# **1. Introduction to ResponseEntity**

**🔹 Why It Came**

* **Fine-Grained Control**: Early Spring MVC controller methods returned domain objects (or views) and relied on default status codes (usually 200 OK). As RESTful practices matured, developers needed a way to control *both* the body *and* the HTTP status (and headers) per response.
* **Full HTTP Semantics**: Proper REST APIs use a variety of status codes (201 Created, 204 No Content, 404 Not Found, 409 Conflict, etc.) and custom headers (Location, ETag, Cache-Control). ResponseEntity<T> wraps all of these in one object.
* **Consistency & Readability**: Using ResponseEntity makes intent explicit in controller code (return ResponseEntity.status(201).body(newUser);), rather than hiding status logic or relying on annotations only.

**2. Core Rules & Best Practices**

| **Rule** | **Description** |
| --- | --- |
| **Always Return ResponseEntity<T>** | For REST endpoints, prefer ResponseEntity<T> over raw domain objects. This makes status control explicit. |
| **Use Static Builders** | ResponseEntity.ok(), .created(URI), .noContent(), .status(HttpStatus) simplify code and improve readability. |
| **Populate Headers When Needed** | e.g. ResponseEntity.created(location).body(entity) sets Location header for newly created resources. |
| **Avoid Overuse** | For simple GETs returning 200, @GetMapping with a returned object is fine. Reserve ResponseEntity when non‐200, headers, or no‐content responses are needed. |
| **Separate Error Handling** | Use @ExceptionHandler or @ControllerAdvice to map exceptions to ResponseEntity<ErrorDto> rather than peppering controllers with try/catch. |

**3. When & Where to Use**

| **Scenario** | **Use Case** |
| --- | --- |
| **Resource Creation (POST)** | Return 201 Created with Location header pointing to the new resource. |
| **Update/Delete Operations** | On successful update: return 200 OK (with updated body) or 204 No Content if no body is needed. |
| **Error Conditions** | Return 404 Not Found, 400 Bad Request, 409 Conflict, etc., with an error payload. |
| **Conditional GET / Caching** | Return 304 Not Modified when If-None-Match/If-Modified-Since conditions are met. |
| **Streaming / File Downloads** | Use ResponseEntity<byte[]> or StreamingResponseBody with appropriate Content-Type and Content-Disposition. |

**4. HTTP Status Codes Overview**

| **Status** | **Name** | **When to Use** |
| --- | --- | --- |
| **200 OK** | Success | Standard response for successful GET, PUT, or POST (when returning a body). |
| **201 Created** | Resource Created | After a successful POST that creates a resource—include Location header. |
| **204 No Content** | No Content | Successful request that does not need to return a body (DELETE, PUT without response body). |
| **400 Bad Request** | Client Error | Malformed syntax, validation errors, missing required parameters. |
| **401 Unauthorized** | Client Error | Authentication required or failed. |
| **403 Forbidden** | Client Error | Authenticated but not allowed to perform the action. |
| **404 Not Found** | Client Error | Resource not found. |
| **409 Conflict** | Client Error | Request conflicts with current state (e.g., duplicate key). |
| **500 Internal Server Error** | Server Error | Unhandled exceptions or failure in server logic. |
| **503 Service Unavailable** | Server Error | Temporary overload or maintenance. |

**5. Real-World Examples**

**Example 1: Creating a New User (POST)**

@PostMapping("/users")

public ResponseEntity<UserDto> createUser(@Valid @RequestBody UserDto in) {

UserDto created = userService.create(in);

URI location = ServletUriComponentsBuilder

.fromCurrentRequest()

.path("/{id}")

.buildAndExpand(created.getId())

.toUri();

return ResponseEntity

.created(location) // 201 Created + Location header

.body(created); // response body with new user

}

* **Status:** 201 Created
* **Headers:** Location: /users/{newId}
* **Body:** The newly created user DTO

**Example 2: Updating a Resource (PUT) with No Body**

@PutMapping("/orders/{id}")

public ResponseEntity<Void> updateOrder(@PathVariable Long id, @Valid @RequestBody OrderDto in) {

boolean exists = orderService.update(id, in);

if (!exists) {

return ResponseEntity.notFound().build(); // 404 if order doesn’t exist

}

return ResponseEntity.noContent().build(); // 204 No Content on success

}

* **404 Not Found:** If the order ID doesn’t exist
* **204 No Content:** Successful update, no response body needed

**Example 3: Searching with Validation and Errors**

@GetMapping("/products")

public ResponseEntity<?> search(

@RequestParam(required = false) String q,

@RequestParam(defaultValue = "0") int page,

@RequestParam(defaultValue = "10") int size) {

if (size > 100) {

return ResponseEntity

.status(HttpStatus.BAD\_REQUEST)

.body(Map.of("error", "Page size must be ≤ 100"));

}

Page<ProductDto> result = productService.search(q, PageRequest.of(page, size));

if (result.isEmpty()) {

return ResponseEntity.noContent().build(); // 204 if no matches

}

return ResponseEntity.ok(result); // 200 with paged results

}

* **400 Bad Request:** If size parameter invalid
* **204 No Content:** If search returns zero results
* **200 OK:** If results found

**6. Summary**

* **ResponseEntity<T>** gives you **full control** over HTTP status, headers, and body.
* Use it whenever you need to return non‐200 statuses, set headers (e.g. Location), or produce *no content* responses.
* Combine appropriate **HTTP status codes** with RESTful semantics to build **clear**, **self‐documenting**, and **standards‐compliant** APIs.

## **1. Why Centralized Exception Handling?**

* **Consistency:** You want all controllers to return uniform error payloads (status codes, JSON structure) rather than ad-hoc try/catch blocks in each handler.
* **Separation of Concerns:** Controllers focus on business logic; error translation is handled elsewhere.
* **Reusability:** Common exception-to-response mapping (e.g. EntityNotFound → 404, ValidationError → 400) lives in one place.

**2. Local Exception Handler**

**🔹 What & Why**

A **local** exception handler is a method annotated with @ExceptionHandler within a specific @Controller (or @RestController). It catches exceptions thrown by that controller’s handler methods.

**📜 How It Works**

@RestController

@RequestMapping("/users")

public class UserController {

@GetMapping("/{id}")

public User get(@PathVariable String id) {

return userService.findById(id)

.orElseThrow(() -> new UserNotFoundException(id));

}

@ExceptionHandler(UserNotFoundException.class)

public ResponseEntity<ApiError> handleNotFound(UserNotFoundException ex) {

ApiError err = new ApiError(404, ex.getMessage());

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

}

}

* Only exceptions raised within this controller are caught by its @ExceptionHandler methods.
* Other controllers require their own handlers or will fall back to global handlers.

**📌 Rules & Best Practices**

1. **Scope:** Handlers in a controller apply only to exceptions thrown by that controller’s request-mapping methods.
2. **Method Signature:** The method may take the exception type, optionally an injected WebRequest or HttpServletRequest.
3. **Return Type:** Can return a domain object (serialized to JSON), ResponseEntity<?>, or even a ModelAndView for server-rendered views.
4. **Ordering:** Local handlers run before any global ones (if both match).

**📍 When to Use Local Handlers**

* You have controller-specific errors (e.g. PaymentController throws InsufficientFundsException with custom recovery steps).
* You need a different response shape for a particular controller.
* You want to embed error-handling logic close to the related business logic.

**✅ Real-World Examples**

1. **Payment Failure in PaymentController**

@ExceptionHandler(InsufficientFundsException.class)

public ResponseEntity<ErrorDto> onInsufficient(InsufficientFundsException ex) {

return status(402).body(new ErrorDto("Payment failed", ex.getBalance()));

}

1. **File Upload Errors in UploadController**

@ExceptionHandler(MaxUploadSizeExceededException.class)

public ResponseEntity<ErrorDto> onTooLarge(MaxUploadSizeExceededException ex) {

return badRequest().body(new ErrorDto("File too large", null));

}

1. **Domain Validation in UserController**

@ExceptionHandler(UsernameTakenException.class)

public ResponseEntity<ErrorDto> onDuplicate(UsernameTakenException ex) {

return conflict().body(new ErrorDto("Username already exists", ex.getUsername()));

}

**3. Global Exception Handler**

**🔹 What & Why**

A **global** exception handler uses @ControllerAdvice (often combined with @RestControllerAdvice in REST apps) to apply exception-handling logic across *all* controllers.

**📜 How It Works**

@RestControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(EntityNotFoundException.class)

public ResponseEntity<ApiError> handleNotFound(EntityNotFoundException ex) {

ApiError err = new ApiError(404, ex.getMessage());

return status(NOT\_FOUND).body(err);

}

@ExceptionHandler(MethodArgumentNotValidException.class)

public ResponseEntity<ApiError> handleValidation(MethodArgumentNotValidException ex) {

List<String> errors = ex.getBindingResult()

.getFieldErrors()

.stream()

.map(f -> f.getField()+": "+f.getDefaultMessage())

.toList();

return badRequest().body(new ApiError(400, "Validation failed", errors));

}

@ExceptionHandler(Exception.class)

public ResponseEntity<ApiError> handleAll(Exception ex) {

return status(INTERNAL\_SERVER\_ERROR)

.body(new ApiError(500, "Unexpected error", ex.getMessage()));

}

}

* Applies to every controller in the application context.
* You can limit its scope via attributes (e.g. basePackages, annotations).

**📌 Rules & Best Practices**

1. **Granularity:** Define handlers for broad categories (e.g. all RuntimeException) and specific exceptions.
2. **Ordering:** If multiple @ControllerAdvice exist, you can use @Order or Ordered interface.
3. **Exception Hierarchy:** The most specific @ExceptionHandler method is chosen (e.g. handler for UserNotFoundException over one for RuntimeException).
4. **Avoid Business Logic:** Keep handlers focused on translating exceptions to responses, not recovery.

**📍 When to Use Global Handlers**

* You need a **consistent** error API across all endpoints (common JSON error shape).
* You want to centralize cross-cutting concerns like logging exceptions or sanitizing error messages.
* You prefer **DRY** (Don’t Repeat Yourself)—no need to replicate similar @ExceptionHandler methods in every controller.

**✅ Real-World Examples**

1. **Uniform 404 Handling**

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<ErrorDto> onNotFound(ResourceNotFoundException ex) {

return status(NOT\_FOUND).body(new ErrorDto("Resource missing", ex.getResourceId()));

}

1. **Global Validation Failure**

@ExceptionHandler(BindException.class)

public ResponseEntity<ErrorDto> onBindError(BindException ex) { … }

1. **Fallback for Uncaught Exceptions**

@ExceptionHandler(Exception.class)

public ResponseEntity<ErrorDto> onAnyError(Exception ex) {

log.error("Unhandled exception", ex);

return status(INTERNAL\_SERVER\_ERROR)

.body(new ErrorDto("Internal error", ex.getMessage()));

}

**4. Local vs. Global: Key Differences**

| **Aspect** | **Local (@ExceptionHandler in Controller)** | **Global (@ControllerAdvice)** |
| --- | --- | --- |
| **Scope** | Only that controller’s exceptions | All controllers (unless filtered) |
| **Co-location** | Proximity to related business logic | Centralized in one or few classes |
| **Reuse** | Not reusable | Fully reusable across the app |
| **Customization** | Easy per-controller variation | Requires conditional logic or multiple advice classes |
| **Complexity** | Simple for one controller | Better for large apps with many controllers |

**5. Decision Guide**

* **Use Local Handler** when:
  + Error semantics are *unique* to one controller.
  + You prefer keeping error logic next to related endpoints.
  + You have only one or two endpoints and simple error needs.
* **Use Global Handler** when:
  + You want *uniform* error responses across your API.
  + You have *many* controllers sharing error-mapping logic.
  + You need centralized logging or metrics on all exceptions.

# **✅ Spring Boot: ResponseEntity & Error Handling Quiz**

**1. What is the purpose of ResponseEntity in Spring Boot?**  
A. To render HTML templates  
B. To handle database queries  
C. To represent the entire HTTP response, including status, headers, and body  
D. To log incoming requests

**Answer:** C. To represent the entire HTTP response, including status, headers, and body

**2. Which method is used to create a ResponseEntity with a body and HTTP status code?**  
A. ResponseEntity.statusCode()  
B. ResponseEntity.create()  
C. ResponseEntity.of()  
D. ResponseEntity.status(HttpStatus).body()

**Answer:** D. ResponseEntity.status(HttpStatus).body()

**3. What is the default HTTP status when returning an object directly from a controller method in Spring Boot?**  
A. 201 Created  
B. 200 OK  
C. 204 No Content  
D. 500 Internal Server Error

**Answer:** B. 200 OK

**4. Which annotation is used to handle exceptions globally in a Spring Boot application?**  
A. @ControllerAdvice  
B. @ExceptionMapper  
C. @ErrorHandler  
D. @RestAdvice

**Answer:** A. @ControllerAdvice

**5. What annotation is used to map a method to handle specific exceptions?**  
A. @ErrorHandler  
B. @Catch  
C. @ExceptionHandler  
D. @HandleException

**Answer:** C. @ExceptionHandler

**6. What class would you use to return a 404 NOT FOUND with a custom message?**  
A. ModelAndView  
B. ResponseStatusException  
C. ServletException  
D. IllegalArgumentException

**Answer:** B. ResponseStatusException

**7. Which HTTP status code is typically used to represent a bad request?**  
A. 404  
B. 403  
C. 400  
D. 401

**Answer:** C. 400

**8. How can you return a 204 No Content response using ResponseEntity?**  
A. return ResponseEntity.noContent().build();  
B. return ResponseEntity.ok().build();  
C. return new ResponseEntity<>(HttpStatus.NO\_RESPONSE);  
D. return ResponseEntity.status(204);

**Answer:** A. return ResponseEntity.noContent().build();

**9. In Spring Boot, what does @ResponseStatus on an exception class do?**  
A. Logs the exception  
B. Maps the exception to an HTTP status code  
C. Stops the server  
D. Sends error to the client as HTML

**Answer:** B. Maps the exception to an HTTP status code

**10. What HTTP status code is generally used for a successful creation of a resource?**  
A. 200 OK  
B. 202 Accepted  
C. 201 Created  
D. 204 No Content

**Answer:** C. 201 Created

**11. Which method allows building a ResponseEntity with custom headers and no body?**  
A. ResponseEntity.empty()  
B. ResponseEntity.headersOnly()  
C. ResponseEntity.status().headers()  
D. ResponseEntity.ok().headers(headers).build()

**Answer:** D. ResponseEntity.ok().headers(headers).build()

**12. How would you handle all RuntimeExceptions globally?**  
A. Use @ExceptionHandler(RuntimeException.class) in a @ControllerAdvice class  
B. Catch them in every controller  
C. Annotate exception with @ResponseStatus  
D. Use @GlobalException

**Answer:** A. Use @ExceptionHandler(RuntimeException.class) in a @ControllerAdvice class

**13. What is the advantage of using ResponseEntity<?> over returning just a POJO?**  
A. It simplifies logging  
B. It provides control over response headers and status codes  
C. It improves performance  
D. It generates OpenAPI docs

**Answer:** B. It provides control over response headers and status codes

**14. What is the typical return type of a method annotated with @ExceptionHandler?**  
A. void  
B. String  
C. ResponseEntity<?>  
D. ModelAndView

**Answer:** C. ResponseEntity<?>

# **Assignment for this week:**

**Assignment: Building a “Room Booking” Microservice with Robust Exception Handling**

**📖 Scenario**

You are developing a **Room Booking** microservice for a hotel chain. The service lets clients:

* **Search** for available rooms by date and type
* **Book** a room
* **Cancel** a booking
* **View** an existing booking

You must implement both **local** and **global** exception handling so that clients always receive clear, consistent error responses (status codes, JSON body).

**🎯 Requirements**

1. **Spring Boot REST API**
   * Use spring-boot-starter-web and spring-boot-starter-validation.
2. **Domain Model**

public class RoomBooking {

String bookingId;

String roomType; // e.g. "DELUXE", "STANDARD"

LocalDate checkIn;

LocalDate checkOut;

String guestName;

}

1. **Exceptions to Define**
   * RoomNotAvailableException → 409 Conflict
   * BookingNotFoundException → 404 Not Found
   * InvalidBookingException → 400 Bad Request
2. **Controller**
   * GET /rooms/available?type={type}&from={date}&to={date}
   * POST /bookings → create a new booking
   * GET /bookings/{id}
   * DELETE /bookings/{id}

In BookingController, use **local** @ExceptionHandler for InvalidBookingException (e.g. check-in ≥ check-out).

1. **Service Layer**
   * RoomService.checkAvailability(...) throws RoomNotAvailableException if none free.
   * BookingService.findById(...) throws BookingNotFoundException.
2. **Global Exception Handler**
   * Annotated @RestControllerAdvice
   * Catches RoomNotAvailableException, BookingNotFoundException, Exception
   * Returns JSON: { "timestamp": ..., "status": 409, "error": "Conflict", "message": "...", "path": "..." }

**🛠️ Sample Inputs & Expected Outputs**

1. **Searching for Available Rooms**  
   **Request**

GET /rooms/available?type=DELUXE&from=2025-05-10&to=2025-05-12

**Response**

HTTP/1.1 200 OK

["DELUXE-101","DELUXE-102","DELUXE-105"]

1. **Booking a Room Success**  
   **Request**

POST /bookings

Content-Type: application/json

{

"roomType":"DELUXE",

"checkIn":"2025-05-10",

"checkOut":"2025-05-12",

"guestName":"Alice"

}

**Response**

HTTP/1.1 201 Created

{

"bookingId":"bkg-1234",

"roomType":"DELUXE",

"checkIn":"2025-05-10",

"checkOut":"2025-05-12",

"guestName":"Alice"

}

1. **Booking Failure: Room Not Available**  
   **Request** (same as above but no rooms left)  
   **Response**

HTTP/1.1 409 Conflict

{

"timestamp":"2025-05-01T14:23:05Z",

"status":409,

"error":"Conflict",

"message":"No DELUXE rooms available from 2025-05-10 to 2025-05-12",

"path":"/bookings"

}

1. **Invalid Dates: Local Handler**  
   **Request**

{

"roomType":"DELUXE",

"checkIn":"2025-05-15",

"checkOut":"2025-05-10",

"guestName":"Bob"

}

**Response**

HTTP/1.1 400 Bad Request

{

"bookingId":null,

"roomType":"DELUXE",

"checkIn":"2025-05-15",

"checkOut":"2025-05-10",

"guestName":"Bob",

"error":"Check-out date must be after check-in date"

}

1. **Cancel Non-Existent Booking**  
   **Request**

DELETE /bookings/bkg-9999

**Response**

HTTP/1.1 404 Not Found

{

"timestamp":"2025-05-01T14:25:00Z",

"status":404,

"error":"Not Found",

"message":"Booking with ID bkg-9999 not found",

"path":"/bookings/bkg-9999"

}

**✅ Deliverables**

* **Domain**, **Service**, and **Controller** classes
* **Custom exception** classes
* **Local** handler in BookingController for InvalidBookingException
* **Global** handler (@RestControllerAdvice) for other exceptions