3. 프레젠테이션 계층 (Presentation Layer)

컨트롤러 역할과 책임

🎾 1. 컨트롤러란?

컨트롤러(Controller)는 사용자의 요청을 받아서 적절한 로직 계층(Application/Service)을 호출하고, 그 결과를 사용자에게 반환하는 **웹 계층(Presentation Layer)**의 구성 요소이다.

> 📌 웹에서 `URL → 동작`을 매핑해주는 엔트리포인트 역할

🧭 2. 컨트롤러의 주요 책임

구분	설명
📥 요청 수신	HTTP 요청을 받아 파라미터를 처리
📄 요청 데이터 바인딩	DTO로 변환하거나 `@RequestBody`, `@RequestParam`, `@PathVariable`로 값 추출
💉 입력 검증	`@Valid`, `@Validated`, `BindingResult` 등을 통해 유효성 검증 수행
🔀 유스케이스 실행 위임	Service/Application Layer에 명령을 위임
📤 응답 반환	JSON, HTML, 파일 등 다양한 형태의 응답 생성
⚠️ 예외 처리 위임 또는 자체 처리	예외가 발생하면 `@ExceptionHandler` 또는 `@ControllerAdvice`로 위임

🧩 3. 컨트롤러가 해야 하는 것 (O)

항목	이유
✅ HTTP 파라미터 처리	HTTP 요청은 컨트롤러가 가장 먼저 받음
✅ DTO 변환	클라이언트 요청을 내부 모델과 분리하기 위해
✅ 입력 검증	API의 무결성을 확보하기 위해
✅ 명령 위임	Service 또는 Application Layer에 업무 위임
✅ 응답 DTO 생성	도메인 객체를 직접 노출하지 않기 위함
✅ Swagger 문서화	`@Operation`, `@ApiResponse` 등으로 API 문서 명세 가능

❌ 4. 컨트롤러가 하지 말아야 할 것 (X)

항목	이유
❌ 비즈니스 로직 직접 실행	역할이 분리되지 않으면 테스트와 유지보수 어려움
❌ DB 접근 (Repository 호출)	DB는 Service/Application을 통해 간접 접근
❌ 도메인 객체 직접 생성	DTO → 도메인 변환은 Service나 Mapper에서 수행
❌ 트랜잭션 처리	`@Transactional` 은 Service 계층에서 선언

💡 5. 실전 예제

```
@RestController
@RequestMapping("/users")
public class UserController {

    private final RegisterUserUseCase registerUserUseCase;

    public UserController(RegisterUserUseCase registerUserUseCase) {
        this.registerUserUseCase = registerUserUseCase;
    }

    @PostMapping
    public ResponseEntity<Void> register(@RequestBody @Valid UserRegisterRequest request) {
        registerUserUseCase.register(request.toCommand());
        return ResponseEntity.status(HttpStatus.CREATED).build();
    }

    @GetMapping("/{id}")
    public ResponseEntity<UserResponse> getUser(@PathVariable Long id) {
        UserResponse user = registerUserUseCase.findUserById(id);
        return ResponseEntity.ok(user);
    }
}
```

🔎 구성 분석:

- `@RestController` 로 API 정의
- `@RequestBody`, `@Valid` 로 입력 바인딩 + 검증
- 도메인 직접 조작 ❌ → 유스케이스 호출 O
- 응답 DTO(UserResponse)를 사용하여 도메인 노출 방지

🧬 6. Controller → Application Layer 흐름

```
Client Request
    ↓
Controller
    ↓      ← (DTO 검증 및 바인딩)
Application/Service Layer
    ↓
Domain Layer
    ↓
Repository
```

🛠️ 7. 기술 요소 요약

기술	설명
`@Controller`	HTML 뷰 반환용 (Thymeleaf 등)
`@RestController`	JSON API 응답용 (`@Controller + @ResponseBody`)
`@RequestMapping`, `@GetMapping` 등	URL 매핑 지정
`@RequestBody`, `@RequestParam`, `@PathVariable`	요청 파라미터 바인딩
`@Valid`, `@Validated` + `BindingResult`	요청 유효성 검사
`ResponseEntity<T>`	응답 코드, 헤더, 본문을 포함한 결과 반환
`@ExceptionHandler`, `@ControllerAdvice`	예외 처리 방식 분리 가능

✅ 정리: Controller의 역할과 책임 요약표

역할 구분	설명
✅ 책임 있음	요청 바인딩, 입력 검증, 응답 생성, 유스케이스 호출
✅ 기술 사용	`@RestController`, `@RequestBody`, `@Valid`, `ResponseEntity`
❌ 책임 없음	도메인 상태 변경, DB 접근, 트랜잭션 처리, 비즈니스 판단
💡 핵심 원칙	**얇고 가벼운 계층**으로 유지해야 함

`@Controller`, `@RestController` 차이

1️⃣ `@Controller`

✔ 정의

Spring MVC에서 **웹 페이지(View)**를 반환하는 데 사용되는 **기본 컨트롤러 어노테이션**.

✔ 주요 특징

- 메서드가 리턴하는 값은 **뷰 이름(view name)**으로 간주됨.
- 주로 **Thymeleaf, JSP, Freemarker** 같은 **템플릿 엔진**과 함께 사용.
- **모델 객체 전달** 시 `Model`, `ModelAndView`, `@ModelAttribute` 등을 사용.

✔ 반환 방식

- 리턴값은 문자열 → ViewResolver → HTML 렌더링
- 필요하면 `@ResponseBody`를 붙여서 JSON 응답도 가능

2️⃣ `@RestController`

✔ 정의

Spring 4.0부터 도입된, **REST API 전용 컨트롤러 어노테이션**.
`@Controller + @ResponseBody`의 조합이 내부적으로 결합된 것.

✔ 주요 특징

- 리턴값이 **그대로 HTTP 응답 바디에 직렬화(JSON/XML 등)** 됨.
- RESTful API 서버 개발에 사용됨.
- ViewResolver는 사용되지 않음.

🔍 내부 차이 비교

항목	`@Controller`	`@RestController`
목적	HTML View 반환	JSON/XML 등 응답
리턴값 처리	View 이름으로 해석	객체 → JSON 변환
ViewResolver	적용됨	사용 안 함
`@ResponseBody` 필요	✅ 필요함 (JSON 응답 시)	❌ 자동 적용됨
사용 위치	Web Page, SSR 앱	REST API 서버

🧪 실전 예제 비교

📘 `@Controller` 예시

```
@Controller
public class HomeController {

    @GetMapping("/hello")
    public String hello(Model model) {
        model.addAttribute("name", "Spring");
        return "hello"; // → templates/hello.html
    }

    @GetMapping("/api/data")
    @ResponseBody
    public DataDto data() {
        return new DataDto("hi", 123); // JSON 반환
    }
}
```

📗 `@RestController` 예시

```
@RestController
public class ApiController {

    @GetMapping("/api/data")
    public DataDto data() {
        return new DataDto("hi", 123); // 자동 JSON 반환
    }
}
```

💡 언제 어떤 걸 써야 할까?

상황	사용 어노테이션
HTML 뷰를 사용자에게 보여줄 때 (SSR)	`@Controller`
JSON 기반 REST API 응답이 필요할 때	`@RestController`

✅ 핵심 요약 정리

항목	`@Controller`	`@RestController`
기본 용도	웹 페이지 반환	API 응답 처리
내부 구성	단독 어노테이션	`@Controller + @ResponseBody`
ViewResolver 사용	O	X

항목	`@Controller`	`@RestController`
리턴 값	View 이름 or JSON (`@ResponseBody`)	JSON/XML 자동 직렬화
주 사용 사례	MVC 웹 앱	REST API 서버

🎯 결론

- **웹 화면(View 기반 페이지)**을 제공할 때: `@Controller`
- **프론트엔드와 JSON으로 통신하는 API 백엔드**를 만들 때: `@RestController`

View 렌더링 vs API 응답 방식

🧩 1. 개념 요약

구분	View 렌더링 방식	API 응답 방식
목적	HTML 페이지 생성	JSON 등 데이터 전달
대상	브라우저 사용자	프론트엔드(JS), 앱 등 클라이언트
반환 형식	템플릿 뷰 (`.html`, `.jsp`)	JSON, XML, Text 등
어노테이션	`@Controller`	`@RestController`

🎨 2. View 렌더링 방식 (서버 사이드 렌더링, SSR)

✔ 동작 흐름

1. 사용자가 URL 접속 → Spring `@Controller`에 요청 도착
2. `Model`에 데이터를 담음
3. 리턴값 = View 이름 (`String`)
4. `ViewResolver`가 HTML/JSP 템플릿을 찾아 렌더링
5. 최종 HTML이 브라우저에 전송됨

✔ 예시

```
@Controller
public class PageController {
    @GetMapping("/hello")
    public String hello(Model model) {
        model.addAttribute("name", "Spring");
        return "hello"; // → templates/hello.html
    }
}
```

템플릿 (Thymeleaf):

```
<p>Hello, [[${name}]]!</p>
```

✔ 특징

항목	설명
ViewResolver 작동	`return "hello"` → `/templates/hello.html`
템플릿 엔진 필요	Thymeleaf, JSP, Freemarker 등
전체 HTML 생성	서버가 완성된 HTML을 만들어 보냄
SEO 친화적	크롤러가 HTML을 바로 해석 가능

⚙️ 3. API 응답 방식 (RESTful API, JSON)

✔ 동작 흐름

1. 프론트엔드가 AJAX, Axios, fetch 등으로 요청
2. `@RestController` 또는 `@ResponseBody`가 적용된 메서드 실행
3. Java 객체를 JSON으로 **자동 직렬화**
4. HTTP 응답 바디에 JSON 전달

✔ 예시

```
@RestController
public class ApiController {
    @GetMapping("/api/hello")
    public Map<String, String> hello() {
        return Map.of("message", "Hello, JSON");
    }
}
```

응답 결과:

```
{
  "message": "Hello, JSON"
}
```

✔ 특징

항목	설명
뷰 렌더링 없음	데이터만 전달
Jackson 사용	객체 → JSON 자동 직렬화
다양한 클라이언트 대응	React, Vue, Android, iOS 등

항목	설명
RESTful 설계 방식 채택 가능	URL + HTTP 메서드로 의미 표현

🧪 비교 표

항목	View 렌더링 (SSR)	API 응답 (JSON)
주요 어노테이션	`@Controller`	`@RestController`, `@ResponseBody`
반환값	View 이름 (`String`)	객체, DTO, Map 등
전송 포맷	HTML	JSON, XML
응답 대상	브라우저	웹 프론트엔드, 모바일 앱
템플릿 엔진 필요	O	X
사용 목적	화면 제공	데이터 제공
예시	게시판 HTML 페이지	게시글 목록 JSON

💡 언제 어떤 방식을 써야 할까?

상황	선택 방식
웹 사이트 중심 (관리자 페이지, 간단한 SSR)	View 렌더링 (`@Controller`)
SPA(React/Vue), 앱 연동 백엔드, API 서버	API 응답 (`@RestController`)

🚀 혼합 사용도 가능

Spring에서는 하나의 프로젝트 내에서 **두 방식 모두 병행**할 수 있다.

```
@Controller
public class ViewController {
    @GetMapping("/home")
    public String home() { return "home"; }
}

@RestController
@RequestMapping("/api")
public class ApiController {
    @GetMapping("/users")
    public List<UserDto> getUsers() { ... }
}
```

✅ 최종 요약 정리

구분	View 렌더링	API 응답
Controller 종류	`@Controller`	`@RestController`
데이터 처리	Model → View	DTO → JSON
목적	HTML 화면 제공	데이터 응답
대상	브라우저	앱, SPA, 외부 시스템
템플릿 필요	O	X
JSON 반환	`@ResponseBody` 필요	기본 적용됨

클라이언트 요청 매핑: `@RequestMapping`, `@GetMapping`, `@PostMapping`

🔍 1. 공통 목적

모두 **HTTP 요청 URL과 컨트롤러 메서드를 매핑**하기 위한 어노테이션이다.

> 📌 클라이언트가 특정 URL로 요청했을 때 어떤 메서드가 실행될지를 지정함.

🧩 2. `@RequestMapping` – 범용 매핑 어노테이션

✔ 개요

가장 기본적이며 강력한 매핑 어노테이션.
모든 HTTP 메서드(GET, POST, PUT, DELETE 등)를 처리할 수 있다.

✔ 주요 속성

속성	설명	예시
`value` or `path`	요청 URL 경로	`/users`, `/api/items`
`method`	HTTP 메서드 지정	`RequestMethod.GET`
`params`	요청 파라미터 조건	`params = "mode=debug"`
`headers`	요청 헤더 조건	`headers = "mode=debug"`
`consumes`	요청 Content-Type 제약	`consumes = "application/json"`
`produces`	응답 Content-Type 제약	`produces = "application/json"`

✔ 예시

```
@RequestMapping(value = "/hello", method = RequestMethod.GET)
public String hello() {
    return "hello";
}
```

📘 3. `@GetMapping`, `@PostMapping` 등 – 축약형 매핑

✔ 개요

Spring 4.3부터 등장한 **HTTP 메서드 전용 단축 어노테이션**.
가독성과 선언 간결성을 위해 사용됨.

축약형 어노테이션	의미
`@GetMapping`	`@RequestMapping(method = RequestMethod.GET)`
`@PostMapping`	`@RequestMapping(method = RequestMethod.POST)`
`@PutMapping`	`@RequestMapping(method = RequestMethod.PUT)`
`@DeleteMapping`	`@RequestMapping(method = RequestMethod.DELETE)`
`@PatchMapping`	`@RequestMapping(method = RequestMethod.PATCH)`

✔ 예시

```
@GetMapping("/users")
public List<UserDto> getUsers() { ... }

@PostMapping("/users")
public void createUser(@RequestBody UserDto dto) { ... }
```

🧪 4. 예시 비교

`@RequestMapping` 방식	축약형 방식
`@RequestMapping(value="/api", method=RequestMethod.GET)`	`@GetMapping("/api")`
`@RequestMapping(value="/api", method=RequestMethod.POST)`	`@PostMapping("/api")`

🧠 5. 클래스 vs 메서드 레벨 사용

✔ 클래스 레벨

```java
@RestController
@RequestMapping("/users") // 공통 prefix
public class UserController {

    @GetMapping("/{id}")
    public UserDto get(@PathVariable Long id) { ... }

    @PostMapping
    public void create(@RequestBody UserDto dto) { ... }
}
```

> 결과적으로:
> `/users/{id}`, `/users` 등으로 경로가 자동 조합됨

⚠️ 6. 우선순위 및 충돌 방지

- 동일한 URL에 대해 **동일한 HTTP 메서드가 중복**되면 `Ambiguous mapping` 오류 발생
- 가급적이면 **명확한 HTTP 메서드 지정** (`@GetMapping`, `@PostMapping` 사용 권장)
- `params`, `headers`, `consumes`, `produces` 조건으로 정밀하게 조정 가능

🛠️ 7. 고급 속성 예시

```java
// 특정 헤더와 Content-Type 조건으로만 매핑됨
@RequestMapping(
  value = "/api/data",
  method = RequestMethod.POST,
  consumes = "application/json",
  produces = "application/json",
  headers = "X-Request-Type=internal"
)
public ResponseEntity<Data> handle(@RequestBody Data data) {
    return ResponseEntity.ok(data);
}
```

✅ 최종 요약 비교표

항목	`@RequestMapping`	`@GetMapping`, `@PostMapping` 등
지원 HTTP 메서드	모든 메서드(GET, POST, PUT, DELETE, ...)	하나의 특정 메서드만
가독성	다소 장황함	간결하고 직관적

항목	`@RequestMapping`	`@GetMapping`, `@PostMapping` 등
설정 유연성	✅ 매우 세밀한 제어 가능	❌ 세부 속성은 제한적
일반 용도	REST API, 특수 조건 핸들링	단순 CRUD, RESTful 설계

📌 언제 어떤 걸 써야 할까?

상황	선택 어노테이션
REST API 기본 CRUD	`@GetMapping`, `@PostMapping` 등 축약형
Content-Type/헤더 조건 제어 필요	`@RequestMapping`
하나의 메서드가 여러 방식 지원	`@RequestMapping(method={...})`

요청/응답 변환: `@RequestBody`, `@ResponseBody`, JSON 직렬화

🧩 1. 개요

어노테이션	역할
`@RequestBody`	HTTP 요청 본문(JSON)을 **Java 객체로 변환** (역직렬화)
`@ResponseBody`	Java 객체를 **HTTP 응답 본문(JSON)으로 변환** (직렬화)

이 두 어노테이션은 Spring이 제공하는 `HttpMessageConverter`를 기반으로 동작함.

💉 2. `@RequestBody` – 요청 본문 → 객체 (Deserialization)

✔ 개념

HTTP 요청의 body(JSON/XML 등)를 Java 객체로 변환함.
대개 JSON → DTO 로 역직렬화하는 데 사용됨.

```
@PostMapping("/users")
public ResponseEntity<Void> createUser(@RequestBody UserCreateRequest request) {
    // request.name, request.email 사용 가능
    ...
}
```

요청 본문 예시 (`application/json`)

```
{
  "name": "Alice",
  "email": "alice@example.com"
}
```

매핑되는 DTO 클래스

```
public class UserCreateRequest {
    private String name;
    private String email;
    // getter, setter 필수
}
```

> ✔ 내부에서 `MappingJackson2HttpMessageConverter`가 Jackson을 사용해 JSON → 객체로 변환함.

📤 3. `@ResponseBody` – 객체 → 응답 본문 (Serialization)

✔ 개념

Java 객체를 HTTP 응답 body에 **직렬화된 JSON 형태로 출력**함.
즉, JSON API 응답을 생성할 때 사용됨.

```
@GetMapping("/users/{id}")
@ResponseBody
public UserResponse getUser(@PathVariable Long id) {
    return new UserResponse("Alice", "alice@example.com");
}
```

> `@RestController`는 모든 메서드에 `@ResponseBody`가 **자동 적용**됨.

🔁 4. JSON 변환 흐름 (전체 과정)

```
클라이언트 → JSON 요청
           ↓
  @RequestBody → Java 객체로 역직렬화
           ↓
     비즈니스 처리
           ↓
   Java 객체 → @ResponseBody → JSON 직렬화
           ↓
      클라이언트 응답
```

⚙️ 5. 작동 원리: `HttpMessageConverter`

Spring MVC는 내부적으로 `HttpMessageConverter`라는 인터페이스를 통해
요청/응답 변환을 처리함.

컨버터	설명
`MappingJackson2HttpMessageConverter`	JSON ↔ Java 객체 (Jackson 사용)
`StringHttpMessageConverter`	문자열 ↔ String

컨버터	설명
`FormHttpMessageConverter`	폼 데이터 ↔ 객체

Spring Boot는 자동으로 JSON 변환용 Jackson 라이브러리를 등록함.

🔐 6. 필수 조건 및 주의사항

📌 `@RequestBody` 사용 시 주의사항

항목	설명
getter/setter 필요	Jackson이 객체 필드 접근에 필요
기본 생성자 필요	JSON 역직렬화 시 반드시 필요
`@Valid` 연동 가능	입력 값 검증 시 `@Valid`, `BindingResult` 사용
Content-Type 체크	요청의 `Content-Type`이 `application/json`이어야 함

📌 `@ResponseBody` 사용 시 주의사항

항목	설명
객체가 JSON으로 직렬화됨	toString()이 아니라 JSON 형태
순환 참조 주의	엔티티에서 양방향 참조 시 `@JsonManagedReference`, `@JsonIgnore` 사용
예외 처리 시 JSON으로 응답	`@ControllerAdvice`, `@ExceptionHandler`와 조합하면 좋음

🧬 7. 실전 종합 예제

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @PostMapping
    public ResponseEntity<UserResponse> register(
        @RequestBody @Valid UserCreateRequest request) {

        // 역직렬화 완료된 request 객체 사용
        User user = userService.register(request);
        return ResponseEntity.ok(new UserResponse(user));
    }
}
```

요청 DTO

```
public class UserCreateRequest {
    private String name;
    private String email;
    // getter, setter, 기본 생성자
}
```

응답 DTO

```
public class UserResponse {
    private String name;
    private String email;
    public UserResponse(User user) {
        this.name = user.getName();
        this.email = user.getEmail();
    }
}
```

✅ 정리 표

항목	@RequestBody	@ResponseBody
방향	JSON → 객체	객체 → JSON
사용 위치	파라미터	메서드 or 클래스
작동 조건	요청 Content-Type이 JSON	응답 Content-Type이 JSON
자동 처리	`@RestController`에서 자동 사용	`@RestController`에서 자동 사용
기반 기술	`HttpMessageConverter` → Jackson	

🛠️ 추가 심화 주제

- Jackson 커스터마이징 (`@JsonProperty`, `@JsonIgnore`, `ObjectMapper 설정`)
- XML 변환용 `Jaxb2RootElementHttpMessageConverter`
- 요청 로그 디버깅 (`ContentCachingRequestWrapper`)
- `@ModelAttribute` vs `@RequestBody` 차이

입력 검증 및 오류 처리: `@Valid`, `BindingResult`, `@ExceptionHandler`

🧩 1. 개요

요소	역할
`@Valid` / `@Validated`	요청 DTO의 **유효성 검사** 수행 (JSR-303 Bean Validation)
`BindingResult`	검증 실패 시 **오류 정보 수집**
`@ExceptionHandler`	검증 실패 또는 예외 발생 시 **전역/개별 예외 처리**

✅ 전체 흐름 요약

```
1. 클라이언트 요청(JSON)
    ↓
2. @RequestBody + @Valid → 입력 DTO 유효성 검사
    ↓
3. BindingResult에 오류 담김 (컨트롤러 내부 처리 가능)
    ↓
4. 처리하지 않으면 MethodArgumentNotValidException 발생
    ↓
5. @ExceptionHandler or @ControllerAdvice 에서 처리
    ↓
6. JSON 형식으로 오류 응답 반환
```

🧪 2. `@Valid` / `@Validated` – 유효성 검증 트리거

✔ 역할

- `javax.validation.Valid`
- DTO에 선언된 **제약 조건(annotation)**을 기준으로 **자동 검증** 수행

✔ 지원하는 제약 어노테이션

어노테이션	설명
`@NotNull`	null이 아니어야 함
`@NotBlank`	공백/빈문자 X
`@Size(min, max)`	길이 제한
`@Email`	이메일 형식
`@Pattern`	정규식

어노테이션	설명
`@Min`, `@Max`	숫자 범위

✔ 예시

```
public class UserCreateRequest {
    @NotBlank(message = "이름은 필수입니다.")
    private String name;

    @Email(message = "이메일 형식이 올바르지 않습니다.")
    private String email;

    // getter/setter
}
```

📥 3. `BindingResult` – 유효성 실패 정보 수집

✔ 동작 방식

컨트롤러 메서드에서 `@Valid` 바로 뒤에 `BindingResult`를 선언하면
→ 검증 실패 시에도 **예외가 발생하지 않고 메서드 내부에서 처리 가능**.

```
@PostMapping("/users")
public ResponseEntity<?> createUser(
    @RequestBody @Valid UserCreateRequest request,
    BindingResult bindingResult
) {
    if (bindingResult.hasErrors()) {
        return ResponseEntity.badRequest().body(bindingResult.getAllErrors());
    }

    userService.create(request);
    return ResponseEntity.ok().build();
}
```

> 📌 `BindingResult`가 없으면 → `MethodArgumentNotValidException`이 발생하고, 예외 처리로 넘어감.

⚠️ 4. 예외 미처리 시 발생하는 기본 예외

예외 클래스	발생 조건
`MethodArgumentNotValidException`	`@RequestBody + @Valid` 오류 발생 시
`ConstraintViolationException`	`@Validated`가 `@PathVariable`, `@RequestParam`에 적용될 때

⚠️ 5. `@Validated`와 `@Valid` 차이

항목	`@Valid`	`@Validated`
패키지	`javax.validation.Valid`	`org.springframework.validation.annotation.Validated`
그룹 지정	❌ 불가능	✅ 가능 (`@Validated(Group1.class)`)
AOP 동작 가능	❌	✅ (예: Service에서 메서드 파라미터 검증 가능)

🛡️ 6. `@ExceptionHandler` – 예외 직접 처리

✔️ 개별 컨트롤러 내에서 처리

```
@RestController
public class UserController {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<?> handleInvalid(MethodArgumentNotValidException ex) {
        List<String> errors = ex.getBindingResult()
                                .getFieldErrors()
                                .stream()
                                .map(err -> err.getField() + ": " +
err.getDefaultMessage())
                                .toList();

        return ResponseEntity.badRequest().body(errors);
    }
}
```

🧱 전역 처리 (`@ControllerAdvice`)

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<?> handleValidationErrors(MethodArgumentNotValidException ex)
{
        List<ErrorResponse> errors = ex.getBindingResult().getFieldErrors().stream()
            .map(e -> new ErrorResponse(e.getField(), e.getDefaultMessage()))
            .toList();

        return ResponseEntity.badRequest().body(errors);
    }
}
```

📦 오류 응답 포맷 정의 예시

```
public class ErrorResponse {
    private String field;
    private String message;

    public ErrorResponse(String field, String message) {
        this.field = field;
        this.message = message;
    }
}
```

✅ 종합 정리 표

항목	설명
`@Valid`, `@Validated`	DTO의 필드에 대해 자동 검증 수행
`BindingResult`	검증 실패 시 예외 발생 없이 오류 수집 가능
`@ExceptionHandler`	특정 예외 발생 시 직접 JSON 응답 처리 가능
`@ControllerAdvice`	전역 예외 처리자 등록 가능 (통일된 API 오류 포맷 제공)
기본 예외 클래스	`MethodArgumentNotValidException`, `ConstraintViolationException`

🎾 정리 흐름

```
1. 요청 DTO → @Valid → Bean Validation (@NotNull, @Size 등)
2. 실패
  → BindingResult → 직접 처리
  → or MethodArgumentNotValidException → @ExceptionHandler
3. JSON 응답 → 클라이언트 전달
```