### Exception Handling in Spring Boot (using `@ControllerAdvice`)

Exception handling in Spring Boot can be done using the `@ControllerAdvice` annotation, which allows you to handle exceptions globally across your application. It’s a way to separate error-handling logic from your core business logic. This leads to cleaner, more maintainable code.

---

### Key Concepts in Exception Handling with Spring Boot

- \*\*`@ControllerAdvice`\*\*: This annotation is used to create a global error-handling mechanism that applies to all controllers. It works across all controllers in your Spring Boot application.

- \*\*`@ExceptionHandler`\*\*: This annotation is used inside a class annotated with `@ControllerAdvice` (or directly in a controller) to handle specific exceptions and define custom responses.

- \*\*`ResponseEntityExceptionHandler`\*\*: This is a convenient base class provided by Spring Boot that you can extend to override and customize default exception-handling behavior.

---

### How `@ControllerAdvice` Works

`@ControllerAdvice` can intercept exceptions thrown by controllers and return appropriate HTTP responses. This is especially useful when you need to provide consistent error messages across your application.

#### Example of a Global Exception Handler using `@ControllerAdvice`:

```java

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<ErrorResponse> handleResourceNotFound(ResourceNotFoundException ex) {

ErrorResponse error = new ErrorResponse("NOT\_FOUND", ex.getMessage());

return new ResponseEntity<>(error, HttpStatus.NOT\_FOUND);

}

@ExceptionHandler(Exception.class)

public ResponseEntity<ErrorResponse> handleGlobalException(Exception ex) {

ErrorResponse error = new ErrorResponse("INTERNAL\_SERVER\_ERROR", "Something went wrong!");

return new ResponseEntity<>(error, HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

class ErrorResponse {

private String errorCode;

private String errorMessage;

// Constructors, getters, and setters

public ErrorResponse(String errorCode, String errorMessage) {

this.errorCode = errorCode;

this.errorMessage = errorMessage;

}

// Getters and Setters

}

```

In this example:

- The `GlobalExceptionHandler` class is annotated with `@ControllerAdvice`, making it a global exception handler.

- \*\*`@ExceptionHandler(ResourceNotFoundException.class)`\*\* is used to handle custom exceptions, in this case, when a resource is not found.

- The method `handleGlobalException` is a fallback handler for any other uncaught exceptions, returning a generic error message and a `500 Internal Server Error` status.

---

### Custom Exception Class Example

You can create custom exception classes to represent specific errors in your application. For example, a `ResourceNotFoundException` can be thrown when a resource is missing.

#### Example:

```java

public class ResourceNotFoundException extends RuntimeException {

public ResourceNotFoundException(String message) {

super(message);

}

}

```

Then, you can throw this exception from your controllers when needed:

```java

@GetMapping("/users/{id}")

public User getUserById(@PathVariable String id) {

return userRepository.findById(id)

.orElseThrow(() -> new ResourceNotFoundException("User not found with ID: " + id));

}

```

If the `id` is not found in the database, the `ResourceNotFoundException` is thrown, and it will be handled by the global exception handler in `GlobalExceptionHandler`.

---

### Handling Multiple Exceptions

You can handle multiple exceptions by defining multiple `@ExceptionHandler` methods inside the same `@ControllerAdvice` class.

#### Example:

```java

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<ErrorResponse> handleResourceNotFound(ResourceNotFoundException ex) {

ErrorResponse error = new ErrorResponse("NOT\_FOUND", ex.getMessage());

return new ResponseEntity<>(error, HttpStatus.NOT\_FOUND);

}

@ExceptionHandler(InvalidInputException.class)

public ResponseEntity<ErrorResponse> handleInvalidInput(InvalidInputException ex) {

ErrorResponse error = new ErrorResponse("INVALID\_INPUT", ex.getMessage());

return new ResponseEntity<>(error, HttpStatus.BAD\_REQUEST);

}

@ExceptionHandler(Exception.class)

public ResponseEntity<ErrorResponse> handleGlobalException(Exception ex) {

ErrorResponse error = new ErrorResponse("INTERNAL\_SERVER\_ERROR", "Something went wrong!");

return new ResponseEntity<>(error, HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

```

- `ResourceNotFoundException` returns a `404 Not Found` response.

- `InvalidInputException` returns a `400 Bad Request` response.

- `Exception` catches all other exceptions and returns a `500 Internal Server Error`.

---

### Returning Custom Error Responses

The error responses returned can be customized by creating a specific response structure (e.g., `ErrorResponse`) and setting the desired fields like error code, message, timestamp, etc.

#### Example:

```java

public class ErrorResponse {

private String errorCode;

private String errorMessage;

private LocalDateTime timestamp;

public ErrorResponse(String errorCode, String errorMessage) {

this.errorCode = errorCode;

this.errorMessage = errorMessage;

this.timestamp = LocalDateTime.now();

}

// Getters and Setters

}

```

This can then be used to return more detailed error messages, including timestamps, in the exception handler:

```java

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<ErrorResponse> handleResourceNotFound(ResourceNotFoundException ex) {

ErrorResponse error = new ErrorResponse("NOT\_FOUND", ex.getMessage());

return new ResponseEntity<>(error, HttpStatus.NOT\_FOUND);

}

```

---

### Example: Full REST API with Exception Handling

Here’s a complete example that shows how to handle exceptions in a typical REST API:

#### 1. \*\*User Controller\*\*:

```java

@RestController

@RequestMapping("/api/users")

public class UserController {

private Map<String, User> users = new HashMap<>();

@GetMapping("/{id}")

public ResponseEntity<User> getUserById(@PathVariable String id) {

User user = users.get(id);

if (user == null) {

throw new ResourceNotFoundException("User not found with ID: " + id);

}

return ResponseEntity.ok(user);

}

@PostMapping

public ResponseEntity<User> createUser(@RequestBody User user) {

if (user.getName() == null || user.getName().isEmpty()) {

throw new InvalidInputException("User name is required");

}

users.put(user.getId(), user);

return ResponseEntity.status(HttpStatus.CREATED).body(user);

}

}

```

#### 2. \*\*Global Exception Handler\*\*:

```java

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<ErrorResponse> handleResourceNotFound(ResourceNotFoundException ex) {

ErrorResponse error = new ErrorResponse("NOT\_FOUND", ex.getMessage());

return new ResponseEntity<>(error, HttpStatus.NOT\_FOUND);

}

@ExceptionHandler(InvalidInputException.class)

public ResponseEntity<ErrorResponse> handleInvalidInput(InvalidInputException ex) {

ErrorResponse error = new ErrorResponse("INVALID\_INPUT", ex.getMessage());

return new ResponseEntity<>(error, HttpStatus.BAD\_REQUEST);

}

@ExceptionHandler(Exception.class)

public ResponseEntity<ErrorResponse> handleGlobalException(Exception ex) {

ErrorResponse error = new ErrorResponse("INTERNAL\_SERVER\_ERROR", "An unexpected error occurred");

return new ResponseEntity<>(error, HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

```

#### 3. \*\*Custom Exceptions\*\*:

```java

public class ResourceNotFoundException extends RuntimeException {

public ResourceNotFoundException(String message) {

super(message);

}

}

public class InvalidInputException extends RuntimeException {

public InvalidInputException(String message) {

super(message);

}

}

```

#### 4. \*\*ErrorResponse Class\*\*:

```java

public class ErrorResponse {

private String errorCode;

private String errorMessage;

private LocalDateTime timestamp;

public ErrorResponse(String errorCode, String errorMessage) {

this.errorCode = errorCode;

this.errorMessage = errorMessage;

this.timestamp = LocalDateTime.now();

}

// Getters and Setters

}

```

### How This Works:

1. \*\*`GET /api/users/{id}`\*\*: If the user with the given ID is not found, the `ResourceNotFoundException` is thrown, and the `GlobalExceptionHandler` handles it.

2. \*\*`POST /api/users`\*\*: If the user input is invalid, an `InvalidInputException` is thrown, which is handled by the `GlobalExceptionHandler` to return a `400 Bad Request`.

3. \*\*Other exceptions\*\*: If any other unhandled exception occurs, the global handler will return a `500 Internal Server Error` with a generic message.

---

### Conclusion

Exception handling in Spring Boot is clean and powerful when using `@ControllerAdvice` and `@ExceptionHandler`. It allows you to manage exceptions globally, ensuring consistent error messages and reducing boilerplate code in your controllers. By customizing error responses, you can provide detailed and user-friendly error messages to clients of your API.