Spring Boot Global Exception Handling - Implementation Guide

# 1. Introduction

This guide explains how to implement a centralized and structured exception handling mechanism in a Spring Boot application using a Global Exception Handler. It is designed to help developers maintain consistency, improve debugging, and return clear error messages to API consumers.

# 2. Why Use Global Exception Handling?

- Centralizes error handling logic  
- Reduces repetitive code across controllers  
- Provides a uniform error response structure  
- Makes debugging easier with traceable logs and error codes  
- Enhances maintainability and scalability

# 3. Project Structure Overview

Here's a simplified view of the classes involved:

- GlobalExceptionHandler.java → Captures all exceptions thrown globally.  
- ExceptionTestController.java → Provides endpoints to trigger different exceptions.  
- ApiCustomException.java → Custom exception class.  
- ErrorMessage.java → Standard error response model.  
- ValidationDetails.java → Contains field-level validation info.  
- ErrorConstants.java → Central error codes and constants.  
- ErrorHandlingUtil.java → Utility for extracting error info.

# 4. Sample Exception Test Controller

This controller has multiple endpoints to simulate and test various exception scenarios:

@RestController  
@RequestMapping("/api/test")  
public class ExceptionTestController {  
  
 @GetMapping("/generic")  
 public String throwGenericException() {  
 throw new RuntimeException("Generic internal error occurred.");  
 }  
  
 @PostMapping("/method-not-allowed")  
 public String methodNotAllowed() {  
 return "Use POST only";  
 }  
  
 @PostMapping(value = "/media-type", consumes = MediaType.APPLICATION\_XML\_VALUE)  
 public String mediaTypeNotSupported() {  
 return "Consumes only XML";  
 }  
  
 @PostMapping("/validation")  
 public String validateInput(@Valid @RequestBody TestRequest request) {  
 return "Validation passed";  
 }  
  
 @GetMapping("/not-found/{id}")  
 public String recordNotFound(@PathVariable String id) {  
 if (!"123".equals(id)) {  
 ApiCustomException ex = new ApiCustomException("No record found");  
 ex.code("ERROR\_MAPPING\_NOT\_FOUND")  
 .httpStatus(HttpStatus.NOT\_FOUND);  
 throw ex;  
 }  
 return "Record found";  
 }  
}

# 5. Global Exception Handler

This class uses @ControllerAdvice and @ExceptionHandler to intercept various exceptions.

@ControllerAdvice  
public class GlobalExceptionHandler {  
  
 @ExceptionHandler(HttpRequestMethodNotSupportedException.class)  
 public ResponseEntity<ErrorMessage> handleMethodTypeException(...) {  
 ...  
 return buildErrorResponse(HttpStatus.METHOD\_NOT\_ALLOWED, ...);  
 }  
  
 @ExceptionHandler(MethodArgumentNotValidException.class)  
 public ResponseEntity<ErrorMessage> handleMethodArgumentNotValidException(...) {  
 ...  
 }  
  
 @ExceptionHandler(ApiCustomException.class)  
 public ResponseEntity<ErrorMessage> handleApiCustomException(ApiCustomException ex) {  
 ...  
 }  
  
 @ExceptionHandler(Exception.class)  
 public ResponseEntity<ErrorMessage> handleAnyException(Exception ex) {  
 ...  
 }  
}

# 6. Custom Error Response Format

Here’s what a typical error response might look like:

{  
 "refId": "a1234-b5678-c9012",  
 "code": "ERROR\_MAPPING\_NOT\_FOUND",  
 "message": "No record found for given ID",  
 "severity": "HIGH",  
 "status": 404,  
 "errorDetails": [  
 {  
 "field": "recordId",  
 "message": "Record not found for ID: 101"  
 }  
 ]  
}

# 7. How to Test Each Scenario

- 500 Error: GET /api/test/generic  
- 405 Error: Use GET on /api/test/method-not-allowed  
- 415 Error: Send JSON to /api/test/media-type expecting XML  
- 400 Error: POST invalid data to /api/test/validation  
- 404 Error: GET /api/test/not-found/999  
- 428 Precondition Failed: Send missing/invalid header to /api/test/precondition-failed  
- 412 Duplicate Check: POST known email to /api/test/duplicate-check

# 8. Logging Format

The logging pattern includes traceId, errorCode, file name, and line number.

{  
 "timestamp": "2025-07-23T14:55:30.231",  
 "level": "ERROR",  
 "traceId": "abc-123-xyz",  
 "project": "MyProject",  
 "class": "GlobalExceptionHandler",  
 "location": "GlobalExceptionHandler.java:42",  
 "errorCode": "ERROR\_MAPPING\_NOT\_FOUND",  
 "message": "No record found for the given ID"  
}

# 9. Conclusion

With this approach, your Spring Boot application can handle all exceptions consistently and provide helpful messages for API consumers while logging meaningful error data for developers. You can now easily trace issues using error codes, trace IDs, and structured responses.