# 🧩 @ExceptionHandler — Complete Explanation

## 🌱 1️⃣ What is @ExceptionHandler?

👉 @ExceptionHandler is an annotation in Spring Boot that helps you **handle errors and exceptions in a clean, centralized, and readable way**.

It is used inside a controller (or globally) to **catch specific exceptions** and return a **custom error message or response** to the client.

### 🔹 In Simple Words:

It’s like a **safety net** in your application.

Whenever your backend code throws an exception (like “Student not found” or “Invalid input”),  
@ExceptionHandler **catches it**, and instead of showing a long Java error page (stack trace),  
it sends a **clear, user-friendly error message** back to the client.

### 🔹 Real-world Analogy:

Imagine you’re running a restaurant 🍽️.  
If a chef accidentally drops food, a **cleaner** immediately steps in to clean it up before customers notice.

Similarly —  
When your backend code fails (throws an exception),  
@ExceptionHandler **steps in immediately** to handle it gracefully before the client sees an ugly error message.

## 💥 2️⃣ Why Do We Need It? (Problem Without It)

### ❌ Problem Without @ExceptionHandler:

* If an error happens in your controller or service,  
  Java throws an exception like this:
* java.lang.NullPointerException: Cannot invoke ...
* The client (like Postman or frontend app) receives an **HTML error page or stack trace**,  
  which looks messy and confusing.
* No proper **HTTP status code** or **JSON structure** is returned.

Example (ugly response without handler):

Whitelabel Error Page

This application has no explicit mapping for /error...

### ✅ With @ExceptionHandler:

You can catch that error and return a **clean, structured JSON response** like:

{

"error": "Student not found",

"status": 404,

"timestamp": "2025-10-20T10:15:30"

}

👉 This looks professional, readable, and REST-compliant.

## 🌟 3️⃣ Advantages of @ExceptionHandler

| **Advantage** | **Explanation** |
| --- | --- |
| **1. Centralized Error Handling** | You can handle all exceptions from one place. |
| **2. Clean Responses** | Returns neat JSON messages instead of raw error pages. |
| **3. Custom Messages** | You can create your own message for each exception. |
| **4. REST Compliance** | You can set correct HTTP status codes (404, 400, 500, etc.). |
| **5. Maintenance Friendly** | No need to write try-catch in every method. |

## ⚙️ 4️⃣ How It Works Internally

When a controller method throws an exception:

1. Spring checks if any @ExceptionHandler is defined for that exception type.
2. If found, that method is called automatically.
3. That method can then:
   * Create a custom message.
   * Return a custom HTTP response.
4. The client receives a **user-friendly response** instead of a raw Java error.

## 🧠 5️⃣ Step-by-Step Example

We’ll continue with our **StudentController** example.

### 🧩 Step 1: Custom Exception Class

(We’ll create our own exception to handle “Student not found” case)

package in.orcas.exception;

public class StudentNotFoundException extends RuntimeException {

public StudentNotFoundException(String message) {

super(message);

}

}

### 🧩 Step 2: Service Layer (throwing exception)

package in.orcas.service;

import java.util.HashMap;

import java.util.Map;

import org.springframework.stereotype.Service;

import in.orcas.model.Student;

import in.orcas.exception.StudentNotFoundException;

@Service

public class StudentService {

private Map<Integer, Student> db = new HashMap<>();

public String saveStudent(Student student) {

db.put(student.getId(), student);

return "Student saved successfully!";

}

public Student getStudentById(Integer id) {

Student student = db.get(id);

if (student == null) {

throw new StudentNotFoundException("Student not found with ID: " + id);

}

return student;

}

public String deleteStudent(Integer id) {

if (!db.containsKey(id)) {

throw new StudentNotFoundException("Cannot delete. Student not found with ID: " + id);

}

db.remove(id);

return "Student deleted successfully with ID: " + id;

}

}

### 🧩 Step 3: Controller (without try-catch)

package in.orcas.controller;

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

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

import in.orcas.model.Student;

import in.orcas.service.StudentService;

@RestController

@RequestMapping("/students")

public class StudentController {

@Autowired

private StudentService service;

@PostMapping

public String createStudent(@RequestBody Student student) {

return service.saveStudent(student);

}

@GetMapping("/{id}")

public Student getStudent(@PathVariable Integer id) {

return service.getStudentById(id);

}

@DeleteMapping("/{id}")

public String deleteStudent(@PathVariable Integer id) {

return service.deleteStudent(id);

}

}

💡 Notice: There are **no try-catch blocks** — the controller just throws exceptions naturally.

### 🧩 Step 4: Exception Handler

#### Option 1️⃣ — Controller-level handler (only for this controller)

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

import in.orcas.exception.StudentNotFoundException;

@RestController

@RequestMapping("/students")

public class StudentController {

// ... same endpoints above

@ExceptionHandler(StudentNotFoundException.class)

public ResponseEntity<String> handleStudentNotFound(StudentNotFoundException ex) {

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

}

}

✅ Output when ID doesn’t exist:

Student not found with ID: 101

(Status: 404 Not Found)

#### Option 2️⃣ — Global Exception Handler (recommended for real-time apps)

Create a **separate class** to handle all exceptions for the whole project.

package in.orcas.exception;

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 java.time.LocalDateTime;

import java.util.HashMap;

import java.util.Map;

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(StudentNotFoundException.class)

public ResponseEntity<Map<String, Object>> handleStudentNotFound(StudentNotFoundException ex) {

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

body.put("error", ex.getMessage());

body.put("status", HttpStatus.NOT\_FOUND.value());

body.put("timestamp", LocalDateTime.now());

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

}

// handle other exceptions

@ExceptionHandler(Exception.class)

public ResponseEntity<Map<String, Object>> handleGeneralException(Exception ex) {

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

body.put("error", ex.getMessage());

body.put("status", HttpStatus.INTERNAL\_SERVER\_ERROR.value());

body.put("timestamp", LocalDateTime.now());

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

}

}

### 🧪 Step 5: Testing in Postman

**GET Request:**

http://localhost:8080/students/999

✅ Output (from Global Exception Handler):

{

"error": "Student not found with ID: 999",

"status": 404,

"timestamp": "2025-10-20T12:45:12.123"

}

## 🔒 6️⃣ Real-Time Usage

In real enterprise APIs:

* Every project has **one global exception handler** (@ControllerAdvice class).
* Each exception type (like UserNotFound, InvalidInput, DatabaseException) gets its own handler method.
* It helps developers maintain **clean controllers** and **consistent error format** across all endpoints.

## 🧭 7️⃣ Summary Table

| **Concept** | **Description** |
| --- | --- |
| Annotation | @ExceptionHandler |
| Purpose | Handle exceptions and return custom response |
| Works With | Controllers and Services |
| Input | Exception class type |
| Response | Custom message / object / ResponseEntity |
| Commonly Combined With | @ControllerAdvice (for global handling) |
| Benefit | Cleaner APIs, professional error handling |
| REST Benefit | Returns proper HTTP status and JSON structure |

## 🎯 Final Takeaway

@ExceptionHandler helps you build **robust**, **clean**, and **professional REST APIs**.  
Without it — your app crashes and sends ugly error pages.  
With it — your backend gracefully explains what went wrong in a user-friendly way.

**🧩 1. Core Principle of @RestControllerAdvice**

**🌱 Simple Definition:**

@RestControllerAdvice is a **special class** in Spring Boot that allows you to **handle errors, exceptions, or any cross-cutting concerns** (like data formatting, validation messages, etc.) for **all controllers in one place**.

**🔍 Core Concept:**

* Normally, when something goes wrong in your REST API, exceptions are thrown (like *StudentNotFoundException* or *InvalidInputException*).
* Without @RestControllerAdvice, each controller must handle those exceptions separately — which leads to **duplicate code** and **poor maintenance**.
* With @RestControllerAdvice, you create **one centralized class** that can catch and process **exceptions from all controllers** in your application.

👉 In short:  
@RestControllerAdvice acts as a **global supervisor** that watches all your REST controllers and helps them handle exceptions gracefully.

**🧠 Think of it like this:**

Imagine your application has many controllers like StudentController, TeacherController, CourseController, etc.  
Instead of each one writing its own error-handling logic,  
@RestControllerAdvice works like a **“global customer service desk”** —  
any problem from any controller is sent to this single desk to handle it properly and give a clean, friendly response.

**⚙️ 2. Key Features**

| **Feature** | **Description** |
| --- | --- |
| **Global Exception Handling** | Handles exceptions from all controllers in one place. |
| **Data Formatting** | You can modify or format response data for all controllers. |
| **Centralized Code** | Keeps controller classes clean and focused only on business logic. |
| **Works with @ExceptionHandler** | You use @ExceptionHandler methods **inside** the @RestControllerAdvice class to handle specific errors. |

**🧱 3. Complete Working Example (Step by Step)**

Let’s build a simple project that explains it clearly.

**🗂️ Project Structure**

in.orcas

├── controller

│ └── StudentController.java

├── service

│ └── StudentService.java

├── model

│ └── Student.java

├── exception

│ ├── StudentNotFoundException.java

│ └── GlobalExceptionHandler.java <-- @RestControllerAdvice class

└── SpringbootApp.java

**🧩 (1) Model Class — Student.java**

package in.orcas.model;

public class Student {

private Integer id;

private String name;

private String email;

public Student() {}

public Student(Integer id, String name, String email) {

this.id = id;

this.name = name;

this.email = email;

}

// Getters and Setters

public Integer getId() { return id; }

public void setId(Integer id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getEmail() { return email; }

public void setEmail(String email) { this.email = email; }

}

**🧩 (2) Custom Exception — StudentNotFoundException.java**

package in.orcas.exception;

public class StudentNotFoundException extends RuntimeException {

public StudentNotFoundException(String message) {

super(message);

}

}

This is a **custom exception** thrown when a student is not found.

**🧩 (3) Service Class — StudentService.java**

package in.orcas.service;

import java.util.\*;

import org.springframework.stereotype.Service;

import in.orcas.model.Student;

import in.orcas.exception.StudentNotFoundException;

@Service

public class StudentService {

private static Map<Integer, Student> studentDB = new HashMap<>();

static {

studentDB.put(101, new Student(101, "Pavan", "pavan@gmail.com"));

studentDB.put(102, new Student(102, "Kalyan", "kalyan@gmail.com"));

}

public Student getStudentById(Integer id) {

Student s = studentDB.get(id);

if (s == null) {

throw new StudentNotFoundException("No student found with ID: " + id);

}

return s;

}

}

**🧩 (4) Controller — StudentController.java**

package in.orcas.controller;

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

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

import in.orcas.model.Student;

import in.orcas.service.StudentService;

@RestController

@RequestMapping("/students")

public class StudentController {

@Autowired

private StudentService service;

@GetMapping("/{id}")

public Student getStudent(@PathVariable Integer id) {

return service.getStudentById(id);

}

}

This controller simply calls the service.  
If the service throws an exception, we won’t handle it here —  
it will automatically be caught by our @RestControllerAdvice.

**🧩 (5) Global Exception Handler — GlobalExceptionHandler.java**

package in.orcas.exception;

import org.springframework.http.\*;

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

import java.time.LocalDateTime;

import java.util.\*;

@RestControllerAdvice

public class GlobalExceptionHandler {

// Handle a specific exception

@ExceptionHandler(StudentNotFoundException.class)

public ResponseEntity<Map<String, Object>> handleStudentNotFound(StudentNotFoundException ex) {

Map<String, Object> body = new LinkedHashMap<>();

body.put("timestamp", LocalDateTime.now());

body.put("error", "Student Not Found");

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

body.put("status", HttpStatus.NOT\_FOUND.value());

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

}

// Handle any other exception (fallback)

@ExceptionHandler(Exception.class)

public ResponseEntity<Map<String, Object>> handleGenericError(Exception ex) {

Map<String, Object> body = new LinkedHashMap<>();

body.put("timestamp", LocalDateTime.now());

body.put("error", "Something went wrong");

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

body.put("status", HttpStatus.INTERNAL\_SERVER\_ERROR.value());

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

}

}

**🧩 (6) Main Application — SpringbootApp.java**

package in.orcas;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SpringbootApp {

public static void main(String[] args) {

SpringApplication.run(SpringbootApp.class, args);

}

}

**🧪 4. Testing in Postman**

**✅ Case 1: Student Found**

**URL:**  
GET http://localhost:8080/students/101

**Response:**

{

"id": 101,

"name": "Pavan",

"email": "pavan@gmail.com"

}

**❌ Case 2: Student Not Found**

**URL:**  
GET http://localhost:8080/students/999

**Response (Handled by @RestControllerAdvice):**

{

"timestamp": "2025-10-21T19:45:22.140",

"error": "Student Not Found",

"message": "No student found with ID: 999",

"status": 404

}

No error stack trace is shown —  
clean, user-friendly message.

**📚 5. How It Actually Works Internally**

When you send a request:

1. StudentController is called.
2. It calls StudentService.
3. If service throws an exception → Spring looks for a **matching @ExceptionHandler** inside @RestControllerAdvice.
4. If found → that method is executed and returns a well-formatted response.
5. If not found → Spring sends a generic 500 error.

**🧠 6. Key Points to Remember**

| **Concept** | **Meaning** |
| --- | --- |
| @RestControllerAdvice | A class-level annotation to globally handle REST errors. |
| @ExceptionHandler | A method-level annotation inside advice that defines how to handle a specific exception. |
| Scope | Applies to all controllers automatically. |
| Advantage | Centralized, clean, reusable, consistent error responses. |
| Common Pair | Usually used together with custom exceptions. |

**✅ Real-Life Analogy**

Think of each **controller** as a **shop counter** in a mall.  
If a customer faces any problem at any counter,  
they don’t deal with it there — they go to the **Customer Help Desk**.  
That **help desk** is your @RestControllerAdvice class.  
It listens for complaints (exceptions) from every counter (controller)  
and gives the right solution or message.

* @RestControllerAdvice  
  👉 Acts as a **global catch block** (like a safety net) for all controllers in your application.
* But this net doesn’t know **what** to catch or **how** to respond by itself.  
  👉 That’s why it **needs** methods marked with @ExceptionHandler,  
  which tell it:  
  “If this specific type of exception happens — handle it in this way.”

**💡 So the Relationship Is:**

| **Annotation** | **Role** | **Analogy** |
| --- | --- | --- |
| @RestControllerAdvice | The global listener that looks for any exceptions from controllers | The *main catch zone* |
| @ExceptionHandler | The actual handler that processes a **specific type** of exception | The *individual catch blocks* inside that zone |

**⚙️ Example to Reinforce**

@RestControllerAdvice

public class GlobalExceptionHandler {

// Handles specific type

@ExceptionHandler(StudentNotFoundException.class)

public ResponseEntity<String> handleStudentNotFound(StudentNotFoundException ex) {

return ResponseEntity.status(HttpStatus.NOT\_FOUND).body(ex.getMessage());

}

// Handles all other unknown types

@ExceptionHandler(Exception.class)

public ResponseEntity<String> handleAnyException(Exception ex) {

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

.body("Something went wrong: " + ex.getMessage());

}

}

✅ @RestControllerAdvice = the global place where exceptions are caught.  
✅ @ExceptionHandler(...) = the specific rule inside that global place for each exception type.

🧠 **In one sentence:**

@RestControllerAdvice provides the “where” to handle exceptions globally,  
@ExceptionHandler provides the “how” and “which one” to handle.

Eg: ExceptionalHandlerRestController