# 10. Spring REST Annotations

# ✅ @RestController

## ✅ What It Does

* When you mark a class with @RestController, you are telling Spring:

“This class is a **REST API component**. It will accept HTTP requests (like GET, POST, PUT, DELETE) and give back **raw data** (JSON, XML, or text).”

So the class becomes a **distributed component** (it can be called by any client over the network).

## ✅ Technical Meaning

* @RestController = @Controller + @ResponseBody

1. **@Controller**
   * Used in Spring MVC.
   * Maps URLs to Java methods.
   * But normally, it returns **view names** (like home.jsp, index.html).
2. **@ResponseBody**
   * Tells Spring to send the **return value directly as raw data** in the HTTP response body.
   * No view resolution happens.

So combining them →

* **@RestController** means:
  + Handle requests like a controller,
  + But always return **raw data** (JSON, XML, text).

## ✅ When to Use

* **@Controller** → For web applications (JSP/Thymeleaf pages).
* **@RestController** → For REST APIs (data services like JSON/XML).

## ✅ Difference Between @Controller and @RestController

| **Annotation** | **What it Returns** | **Use Case Example** |
| --- | --- | --- |
| @Controller | Returns **viewName** (like home.jsp) | Web applications with JSP/Thymeleaf |
| @RestController | Returns **raw data** (like JSON/XML) | RESTful APIs for B2B/B2C systems |

👉 Example:

@Controller

public class WebController {

@GetMapping("/home")

public String homePage() {

return "home"; // View resolver finds home.jsp

}

}

@RestController

public class ApiController {

@GetMapping("/greet")

public String greet() {

return "Hello REST!"; // Directly returns raw text

}

}

# ✅ Binding Methods to HTTP Requests

## ✅ What Happens

* In REST, every HTTP request (GET, POST, PUT, DELETE) should be handled by a **specific method** in the controller.
* Spring Boot provides **mapping annotations** that directly bind Java methods to these HTTP requests.
* Each request also has a **unique URL pattern** (endpoint), so the server knows which method to call.

## ✅ Mapping Annotations in REST APIs

1. **@GetMapping** → Handles **GET** requests
   * Used to **fetch** data.
   * Example: fetching a product by its ID.
2. **@PostMapping** → Handles **POST** requests
   * Used to **create** new data.
   * Example: adding a new product.
3. **@PutMapping** → Handles **PUT** requests
   * Used to **update** existing data.
   * Example: updating a product by ID.
4. **@DeleteMapping** → Handles **DELETE** requests
   * Used to **remove** data.
   * Example: deleting a product by ID.

## ✅ Example REST Controller

@RestController

@RequestMapping("/products")

public class ProductController {

// GET - fetch product by id

@GetMapping("/{id}")

public String getProduct(@PathVariable int id) {

return "Fetching product with ID: " + id;

}

// POST - create new product

@PostMapping

public String addProduct(@RequestBody String product) {

return "Product added: " + product;

}

// PUT - update product

@PutMapping("/{id}")

public String updateProduct(@PathVariable int id, @RequestBody String product) {

return "Product updated with ID " + id + ": " + product;

}

// DELETE - remove product

@DeleteMapping("/{id}")

public String deleteProduct(@PathVariable int id) {

return "Product deleted with ID: " + id;

}

}

👉 Key Points in the Example:

* @RequestMapping("/products") → Base URL for all methods in this controller.
* /{id} → Dynamic path variable (e.g., /products/10).
* @PathVariable → Reads values from URL.
* @RequestBody → Reads data from request body (JSON, text, XML).

## ✅ Configuring Properties

Spring Boot runs on an **embedded Tomcat server**.

* Default port = **8080**
* We can change it in application.properties or application.yml.

👉 Example (application.properties):

server.port=9999

👉 Example (application.yml):

server:

port: 9999

So now, instead of:

http://localhost:8080/products/10

You’ll use:

http://localhost:9999/products/10

## ✅ Simple Analogy

* **@RestController** → Like a **delivery counter** in a restaurant.
  + You ask for food (request) → They give you food directly in a box (raw data like JSON/XML).
* **@Controller** → Like a **dining hall** in a restaurant.
  + You ask for food (request) → They serve it decorated on a plate at your table (HTML/JSP page).

# 🔑 Final Wrap-Up

1. @RestController = @Controller + @ResponseBody → Always returns raw data (JSON/XML/Text).
2. @Controller → Used for web apps, returns **view names** (JSP/Thymeleaf pages).
3. **HTTP Method Annotations**:
   * @GetMapping → Fetch data
   * @PostMapping → Create data
   * @PutMapping → Update data
   * @DeleteMapping → Remove data
4. **Configuring server.port** → Change default port (8080) in application.properties or application.yml.
5. REST APIs in Spring Boot are **lightweight, easy to build, and run without extra jars** (just spring-boot-starter-web).

# 11. Concluding Notes on REST

## ✅ RESTful API = RESTful Service

* Both words mean the **same thing**.
* REST is just an **architecture style** (a set of principles).
* When we implement those principles in real code, we call it either a **RESTful API** or a **RESTful Service**.
* So don’t get confused — they are interchangeable terms.

👉 Example:

http://localhost:9999/welcome/msg

If you run this in a browser, you get:

Welcome to Ineuron

Breakdown:

* localhost:9999 → Your server running on port 9999.
* /welcome/msg → The REST endpoint (controller method).
* Response → **Raw Data** (not a JSP/HTML page).

## ✅ Output of RESTful API is Always Raw Data

* REST APIs do not return **views** (JSP, Thymeleaf, HTML pages).
* They return **data only** → which clients (like browsers, mobile apps, or other systems) can use.
* This raw data can be in different formats:

1. **Plain Text**
2. Welcome to Ineuron
3. **JSON**
4. { "message": "Welcome to Ineuron" }
5. **XML**
6. <message>Welcome to Ineuron</message>

👉 Example REST Controller:

@RestController

public class WelcomeController {

@GetMapping("/welcome/msg")

public String welcomeMsg() {

return "Welcome to Ineuron"; // raw text response

}

}

When you visit →  
http://localhost:9999/welcome/msg

You see:

Welcome to Ineuron

## ✅ Testing REST APIs (Postman Tool)

* Browsers are good only for **GET requests**.
* But REST supports **POST, PUT, DELETE** → these can’t be tested properly with just a browser.
* For this, developers use **Postman**.

👉 With Postman you can:

* Send GET, POST, PUT, DELETE requests.
* Add **headers** (like Authorization token).
* Add **body data** (JSON/XML).
* See raw responses from the server.

👉 How to open Postman:

* Older Chrome version → chrome://apps (if installed via Chrome Web Store).
* Modern practice → Download the **standalone Postman desktop app** (recommended).

## ✅ Simple Analogy

Think of REST API as a **juice machine**:

* You press a button (send a request).
* It gives you **raw juice** (JSON/XML data).
* It does not serve you on a decorated plate with fruit slices (that’s what **MVC with JSP** does — prepares a full decorated view).

# 🔑 Final Wrap-Up

1. **RESTful API = RESTful Service** → both mean the same.
2. REST always returns **raw data**, not web pages.
3. Data formats supported → **Text, JSON, XML, YAML, etc.**
4. For testing, **use Postman**, especially for POST/PUT/DELETE requests.
5. In Spring Boot, building REST APIs is **simple** → just add @RestController and mapping annotations.

👉 This is the **last and most important takeaway**:  
REST is about **resources and raw data exchange**, while Spring Boot makes it **easy and fast** to implement these principles.

# 1. What is a RestResource?

* A **RestResource** is like a **service provider** in a system where many applications talk to each other (distributed application).
* It is a **class or object** in your program that **shares business work (services/logic)** with others through the internet/network using **REST rules**.

### Example:

If you create a service that gives **student details** through a link like /students/1, then that service class (controller) is a **RestResource**.

👉 **Easy way to remember:**  
RestResource = **Shopkeeper** who keeps goods (business services) and gives them when someone asks.

# 2. What is a RestClient?

* A **RestClient** is the one who **uses or consumes** the services given by the RestResource.
* The RestClient **sends a request** (GET, POST, PUT, DELETE) and then gets back an **answer (response)**.

### RestClient can be:

* A **browser** (when you type a web link/URL).
* Tools like **Postman** or **SoapUI**.
* Another **application/program**.
* Even a **human being**, because we send the request using a tool or browser.

👉 **Easy way to remember:**  
RestClient = **Customer** who goes to the shopkeeper and asks for an item (sends request to get services).

# 3. Method-level URL patterns in REST Controllers

When we create REST methods, we use URL patterns to connect them to web links.

### Case A: Writing a URL pattern

@GetMapping("/welcome")

public String welcomeMessage() {

return "Welcome to Restful Services from Ineuron ... ";

}

* This method will run when someone sends a request to: **GET /welcome**.
* Here, /welcome is the **endpoint** (the unique web address for that service).

### Case B: No URL pattern at method level

@GetMapping

public String welcomeMessage() {

return "Welcome to Restful Services from Ineuron ... ";

}

* Here, the method is linked only to **GET request**, but there is no path written.
* In this case:
  + If there is a **class-level URL**, that will be used.
  + If there is no class-level URL, then the method will be connected to the **root URL /**.

👉 **Important Note:**  
If you write **two methods without URL patterns**, then both methods will try to use the same URL.

### Example:

@GetMapping

public String method1() { return "Hello"; }

@GetMapping

public String method2() { return "Hi"; }

* Both methods are connected to the same URL (/).
* Now, Spring cannot decide which one to run.
* This causes **confusion/ambiguity** and will throw an error.

✅ That is why we must always give at least **one unique URL pattern**.

# 4. Why should we write both class-level and method-level mappings?

### Example:

@RestController

@RequestMapping("/api")

public class MyController {

@GetMapping("/welcome")

public String welcome() {

return "Welcome!";

}

@GetMapping("/greet")

public String greet() {

return "Good Afternoon";

}

}

👉 **Final URLs:**

* /api/welcome
* /api/greet

### Advantages of using both mappings:

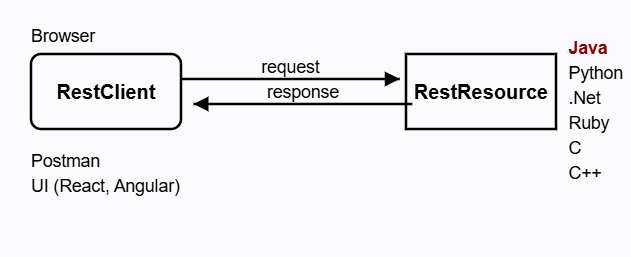
1. **Uniqueness** – Each method has its own clear path. No conflicts.
2. **Clarity** – It becomes easy to read and understand the structure.
3. **Modularity (Grouping)** – You can group similar services under one common class.
   * Example: @RequestMapping("/students") → all methods inside are related to **students**.
4. **Scalability (Growth)** – When your project grows bigger, it becomes simple to manage and maintain.

👉 **Easy analogy:**

* **Class-level mapping** = **Building name** (e.g., “Sunrise Apartments”).
* **Method-level mapping** = **Flat number** (e.g., Flat 101, Flat 102).
* Together = **Unique full address** (e.g., Sunrise Apartments, Flat-101).

# 📌 Notes (Quick Revision in Simple Words)

* **RestResource** = Service provider (like shopkeeper).
* **RestClient** = Service consumer (like customer).
* If no URL pattern is given at method level:
  + Class-level mapping (if present) is used.
  + If no class-level mapping → it connects to root /.
* **Ambiguity issue**: Happens when two or more methods don’t have their own URL patterns (Spring won’t know which one to run).
* **Best practice**: Always write both **class-level mapping** and **method-level mapping** to keep URLs **clear, unique, and well-organized**.



### Diagram Explanation

1. **RestClient (Left side)**
   * This is the **consumer** or **user** who sends requests.
   * Examples of RestClients:
     + **Browser** (when you type a URL).
     + **Postman** (a tool for testing APIs).
     + **UI Applications** built with **React, Angular**, etc.

👉 RestClient = **Customer** who goes to the shopkeeper and asks for goods.

1. **RestResource (Right side)**
   * This is the **service provider** or **server** that answers the request.
   * It can be built in **any programming language** like:
     + Java
     + Python
     + .Net
     + Ruby
     + C
     + C++

👉 RestResource = **Shopkeeper** who provides goods (services) when asked.

1. **Request →**
   * The **RestClient sends a request** (example: GET /students/1).
   * The request goes **from left (RestClient) to right (RestResource)**.
   * Request usually contains information like:
     + Which service you want (URL).
     + What type of action you want (GET, POST, PUT, DELETE).
     + Sometimes extra data (like JSON in the body).

👉 Example: A customer asks the shopkeeper: “Please give me 1kg rice.”

1. **Response ←**
   * The **RestResource prepares an answer** and sends it back.
   * The response goes **from right (RestResource) to left (RestClient)**.
   * Response usually contains:
     + The requested data (example: student details in JSON format).
     + Or a success message (like "Student saved successfully").
     + Or an error (if something goes wrong).

👉 Example: The shopkeeper gives the customer the **rice packet** as the response.

### Putting it All Together

* **RestClient** (Customer/browser/Postman/UI app) **asks** for something → (Request).
* **RestResource** (Shopkeeper/server application in Java, Python, etc.) **provides** it → (Response).

📌 **Simple Analogy:**

* Customer = RestClient.
* Shopkeeper = RestResource.
* Customer’s request = “Give me rice” (Request).
* Shopkeeper’s answer = “Here is your rice” (Response).