6. 예외 처리

전역 예외 처리: `@ControllerAdvice`

Spring Boot에서 **전역 예외 처리**를 깔끔하게 구현하려면 `@ControllerAdvice`와 `@ExceptionHandler`를 사용하는 것이 가장 정석적인 방법이다.
이 기능은 **컨트롤러 전역의 예외를 한 곳에서 처리**할 수 있게 해 주며,
REST API 설계에서 **에러 응답을 통일된 형식으로 제공**할 수 있도록 해 준다.

✅ 1. `@ControllerAdvice`란?

> **모든 컨트롤러에서 발생하는 예외를 한 곳에서 처리**하기 위한 전역 설정 클래스

- `@ExceptionHandler`와 함께 사용됨
- 특정 패키지나 클래스만 대상으로 설정도 가능
- JSON 응답을 위해 보통 `@RestControllerAdvice`를 사용함

✅ 2. 기본 구조

```
@RestControllerAdvice // @ControllerAdvice + @ResponseBody
public class GlobalExceptionHandler {

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<String> handleIllegalArgument(IllegalArgumentException ex) {
        return ResponseEntity.badRequest().body("잘못된 요청: " + ex.getMessage());
    }
}
```

→ 모든 컨트롤러에서 `IllegalArgumentException` 발생 시 해당 메서드가 실행됨

✅ 3. 예외별 분리 처리

```
@ExceptionHandler(EntityNotFoundException.class)
public ResponseEntity<String> handleNotFound(EntityNotFoundException ex) {
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body("리소스를 찾을 수 없습니다.");
}

@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<String> handleValidationError(MethodArgumentNotValidException ex)
{
    return ResponseEntity.badRequest().body("유효성 검증 실패");
}
```

✅ 4. 커스텀 에러 응답 DTO 사용

```
@Getter @AllArgsConstructor
public class ErrorResponse {
    private int status;
    private String message;
    private LocalDateTime timestamp;
}
```

```
@ExceptionHandler(MyBusinessException.class)
public ResponseEntity<ErrorResponse> handleBusinessException(MyBusinessException ex) {
    ErrorResponse error = new ErrorResponse(
        HttpStatus.BAD_REQUEST.value(),
        ex.getMessage(),
        LocalDateTime.now()
    );
    return ResponseEntity.badRequest().body(error);
}
```

✅ 5. 모든 예외 처리 (`Exception`)

```
@ExceptionHandler(Exception.class)
public ResponseEntity<ErrorResponse> handleAll(Exception ex) {
    ErrorResponse error = new ErrorResponse(500, "알 수 없는 서버 오류",
LocalDateTime.now());
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(error);
}
```

✅ 6. @ControllerAdvice 상세 옵션

속성	설명
`assignableTypes`	특정 컨트롤러 클래스 지정
`basePackages`	특정 패키지에만 적용
`annotations`	특정 애노테이션이 붙은 클래스에만 적용

```
@ControllerAdvice(basePackages = "com.example.api")
```

✅ 7. 실무 예외 처리 전략

예외	처리 방법
유효성 검증 실패	`MethodArgumentNotValidException`
존재하지 않는 리소스	`EntityNotFoundException` 또는 커스텀 예외
인증/인가 오류	`AccessDeniedException`, `AuthenticationException`
비즈니스 로직 오류	`MyBusinessException` 등 커스텀 정의
서버 내부 오류	`Exception` 또는 `RuntimeException` catch-all

✅ 8. API 응답 일관화 예시

```
{
  "status": 400,
  "message": "입력 값이 잘못되었습니다.",
  "timestamp": "2024-05-01T15:30:00"
}
```

→ 모든 예외 응답을 위와 같이 통일된 형식으로 관리하면
프론트엔드, 모바일 클라이언트, 외부 시스템과의 연동에서 **표준화된 에러 응답 처리**가 가능해짐

✅ 9. 결론

- `@ControllerAdvice`는 Spring의 **전역 예외 처리 핵심 도구**다.
- `@ExceptionHandler`와 조합하여 예외별 응답 처리 로직을 구성하면,
  애플리케이션 전반에 걸쳐 **깨끗하고 일관된 예외 처리 구조**를 설계할 수 있다.
- **REST API를 제공하는 경우에는 `@RestControllerAdvice`를 기본으로 사용**하고,
  응답 포맷을 명확하게 정의한 DTO를 반환하는 방식이 가장 바람직하다.

커스텀 예외 정의

Spring Boot에서 **커스텀 예외(Custom Exception)**를 정의하는 것은
비즈니스 로직에서 발생하는 **의미 있는 에러 상황을 명확하게 구분**하고,
전역 예외 처리(`@ControllerAdvice`)와 함께 사용하여 **응답 구조의 일관성**을 유지하는 핵심 전략이다.

✅ 1. 왜 커스텀 예외를 정의하는가?

이유	설명
명확한 책임 구분	단순한 `RuntimeException`보다 의미를 분명히 표현
예외 메시지 통제	사용자가 이해하기 쉬운 메시지 제공

이유	설명
HTTP 상태 코드 제어	상황에 따라 400, 403, 404, 500 등 분기 처리
에러 응답 포맷 통일	`ErrorResponse` 객체와 함께 응답 형식 유지 가능

✅ 2. 커스텀 예외 기본 예시

```
public class MemberNotFoundException extends RuntimeException {
    public MemberNotFoundException(String message) {
        super(message);
    }
}
```

사용 예:

```
Member member = memberRepository.findById(id)
    .orElseThrow(() -> new MemberNotFoundException("회원을 찾을 수 없습니다."));
```

✅ 3. 공통 ErrorCode Enum 설계 (권장)

```
@Getter
@AllArgsConstructor
public enum ErrorCode {
    MEMBER_NOT_FOUND(HttpStatus.NOT_FOUND, "회원을 찾을 수 없습니다."),
    INVALID_INPUT(HttpStatus.BAD_REQUEST, "입력 값이 올바르지 않습니다."),
    INTERNAL_SERVER_ERROR(HttpStatus.INTERNAL_SERVER_ERROR, "서버 오류");

    private final HttpStatus status;
    private final String message;
}
```

✅ 4. 통합 커스텀 예외 클래스

```
@Getter
public class CustomException extends RuntimeException {
    private final ErrorCode errorCode;

    public CustomException(ErrorCode code) {
        super(code.getMessage());
        this.errorCode = code;
    }
}
```

사용 예:

```
if (!user.isActive()) {
    throw new CustomException(ErrorCode.INVALID_INPUT);
}
```

✅ 5. 전역 예외 처리에 연결

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<ErrorResponse> handleCustomException(CustomException ex) {
        ErrorCode code = ex.getErrorCode();
        ErrorResponse error = new ErrorResponse(code.getStatus().value(),
code.getMessage(), LocalDateTime.now());
        return ResponseEntity.status(code.getStatus()).body(error);
    }
}
```

✅ 6. ErrorResponse DTO 예시

```
@Getter
@AllArgsConstructor
public class ErrorResponse {
    private int status;
    private String message;
    private LocalDateTime timestamp;
}
```

✅ 7. 확장 가능 구조

- `BusinessException`, `AccessDeniedException`, `TokenExpiredException` 등 계층적 예외 구성 가능
- enum `ErrorCode`에 코드 번호 추가 → 클라이언트 오류 구분 처리
- API 테스트 시 동일한 JSON 포맷으로 클라이언트 응답

```
{
  "status": 404,
  "message": "회원을 찾을 수 없습니다.",
  "timestamp": "2025-05-22T22:15:30"
}
```

✅ 8. 실무 패턴 정리

구성 요소	역할
`CustomException`	공통 예외 슈퍼클래스
`ErrorCode` enum	예외 종류/메시지/상태 코드 정의
`ErrorResponse`	표준 응답 포맷 정의
`@RestControllerAdvice`	전역 예외 처리 구현
도메인별 예외	`UserNotFoundException`, `OrderInvalidException` 등 정의 가능

✅ 결론

커스텀 예외는 단순히 `RuntimeException`을 던지는 것보다
도메인별 오류를 명확하게 구분하고,
일관된 응답 구조로 API 품질을 향상시키는 핵심적인 설계 전략이다.

> 잘 설계된 커스텀 예외 구조는 유지보수성, 디버깅, 클라이언트 개발 효율성까지 전부 향상시켜준다.

HTTP 상태 코드 매핑

Spring Boot에서 예외를 처리할 때 **적절한 HTTP 상태 코드(HttpStatus)**를 함께 매핑하는 것은 매우 중요합니다.
이것은 클라이언트(브라우저, 앱, API 소비자)에게 **오류의 종류를 명확하게 전달**하고,
RESTful API의 **표준을 지키는 핵심 요소**입니다.

아래는 자주 사용하는 **HTTP 상태 코드와 그에 대응하는 Spring 예외/커스텀 예외 매핑 전략**을 정리한 내용입니다.

✅ 1. HTTP 상태 코드 개요

상태 코드	의미	설명
`200 OK`	성공	일반적인 요청 성공 응답
`201 Created`	생성됨	POST 요청으로 자원이 생성됨
`204 No Content`	내용 없음	응답 본문이 없을 때
`400 Bad Request`	잘못된 요청	파라미터 오류, 형식 오류 등
`401 Unauthorized`	인증 실패	로그인 필요 또는 토큰 오류
`403 Forbidden`	인가 실패	권한 없음 (로그인은 했으나 거부됨)
`404 Not Found`	리소스 없음	대상 엔티티 없음
`405 Method Not Allowed`	메서드 허용 안됨	POST만 가능한데 GET으로 요청한 경우
`409 Conflict`	충돌	중복 데이터, 무결성 위반 등

상태 코드	의미	설명
`422 Unprocessable Entity`	처리 불가	형식은 맞지만 의미상 오류
`500 Internal Server Error`	서버 오류	서버 내부 로직 실패

✅ 2. 전형적인 매핑 예시

예외 상황	상태 코드	설명
필수 파라미터 누락, 타입 불일치	`400`	`MethodArgumentNotValidException`, `IllegalArgumentException`
로그인 정보 없음/토큰 없음	`401`	`AuthenticationException`, 커스텀 `UnauthenticatedException`
권한 없음 (접근 차단)	`403`	`AccessDeniedException`, 커스텀 `ForbiddenException`
데이터 없음	`404`	`EntityNotFoundException`, `MemberNotFoundException`
중복 이메일, ID	`409`	`DuplicateMemberException`, `ConflictException`
DB 예외, NullPointerException	`500`	일반 `Exception` 또는 `RuntimeException`

✅ 3. 예외와 상태 코드 매핑 실전 구현

ErrorCode enum

```java
@Getter
@AllArgsConstructor
public enum ErrorCode {
    INVALID_INPUT(HttpStatus.BAD_REQUEST, "잘못된 입력입니다."),
    UNAUTHORIZED(HttpStatus.UNAUTHORIZED, "인증이 필요합니다."),
    FORBIDDEN(HttpStatus.FORBIDDEN, "접근 권한이 없습니다."),
    NOT_FOUND(HttpStatus.NOT_FOUND, "리소스를 찾을 수 없습니다."),
    CONFLICT(HttpStatus.CONFLICT, "중복된 요청입니다."),
    INTERNAL_ERROR(HttpStatus.INTERNAL_SERVER_ERROR, "서버 오류입니다.");

    private final HttpStatus status;
    private final String message;
}
```

CustomException 예시

```
@Getter
public class CustomException extends RuntimeException {
    private final ErrorCode errorCode;

    public CustomException(ErrorCode errorCode) {
        super(errorCode.getMessage());
        this.errorCode = errorCode;
    }
}
```

전역 처리: @RestControllerAdvice

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<ErrorResponse> handleCustom(CustomException ex) {
        ErrorCode code = ex.getErrorCode();
        ErrorResponse body = new ErrorResponse(code.getStatus().value(),
code.getMessage(), LocalDateTime.now());
        return ResponseEntity.status(code.getStatus()).body(body);
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ErrorResponse>
handleValidation(MethodArgumentNotValidException ex) {
        return ResponseEntity.badRequest().body(new ErrorResponse(400, "입력값이 유효하지
않습니다", LocalDateTime.now()));
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleAll(Exception ex) {
        return ResponseEntity.internalServerError().body(new ErrorResponse(500, "서버 오
류", LocalDateTime.now()));
    }
}
```

✅ 4. 응답 포맷 예시

```
{
  "status": 404,
  "message": "리소스를 찾을 수 없습니다.",
  "timestamp": "2025-05-22T23:30:00"
}
```

✅ 5. 실무 팁

팁	설명
HTTP 상태 코드는 명확하게 구분하자	400 vs 403 vs 404 혼동 금지
예외마다 코드와 메시지를 Enum으로 정리	일관성 유지 및 유지보수 용이
클라이언트와 응답 포맷 사전 협의	status, message, code, detail 등을 표준화
`@Valid` + `MethodArgumentNotValidException` 조합 활용	DTO 유효성 검증 후 400 응답

✅ 결론

- 예외 상황에 맞는 **HTTP 상태 코드 매핑**은 API 품질의 핵심이다.
- `@ControllerAdvice + @ExceptionHandler + ErrorCode enum + DTO 응답` 구조로
  모든 예외를 통일된 방식으로 관리하는 것이 실무적으로 가장 안정적이고 확장 가능성이 높다.

Validation & BindingResult 처리

Spring Boot에서는 요청 데이터의 유효성을 검증하기 위해 `@Valid` 또는 `@Validated`를 사용하고,
검증 실패 시 오류 정보를 `BindingResult` 또는 `MethodArgumentNotValidException`으로 받아 처리할 수 있습니다.
이는 **폼 검증, API 파라미터 검증, DTO 유효성 검사 등**에 모두 적용되는 **핵심 기술**입니다.

✅ 1. 유효성 검증을 위한 주요 애노테이션

Spring은 **JSR-380 (Bean Validation 2.0, Hibernate Validator)**을 사용하여 다음 애노테이션을 지원합니다.

애노테이션	설명
`@NotNull`	값이 `null`이면 에러
`@NotBlank`	공백/빈 문자열/`null` 모두 에러
`@NotEmpty`	빈 문자열/컬렉션 불가 (null 허용)
`@Size(min, max)`	길이 또는 컬렉션 크기 제한
`@Email`	이메일 형식 검증
`@Pattern(regexp)`	정규식 검증
`@Min`, `@Max`	숫자 범위 검증
`@Positive`, `@Negative`	양수/음수 검증
`@AssertTrue`, `@AssertFalse`	불리언 조건 검증

✅ 2. DTO에 유효성 검증 애노테이션 적용

```
@Getter
public class MemberRequest {

    @NotBlank(message = "이름은 필수입니다.")
    private String name;

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

    @Min(value = 18, message = "나이는 18세 이상이어야 합니다.")
    private int age;
}
```

✅ 3. `@Valid` + `BindingResult` 사용

```
@PostMapping("/members")
public ResponseEntity<?> create(@Valid @RequestBody MemberRequest dto, BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        // 모든 필드 에러 출력
        List<String> errors = bindingResult.getFieldErrors().stream()
            .map(err -> err.getField() + ": " + err.getDefaultMessage())
            .toList();

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

    // 검증 성공 로직
    return ResponseEntity.ok("등록 완료");
}
```

특징	설명
`@Valid`	DTO의 유효성 검증 수행
`BindingResult`	오류가 존재하면 여기에 바인딩됨
→ `Exception` 발생 없이 오류 처리 가능	(자동 예외 발생 X)

✅ 4. 자동 예외 처리: `@ExceptionHandler`

`BindingResult`를 생략하면, Spring은 자동으로 예외를 던집니다:
→ `MethodArgumentNotValidException` (for `@RequestBody`)
→ `BindException` (for `@ModelAttribute`)

전역 예외 처리 예시:

```java
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<ErrorResponse> handleValidation(MethodArgumentNotValidException ex) {
    List<String> errors = ex.getBindingResult().getFieldErrors().stream()
        .map(e -> e.getField() + ": " + e.getDefaultMessage())
        .toList();

    return ResponseEntity.badRequest()
        .body(new ErrorResponse(400, "검증 실패", LocalDateTime.now(), errors));
}
```

✅ 5. 응답 DTO 포맷 예시

```json
{
  "status": 400,
  "message": "검증 실패",
  "timestamp": "2025-05-22T23:55:00",
  "errors": [
    "name: 이름은 필수입니다.",
    "email: 이메일 형식이 올바르지 않습니다."
  ]
}
```

✅ 6. `@Validated` vs `@Valid`

항목	`@Valid`	`@Validated`
표준	JSR-380 (javax)	Spring (org.springframework)
그룹 검증	❌	✅
일반 사용	DTO 검증	그룹 조건, 복합 조건
위치	컨트롤러, 서비스	컨트롤러, 서비스, AOP 레벨 가능

✅ 7. 실무 전략 정리

항목	권장 방법
API DTO 검증	`@Valid` + `BindingResult`
전역 실패 처리	`@ExceptionHandler(MethodArgumentNotValidException.class)`
복합 조건/그룹 검증	`@Validated + groups`
테스트에서 검증	`Validator.validate()` 활용

✅ 결론

- Spring에서의 `@Valid` / `BindingResult` 조합은 **API 요청에 대한 안전한 입력 필터 역할**을 한다.
- 전역 예외 처리와 연동하여 **일관된 에러 응답**을 제공하고,
  클라이언트와의 명확한 API 계약을 보장하는 핵심 기반이다.

Bean Validation: `@valid`, `@validated`

Spring에서는 **Bean Validation(JSR 380)** 기반으로 요청 객체의 유효성을 검증할 수 있으며,
이때 사용하는 대표적인 애노테이션이 바로 `@valid`와 `@validated`다.
두 애노테이션은 비슷해 보이지만, **용도와 기능 범위에 분명한 차이점**이 있다.

✅ 1. `@valid`란?

- **javax.validation.constraints**에서 제공하는 표준 JSR 380 애노테이션
- 기본적인 Bean Validation만 수행 (그룹 기능은 없음)
- 컨트롤러, 서비스, 내부 메서드 파라미터 등에서 사용 가능

```
@PostMapping("/members")
public ResponseEntity<?> create(@Valid @RequestBody MemberRequest dto) {
    // DTO의 필드 유효성 검증
}
```

✅ 2. `@validated`란?

- **Spring 전용 애노테이션 (org.springframework.validation.annotation.Validated)**
- `@valid` 기능 포함 + **그룹(Group) 조건 검증**을 지원
- 유효성 검증을 보다 **정교하게 제어**하고자 할 때 사용

```
@PostMapping("/members")
public ResponseEntity<?> create(@Validated(Create.class) @RequestBody MemberRequest dto)
{
    // Create 그룹에만 해당하는 필드 검증 실행
}
```

✅ 3. 주요 차이점 비교

항목	`@valid`	`@validated`
출처	Java 표준 (JSR-380)	Spring 자체 애노테이션
기본 기능	Bean Validation	Bean Validation
그룹 검증 지원	❌ 불가	✅ 가능 (`@Validated(Group.class)`)

항목	`@Valid`	`@Validated`
적용 대상	컨트롤러, 서비스	컨트롤러, 서비스, AOP 등
커스텀 Validator	지원됨	지원됨

✅ 4. 그룹 검증 사용 예시 (`@Validated` 전용 기능)

1) 그룹 정의

```
public interface Create {}
public interface Update {}
```

---

2) DTO 정의

```
@Getter
public class MemberRequest {

    @NotBlank(groups = Create.class)
    private String name;

    @Email(groups = {Create.class, Update.class})
    private String email;

    @Min(value = 18, groups = Create.class)
    private int age;
}
```

---

3) 컨트롤러 적용

```
@PostMapping("/members")
public ResponseEntity<?> create(@Validated(Create.class) @RequestBody MemberRequest dto)
{
    // Create 조건만 적용됨
}

@PutMapping("/members/{id}")
public ResponseEntity<?> update(@Validated(Update.class) @RequestBody MemberRequest dto)
{
    // Update 조건만 적용됨
}
```

✅ 5. 메서드 수준 유효성 검증

서비스, 컴포넌트에서도 검증 적용 가능:

```
@Validated
@Service
public class MemberService {

    public void save(@Valid MemberRequest dto) {
        // 내부에서도 검증 동작
    }
}
```

> 서비스 메서드에서 `@Validated`는 **클래스에 붙여야 작동**한다.
> (프록시 기반 AOP이므로)

✅ 6. 중첩 객체 유효성 검증

```
public class OrderRequest {
    @Valid
    private MemberRequest member; // 내부 DTO도 검증
}
```

✅ 7. BindingResult 병행 사용 가능

```
public ResponseEntity<?> create(@Valid MemberRequest dto, BindingResult result) {
    if (result.hasErrors()) {
        // 수동 처리
    }
}
```

✅ 8. 결론

항목	설명
`@Valid`	가장 기본적인 유효성 검증 (JSR 380)
`@Validated`	그룹 기능 포함 + Spring 내부까지 확장 가능
실무 권장	**단순 검증은** `@Valid`, 그룹/서비스 레이어 확장 시 `@Validated`