**Exception Handling with Controlleradvice**

controlleradvice for exceptions in spring boot with example

###### **ChatGPT said:**

In **Spring Boot**, @ControllerAdvice is used for **global exception handling**. It allows you to centralize exception handling logic instead of writing try-catch blocks in every controller.

## **Example of @ControllerAdvice for Exception Handling**

### **Step 1: Create a Custom Exception**

java

CopyEdit

public class ResourceNotFoundException extends RuntimeException {

public ResourceNotFoundException(String message) {

super(message);

}

}

### **Step 2: Create a Global Exception Handler**

java

CopyEdit

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.ExceptionHandler;

import org.springframework.web.bind.annotation.ControllerAdvice;

import org.springframework.web.context.request.WebRequest;

import java.util.HashMap;

import java.util.Map;

@ControllerAdvice

public class GlobalExceptionHandler {

// Handle specific exception

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<Object> handleResourceNotFoundException(ResourceNotFoundException ex, WebRequest request) {

Map<String, Object> response = new HashMap<>();

response.put("error", "Resource Not Found");

response.put("message", ex.getMessage());

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

}

// Handle global exception

@ExceptionHandler(Exception.class)

public ResponseEntity<Object> handleGlobalException(Exception ex, WebRequest request) {

Map<String, Object> response = new HashMap<>();

response.put("error", "Internal Server Error");

response.put("message", ex.getMessage());

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

}

}

### **Step 3: Use the Exception in a Controller**

java

CopyEdit

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/api")

public class SampleController {

@GetMapping("/find")

public String findResource(@RequestParam String id) {

if ("123".equals(id)) {

return "Resource Found";

} else {

throw new ResourceNotFoundException("Resource with ID " + id + " not found.");

}

}

}

## **Testing the Exception Handling**

### **Request:**

bash

CopyEdit

GET /api/find?id=999

### **Response:**

json

CopyEdit

{

"error": "Resource Not Found",

"message": "Resource with ID 999 not found."

}

## **Key Benefits of @ControllerAdvice**

1. **Centralized Exception Handling** – Keeps controllers clean.
2. **Handles Multiple Exceptions** – Can handle specific and global exceptions.
3. **More Readable Code** – No need for try-catch in each controller.

Let me know if you need more details! 🚀

##### **You said:**

how controlleradvice is beneficial from normal exception handling

###### **ChatGPT said:**

Using @ControllerAdvice for **global exception handling** in Spring Boot provides several advantages over **normal exception handling** within individual controllers. Here’s a detailed comparison:

## **1. Centralized Exception Handling**

### **🔹 With @ControllerAdvice (Global Handling)**

* All exception-handling logic is in a single place.
* No need to repeat exception-handling code in multiple controllers.

java

CopyEdit

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<Object> handleResourceNotFound(ResourceNotFoundException ex) {

return new ResponseEntity<>(Map.of("error", ex.getMessage()), HttpStatus.NOT\_FOUND);

}

}

### **🔹 Without @ControllerAdvice (Local Handling)**

* Each controller must handle exceptions separately, leading to **code duplication**.

java

CopyEdit

@RestController

@RequestMapping("/api")

public class SampleController {

@GetMapping("/find")

public ResponseEntity<Object> findResource(@RequestParam String id) {

try {

if (!id.equals("123")) {

throw new ResourceNotFoundException("Resource not found");

}

return ResponseEntity.ok("Resource Found");

} catch (ResourceNotFoundException ex) {

return new ResponseEntity<>(Map.of("error", ex.getMessage()), HttpStatus.NOT\_FOUND);

}

}

}

### **✅ Benefit of @ControllerAdvice**

* No need to **repeat** try-catch in every controller.
* **Cleaner & maintainable code.**

## **2. Separation of Concerns**

* **Without @ControllerAdvice**: Exception handling is mixed with business logic in controllers.
* **With @ControllerAdvice**: Business logic and error handling are separated, making the code more modular.

## **3. Better Maintainability**

* If exception-handling logic needs to be updated, it only needs to be modified in **one place** (GlobalExceptionHandler).
* Without @ControllerAdvice, you would have to update exception-handling logic in multiple controllers.

## **4. Improved Reusability**

* @ControllerAdvice can handle multiple exceptions across different controllers.
* No need to rewrite handling logic for each controller.

## **5. Handles Multiple Exception Types Efficiently**

Instead of handling different exceptions separately in each controller, @ControllerAdvice allows handling all exceptions in one class.

java

CopyEdit

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<Object> handleResourceNotFound(ResourceNotFoundException ex) {

return new ResponseEntity<>(Map.of("error", ex.getMessage()), HttpStatus.NOT\_FOUND);

}

@ExceptionHandler(IllegalArgumentException.class)

public ResponseEntity<Object> handleBadRequest(IllegalArgumentException ex) {

return new ResponseEntity<>(Map.of("error", "Invalid input"), HttpStatus.BAD\_REQUEST);

}

@ExceptionHandler(Exception.class)

public ResponseEntity<Object> handleGlobalException(Exception ex) {

return new ResponseEntity<>(Map.of("error", "Something went wrong"), HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

### **✅ Benefit of @ControllerAdvice**

* One place to manage **multiple exceptions**.
* Reduces **boilerplate code** in controllers.

## **6. Provides Consistent Error Responses**

Using @ControllerAdvice, you can ensure that all exceptions return a **uniform response format**.

Example of a **consistent error response**:

json

CopyEdit

{

"error": "Resource Not Found",

"message": "Resource with ID 999 not found."

}

Without @ControllerAdvice, different controllers might return different error structures, making it harder for clients to handle errors consistently.

## **7. More Readable & Testable Code**

* Makes **unit testing easier** because the exception-handling logic is separate from controllers.
* Improves **readability** by removing redundant try-catch blocks.

## **Conclusion: Why Use @ControllerAdvice?**

| **Feature** | **With @ControllerAdvice** | **Without @ControllerAdvice** |
| --- | --- | --- |
| **Code Duplication** | ❌ Avoided | ✅ Present |
| **Separation of Concerns** | ✅ Yes | ❌ No |
| **Maintainability** | ✅ High | ❌ Low |
| **Error Consistency** | ✅ Uniform error responses | ❌ Inconsistent |
| **Centralized Exception Handling** | ✅ Yes | ❌ No |
| **Readability & Testability** | ✅ Better | ❌ Worse |

### **Final Verdict:**

✅ @ControllerAdvice is the **best practice** for handling exceptions in Spring Boot because it simplifies code, improves maintainability, and ensures consistent error responses across controllers.

Would you like an example with logging or a custom error response structure? 🚀

**controlleradvice for sprinwebflux mono response in servcie**

###### **ChatGPT said:**

### **Handling Exceptions in Spring WebFlux (Mono Response) Using @ControllerAdvice**

In **Spring WebFlux**, exceptions need to be handled **reactively** when using Mono and Flux. @ControllerAdvice works similarly to Spring MVC, but we should return **Mono<ResponseEntity>** instead of just ResponseEntity.

### **📌 Steps to Handle Exceptions in WebFlux**

1. **Create a Custom Exception (ResourceNotFoundException)**
2. **Throw the Exception in the Service Layer**
3. **Create a Global Exception Handler Using @ControllerAdvice**
4. **Ensure the Controller Uses Mono Properly**

## **1️⃣ Create a Custom Exception**

Create a custom exception that extends RuntimeException (since WebFlux prefers non-blocking exceptions).

java

CopyEdit

public class ResourceNotFoundException extends RuntimeException {

public ResourceNotFoundException(String message) {

super(message);

}

}

## **2️⃣ Service Layer (Returning Mono)**

* Instead of throwing exceptions directly, use Mono.error() to return the error reactively.

java

CopyEdit

import org.springframework.stereotype.Service;

import reactor.core.publisher.Mono;

@Service

public class UserService {

public Mono<String> getUserById(String id) {

if ("123".equals(id)) {

return Mono.just("User Found");

} else {

return Mono.error(new ResourceNotFoundException("User with ID " + id + " not found"));

}

}

}

## **3️⃣ Global Exception Handler Using @ControllerAdvice**

Use @ControllerAdvice to handle exceptions globally in a **reactive manner**.  
 Instead of returning ResponseEntity, return Mono<ResponseEntity<?>>.

java

CopyEdit

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.ControllerAdvice;

import org.springframework.web.bind.annotation.ExceptionHandler;

import reactor.core.publisher.Mono;

import java.util.Map;

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

public Mono<ResponseEntity<Map<String, String>>> handleResourceNotFoundException(ResourceNotFoundException ex) {

return Mono.just(ResponseEntity.status(HttpStatus.NOT\_FOUND)

.body(Map.of("error", "Not Found", "message", ex.getMessage())));

}

@ExceptionHandler(Exception.class)

public Mono<ResponseEntity<Map<String, String>>> handleGlobalException(Exception ex) {

return Mono.just(ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR)

.body(Map.of("error", "Internal Server Error", "message", ex.getMessage())));

}

}

## **4️⃣ Controller Returning Mono<ResponseEntity>**

Make sure the controller **returns Mono<ResponseEntity>** and does not block the response.

java

CopyEdit

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import reactor.core.publisher.Mono;

@RestController

@RequestMapping("/api/users")

public class UserController {

@Autowired

private UserService userService;

@GetMapping("/{id}")

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

return userService.getUserById(id)

.map(ResponseEntity::ok); // If success, return 200 OK

}

}

## **🛠 How It Works**

| **Scenario** | **Behavior** |
| --- | --- |
| GET /api/users/123 | Returns "User Found" (200 OK) |
| GET /api/users/999 | Throws ResourceNotFoundException, handled by @ControllerAdvice (404 Not Found) |
| Any unhandled exception | Handled by @ExceptionHandler(Exception.class) (500 Internal Server Error) |

## **📌 Benefits of This Approach**

✅ **Fully Reactive** – Does not block the response thread.  
 ✅ **Centralized Error Handling** – All exceptions handled in one place.  
 ✅ **Consistent Error Responses** – Ensures a structured response format.  
 ✅ **Works with WebFlux (Mono)** – Ensures proper reactive exception propagation.

### **🚀 Final Thoughts**

This is the best practice for handling exceptions **reactively** in **Spring WebFlux**.  
 Would you like to extend it with logging (SLF4J) or a global error response format (e.g., ErrorResponse DTO)? 😊

##### **You said:**

how to make this dynamic Mono<ResponseEntity<String>> since controller can return anything string or any object

###### **ChatGPT said:**

To make Mono<ResponseEntity<T>> **dynamic** so that it can return **any object type** (String, DTO, etc.), you can use **generics (T)** in the response.

## **📌 Updated Controller with Dynamic Response (Mono<ResponseEntity<?>>)**

We update the controller to handle **any type of response**, not just String.

java

CopyEdit

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import reactor.core.publisher.Mono;

@RestController

@RequestMapping("/api/users")

public class UserController {

@Autowired

private UserService userService;

@GetMapping("/{id}")

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

return userService.getUserById(id)

.map(ResponseEntity::ok) // Dynamically handles different response types

.cast(ResponseEntity.class); // Ensures the return type is ResponseEntity<?>

}

}

## **📌 Updated Service to Support Multiple Response Types**

Now, the service can return **either a String or a User DTO**.

java

CopyEdit

import org.springframework.stereotype.Service;

import reactor.core.publisher.Mono;

@Service

public class UserService {

public Mono<Object> getUserById(String id) {

if ("123".equals(id)) {

return Mono.just(new User("123", "John Doe", "john@example.com")); // Return DTO

} else if ("456".equals(id)) {

return Mono.just("Just a string response!"); // Return a String

} else {

return Mono.error(new ResourceNotFoundException("User with ID " + id + " not found"));

}

}

}

## **📌 Updated Global Exception Handler (Mono<ResponseEntity<?>>)**

Modify the exception handler to return a **generic response**.

java

CopyEdit

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.ControllerAdvice;

import org.springframework.web.bind.annotation.ExceptionHandler;

import reactor.core.publisher.Mono;

import java.util.Map;

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

public Mono<ResponseEntity<?>> handleResourceNotFoundException(ResourceNotFoundException ex) {

return Mono.just(ResponseEntity.status(HttpStatus.NOT\_FOUND)

.body(Map.of("error", "Not Found", "message", ex.getMessage())));

}

@ExceptionHandler(Exception.class)

public Mono<ResponseEntity<?>> handleGlobalException(Exception ex) {

return Mono.just(ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR)

.body(Map.of("error", "Internal Server Error", "message", ex.getMessage())));

}

}

## **📌 User DTO Example**

If the response is an object, it will be returned as JSON.

java

CopyEdit

public class User {

private String id;

private String name;

private String email;

public User(String id, String name, String email) {

this.id = id;

this.name = name;

this.email = email;

}

// Getters and Setters

}

## **📌 Expected API Responses**

| **Request** | **Response Type** | **Example Response** |
| --- | --- | --- |
| GET /api/users/123 | User DTO | {"id":"123","name":"John Doe","email":"john@example.com"} |
| GET /api/users/456 | String | "Just a string response!" |
| GET /api/users/999 | Error JSON | {"error": "Not Found", "message": "User with ID 999 not found"} |

## **🚀 Key Benefits of This Approach**

✅ **Dynamic Response Handling** – Supports both **Strings & DTOs** ✅ **Fully Reactive** – Works well with Mono<ResponseEntity<?>>  
 ✅ **Centralized Exception Handling** – Keeps controllers clean  
 ✅ **Type Safety** – Ensures correct response structure

Would you like to add more enhancements, such as logging or custom error response models?