# 📘 Notes: Spring Boot REST Example

### 1. What is REST in simple terms?

* REST = **Representational State Transfer**
* It’s a style of building web services where we use **HTTP methods (GET, POST, PUT, DELETE)** to perform operations.
* Example:
  + GET /msg → Fetch a message
  + POST /student → Create student
  + DELETE /student/10 → Delete student with id 10

### 2. Important Spring Boot Annotations used

* **@SpringBootApplication**
  + Marks the main class.
  + Combines @Configuration, @EnableAutoConfiguration, and @ComponentScan.
  + Starts Spring Boot.
* **@RestController**
  + Special controller where each method returns data directly (instead of view).
  + Equivalent to: @Controller + @ResponseBody.
* **@GetMapping("/msg")**
  + Maps HTTP GET request to /msg.
  + Example: http://localhost:8080/msg.

### 3. Project Structure

rest-demo

└─ src

└─ main

├─ java

│ └─ in/ineuron

│ ├─ RestDemoApplication.java <-- Main app

│ └─ controller

│ └─ WishController.java <-- REST Controller

└─ resources

└─ application.properties <-- Config file

### 4. Flow of Execution

1. Run **RestDemoApplication** → Starts Spring Boot (Tomcat server runs on port 8080).
2. Request comes to → http://localhost:8080/msg.
3. Spring finds → WishController and generateWishMessage() method.
4. Method returns → "Welcome to Restful Services from Ineuron...".
5. Spring sends this string as HTTP response to browser / Postman.

### 5. pom.xml Purpose

* Manages dependencies.
* Important dependency used:
* <dependency>
* <groupId>org.springframework.boot</groupId>
* <artifactId>spring-boot-starter-web</artifactId>
* </dependency>

→ This adds Spring MVC + Tomcat + REST support.

### 6. application.properties

* Used to configure application.
* Example:
* server.port=9090
* spring.application.name=RestDemo
* If no port is given, default = **9090**.

### 7. Testing REST

* You can test using:
  1. Browser → http://localhost:9090/msg
  2. Postman / Curl.

### 8. Output

When you hit /msg, you get:

Welcome to Restful Services from Ineuron...

### 9. Key Points to Remember

* REST services return **data (JSON, String, XML)**, not views (like JSP).
* @RestController is most commonly used.
* Always keep dependencies and port settings in mind.
* Spring Boot auto-configures most things, you just focus on writing controllers and business logic.

Eg: RestDemoApplication

**📘 Detailed Notes: Spring Boot REST Example with ResponseEntity**

**1. Introduction to REST and Spring Boot**

* **REST (Representational State Transfer)** → architectural style to design APIs using HTTP.
* **Spring Boot** → simplifies REST service development by:
  + Auto-configuring web server (Tomcat).
  + Handling requests with annotations.
  + Returning responses easily.

**2. What is ResponseEntity?**

* ResponseEntity<T> = A **generic class** provided by Spring (org.springframework.http).
* Represents the **entire HTTP response**.
* It allows us to set:
  1. **Body** → the actual data we want to send.
  2. **Status Code** → tells client whether request succeeded, failed, or something else.
  3. **Headers** → additional metadata like authentication, caching, or custom info.

👉 Without ResponseEntity → we can only send body.  
👉 With ResponseEntity → we send a **complete response**.

**3. Why do we need ResponseEntity in Real-Time?**

1. **Different Outcomes Need Different Status Codes**
   * Success → 200 OK
   * New data created → 201 CREATED
   * Record not found → 404 NOT FOUND
   * Validation error → 400 BAD REQUEST
   * Server error → 500 INTERNAL SERVER ERROR
2. **Professional APIs**
   * REST APIs are consumed by **other apps** (mobile, web, microservices).
   * They expect **correct HTTP status codes**, not just messages.
3. **Custom Headers**
   * Sometimes extra data must be sent. Example:
     + X-App-Version: 1.0
     + X-Request-ID: 12345
4. **Better Error Handling**
   * We can clearly tell the client **why something failed**.

**4. Example Code**

**Controller**

@RestController

@RequestMapping("/api") // Class-level mapping

public class WishController {

@GetMapping("/msg")

public ResponseEntity<String> generateWishMessage() {

String msg = "Welcome to Restful Services from Ineuron...";

// Returning ResponseEntity with body + status code

return new ResponseEntity<>(msg, HttpStatus.OK);

}

}

**5. Working of return new ResponseEntity<>(msg, HttpStatus.OK);**

* msg → the **body** of response.
* HttpStatus.OK → status code 200, means "Request successful".
* Together, Spring Boot generates a **proper HTTP response**:
* HTTP/1.1 200 OK
* Content-Type: text/plain;charset=UTF-8
* Content-Length: 45
* Welcome to Restful Services from Ineuron...

**6. What is HttpStatus.OK?**

* **HttpStatus** is an **enum** in Spring containing standard HTTP codes.
* Commonly used values:
  + HttpStatus.OK → 200 (Success)
  + HttpStatus.CREATED → 201 (Resource created successfully)
  + HttpStatus.NO\_CONTENT → 204 (Request successful but no data to return)
  + HttpStatus.BAD\_REQUEST → 400 (Invalid input from client)
  + HttpStatus.NOT\_FOUND → 404 (Resource not found)
  + HttpStatus.INTERNAL\_SERVER\_ERROR → 500 (Unexpected server error)

👉 These codes are **universal standards**, so any client (browser, Postman, mobile app) understands them.

**7. Flow of Execution**

1. Client sends request → http://localhost:8080/api/msg.
2. Spring Boot dispatches request to → WishController.generateWishMessage().
3. Method prepares → "Welcome to Restful Services from Ineuron...".
4. Wraps it inside → ResponseEntity<String>(msg, HttpStatus.OK).
5. Returns to client → Response with **body + status**.

**8. Analogy for Better Understanding**

Think of **sending a parcel**:

* **Only message** (without ResponseEntity) = Sending parcel without label. Postman assumes everything is fine.
* **With ResponseEntity** = Sending parcel with:
  + ✅ Contents (the data/body)
  + ✅ Stamp saying "Delivered Successfully" or "Delivery Failed" (status code)
  + ✅ Extra stickers like "Fragile", "Urgent" (headers)

👉 Clients trust the parcel **with labels and stamps**, not just the contents.

**9. Real-Time Usage Example**

Suppose we have an API to fetch a student by ID:

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

public ResponseEntity<?> getStudent(@PathVariable Integer id) {

Student student = service.findStudentById(id);

if (student != null) {

// Found → 200 OK with student JSON

return ResponseEntity.ok(student);

} else {

// Not found → 404 with message

return ResponseEntity.status(HttpStatus.NOT\_FOUND)

.body("Student not found with id: " + id);

}

}

* If student exists:
* Status: 200 OK
* Body: { "id": 1, "name": "Pavan" }
* If student missing:
* Status: 404 NOT FOUND
* Body: Student not found with id: 1

**10. Quick Comparison Table**

| **Feature** | **Without ResponseEntity** | **With ResponseEntity** |
| --- | --- | --- |
| Response Body | ✅ Yes | ✅ Yes |
| HTTP Status Control | ❌ Always 200 | ✅ Customizable (200, 201, 404, etc.) |
| Headers | ❌ Cannot set | ✅ Can set easily |
| Real-Time Usage | Rare (for demo/learning only) | ✅ Standard for professional APIs |

**11. Key Takeaways**

* **Need**: ResponseEntity is needed when you want to build **real-world REST APIs** with proper status codes and headers.
* **Working**: It wraps the **body + status + headers** into one object that Spring sends as HTTP response.
* **HttpStatus.OK**: Standard status code (200) → means success.

👉 In **practice**:

* Small demo → returning String or Object is okay.
* **Real-time project** → always use ResponseEntity.

Eg: ResponseEntity

**HTTP Methods in Spring REST Controller**

**🔹 Class Overview**

@RestController

@RequestMapping("/customer")

* @RestController → Marks this class as a REST endpoint.
  + Combines @Controller + @ResponseBody.
  + So every method returns data directly as an HTTP response (no view page).
* @RequestMapping("/customer") → Common base path for all methods.

**🔹 1. GET – Read Data**

@GetMapping("/report")

public ResponseEntity<String> showCustomerReport() {

return new ResponseEntity<>("FROM GET - showCustomerReport()", HttpStatus.OK);

}

✅ **Use Case:**

* When client wants to **read/fetch** data (no modification).
* Example: /customer/report

✅ **HTTP Status:**

* HttpStatus.OK (200) → Request successful, data returned.

✅ **Real-time Example:**

* Get customer details by ID or get all customers list.

**🔹 2. POST – Create New Data**

@PostMapping("/report")

public ResponseEntity<String> registerCustomer() {

return new ResponseEntity<>("FROM POST - registerCustomer()", HttpStatus.CREATED);

}

✅ **Use Case:**

* When client sends data to **create a new record** in DB.
* Example: /customer/report with JSON body (like customer info).

✅ **HTTP Status:**

* HttpStatus.CREATED (201) → Resource successfully created.

✅ **Real-time Example:**

* Register a new customer.
* Submit a new order or payment.

**🔹 3. PUT – Full Update**

@PutMapping("/modify")

public ResponseEntity<String> updateCustomer() {

return new ResponseEntity<>("FROM PUT - updateCustomer()", HttpStatus.OK);

}

✅ **Use Case:**

* To **replace the existing record completely** with new data.
* All fields of resource are updated.

✅ **HTTP Status:**

* HttpStatus.OK (200) → Update successful.

✅ **Real-time Example:**

* Update entire customer details (name, address, phone, etc.).

**🔹 4. PATCH – Partial Update**

@PatchMapping("/pmodify")

public ResponseEntity<String> updateCustomerByNo() {

return new ResponseEntity<>("FROM PATCH - updateCustomerByNo()", HttpStatus.OK);

}

✅ **Use Case:**

* When you want to **update only few fields** instead of the full object.

✅ **HTTP Status:**

* HttpStatus.OK (200)

✅ **Real-time Example:**

* Update only customer’s phone number or email.

**🔹 5. DELETE – Remove Data**

@DeleteMapping("/remove")

public ResponseEntity<String> deleteCustomer() {

return new ResponseEntity<>("FROM DELETE - deleteCustomer()", HttpStatus.NO\_CONTENT);

}

✅ **Use Case:**

* To **delete** a resource from the server.

✅ **HTTP Status:**

* HttpStatus.NO\_CONTENT (204) → Successfully deleted, no body returned.

✅ **Real-time Example:**

* Delete a customer record or cancel a booking.

**🔹 6. HEAD – Headers Only**

@RequestMapping(value = "/head", method = RequestMethod.HEAD)

public ResponseEntity<Void> headCustomer() {

return new ResponseEntity<>(HttpStatus.OK);

}

✅ **Use Case:**

* To **check if a resource exists** without downloading the body.
* Returns only **headers**, no body content.

✅ **HTTP Status:**

* HttpStatus.OK (200)

✅ **Real-time Example:**

* API health checks or validation before fetching large data.

**🔹 7. OPTIONS – Allowed Methods**

@RequestMapping(value = "/options", method = RequestMethod.OPTIONS)

public ResponseEntity<Void> optionsCustomer() {

return ResponseEntity.ok().build();

}

✅ **Use Case:**

* To know **which HTTP methods are allowed** for a specific endpoint.

✅ **HTTP Status:**

* HttpStatus.OK (200) → “Allow” header auto-added by Spring.

✅ **Real-time Example:**

* Used in browsers during **CORS preflight requests**.

**🔹 8. TRACE – Request Debugging**

@RequestMapping(value = "/trace", method = RequestMethod.TRACE)

public ResponseEntity<String> traceCustomer() {

return new ResponseEntity<>("FROM TRACE - traceCustomer()", HttpStatus.OK);

}

✅ **Use Case:**

* Used for **debugging** – the server echoes the request it received.
* Rarely used in production (can be disabled for security).

✅ **HTTP Status:**

* HttpStatus.OK (200)

**🧩 About ResponseEntity**

ResponseEntity<T> → A wrapper to build complete HTTP responses.  
It allows you to set:

* Response **Body** (e.g., message or data)
* **Status Code** (OK, CREATED, etc.)
* **Headers** (optional)

Example:

return new ResponseEntity<>("Success", HttpStatus.OK);

**✅ Summary Table**

| **HTTP Method** | **Purpose** | **Typical Status** | **Example Action** |
| --- | --- | --- | --- |
| GET | Read data | 200 OK | Get customer info |
| POST | Create new | 201 Created | Register new customer |
| PUT | Full update | 200 OK | Update all fields |
| PATCH | Partial update | 200 OK | Update one field |
| DELETE | Delete resource | 204 No Content | Remove record |
| HEAD | Headers only | 200 OK | Check resource availability |
| OPTIONS | Supported methods | 200 OK | CORS preflight check |
| TRACE | Debug request | 200 OK | Request echo (for testing) |

Eg: HttpMethods