Exception Handling in Spring Boot

Introduction

Exception handling is crucial for building robust applications, as it helps in managing errors gracefully and providing meaningful feedback to users.

We'll cover:
• Global exception handling
• Custom exception handling
• Returning custom error responses
• Best practices

Global and Custom Exception Handling

In this example, we handle two types of exceptions: **ResourceNotFoundException** and generic **Exception**.

The **@ExceptionHandler** annotation specifies which exception to handle, and the method returns a custom response with error details.

Annotations Explained

@ExceptionHandler
• Tells Spring Boot which exception a method should handle.
• Used inside @ControllerAdvice or directly in controllers.
Example:

```
@ExceptionHandler(ResourceNotFoundException.class)
public ResponseEntity<?> handleResourceNotFound(ResourceNotFoundException
ex) { ... }
```

@ControllerAdvice
• A specialized annotation used to handle exceptions globally across the application.
• Helps centralize error-handling logic in one place.
Example:

```
@ControllerAdvice
public class GlobalExceptionHandler { ... }
```

ErrorDetails Class

Custom exception handling provides a way to represent specific error conditions in your application.

For example, **ResourceNotFoundException** extends RuntimeException and is used to indicate

that a requested resource was not found.

Annotations Explained (ErrorDetails)

@ResponseStatus
• Maps an exception to a specific HTTP status code.
• Useful for directly associating exceptions with responses.
Example:

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException { ... }
```

Returning Custom Error Responses

Spring Boot allows you to customize the error responses returned by your application.

Example: We define a custom error response class **CustomErrorResponse** and use it in our global exception handler to provide more detailed error information.

Annotations Explained (Custom Error Responses)

@ResponseBody
• Ensures that the return value of a method is serialized into JSON (or XML) and sent back as the HTTP response body.
Example:

```
@ResponseBody
public CustomErrorResponse handleException(Exception ex) { ... }
```

How the Flow Works When an Exception Occurs

High-level sequence (happy → error path)

1. **Request enters DispatcherServlet**
Spring MVC's front controller receives the HTTP request.

2. **Handler mapping & controller method**
The request is routed to a controller method (via @RequestMapping/@GetMapping/etc.). Inside your controller → service → repository call chain, something goes wrong and an exception is thrown.

3. **Exception bubbles up**
The exception propagates back up the call stack to the DispatcherServlet. Spring does not crash—instead it invokes the exception resolution chain.

4. **Exception is resolved by the first matching resolver**
Spring iterates through its configured HandlerExceptionResolvers in order:

• @ExceptionHandler in the same controller (method-level)
• @ControllerAdvice global @ExceptionHandler methods
• Framework resolvers (e.g., DefaultHandlerExceptionResolver)
• Fallback: BasicErrorController (Spring Boot's error endpoint)

5. **A ResponseEntity or body + status is produced**
Your exception handler returns a ResponseEntity<CustomErrorResponse> (or a body + @ResponseStatus).
Spring picks an HttpMessageConverter (usually Jackson) to serialize your error object to JSON.

6. **Response written to client**
The DispatcherServlet writes status code + headers + JSON body to the HTTP response.
Logging happens based on your code and logging config.

Best Practices for Error Handling

1. **Use Meaningful Exception Names:** Custom exceptions should have names that clearly indicate the error condition.
2. **Provide Clear Error Messages:** Error messages should be user-friendly and provide enough information to understand the error.
3. **Log Exceptions:** Always log exceptions, especially unexpected ones, to help with debugging and monitoring.
4. **Avoid Revealing Sensitive Information:** Do not include sensitive information in error messages or responses.
5. **Use HTTP Status Codes Appropriately:** Use appropriate HTTP status codes to indicate the type of error (e.g., 404 for not found, 500 for server errors).
6. **Centralize Exception Handling:** Use @ControllerAdvice to centralize exception handling logic.