



It's Your Life

with





전체 스프링의

구조와 흐름!





Client

<http://localhost:8080/post>

1. HTTP Request

DispatcherServlet

2. Request 요청에
맞는 Controller 를 찾기



잠깐! 우리는 크게 2가지의
스프링 응답에 대해서 배웠습니다!

그 2가지가 뭐였죠!?



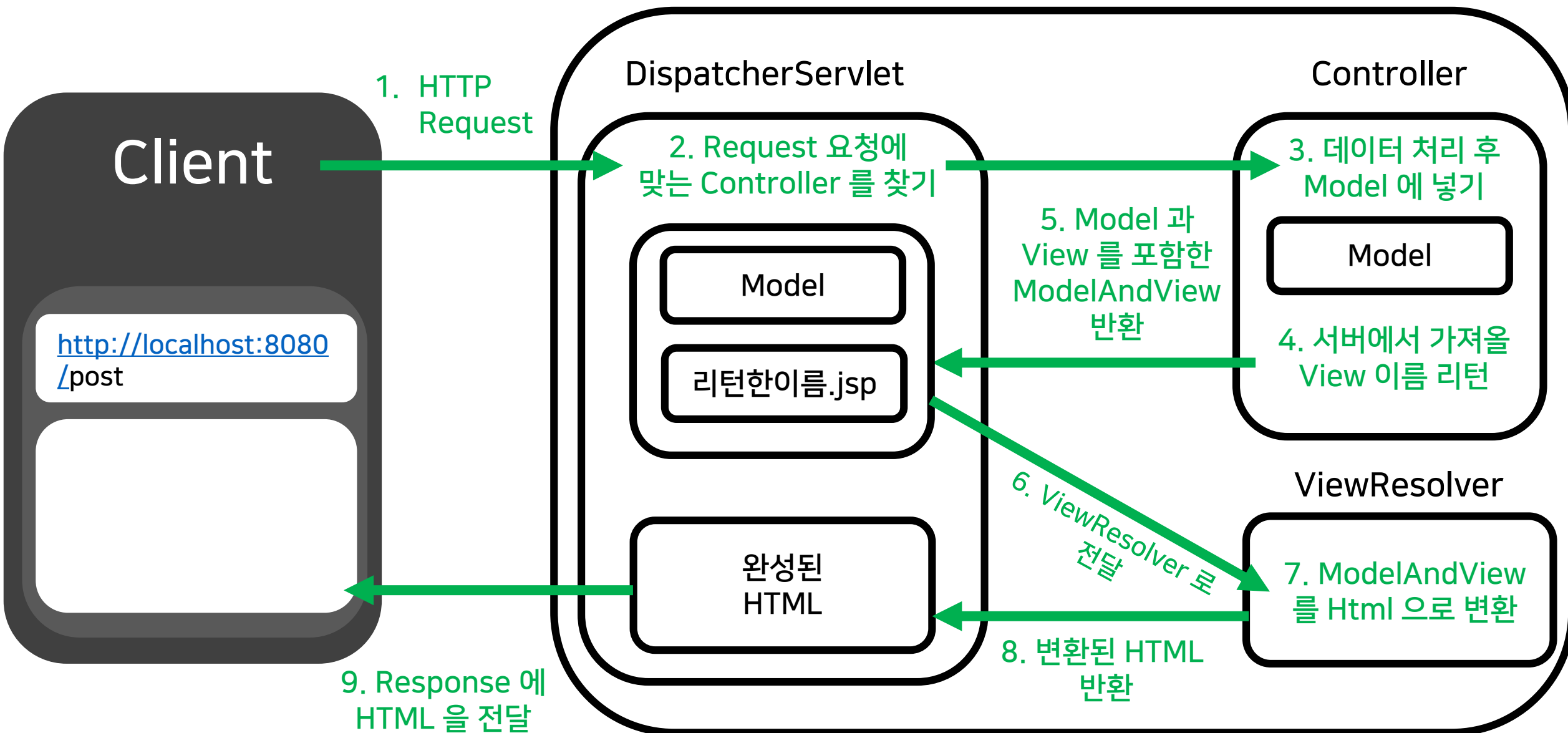
그런데 말입니다



MVC에 의한

View 전달

Tomcat Server

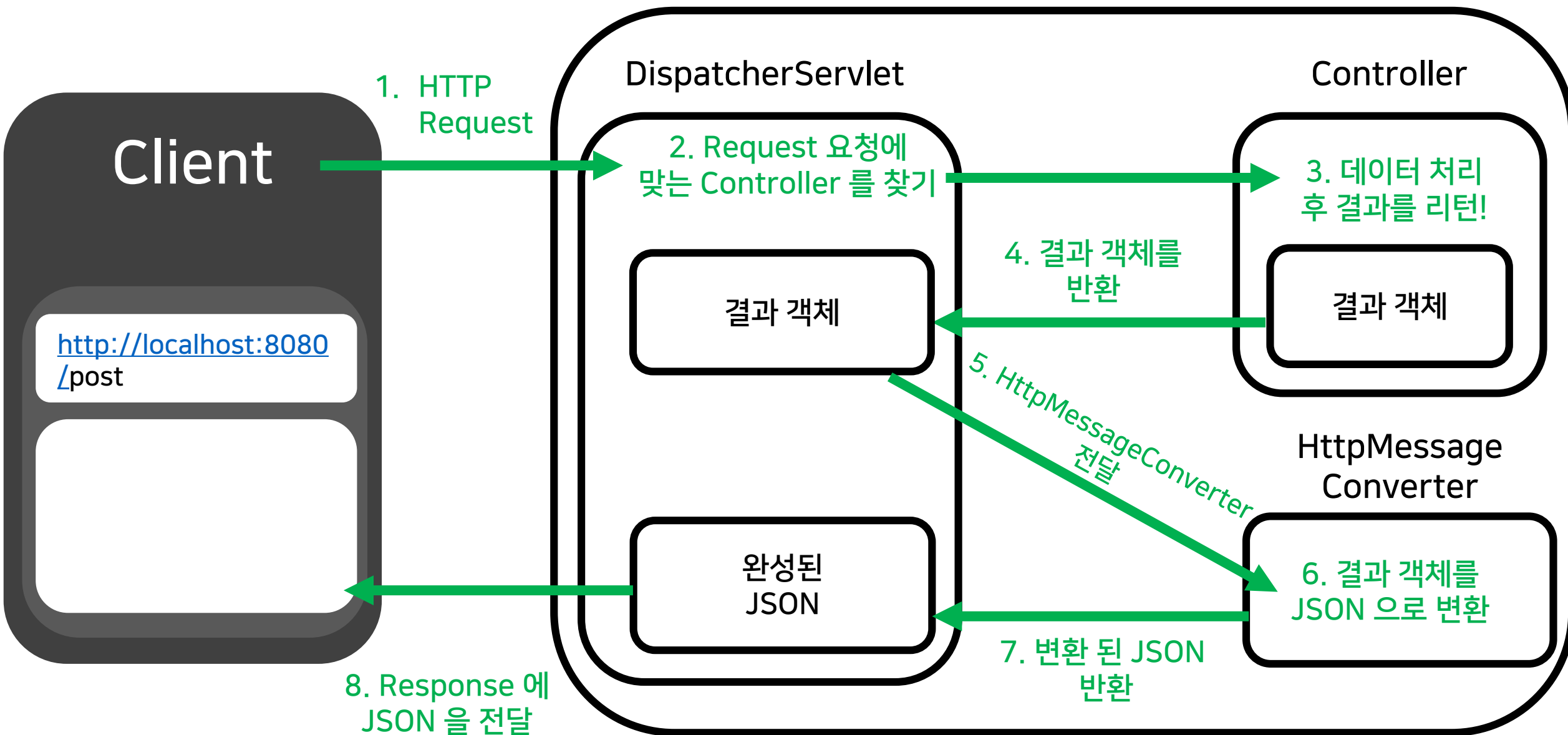




REST 방식에 의한

JSON 전달

Tomcat Server





스프링의 흐름

자세히 보기

스프링이 컨트롤러를



배정하는 순간!



Client

<http://localhost:8080/post>

1. HTTP Request

DispatcherServlet

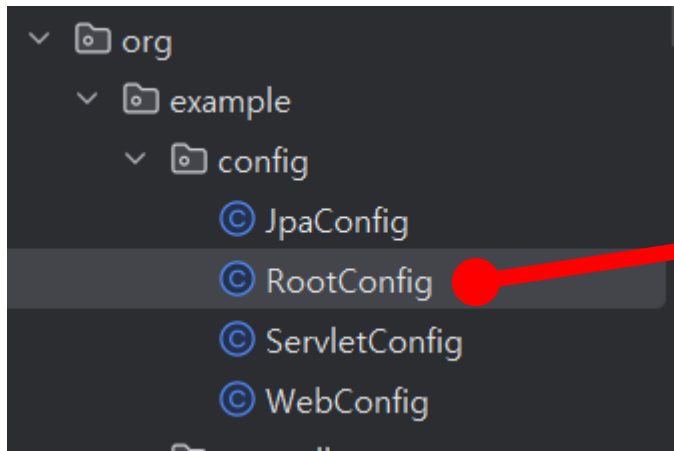
2. Request 요청에
맞는 Controller 를 찾기

주옥같은 뮤지컬 넘버

뮤지컬 지킬앤하이드

“지금 이 순간”





스프링 설정을 확인하기 위해
RootConfig 를 봅시다!

바로 요 @ComponentScan 이 중요합니다!

가장 기본적인 스프링의 빈 들을
어느 패키지에서 찾을지를 설정하는 어노테이션!

```
26  @Configuration
27  @ComponentScan(basePackages = "org.example")
28  @MapperScan(basePackages = {"org.example.mapper"})
29  @PropertySource("classpath:application.properties")
30  public class RootConfig {
```



그런데 스프링에서

Component 는 무엇을 의미할까요!?

Component 는 가장 기본적인
스프링의 Bean의 형태입니다!

Component 는 아래의 빈들을 포함

1. Component
2. Service
3. Repository
4. Controller
5. RestController
6. Configuration
7. Bean

그런데 말입니다



그란데 말입니다



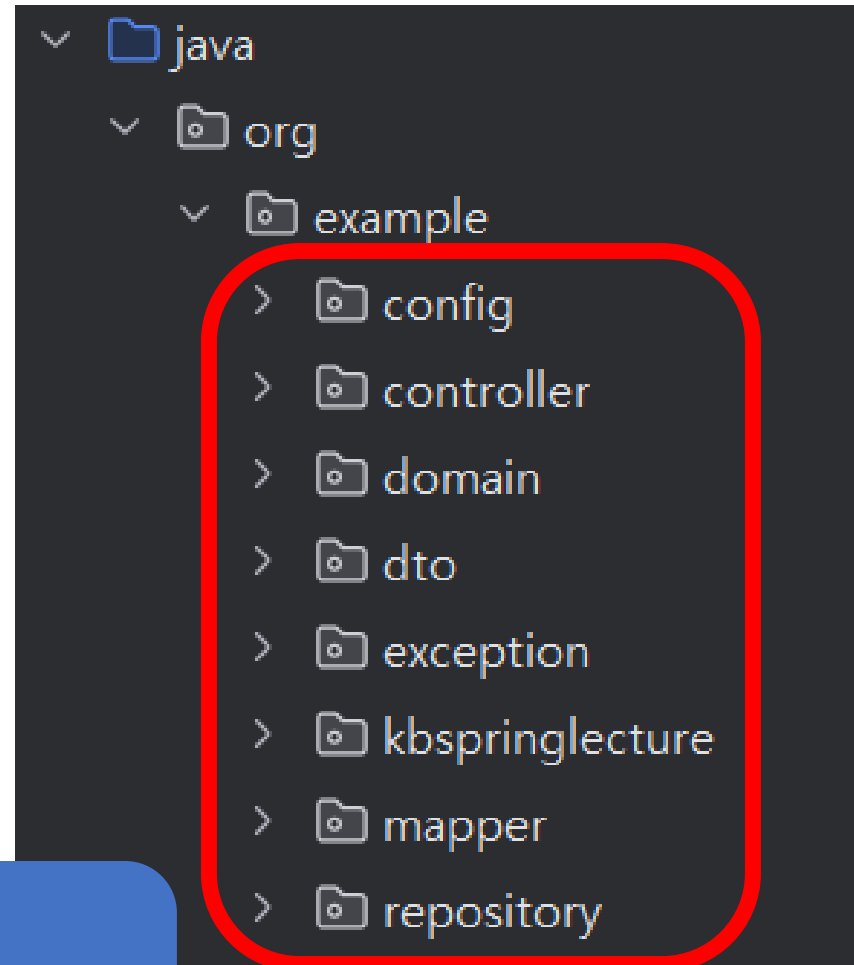
그렇다면
@ComponentScan 을 지정하면
해당 패키지에 있는 어떤 Bean 을 등록 할까요!?

넵! 당연히 아래의 Bean 어노테이션이 붙은
클래스들을 전부 찾아서 등록합니다!!

1. @Component
2. @Service
3. @Repository
4. @Controller
5. @RestController
6. @Configuration
7. @Bean



@ComponentScan 어노테이션으로 인해
Spring 은 지정한 example 패키지 하위의
모든 패키지로 부터 Bean 들을 찾아서 등록 합니다!



```
@Controller  👤 kdtTetz *  
@Slf4j  
@RequiredArgsConstructor  
@RequestMapping(🌐"/post/v1")  
public class PostController {
```

잡았다 요놈



```
@RestController  👤 Tetz  
@Slf4j  
@RequiredArgsConstructor  
@CrossOrigin(origins = "http://localhost")  
@RequestMapping(🌐"/post/v1/rest")  
public class RestPostController {
```

잡았다 요놈



```
✓ @Repository  👤 kdtTetz  
@RequiredArgsConstructor  
public class PostRepository {
```

잡았다 요놈



@Service Bean 목록

1. PostService
2. TodoService
3. LoginService
- ...
- ...

@Controller Bean 목록

1. PostController
2. TodoController
3. LoginController
- ...
- ...

@Repository Bean 목록

1. PostRepository
2. TodoRepository
3. LoginRepository
- ...
- ...



스프링이 구동이 되면 자동으로
지정한 패키지에서 Bean 을 찾아서
위와 같이 등록하고 목록을 가지고 있습니다!



WATCHA PLAY

ZANAM



“나 다시 돌아갈래!!”



Client

<http://localhost:8080/post>

1. HTTP Request

주소 요청은
/post

DispatcherServlet

2. Request 요청에
맞는 Controller 를 찾기

```
✓ @Controller kdtTetz *  
@Slf4j  
@RequiredArgsConstructor  
@RequestMapping(🌐✓"/post/v1")  
public class PostController {
```

@Controller 로 등록 된 Bean 중에서
/post 주소에 매핑 된 컨트롤러가 있네요!?





스프링은 Request 주소 요청을 확인하고
해당 주소(/board)로 매핑이 되어있는
Controller Bean 이 있는지를 찾습니다!



@Service Bean 목록

1. PostService
2. TodoService
3. LoginService
- ...
- ...

@Controller Bean 목록

1. PostController
2. TodoController
3. LoginController
- ...
- ...

@Repository Bean 목록

1. PostRepository
2. TodoRepository
3. LoginRepository
- ...
- ...



Client

<http://localhost:8080/post>

1. HTTP Request

주소 요청은
/post

DispatcherServlet

2. Request 요청에
맞는 Controller 를 찾기

Tomcat Server



Client

[http://localhost:8080/
board](http://localhost:8080/board)

1. HTTP
Request

DispatcherServlet

2. Request 요청에
맞는 Controller 를 찾기

PostController





스프링이 Bean을

주입하는 순간!



여러분은 이제 컨트롤러가 어떤 식으로
배정이 되는지 완-벽-히 이해했습니다!!





그란데 말입니다

그렇다면
스프링의 의존성 주입이 무엇인지
설명 가능하신 분!?

스프링은 실제 서비스의 뼈대만 제공

실제로 서비스는 구현 된 구현체를
스프링에 끼워 넣는(= 주입) 형태로
서비스가 정해지는 구조를 말합니다!

스프링에 게시판 서비스를 주입하면?
→ 게시판 서비스

스프링에 Todo 서비스를 주입하면?
→ Todo 서비스

@Service Bean 목록

1. PostService
2. TodoService
3. LoginService
- ...
- ...

@Controller Bean 목록

1. PostController
2. TodoController
3. LoginController
- ...
- ...

@Repository Bean 목록

1. PostRepository
2. TodoRepository
3. LoginRepository
- ...
- ...



스프링은 이렇게 자신이 사용할 구현체들을
Bean 목록에 하나씩만 등록해 놓습니다!



그런데 말입니다

스프링은 왜 하나씩만 구현체를
Bean 으로 등록하나요!?

+ 싱글톤 패턴이 뭐였죠!?

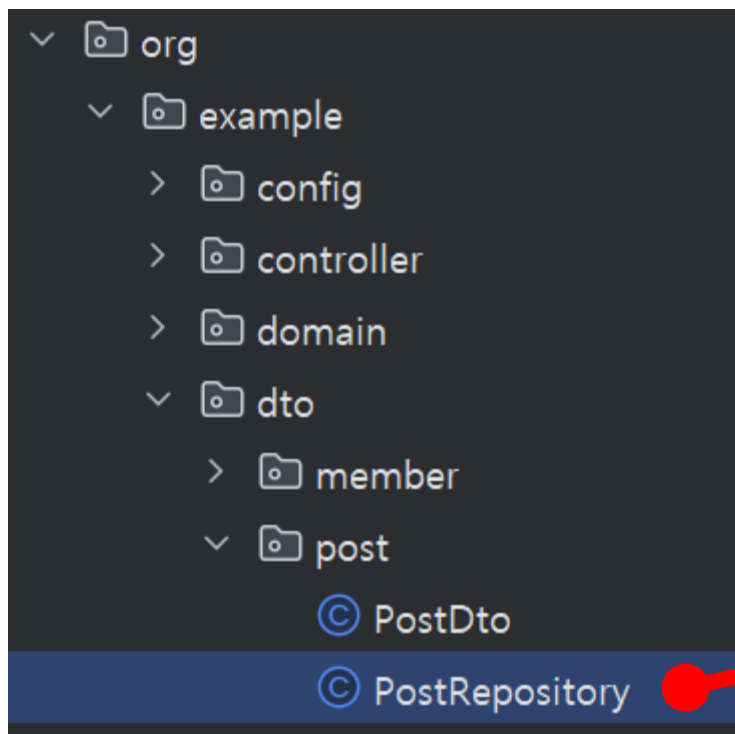
똑같은 Bean 이 여러 개면!?

1. 관리가 힘들어 집니다
2. 특정 순간에 어떤 Bean 을
선택해야할지 헷갈립니다
3. Bean 별로 다른 데이터를 가지게
되어서 혼선이 발생합니다



실제 Bean 이 주입(= 의존성 주입)이
일어나는 코드를 봅시다!!





Bean 등록이 되는 코드를 확인하기 위해
PostRepository 로 오시면 됩니다!

인텔리제이에서 해당 클래스가
Bean 으로 등록 되었음을
나타내는 아이콘 입니다!

해당 PostRepository 클래스는
스프링 Bean 에 Repository 로 등록 되어 있습니다!



```
@Repository kdtTetz
@RequiredArgsConstructor
public class PostRepository {
    private final PostMapper postMapper;
```

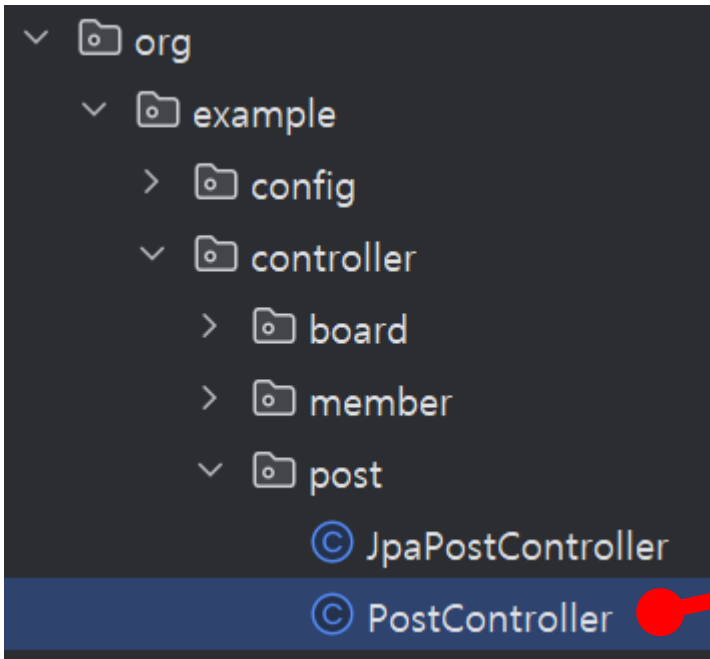
The screenshot shows a code editor with the following code: `@Repository kdtTetz`, `@RequiredArgsConstructor`, `public class PostRepository {`, and `private final PostMapper postMapper;`. A red arrow points from the `@Repository` annotation to the text box on the left. Another red arrow points from the `PostRepository` class name to the text box on the right. There are also some icons on the left side of the code editor, including a green circle with a diagonal line and a green circle with a right arrow.

@Repository Bean 목록

1. PostRepository
2. TodoRepository
3. LoginRepository
- ...
- ...

스프링의 Repository Bean 목록은
아마도 이런 식으로 관리되고 있을 겁니다!





Bean 이 주입되는 코드 확인을 위해
PostController 로 오시면 됩니다!

의존성 주입을 받아 사용할 인스턴스를
private final 로 선언합니다!

```
@Controller kdtTetz *
@Slf4j
@RequestMapping("/post/v1")
public class PostController {
    private final PostRepository postRepository; 7 usages
    private String context = "/post/v1"; 5 usages

    @Autowired new *
    public PostController(PostRepository postRepository) {
        this.postRepository = postRepository;
    }
}
```

PostController 인스턴스가 생성되는 시점에
Bean 목록으로 부터 PostRepository 를
매개변수로 전달 받아서 의존성을 주입!

```
@Controller kdtTetz *  
@Slf4j  
@RequestMapping("/post/v1")  
public class PostController {  
    private final PostRepository postRepository; 7 usages  
    private String context = "/post/v1"; 5 usages  
  
    @Autowired new *  
    public PostController(PostRepository postRepository) {  
        this.postRepository = postRepository;  
    }  
}
```

현재 생성자에서 의존성 주입이 일어나고 있다는 것을
인텔리제이가 표시! + @Autowired 어노테이션으로 명시

@Repository Bean 목록

1. PostRepository
2. TodoRepository
3. LoginRepository
- ...
- ...



어!? 클래스가 생성 될 때
PostRepository 타입의
멤버 변수를 생성하네!?

이게 Bean 목록에 있나!?

```
@Autowired new *  
public PostController(PostRepository postRepository) {  
    this.postRepository = postRepository;  
}
```

@Repository Bean 목록

1. PostRepository
2. TodoRepository
3. LoginRepository
- ...
- ...



어라 있네!?
그럼 Bean 에 등록 된 걸
고대로 쓰면 되겠구나!

```
@Autowired new *  
public PostController(PostRepository postRepository) {  
    this.postRepository = postRepository;  
}
```


@Repository Bean 목록

1. PostRepository

2. TodoRepository

3. LoginRepository

...

...

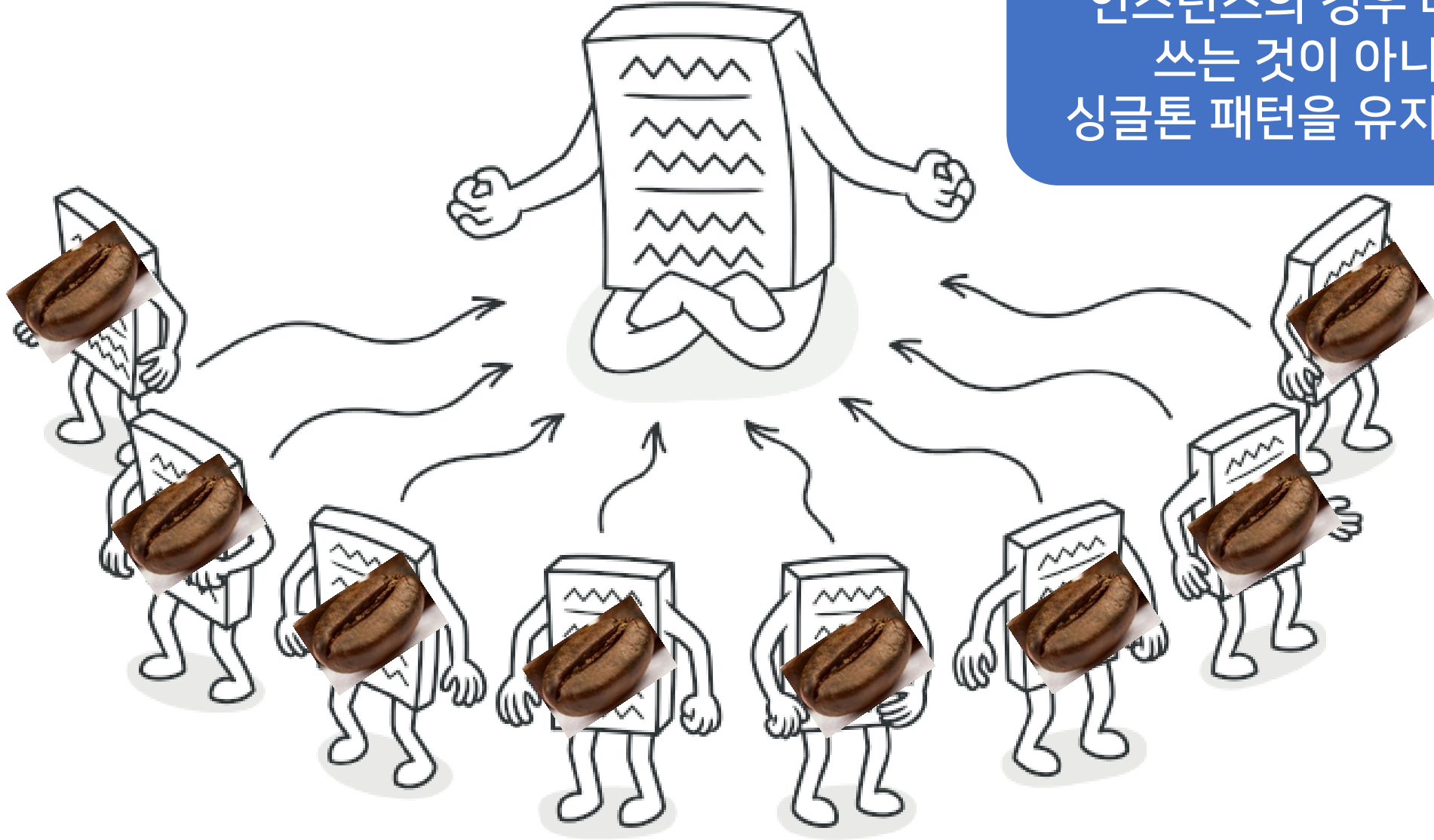


그럼 Bean 목록에 있는 걸
매개 변수로 전달해서
주입 시켜야지!

```
@Autowired new *  
public PostController(PostRepository postRepository) {  
    this.postRepository = postRepository;  
}
```



스프링은 자신의 Bean 에 등록된
인스턴스의 경우 매번 새로 생성해서
쓰는 것이 아니라 주입을 통해
싱글톤 패턴을 유지하면서 운영이 가능

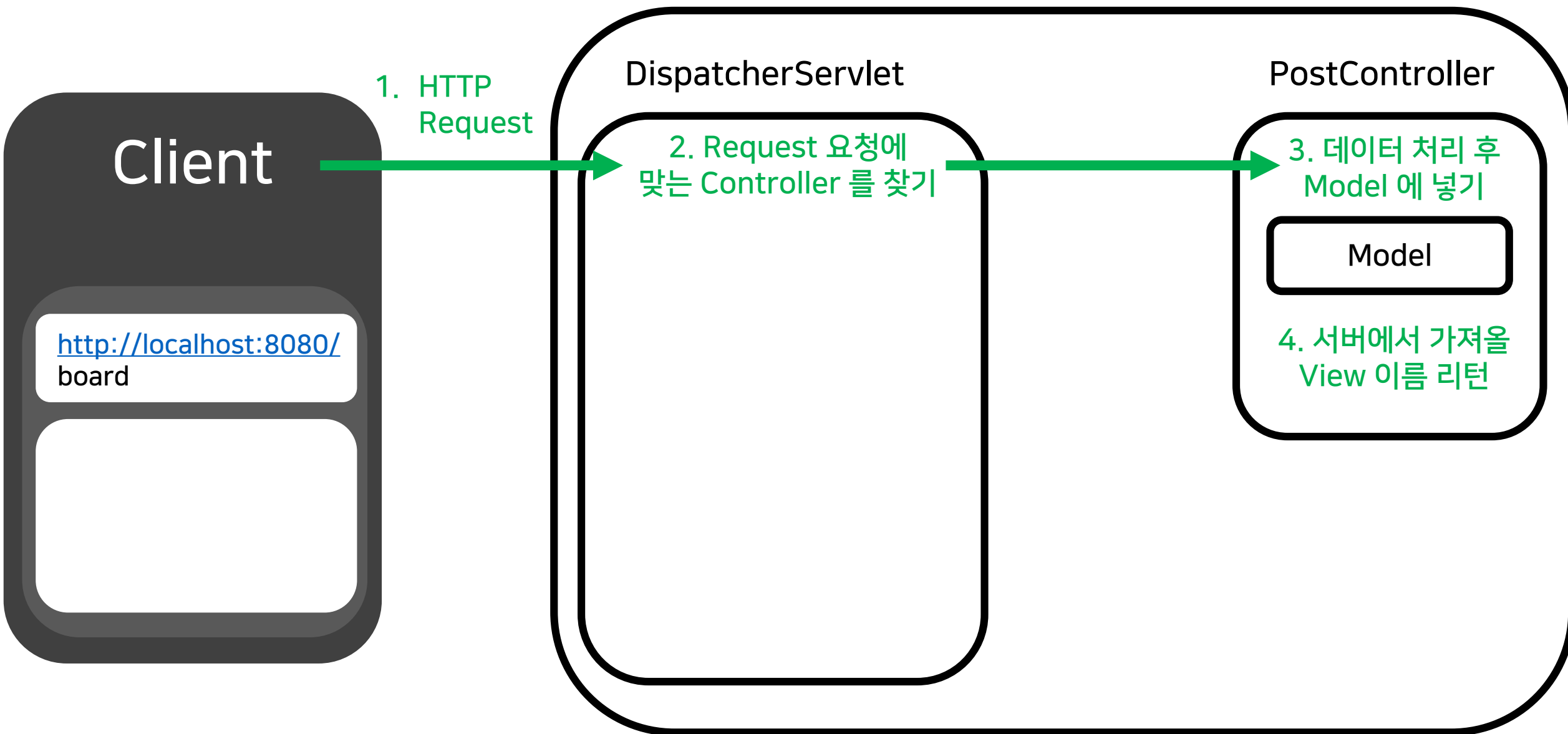


스프링이 View 파일을



배정하는 순간!

Tomcat Server

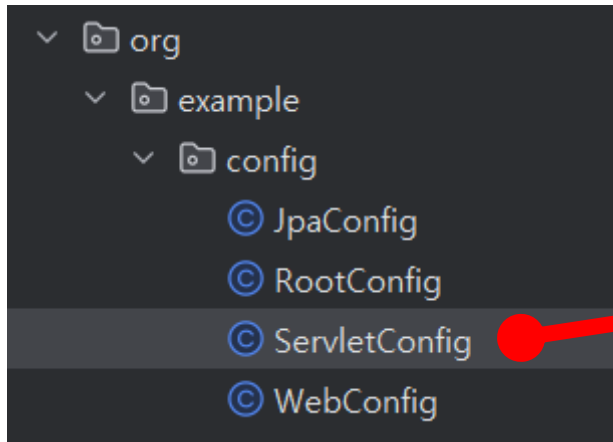


주옥같은 뮤지컬 넘버

뮤지컬 지킬앤하이드

“지금 이 순간”





스프링 설정을 확인하기 위해
ServletConfig 를 봅시다!



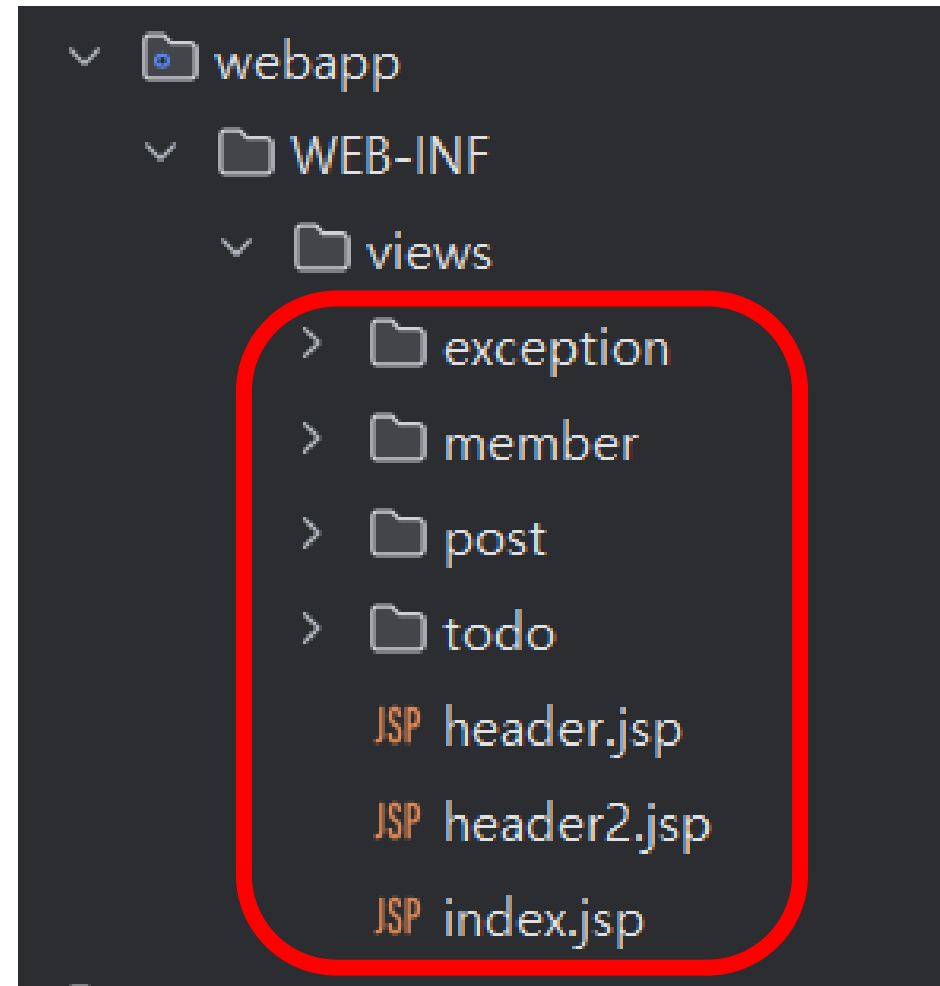
결국 View 파일을 HTML 로 변환하는 건
ViewResolver 가 하는 일이므로
ViewResolver 에 설정을 해줍니다!

```
// jsp view resolver 설정
@Override 3 usages kdtTetz
public void configureViewResolvers(ViewResolverRegistry registry){
    InternalResourceViewResolver bean = new InternalResourceViewResolver();
    bean.setViewClass(JstlView.class);
    bean.setPrefix("/WEB-INF/views/");
    bean.setSuffix(".jsp");
    registry.viewResolver(bean);
}
```

어떤 폴더에서 → /WEB-INF/views
어떤 파일을 → 전달 받은 파일명.jsp
를 찾아서 전달하라고 설정합니다!



ServletConfig 의 설정으로 인하여
Spring 은 지정한 WEB-INF/views 폴더의
하위 폴더로 부터 전달 받은 이름 + .jsp
파일을 전달 합니다!



Tomcat Server



Client

<http://localhost:8080/board>

1. HTTP Request

DispatcherServlet

2. Request 요청에
맞는 Controller 를 찾기

Model

리턴한이름.jsp

Controller

3. 데이터 처리 후
Model 에 넣기

Model

4. 서버에서 가져올
View 이름 리턴

ViewResolver

7. ModelAndView
를 Html 으로 변환

5. Model 과
View 를 포함한
ModelAndView
반환

6. ViewResolver 보
전달

```
// 게시글 목록
```

```
@GetMapping(🌐"/show")  👤 kdtTetz *
```

```
public String postList(HttpServletRequest request, Model model) {
```

```
    log.info("=====> 게시글 목록 페이지 호출, " + request.getRequestURI());
```

```
    model.addAttribute( attributeName: "postList", postRepository.findAll());
```

```
    return context + "/post/post-show";
```

```
}
```

데이터 처리 결과를
Model 에 추가하여 전달!

/WEB-INF/views 폴더에서 찾을 파일명을 리턴



```
// 게시글 목록
```

```
@GetMapping(🌐"/show")  👤 kdtTetz *
```

```
public String postList(HttpServletRequest request, Model model) {  
    log.info("=====> 게시글 목록 페이지 호출, " + request.getRequestURI());  
  
    model.addAttribute( attributeName: "postList", postRepository.findAll();  
    return context + "/post/post-show";  
}
```

/WEB-INF/views 폴더에서 찾을 파일명을 리턴

→ /post/post-show.jsp 파일!

```
✓ WEB-INF  
  ✓ views  
    > exception  
    > member  
    ✓ post  
      > v1  
      > v2
```

JSP post-new.jsp

JSP post-show.jsp

JSP post-update.jsp

post-show.jsp 파일을 찾아서
ViewResolver 에 전달 합니다!



잠깐, 웹 브라우저는
어떤 언어만 해석할 수 있죠!?

1. HTML
2. CSS
3. JS

그런데 말입니다



저기 혹시 JAVA 할 줄 아세요!?



그럼 JSTL 가능 하신가요!?



아 그럼 제가 무지 편할텐데.....



HTML/CSS/JS



웹 브라우저는 JSTL 혹은 JVA 문법을 당연히 모릅니다
+ 전달 받은 Model 의 데이터는 서버 내부에서만 알 수 있습니다!



```
<tbody>
<c:forEach var="post" items="${postList}">
  <tr>
    <td>${post.id}</td>
    <td>${post.title}</td>
    <td>${post.content}</td>
    <td>
      <form action="/post/v1/update" method="get" style="display:inline;">
        <input type="hidden" name="id" value="${post.id}">
        <input type="submit" value="수정" class="update-button">
      </form>
    </td>
  </tr>
</c:forEach>
</tbody>
```

ViewResolver

7. ModelAndView 를 Html 으로 변환

따라서 ViewResolver 는
위와같은 JSP 파일을
순수한 HTML 으로 변환 시킵니다!

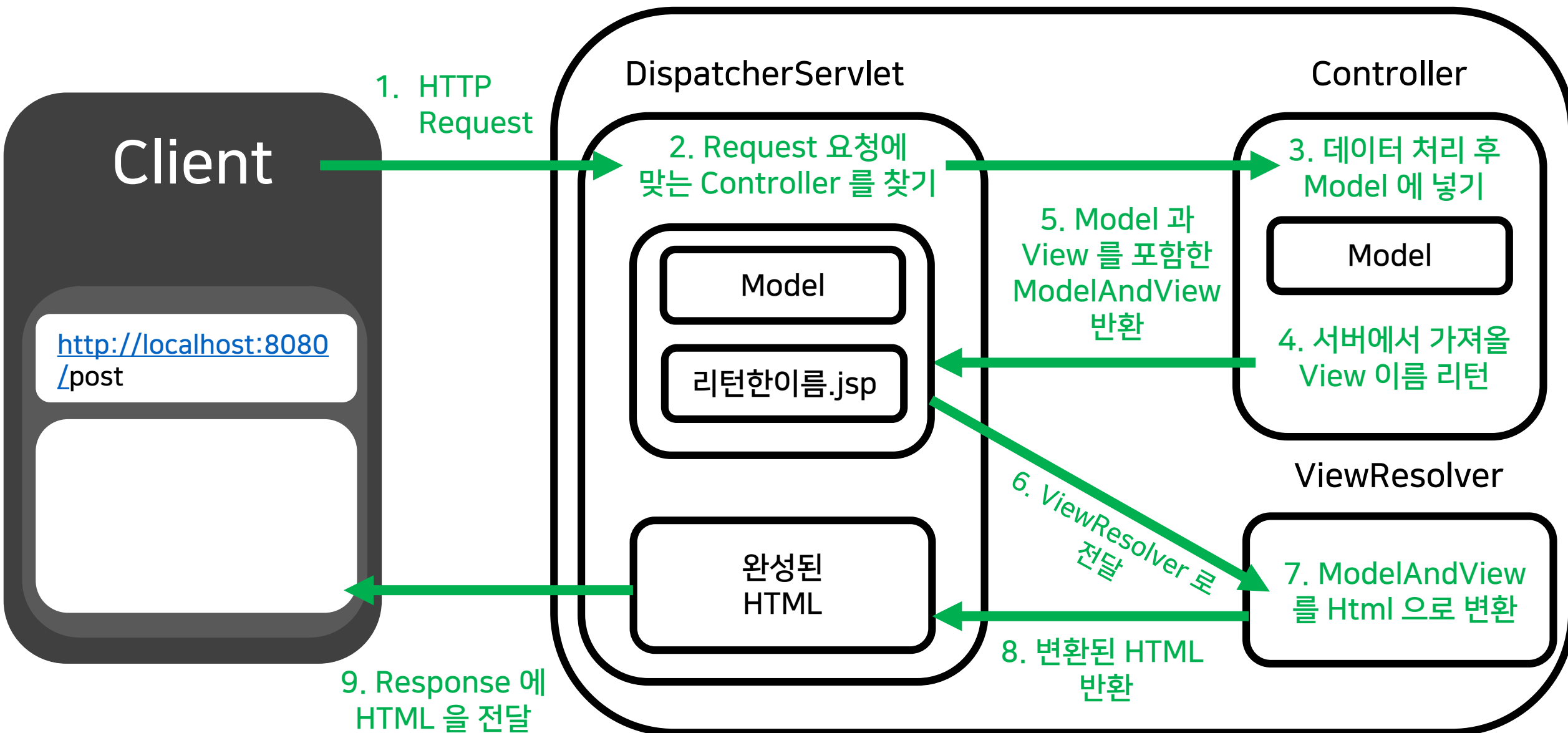




마지막

Response 로 전달!

Tomcat Server





REST API 의

경우



그런데 말입니다

그런데 말입니다!
REST API 는 어떤 차이를 가질까요?

사실 Controller 까지는
정확하게 동일합니다!

대신 View 파일을 HTML 로 변환해서
전달 X

데이터를 JSON 으로 변환하여
전달 O

Tomcat Server



Client

<http://localhost:8080/post>

1. HTTP Request

DispatcherServlet

2. Request 요청에
맞는 Controller 를 찾기

결과 객체

완성된
JSON

Controller

3. 데이터 처리
후 결과를 리턴!

결과 객체

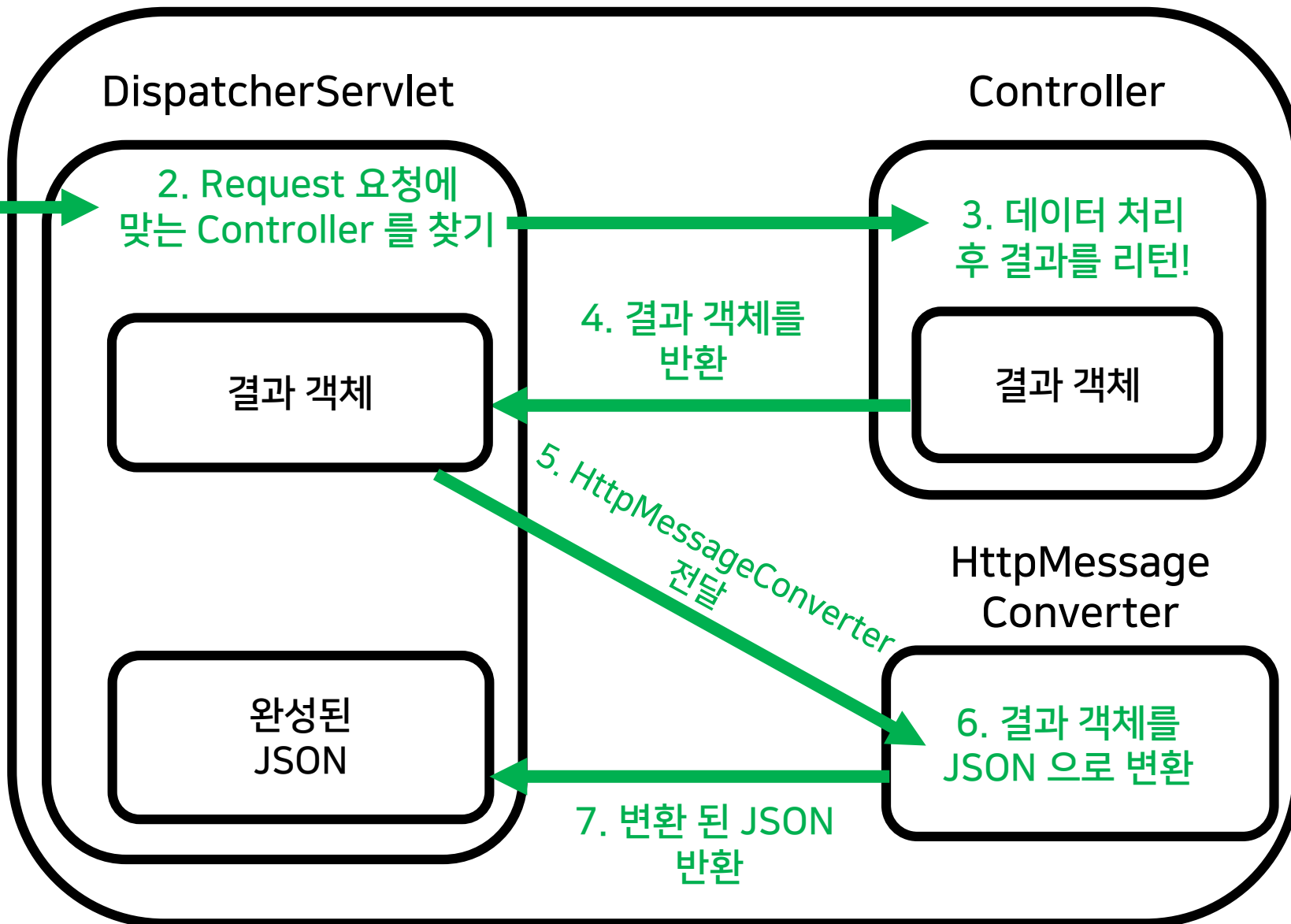
HttpMessage
Converter

6. 결과 객체를
JSON 으로 변환

4. 결과 객체를
반환

5. HttpMessageConverter
전달

7. 변환 된 JSON
반환





```
// 게시글 목록
```

```
@GetMapping(🌐"/show")  👤 Tetz
```

```
public List<PostDto> postList(HttpServletRequest request, Model model) {
```

```
    log.info("=====> 게시글 목록 페이지 호출, " + request.getRequestURI());
```

```
    return postRepository.findAll();
```

```
}
```

리턴 값은 아래 그림과 같이 PostDto 객체를 가지는
List 컬렉션이 될 것입니다!

List<PostDto>

id: 1

첫 번째 글

첫 번째 글의 내용입니다.

id: 2

두 번째 글

두 번째 글의 내용입니다.

id: 3

세 번째 글

세 번째 글의 내용입니다.



그런데 말입니다

그런데 말입니다!
웹 브라우저는 JAVA 를 이해하나요!?

따라서 JS 가 간편하게
객체로 변환이 가능한

JSON 형태로 변환하여
전달이 필요합니다!



저기 혹시 JAVA 할 줄 아세요!?



혹시 객체 아세요!?



아 그럼 제가 무지 편할텐데.....



JSON!!



HttpMessage Converter

6. 결과 객체를
JSON 으로 변환

자바의 객체를
JSON 형태로 변환!



List<PostDto>

id: 1

첫 번째 글

첫 번째 글의 내용입니다.

id: 2

두 번째 글

두 번째 글의 내용입니다.

id: 3

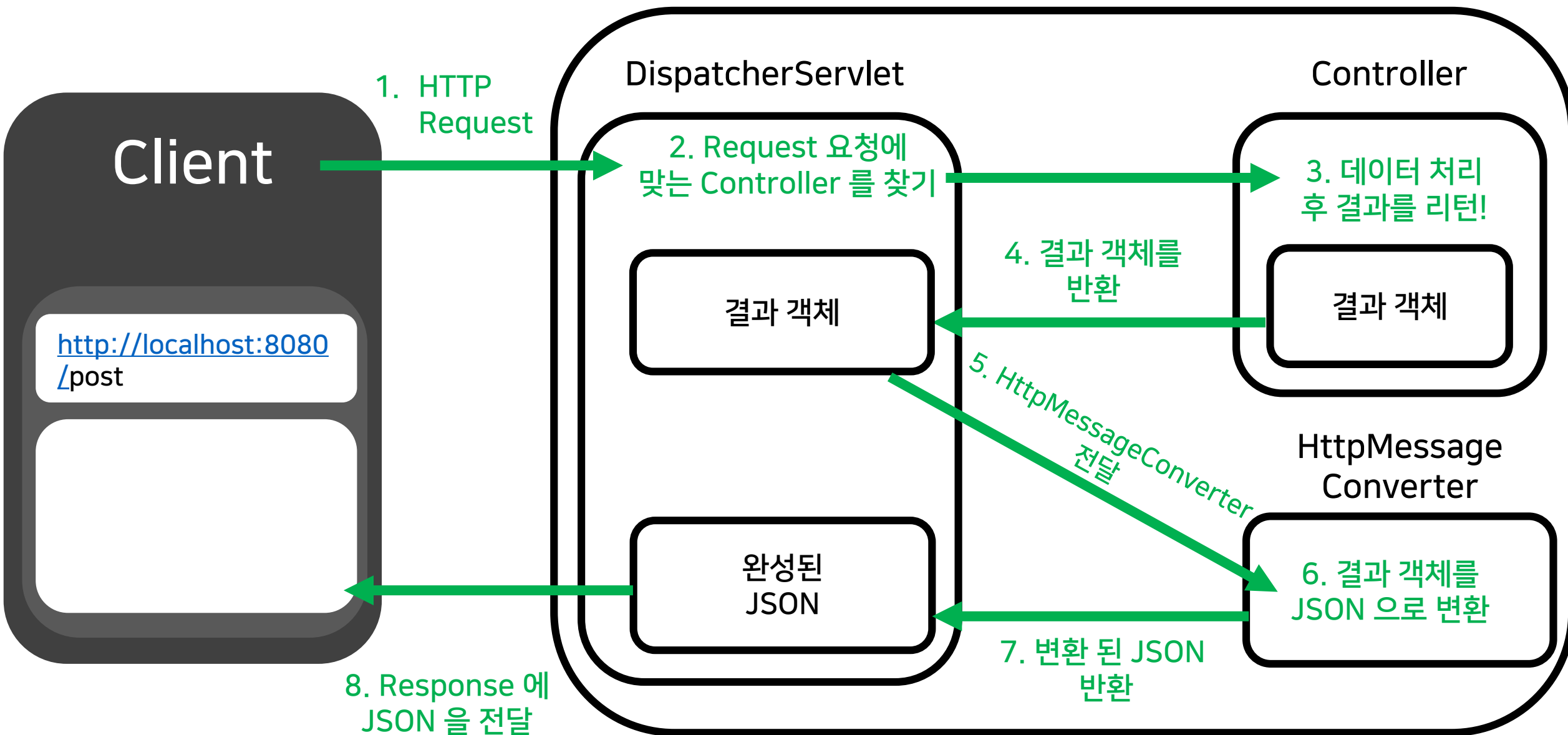
세 번째 글

세 번째 글의 내용입니다.



```
[
  {
    "id": 1,
    "title": "첫 번째 글",
    "content": "첫 번째 글의 내용입니다."
  },
  {
    "id": 2,
    "title": "두 번째 글",
    "content": "두 번째 글의 내용입니다."
  },
  {
    "id": 3,
    "title": "세 번째 글",
    "content": "세 번째 글의 내용입니다."
  }
]
```

Tomcat Server





```
[
  {
    "id": 1,
    "title": "첫 번째 글",
    "content": "첫 번째 글의 내용입니다."
  },
  {
    "id": 2,
    "title": "두 번째 글",
    "content": "두 번째 글의 내용입니다."
  },
  {
    "id": 3,
    "title": "세 번째 글",
    "content": "세 번째 글의 내용입니다."
  }
]
```



"[{W"idW":1,W"titleW":W"첫 번째 글
W",W"contentW":W"첫 번째 글의 내용입니
다.W"},{W"idW":2,W"titleW":W"두 번째 글
W",W"contentW":W"두 번째 글의 내용입니
다.W"},{W"idW":3,W"titleW":W"세 번째 글
W",W"contentW":W"세 번째 글의 내용입니다.W"}]"

어노테이션 정리!





그런데 말입니다
지금까지 우리가 쓴 어노테이션은
어떤 것들이 있었나요!?



그런데 말입니다



```
@Configuration  👤 xenosign +1
@MapperScan(basePackages = {"org.example.mapper"})

@EnableWebMvc

@RestController
@RequiredArgsConstructor
@RequestMapping(🌐✓"/api/board")
@Slf4j
@Controller

@GetMapping(🌐✓"/search")
@PostMapping(🌐✓"/new")
@AllArgsConstructor

@Repository  👤

@ControllerAdvice
@Mapper
```



위의 어노테이션들이 뭘 하는지
전부 아시는 분!?

그래도 일단 어노테이션의 기능을
하나하나 떠올려 봅시다!

그런데 말입니다



설정

어노테이션!


```
@Configuration  👤 Tetz +1  
@ComponentScan(basePackages = "org.example")  
@MapperScan(basePackages = {"org.example.mapper"})  
@PropertySource("classpath:application.properties")  
@EnableWebMvc  
@ControllerAdvice
```



설정 관련 어노테이션



- @Configuration
 - 이 클래스가 Spring의 설정 클래스임을 나타냄
 - Bean 정의나 추가적인 설정 메소드를 포함 → 참고로 이 친구도 Bean 으로 등록!
- @ComponentScan
 - 패키지과 그 하위 패키지에서 Spring 컴포넌트(@Component, @Service, @Repository, @Controller 등)를 자동으로 스캔하고 Bean으로 등록
- @MapperScan
 - MyBatis의 매퍼 인터페이스를 스캔
 - 패키지에서 매퍼 인터페이스를 찾아 MyBatis와 연동

설정 관련 어노테이션



- @PropertySource
 - 애플리케이션의 프로퍼티 파일을 지정
- @EnableWebMvc
 - Spring MVC 구성을 활성화
 - Web 애플리케이션 개발에 필요한 다양한 기능과 설정을 자동으로 구성
- @ControllerAdvice
 - Spring 의 전역 예외 처리 및 공통 기능을 제공하기 위해 사용



Bean 관련 어노테이션!

```
@Component  
@Service  
@Bean  
@Controller  
@RestController  
@Repository
```



Bean 관련 어노테이션



- @Component
 - 가장 기본적인 Spring 컴포넌트
 - 일반적인 Bean으로 등록되어 Spring이 관리
 - 다른 특수화된 어노테이션들의 기본
- @Bean
 - 외부 라이브러리 등을 Bean으로 등록할 때 사용
 - @Configuration 클래스 내의 메서드에 사용되어 Bean을 생성하고 구성
- @Service
 - 비즈니스 로직을 담당하는 서비스 계층의 컴포넌트
 - @Component의 특화된 형태로, 비즈니스 로직이 여기에 위치함을 명시

Bean 관련 어노테이션



- @Controller
 - Spring MVC의 컨트롤러
 - 주로 웹 요청을 처리하고 응답을 반환하는 역할
- @RestController
 - @Controller와 @ResponseBody를 결합한 어노테이션
 - RESTful 웹 서비스에서 사용되며, 메서드 반환 값을 JSON 형태로 전달
- @Repository
 - 데이터 접근 계층(DAO)의 컴포넌트



컨트롤러 매핑 관련

어노테이션!

@GetMapping 🌐 ✓ 👤

@PostMapping

@PutMapping

@DeleteMapping



컨트롤러 매핑 관련 어노테이션



- @GetMapping

- HTTP GET 요청을 특정 핸들러 메서드에 매핑 / 데이터 조회

- @PostMapping

- HTTP POST 요청을 특정 핸들러 메서드에 매핑 / 데이터 생성

- @PutMapping

- HTTP PUT 요청을 특정 핸들러 메서드에 매핑 / 데이터 수정


- @DeleteMapping

- HTTP DELETE 요청을 특정 핸들러 메서드에 매핑 / 데이터 삭제



몸복

어노테이션!

@Data 14 usages  kdtTetz

@AllArgsConstructor

@RequiredArgsConstructor

@NoArgsConstructor

@Builder

@Log4j

@Slf4j



롬복 어노테이션



- @Data
 - 자동으로 getter, setter, equals(), hashCode(), toString() 메서드를 생성
- @AllArgsConstructor
 - 모든 필드를 매개변수로 받는 생성자를 자동으로 생성
- @RequiredArgsConstructor
 - final 필드만을 매개변수로 받는 생성자를 생성
- @NoArgsConstructor
 - 매개변수가 없는 기본 생성자를 자동으로 생성

롬복 어노테이션



- @Builder
 - 빌더 패턴을 자동으로 구현
 - 클래스 내의 메서드를 체이닝으로 사용할 수 있도록 설정
- @Log4j
 - Log4j 로깅 프레임워크를 위한 logger 인스턴스를 자동으로 생성
- @Slf4j
 - SLF4J(Simple Logging Facade for Java) 로깅 인스턴스를 자동으로 생성



그란데 말입니다

빌더 패턴이 뭐죠!?
설명 가능하신 분!?



그란데 말입니다

```
@Data 22 usages  👤 kdtTetz *  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
public class BoardVO {  
    private Long no;  
    private String title;  
    private String content;  
    private String writer;  
    private Date regDate;  
    private Date updateDate;
```

@Builder 어노테이션이 달린
BoardVO 클래스를 봅시다!

해당 클래스에는 @Data 어노테이션으로 인해서
Getter / Setter / toString / hashCode 등의
메서드도 자동으로 생성 되어 있습니다!





```
public class BoardVO {  
    private Long no;  
    private String title;  
    private String content;  
    private String writer;  
    private Date regDate;  
    private Date updateDate;  
}
```

빌더 패턴을 쓰면 각각의 필드에 대한
setter 를 아래와 같이 메서드 체이닝을 사용하여
호출하여 좀 더 간단하게 인스턴스를
생성할 수 있습니다!

```
BoardVO board = BoardVO.builder()  
    .no(1L)  
    .title("Spring 프레임워크 소개")  
    .content("Spring은 자바 엔터프라이즈 애플리케이션 개발을 위한 오픈소스 프레임워크입니다.")  
    .writer("홍길동")  
    .regDate(new Date(System.currentTimeMillis()))  
    .updateDate(new Date(System.currentTimeMillis()))  
    .build();
```



```
BoardVO board = BoardVO.builder()  
    .no(1L)  
    .title("Spring 프레임워크 소개")  
    .content("Spring은 자바 엔터프라이즈 애플리케이션 개발을 위한 오픈소스 프레임워크입니다.")  
    .writer("홍길동")  
    .regDate(new Date(System.currentTimeMillis()))  
    .updateDate(new Date(System.currentTimeMillis()))  
    .build();
```

빌더 패턴을 쓴 위 코드와 사용하지 않은 아래 코드를 비교 해보세요

```
BoardVO board = new BoardVO();  
board.setNo(1L);  
board.setTitle("Spring 프레임워크 소개");  
board.setContent("Spring은 자바 엔터프라이즈 애플리케이션 개발을 위한 오픈소스 프레임워크입니다.");  
board.setWriter("홍길동");  
board.setRegDate(new Date(System.currentTimeMillis()));  
board.setUpdateDate(new Date(System.currentTimeMillis()));
```



Mybatis

어노테이션

```
@MapperScan(basePackages = {"org.example.mapper"})  
@Mapper
```



MyBatis 어노테이션



- @MapperScan

- Spring과 MyBatis를 함께 사용할 때 설정 클래스에서 사용
- 지정된 패키지와 그 하위 패키지에서 MyBatis 매퍼 인터페이스를 스캔
- 스캔된 매퍼 인터페이스들을 자동으로 Spring의 빈으로 등록

- @Mapper

- 해당 인터페이스가 MyBatis 매퍼임을 표기
- MyBatis-Spring 모듈이 이 어노테이션이 붙은 인터페이스를 찾아 자동으로 매퍼 구현체를 생성