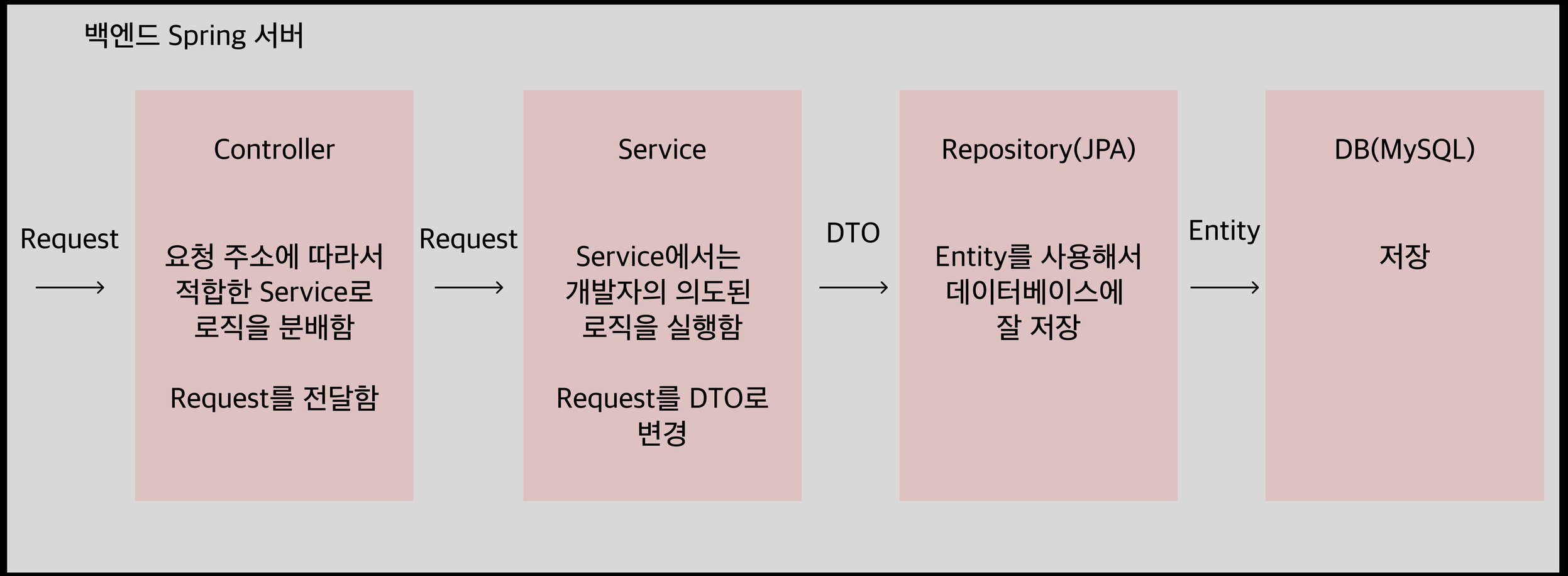
MVC Pattern

- MyBatis와 JPA의 차이점?

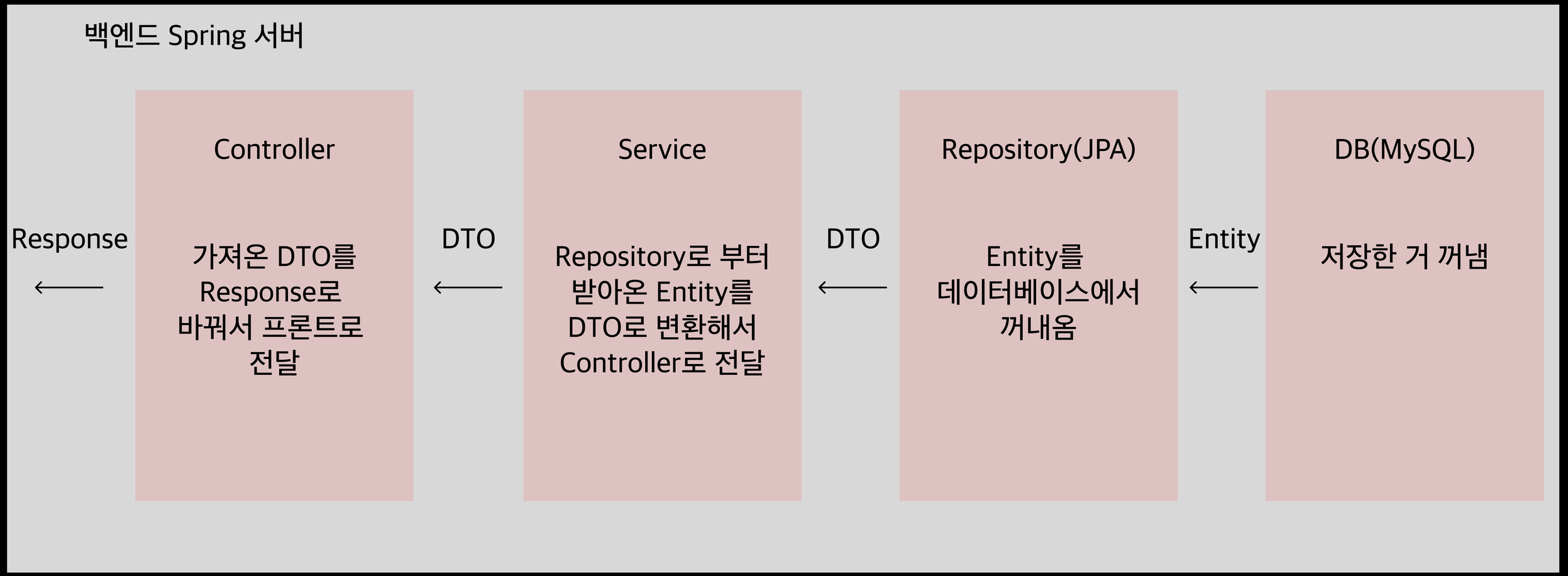
- MyBatis는 데이터베이스와 개발자의 관계에서 데이터베이스가 조금 더 편한 방식을 사용함
  - 쿼리를 다 정의하는 방식

- JPA는 데이터베이스와 개발자의 관계에서 개발자가 조금 더 편한 방식을 사용함
  - 자바에서 사용하는 객체 자체를 데이터베이스에 넣는 방식

MVC Pattern


백엔드 Spring 서버
Request
Controller
요청 주소에 따라서
적합한 Service로
로직을 분배함
Request를 전달함
Request
Service
Service에서는
개발자의 의도된
로직을 실행함
Request를 DTO로
변경
DTO
Repository(JPA)
Entity를 사용해서
데이터베이스에
잘 저장
Entity
DB(MySQL)
저장

MVC Pattern


백엔드 Spring 서버
Controller
Service
Repository(JPA)
DB(MySQL)
Response
DTO
DTO
Entity
가져온 DTO를 Response로 바꿔서 프론트로 전달
Repository로 부터 받아온 Entity를 DTO로 변환해서 Controller로 전달
Entity를 데이터베이스에서 꺼내옴
저장한 거 꺼냄

MVC Pattern

- Entity
  - 데이터베이스에 저장하고 데이터베이스에서 가져오는 객체를 의미함
  - 실제로 저장되는 객체

- DTO
  - 서비스에서 사용하는 형태의 객체
  - Request / Response와 Entity의 중간 다리 역할을 수행함
  - Entity를 바로 사용하게 되면 생기는 문제를 막기 위해서 사용함

- Request / Response
  - 프론트엔드와 소통하는 형태의 객체
  - 프론트에서는 Request에 맞게 JSON을 만들어서 백으로 전달
  - 백은 사전에 서로 협의 된 Response 타입을 Return -> 프론트에서 JSON으로 받아서 사용

DTO 문제?

- Post와 User Table이 있다고 가정
- User는 여러 개의 Post Instance와 관계를 가질 수 있고, Post는 하나의 User를 무조건 가져야 함
  - One to Many Relationship

- 이 때, Entity를 바로 사용하면 생기는 문제?

🦁 MVC Pattern

DTO 문제?

- Post와 User Table이 있다고 가정
- User는 여러 개의 Post Instance와 관계를 가질 수 있고, Post는 하나의 User를 무조건 가져야 함
  - One to Many Relationship

- 이 때, Entity를 바로 사용하면 생기는 문제?

- 하나의 Post를 가지고 왔을 때, 작성자에 대한 정보를 같이 가지고 옴
- 작성자는 여러 게시물을 가지고 있기 때문에 해당되는 모든 게시물을 가지고 옴
- 해당 게시물들은 다시 작성자에 정보를 가지고 있기 때문에 작성자의 정보를 가지고 옴
- 작성자는 다시 ..

===> 무한 참조에 걸리게 됨

🦁 MVC Pattern

DTO 문제?

- Post와 User Table이 있다고 가정
- User는 여러 개의 Post Instance와 관계를 가질 수 있고, Post는 하나의 User를 무조건 가져야 함
  - One to Many Relationship

- 이 때, Entity를 바로 사용하면 생기는 문제?

- 하나의 Post를 가지고 왔을 때, 작성자에 대한 정보를 같이 가지고 옴
- 작성자는 여러 게시물을 가지고 있기 때문에 해당되는 모든 게시물을 가지고 옴
- 해당 게시물들은 다시 작성자에 정보를 가지고 있기 때문에 작성자의 정보를 가지고 옴
- 작성자는 다시 ..

===> 무한 참조에 걸리게 됨

🦁 MVC Pattern

DTO 문제?

- 무한 참조는 FK의 위치와는 상관 없는 문제
  - 이는 데이터베이스의 관점

- 객체의 관점에서 1:N으로 매핑하게 되면, 각 Entity 안에 필드가 생김

```
@ManyToOne(fetch = FetchType.LAZY)
private User user;
```

```
@OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
private List<Post> postList;
```

DTO 문제?

- 무한 참조는 FK의 위치와는 상관 없는 문제
  - 이는 데이터베이스의 관점

- 객체의 관점에서 1:N으로 매핑하게 되면, 각 Entity 안에 필드가 생김

- 이 둘은 서로를 가리키는 양방향 구조를 가지게 됨
  - DB 테이블에 FK가 없어도 Entity 레벨에서 양방향 참조 관계를 가지게 됨

```
@ManyToOne(fetch = FetchType.LAZY)
private User user;
```

```
@OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
private List<Post> postList;
```

🦁 MVC Pattern

DTO 문제?

- @OneToMany 어노테이션을 사용하게 될 경우, 엔티티 레벨에서 참조 관계를 가지게 됨
- FK와 코드상 Entity가 참조하는 여부는 다름

- 이를 해결하기 위해서 참조를 끊어주는 DTO를 사용함

- DTO 안에 User 정보를 참조하지 않도록 설계

```
17 usages  김동규 +1
@Getter
@Setter
@Builder
@AllArgsConstructor
@RequiredArgsConstructor
public class PostDto {
    private Long postId;
    private String title;
    private String content;
    private int type;
    private List<PostFileDto> postFileDtoList;

    김동규
    public static PostDto toResponse(Post post) {
        return PostDto.builder()
                .postId(post.getId())
                .title(post.getTitle())
                .content(post.getContent())
                .type(post.getType())
                .build();
    }
}
```

DTO 문제?

- 다른 해결 방법으로는 양방향 자체를 사용하지 않는 방법이 존재함
- 예를 들어, Entity로 가지고 올 때, Post Entity 안에서 User 자체를 선언하지 않고 사용
  - 그렇게 될 경우, 양방향 무한 참조 문제가 발생하지 않음

- 하지만 양방향 접근은 많은 이점이 있기 때문에 잘 사용하면 편함!!