Class-27: Exception Handling in Spring Boot

by Pial Kanti Samadder

What are Exceptions

- In Java, an exception is an event that **occurs during the execution of a program that disrupts the normal flow of instructions.**
- Exceptions are used to handle errors, unexpected conditions, or other exceptional events that may occur during runtime.
- Java provides a mechanism for handling exceptions through the use of try-catch blocks.

Exception Handling in Spring Boot

Exception handling in Spring Boot helps deal with errors and exceptions present in APIs, delivering a robust enterprise application. It provides a comprehensive framework for handling exceptions, including both built-in and custom exceptions.

Here are some key approaches to exception handling in Spring Boot:

- Default exception handling by Spring Boot
- Using `@ExceptionHandler` annotation
- Using **@ControllerAdvice** for global exception handling

Default Error Handling in Spring Boot

The default error-handling mechanism in **Spring Boot** is designed to provide a standard response for unhandled exceptions in web applications, including REST APIs and web-based user interfaces.

When an unhandled exception occurs in a Spring Boot application, the framework uses its **default error-handling mechanism**, which relies on:

Error Controller:

- The built-in `BasicErrorController` handles error responses.
- This controller processes errors and returns an appropriate response, either in JSON (for REST APIs) or as an HTML error page (for web applications).

Error Attributes:

- The `DefaultErrorAttributes` class collects error details and makes them available for rendering in the response.

Error Pages:

- By default, Spring Boot serves an HTML error page if the client accepts `text/html`.
- For REST APIs, a JSON response is returned if the client accepts `application/json`.

Default JSON Error Response

For REST APIs, the default error response typically includes the following fields:

```
{
  "timestamp": "2025-11-05T17:35:47.354+00:00",
  "status": 500,
  "error": "Internal Server Error",
  "path": "/users"
}
```

Fields Explained:

- **timestamp**: When the error occurred.
- **status**: HTTP status code (e.g., 404 for Not Found, 500 for Internal Server Error).
- **error**: A brief error description.
- **path**: The URL path where the error occurred.

@ExceptionHandler

The `@ExceptionHandler` annotation is used inside a **Spring REST controller** to handle **specific exceptions** and return a **custom response**.

- Prevents app crashes by handling exceptions
- Returns meaningful error messages instead of default stack traces
- Can be applied to a single controller or globally

Handling a Specific Exception using `@ExceptionHandler`

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

    @GetMapping("/user/{id}")
    public String getUser(@PathVariable int id) {
        if (id <= 0) {
            throw new IllegalArgumentException("Invalid user ID");
        }
        return "User details for ID: " + id;
    }

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<String> handleInvalidIdException(IllegalArgumentException ex) {
        return new ResponseEntity<>("Error: " + ex.getMessage(), HttpStatus.BAD_REQUEST);
    }
}
```

If the user sends `/api/user/-1`, the exception handler returns:

`Error: Invalid user ID` (HTTP 400 Bad Request) instead of a default stack trace.

Handling Multiple Exceptions using `@ExceptionHandler`

```
@ExceptionHandler({IllegalArgumentException.class, NullPointerException.class})
public ResponseEntity<String> handleMultipleExceptions(Exception ex) {
    return new ResponseEntity<>("Exception occurred: " + ex.getMessage(), HttpStatus.BAD_REQUEST);
}
```

This method will handle both `IllegalArgumentException` **and** `NullPointerException`**.**

Global Exception Handling with `@ControllerAdvice`

Instead of handling exceptions in **each controller**, you can define a **global exception handler** using `@ControllerAdvice`. It allows you to define global error handling logic that can be applied to multiple controllers in your application.

Creating a Global Exception Handler

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(UserNotFoundException.class)
    public ResponseEntity<String> handleUserNotFoundException(UserNotFoundException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleGenericException(Exception ex) {
        return new ResponseEntity<>("Something went wrong: " + ex.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Now, all controllers in the application will use this exception handler.

`@ControllerAdvice` vs `@RestControllerAdvice`

@ControllerAdvice

- Used for **global exception handling** across multiple controllers.
- Typically used in **Spring MVC (HTML-based)** applications.
- Methods inside the class usually need `@ResponseBody` to return JSON.

Example:

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(Exception.class)
    @ResponseBody // required to return JSON
    public String handleException(Exception ex) {
        return ex.getMessage();
    }
}
```

@RestControllerAdvice

- A specialization of `@ControllerAdvice` for **REST APIs**.
- Automatically adds `@ResponseBody` to all methods.
- No need to manually annotate each handler method to return JSON.

Example:

```
@RestControllerAdvice
public class GlobalRestExceptionHandler {

    @ExceptionHandler(Exception.class)
    public String handleException(Exception ex) {
        return ex.getMessage();
    }
}
```

ProblemDetails

`PoblemDetails` is not just another Spring Boot feature — **it is an IETF standard (RFC 9457) that defines a "problem detail" as a way to carry machine-readable details error in a HTTP response**.

Since Spring Boot 3.0 the `ProblemDetails` are natively supported, making it easier than ever to implement this standard in your applications.

Why should you care?

- Better client — side error handling
- Improved API documentation
- Consistency across your services

Traditional Error Handling Approaches

Before `ProblemDetail`, we often implemented **custom exception handlers and response entities** to handle errors in Spring Boot. We'd create custom error response structures. That resulted in inconsistencies across different APIs.

Also, this approach required a lot of boilerplate code. Moreover, it lacked a standardized way to represent errors, making it difficult for clients to parse and understand error messages uniformly.

`ProblemDetail` Specification

The `ProblemDetail` specification is part of the [RFC 7807 standard](). **It defines a consistent structure for error responses:**

- `type` → URI identifying the problem type
- `title` → Short, human-readable summary
- `status` → HTTP status code
- `detail` → Detailed error message
- `instance` → URI of the specific occurrence

Enabling `ProblemDetail` in Spring Boot

First, we can add a property to enable it. For RESTful service, we add the following property to *application.properties*:

```
spring.mvc.problemdetails.enabled=true
```

- Automatically formats error responses using `ProblemDetail`
- Can be turned off if not needed

Example `ProblemDetail` JSON Response

```
{
   "type": "about:blank",
   "title": "Bad Request",
   "status": 400,
   "detail": "Invalid request content.",
   "instance": "/sales/calculate"
}
```

- **type** → identifies the error type
- **title** → brief description
- **status** → HTTP status code
- **detail** → specific explanation
- **instance** → URI of the failed request

Implementing `ProblemDetail` in Exception Handler

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(EntityNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public ProblemDetail handleEntityNotFoundException(EntityNotFoundException ex, HttpServletRequest request) {
        // Create a ProblemDetail object using Spring 6+ API
        ProblemDetail problem = ProblemDetail.forStatusAndDetail(HttpStatus.NOT_FOUND, ex.getMessage());
        problem.setType(URI.create("https://example.com/problem/entity-not-found"));
        problem.setTitle("Entity Not Found");
        problem.setInstance(URI.create(request.getRequestURI()));
        return problem;
    }
}
```

Spring Boot Exception Handling — Best Practices

1. **Use Specific Exceptions**
   - Avoid generic `Exception` / `RuntimeException`; define custom exceptions.
2. **Global Handling**
   - Use `@ControllerAdvice` / `@RestControllerAdvice` for centralized handling.
3. **Meaningful Error Message**s
   - Clear, informative, user-friendly
   - Never expose stack traces in production
4. **HTTP Status Codes**
   - Use `@ResponseStatus` on custom exceptions
   - Ensures clean, consistent HTTP responses

Running Sample Project

- **Clone the Repository**

If you haven't cloned the repository yet, run the following command (ensure `git` is installed):

```
git clone https://github.com/PialKanti/Ostad-SpringBoot-Course.git
```

Then switch to the correct branch for today's class (replace with the actual branch name, e.g., `class-24-jpa-relationship`):

```
git fetch
git switch class-27-exception-handling
```

Or,

If You Already Have the Repository Cloned, simply open your existing project folder and switch (or checkout) to the appropriate branch:

```
git fetch
git switch class-27-exception-handling
```

- **Set Up and Run PostgreSQL Database**

You can run PostgreSQL **either via Docker** or a **desktop installation**.

Option 1: Run via Docker

A `compose.yml` file is available in the root of the repository.
Run the following command from the project root:

```
docker compose up -d
```

This will start a PostgreSQL container automatically.

Or,

Option 2: Run via PostgreSQL Desktop (Manual Setup)

If you already have PostgreSQL installed locally:

1. Start your PostgreSQL server.
2. Create a new database named `crud_db` if not exists.

- **Open the Project in IntelliJ IDEA**

1. Open IntelliJ IDEA.
2. Click **File → Open** and select the `crud-sample` folder inside the repository.
3. Let IntelliJ import Maven/Gradle dependencies automatically.