What is rest in rest api

Good question 👍 Let's carefully unpack **what REST means in REST API**.

🔹 REST = Representational State Transfer

It is an **architectural style** for designing APIs over HTTP.

- **Representational** → You don't send the actual resource (like a restaurant), you send a **representation** (JSON, XML, etc.) of it.
- **State Transfer** → When the client interacts with the server, the "state" of the resource is transferred (for example, getting restaurant details, updating an order, etc.).

🔹 Key Principles of REST

1. **Client-Server** → Client (Swiggy app) and Server (Swiggy backend) are separate.
2. **Stateless** → Each request from the client contains all the information needed (no server memory of previous requests).
3. **Uniform Interface** → Standard HTTP methods are used (GET, POST, PUT, DELETE).
4. **Representations** → Resources (like `Restaurant`, `Order`) are represented as JSON/XML, not Java objects directly.
5. **Resource-Based** → Everything is treated as a "resource" with a unique URL.

What is rest in rest api

Good question 👍 Let's carefully unpack **what REST means in REST API**.

🔹 REST = Representational State Transfer

It is an **architectural style** for designing APIs over HTTP.

- **Representational** → You don't send the actual resource (like a restaurant), you send a **representation** (JSON, XML, etc.) of it.
- **State Transfer** → When the client interacts with the server, the "state" of the resource is transferred (for example, getting restaurant details, updating an order, etc.).

🔹 Key Principles of REST

1. **Client-Server** → Client (Swiggy app) and Server (Swiggy backend) are separate.
2. **Stateless** → Each request from the client contains all the information needed (no server memory of previous requests).
3. **Uniform Interface** → Standard HTTP methods are used (GET, POST, PUT, DELETE).
4. **Representations** → Resources (like `Restaurant`, `Order`) are represented as JSON/XML, not Java objects directly.
5. **Resource-Based** → Everything is treated as a "resource" with a unique URL.

🔹 REST API in a Swiggy Example 🍔

- **Resource:** Restaurant
- **URL (Resource Identifier):**
  `https://swiggy.com/api/restaurants`
- **Operations (HTTP Methods):**
  - `GET /restaurants` → Get list of restaurants.
  - `GET /restaurants/101` → Get details of restaurant with ID=101.
  - `POST /orders` → Place a new order.
  - `PUT /orders/55` → Update order with ID=55.
  - `DELETE /orders/55` → Cancel order with ID=55.

Here:

- "Restaurant" and "Order" are **resources**.
- JSON returned to the app is the **representation** of those resources.

✅ **In Short:**

- **REST** is a set of rules for how web services should be built.
- **REST API** is an API that follows these REST rules.
- Data is exchanged as **representations (usually JSON)** via **HTTP methods**.

🔹 REST API in a Swiggy Example 🍔

- **Resource:** Restaurant
- **URL (Resource Identifier):**
  `https://swiggy.com/api/restaurants`
- **Operations (HTTP Methods):**
  - `GET /restaurants` → Get list of restaurants.
  - `GET /restaurants/101` → Get details of restaurant with ID=101.
  - `POST /orders` → Place a new order.
  - `PUT /orders/55` → Update order with ID=55.
  - `DELETE /orders/55` → Cancel order with ID=55.

Here:

- "Restaurant" and "Order" are **resources**.
- JSON returned to the app is the **representation** of those resources.

✅ **In Short:**

- **REST** is a set of rules for how web services should be built.
- **REST API** is an API that follows these REST rules.
- Data is exchanged as **representations (usually JSON)** via **HTTP methods**.

What is rest in RestController annotation

Great follow-up 👍

Let's break down what `Rest` **means in** `@RestController`.

🔹 `@RestController` in Spring Boot

- `@RestController` is a **specialized version** of `@Controller`.
- It is basically:

```java
@RestController = @Controller + @ResponseBody
```

Copy Edit

- Meaning: every method in that class returns **data directly in the HTTP response body**, instead of resolving to a view (like JSP/Thymeleaf).

🔹 Why the word REST?

The **"Rest"** in `@RestController` comes from **RESTful Web Services** (Representational State Transfer).

- A class annotated with `@RestController` is meant to **build REST APIs**.
- These APIs typically return **JSON/XML representations** of resources.
- Instead of returning HTML pages (as in MVC `@C oller`), it returns raw data that clients (mobile apps, browsers, other services) consume.

What is rest in RestController annotation

Great follow-up 👍

Let's break down what `Rest` **means in** `@RestController`.

🔹 `@RestController` in Spring Boot

- `@RestController` is a **specialized version** of `@Controller`.
- It is basically:

```java
@RestController = @Controller + @ResponseBody
```

Copy Edit

- Meaning: every method in that class returns **data directly in the HTTP response body**, instead of resolving to a view (like JSP/Thymeleaf).

🔹 Why the word REST?

The **"Rest"** in `@RestController` comes from **RESTful Web Services** (Representational State Transfer).

- A class annotated with `@RestController` is meant to **build REST APIs**.
- These APIs typically return **JSON/XML representations** of resources.
- Instead of returning HTML pages (as in MVC `@C oller`), it returns raw data that clients (mobile apps, browsers, other services) consume.

🔹 Example

java Copy Edit

```java
@RestController
@RequestMapping("/api")
public class RestaurantController {

    @GetMapping("/restaurant")
    public Restaurant getRestaurant() {
        return new Restaurant(1, "Pizza Palace");
    }
}
```

👉 Here, `getRestaurant()` returns a **Java object**.

👉 `@RestController` ensures it is **automatically converted to JSON** and sent in the HTTP response.

✅ **In short:**

- The "Rest" in `@RestController` signals that the controller is designed for **RESTful APIs**.
- Instead of returning views, it returns **resource data** (usually JSON) directly.

🔹 Example

java Copy Edit

```java
@RestController
@RequestMapping("/api")
public class RestaurantController {

    @GetMapping("/restaurant")
    public Restaurant getRestaurant() {
        return new Restaurant(1, "Pizza Palace");
    }
}
```

👉 Here, `getRestaurant()` returns a **Java object**.

👉 `@RestController` ensures it is **automatically converted to JSON** and sent in the HTTP response.

✅ **In short:**

- The "Rest" in `@RestController` signals that the controller is designed for **RESTful APIs**.
- Instead of returning views, it returns **resource data** (usually JSON) directly.

If it return java object directly in http response ..then when it change in json by jackson

Super sharp question 👌 — you're digging into the **internals of Spring Boot**.
Let's walk through step by step.

🔹 Flow of a `@RestController` Response

1. **You return a Java object** from your controller method.

```java
@GetMapping("/restaurant")
public Restaurant getRestaurant() {
    return new Restaurant(1, "Pizza Palace");
}
```

Copy Edit

👉 Here you are just returning a **plain Java object**, not JSON.

---

2. **DispatcherServlet** (the front controller) gets the return value.
   - It doesn't know yet if it should become JSON, XML, or something else.
   - It asks the **HandlerAdapter** to process it.

3. **HandlerAdapter → HttpMessageConverter**
   - Spring Boot looks at:
     - The `Accept` **header** from the client request (`application/json` , `application/xml` , etc.).
     - The **return type** (Java object).
   - Based on this, it picks an appropriate **HttpMessageConverter**.

---

4. **Jackson's MappingJackson2HttpMessageConverter** (default for JSON)
   - If the client asked for JSON (`Accept: application/json`), Spring delegates to **Jackson**.
   - Jackson takes the Java object and converts it into a JSON string.

```json
{
  "id": 1,
  "name": "Pizza Palace"
}
```

---

5. **HTTP Response**
   - The JSON string is written directly to the **HTTP response body**.
   - The client (e.g., Postman, browser, or Swiggy app) receives pure JSON.

🔹 Where does Jackson come in?

Jackson comes in **after your controller method returns** the Java object, but **before Spring writes to the HTTP response**.

It's invoked inside the **HttpMessageConverter layer**.

✅ So in simple terms:

- You return Java object → Spring hands it to `HttpMessageConverter`.
- `HttpMessageConverter` uses **Jackson** to turn it into JSON.
- JSON is sent in the HTTP response body.

🔹 What is `@RestController`?

- `@RestController` is a **specialized version of** `@Controller` in Spring.
- It tells Spring:
  1. **This class is a Controller** (handles HTTP requests).
  2. **Return values of methods should be written directly into the HTTP response body** (instead of rendering a view like JSP/Thymeleaf).

👉 It is basically `@Controller + @ResponseBody` combined.

🔹 Example

java Copy Edit

```java
@RestController
@RequestMapping("/api/hello")
public class HelloController {

    @GetMapping
    public String sayHello() {
        return "Hello, Abhishek! 👋";
    }
}
```

What happens here?

- Client hits: `GET http://localhost:8080/api/hello`
- `sayHello()` returns a **String**.
- Because of `@RestController`, Spring writes `"Hello, Abhishek! 👋"` **directly in the HTTP response body**.
- No ViewResolver / JSP / Thymeleaf needed.

🔹 Real-World Example (Swiggy 🍔)

java Copy Edit

```java
@RestController
@RequestMapping("/api/restaurants")
public class RestaurantController {

    private final RestaurantService service;

    public RestaurantController(RestaurantService service) {
        this.service = service;
    }

    @GetMapping
    public List<Restaurant> getRestaurants() {
        return service.getAllRestaurants();
    }
}
```

- **Client:** Mobile app calls `GET /api/restaurants`.
- **Controller:** Runs `getRestaurants()`.
- **Return:** List of restaurants → automatically converted to JSON by **Jackson**.

Response looks like:

```json
[
  {"id": 1, "name": "Pizza Palace"},
  {"id": 2, "name": "Biryani House"}
]
```

✅ In Short

- `@RestController = @Controller + @ResponseBody`.
- Used in REST APIs (JSON/XML output).
- No **ViewResolver** needed → perfect for APIs.

🔹 What is `@ResponseBody`?

- `@ResponseBody` tells Spring MVC that the **return value of a method** should be **written directly into the HTTP response body**, instead of being resolved as a **view name**.
- It is used on methods inside a `@Controller` class.

👉 Without it, Spring thinks your method is returning the name of a view (like `"home.jsp"`).

🔹 Example 1: Without `@ResponseBody`

java Copy Edit

```java
@Controller
public class HelloController {

    @GetMapping("/hello")
    public String sayHello() {
        return "hello";  // interpreted as view name (hello.jsp or hello.html)
    }
}
```

Here Spring will try to resolve `hello` as a **view template**, not as plain text.

🔹 Example 3: Returning JSON

java Copy Edit

```java
@Controller
public class RestaurantController {

    @GetMapping("/restaurant")
    @ResponseBody
    public Restaurant getRestaurant() {
        return new Restaurant(1, "Pizza Palace");
    }
}
```

Response:

json Copy Edit

```json
{
  "id": 1,
  "name": "Pizza Palace"
}
```

👉 Spring Boot uses **Jackson** under the hood to automatically convert Java objects to JSON when `@ResponseBody` is present.

🔹 How `@RestController` Works

- When you annotate a class with `@RestController`:
  - Every method in that class returns data **directly in the HTTP response body**.
  - Spring Boot automatically uses **HTTP Message Converters** (like Jackson) to turn Java objects into **JSON** (or XML, if requested).

🔹 Example

java

Copy Edit

```java
@RestController
@RequestMapping("/api")
public class RestaurantController {

    @GetMapping("/restaurant")
    public Restaurant getRestaurant() {
        return new Restaurant(1, "Pizza Palace");
    }
}
```

Response to Client (Browser/Postman/Swiggy app):

json

Copy Edit

```json
{
  "id": 1,
  "name": "Pizza Palace"
}
```

👉 No JSP, Thymeleaf, or HTML view involved.

👉 Data goes straight from Java → JSON → HTTP response → Client.

@RestController class method return directly java object

Yes ✅ you're spot on — a method in a `@RestController` class **returns a Java object directly**, and then **Spring Boot does the work of converting that Java object into JSON (or XML, etc.) before sending it in the HTTP response.**

`@GetMapping` is a **shortcut annotation** in Spring MVC/Spring Boot used to **map HTTP GET requests** onto specific controller methods.

It is equivalent to:

```java
@RequestMapping(method = RequestMethod.GET, value = "/path")
```

So basically:

👉 `@GetMapping("/path")` = `@RequestMapping(value="/path", method=RequestMethod.GET)`

🔹 Why GET?

- HTTP **GET** is used to **fetch/read data** from the server.
- It should **not modify** data (idempotent).

Example

java

Copy Edit

```java
@RestController
@RequestMapping("/api")
public class RestaurantController {

    // Maps GET request at /api/restaurant
    @GetMapping("/restaurant")
    public Restaurant getRestaurant() {
        return new Restaurant(1, "Pizza Palace");
    }

    // Maps GET request at /api/restaurants
    @GetMapping("/restaurants")
    public List<Restaurant> getAllRestaurants() {
        return List.of(
            new Restaurant(1, "Pizza Palace"),
            new Restaurant(2, "Burger Hub")
        );
    }
}
```

What happens here:

1. Client calls → `GET /api/restaurant`
   - Controller returns **Java object** `Restaurant`.
   - Spring → `HttpMessageConverter` → Jackson → JSON in response.
2. Client calls → `GET /api/restaurants`
   - Controller returns a **List of objects**.
   - Spring automatically converts the list into a JSON array.

🔹 Real World (Swiggy Example 🍔)

- `GET /api/restaurants` → fetches list of restaurants near you.
- `GET /api/restaurants/101` → fetches details of restaurant with ID = 101.

👉 Both would be handled with `@GetMapping`.

✅ **In short:**

`@GetMapping` = handles **HTTP GET requests** and maps them to controller methods, usually to return **data (JSON)** to the client.

🔹 `@GetMapping` Attributes

The `@GetMapping` annotation is **meta-annotated** with `@RequestMapping`, so it inherits most of the same attributes.

Here are the key ones:

1. `value` / `path`

- Defines the **URI path(s)** for the GET request.
- Both `value` and `path` do the same thing.

java Copy Edit

```java
@GetMapping("/restaurants")
@GetMapping(path = "/restaurants")   // same as above
```

---

2. `params`

- Restricts mapping based on **HTTP request parameters**.

```java
@GetMapping(value = "/restaurants", params = "cuisine=italian")
public String getItalianRestaurants() {
    return "Only Italian restaurants";
}
```

👉 Matches only when query param `?cuisine=italian` is present.

---

3. `headers`

- Restricts mapping based on **HTTP headers**.

```java
@GetMapping(value = "/restaurants", headers = "X-API-VERSION=1")
public String getRestaurantsV1() {
    return "API version 1";
}
```

4. `consumes`

- Defines what **Content-Type** the method can consume (request body).
  👉 Rarely used with `GET` because GET usually has no body.

```java
@GetMapping(value = "/data", consumes = "application/json")
public String consumeJson() {
    return "Consumes JSON input";
}
```

---

5. `produces`

- Defines what **Content-Type** the method produces (response).

```java
@GetMapping(value = "/restaurants", produces = "application/json")
public List<Restaurant> getRestaurants() { ... }

@GetMapping(value = "/restaurants", produces = "application/xml")
public List<Restaurant> getRestaurantsXml() { ... }
```

👉 Useful for returning JSON **or** XML depending on client request.

🔹 Example (with multiple attributes)

java Copy Edit

```java
@GetMapping(
    value = "/restaurants",
    params = "city=delhi",
    headers = "X-API-VERSION=2",
    produces = "application/json"
)
public List<Restaurant> getDelhiRestaurantsV2() {
    return List.of(new Restaurant(1, "Delhi Dhaba"));
}
```

👉 This method will only respond if:

- URL contains `?city=delhi`
- Header has `X-API-VERSION=2`
- Client accepts `application/json`

Summary:

Main attributes of `@GetMapping`:

- `value` / `path` → URI
- `params` → request parameters
- `headers` → request headers
- `consumes` → expected input type
- `produces` → response type

When we use the path attribute in @GetMapping .... What is checked by the dispatcherservlet and which controller methods run ...And what get http request checked with path

Excellent 🔥 — you're now asking **how DispatcherServlet decides which method to call when we use `path` (or `value`) inside `@GetMapping`.**

Let's go step by step:

🔹 What happens when a request comes in?

1. **Client sends HTTP request**

   Example:

   ```bash
   GET /restaurants
   ```

   Copy Edit

   👉 The **path** here is `/restaurants`.

2. **Tomcat receives the HTTP request**
   - Since Spring Boot runs on an embedded Tomcat, the request is passed to **DispatcherServlet** (the front controller).

---

3. **DispatcherServlet checks HandlerMapping**
   - Spring maintains a **HandlerMapping registry** (a big map of URL paths → controller methods).
   - Example mapping table:

Path (from `@GetMapping`)	HTTP Method	Controller Method
`/restaurants`	GET	`getRestaurants()`
`/restaurants/{id}`	GET	`getRestaurantById(Long id)`
`/orders`	POST	`createOrder()`

---

4. **DispatcherServlet matches request path**
   - It takes the path from the HTTP request (`/restaurants`)
   - Compares it against registered mappings (`@GetMapping(path="/restaurants")`).
   - Also checks **HTTP method** (must be GET for `@GetMapping`).

👉 If both path **and** method match → that controller method is chosen.

5. **Matched method runs**

   Example:

```java
@RestController
class RestaurantController {
    @GetMapping(path="/restaurants")
    public List<Restaurant> getRestaurants() {
        return List.of(new Restaurant(1, "Pizza Palace"));
    }
}
```

   - If client calls `GET /restaurants`, DispatcherServlet chooses `getRestaurants()`.
   - If client calls `POST /restaurants`, this method **won't run** (since method mismatch).

🔹 Real Swiggy Example 🍔

- **Request:** `GET /api/restaurants`
  - Path = `/api/restaurants`
  - HTTP Method = GET
- DispatcherServlet looks into mappings and finds:

```java
@GetMapping(path="/api/restaurants")
public List<Restaurant> getAllRestaurants() { ... }
```

✅ Match found → method runs → returns JSON list of restaurants.

🔹 In short:

- `path` / `value` in `@GetMapping` tells DispatcherServlet *which URL this method listens to.*
- DispatcherServlet checks the **request's URL path + HTTP method**.
- If they match → that controller method is executed.

Dynamic URL Paths to get the data in the url

Perfect 🚀 — you're asking about **dynamic URL paths** in Spring Boot (often called **Path Variables**) so you can pass data inside the URL itself.

🔹 What is a Dynamic URL Path?

Instead of always using fixed URLs like:

```bash
GET /restaurants
```

You sometimes want to pass **data inside the URL** like:

```bash
GET /restaurants/5
```

Here `5` is the **restaurant ID**.

🔹 How to Handle in Spring Boot

Spring provides `@PathVariable` for this.

Example

java Copy code

```java
@RestController
@RequestMapping("/restaurants")
public class RestaurantController {

    // GET /restaurants/5
    @GetMapping("/{id}")
    public String getRestaurantById(@PathVariable int id) {
        return "Restaurant with ID = " + id;
    }
}
```

👉 If client calls `GET /restaurants/5`

- DispatcherServlet matches `/{id}`
- Extracts `id = 5`
- Passes it into method parameter

Response:

csharp Copy code

```
Restaurant with ID = 5
```

🔹 Multiple Dynamic Parts

java

Copy code

```java
// GET /restaurants/5/menu/10
@GetMapping("/{restaurantId}/menu/{itemId}")
public String getMenuItem(
        @PathVariable int restaurantId,
        @PathVariable int itemId) {
    return "Restaurant " + restaurantId + ", Menu Item " + itemId;
}
```

Call → `GET /restaurants/5/menu/10`

Response →

mathematica

Copy code

```mathematica
Restaurant 5, Menu Item 10
```

🔹 What is `@PathVariable`?

- `@PathVariable` is used to **extract values from the URL path itself**.
- It maps a **dynamic part of the URL** (like `/users/10`) to a **method parameter** in your controller.
- Works with `@GetMapping`, `@PostMapping`, etc.

🔹 When to Use `@PathVariable`

✅ Use when the value is a **unique identifier** or part of the **resource's identity**.

Examples:

- `/restaurants/5` → Get restaurant with id = 5
- `/orders/123/status` → Get status of order 123
- `/users/42/profile` → Get profile of user 42

❌ Don't use it for **filtering or optional data** (that's better with `@RequestParam` like `/restaurants?city=Delhi`).

🔹 Key Points about `@PathVariable`

1. **Binding:** It binds URL template variables (`/{id}`) to method parameters.
2. **Mandatory by default** → if missing, you get `404 Not Found` .
   - Use `required=false` if optional (but usually for optional values, `@RequestParam` is better).
3. **Works with any HTTP method** → `@GetMapping` , `@PostMapping` , etc.
4. **Type Conversion:** Spring automatically converts from String → int, long, etc.
   - If it fails (e.g., `/restaurants/abc` but expecting `int`), you get `400 Bad Request` .

🔹 REST Best Practice

- **Use PathVariable** for **resource identifiers** (`/users/5` , `/orders/123`)
- **Use RequestParam** for **filtering, searching, pagination, optional parameters** (`/restaurants?city=Delhi&page=2`).

🔹 Example 3: Custom Variable Names

Sometimes your path variable name ≠ method parameter name.

java Copy code

```java
// GET /orders/123
@GetMapping("/orders/{orderId}")
public String getOrder(@PathVariable("orderId") String id) {
    return "Order ID = " + id;
}
```

👉 Path part `{orderId}` maps to parameter `id`.

What `@RequestParam` is

Binds **query-string** or **form data** parameters to controller method arguments.

- Works with:
  - `GET /search?q=pizza&page=2` → `q`, `page` are query params
  - `POST` with `application/x-www-form-urlencoded` or `multipart/form-data`
- Not for JSON bodies — use `@RequestBody` for that.

When to use it (and when not)

Use `@RequestParam` for:

- Filters/search/pagination/sorting: `/restaurants?city=Delhi&page=2&size=20`
- Small scalar inputs (Strings, numbers, booleans, enums)
- Form fields and **file uploads** (`MultipartFile`)
- Optional knobs/toggles

Don't use it for:

- Resource identity: use `@PathVariable` (`/restaurants/42`)
- Complex nested data (JSON): use `@RequestBody`

Core attributes

java Copy code

```java
@GetMapping("/restaurants")
public List<Restaurant> list(
  @RequestParam(name="city") String city,          // 1) name/value
  @RequestParam(required=false) String cuisine,    // 2) required (default true)
  @RequestParam(defaultValue="1") int page,        // 3) defaultValue -> also makes it not required
  @RequestParam(defaultValue="20") int size
) { ... }
```

1. **name/value** (alias): the query key. If omitted, Spring uses the **parameter name**.
2. **required**: default `true`. Missing param ⇒ **400 Bad Request**.
3. **defaultValue**: sets a fallback value **and** implicitly sets `required=false`.

> Tip: If using `required=false`, prefer wrapper types (`Integer`, `Boolean`) so `null` is possible.

2) Custom param name

sql

Copy code

```
GET /search?q=pizza
```

java

Copy code

```
@GetMapping("/search")
public List<Item> search(@RequestParam("q") String keyword) { ... }
```

3) Collections & repeated params

bash

Copy code

```
GET /menu?ids=1&ids=2&ids=3
```

java

Copy code

```
@GetMapping("/menu")
public List<MenuItem> getMenu(@RequestParam List<Integer> ids) { ... }
// also works with Set<Integer>
```

4) Grab all params dynamically

java Copy code

```java
@GetMapping("/report")
public Map<String,String> report(@RequestParam Map<String,String> params) { ... }

// For repeated keys:
public MultiValueMap<String,String> report(@RequestParam MultiValueMap<String,String> params) { ..
```

5) Dates & formatting

vbnet Copy code

```vbnet
GET /orders?from=2025-08-01&to=2025-08-31
```

java Copy code

```java
@GetMapping("/orders")
public List<Order> orders(
  @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE) LocalDate from,
  @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE) LocalDate to
) { ... }
```

6) Booleans and enums

bash

Copy code

```
GET /restaurants?vegOnly=true&sort=RATING
```

java

Copy code

```
enum SortBy { NAME, RATING }
@GetMapping("/restaurants")
public List<Restaurant> list(
  @RequestParam(defaultValue="false") boolean vegOnly,
  @RequestParam(defaultValue="NAME") SortBy sort
) { ... }
```

7) Validation on params

java

Copy code

```
@Validated
@RestController
class RestaurantController {
  @GetMapping("/restaurants")
  public List<Restaurant> list(
    @RequestParam @Min(1) int page,
    @RequestParam @Max(100) int size,
    @RequestParam(required=false) @Pattern(regexp="^(name|rating)$") String sortBy
  ) { ... }
}
```

8) File upload (multipart form)

java

Copy code

```java
@PostMapping(value="/menu/upload", consumes=MediaType.MULTIPART_FORM_DATA_VALUE)
public String upload(@RequestParam("file") MultipartFile file) {
  return "Uploaded " + file.getOriginalFilename();
}
```

9) With `@PathVariable` together

bash

Copy code

```bash
GET /restaurants/42/menu?vegOnly=true
```

java

Copy code

```java
@GetMapping("/restaurants/{id}/menu")
public List<MenuItem> menu(
  @PathVariable long id,
  @RequestParam(defaultValue="false") boolean vegOnly
) { ... }
```

How binding & errors work

- Spring converts strings → target types (int, long, boolean, enum, `LocalDate`, etc.).
- If conversion fails (`page=abc` for `int`) ⇒ **400 Bad Request**.
- Missing required param ⇒ **400**.
- `defaultValue` avoids 400 by supplying a value.
- You can add **custom converters/formatters** if needed.

Best practices (quick checklist)

- **Identity in path**, **filters in query**: `/orders/123` + `?includeItems=true`.
- Provide **defaults** for page/size; cap size server-side.
- **Allowlist** sortable fields to avoid injection (`sort=name` / `rating` only).
- Keep query params **small and simple**; move complex data to a DTO with `@RequestBody`.
- Prefer **wrapper types** for optional params.

```
@GetMapping(path="/employees/{employeeId}")

public EmployeeDTO getEmployeeById(@PathVariable Long
employeeId) {}


@GetMapping(path="/employees/{employeeName}")

27

public EmployeeDTO
getEmployeeByIdd(@PathVariable(name="employeeName") String
Name){}
```

Is these both are considered same as when I am searching it by url in chrome it is showing error

Your code:

```java
@GetMapping(path="/employees/{employeeId}")
public EmployeeDTO getEmployeeById(@PathVariable Long employeeId) { ... }

@GetMapping(path="/employees/{employeeName}")
public EmployeeDTO getEmployeeByIdd(@PathVariable(name="employeeName") String name) { ... }
```

✅ What Spring sees

- Both mappings have the **same URL pattern**:

  ```bash
  /employees/{something}
  ```

  It doesn't matter whether you name it `{employeeId}` or `{employeeName}` — to Spring MVC, both are just **wildcards for one path segment**.
- So Spring can't distinguish them → **Ambiguous mapping exception** at startup (or strange runtime errors if somehow loaded).

That's why in Chrome/Postman when you hit `/emplo /27`, Spring doesn't know whether to call method 1 (expecting an ID) or method 2 (expecting a name). Hence the error.

🚩 Key point:

The **placeholder name** inside `{ }` is only for binding to the method parameter.
For the URL pattern itself, `{employeeId}` and `{employeeName}` are treated the same: one wildcard segment.

So the two methods are **considered the same mapping**.

✅ How to fix / differentiate

If you want **two separate endpoints** (one by ID, one by name), you must make the paths **different**:

java Copy code

```java
@GetMapping(path="/employees/id/{employeeId}")
public EmployeeDTO getEmployeeById(@PathVariable Long employeeId) { ... }

@GetMapping(path="/employees/name/{employeeName}")
public EmployeeDTO getEmployeeByName(@PathVariable String employeeName) { ... }
```

Now Spring can clearly distinguish:

- `/employees/id/27` → goes to the ID method
- `/employees/name/John` → goes to the name me

🔹 What is `@PostMapping`?

- `@PostMapping` is a Spring MVC annotation that handles **HTTP POST requests**.
- It is a **specialized version** of `@RequestMapping(method = RequestMethod.POST)`.
- Typically used to **create a new resource** on the server (e.g., adding an employee, placing an order, registering a user).

🔹 Example 1: Basic usage

java Copy code

```java
@RestController
@RequestMapping("/employees")
public class EmployeeController {

    @PostMapping
    public String createEmployee() {
        return "New employee created successfully!";
    }
}
```

👉 If you send a `POST` request to `/employees`, it will return:

```sql
New employee created successfully!
```

🔹 Example 2: Sending data in request body

Usually, POST requests carry data in the **HTTP request body** (JSON, XML, or form data).

Spring uses `@RequestBody` to read that data and bind it to a Java object.

java Copy code

```java
@PostMapping
public EmployeeDTO createEmployee(@RequestBody EmployeeDTO employee) {
    // Normally you'd save employee in DB here
    return employee;  // return the saved employee as response
}
```

👉 **Request (POST /employees):**

json Copy code

```json
{
  "id": 101,
  "name": "Alice",
  "department": "IT"
}
```

👉 **Response** (JSON automatically returned because of Jackson):

json Copy code

```
{
  "id": 101,
  "name": "Alice",
  "department": "IT"
}
```

🔹 Example 3: With Path + Body

java Copy code

```java
@PostMapping("/{department}")
public EmployeeDTO createEmployeeInDept(
        @PathVariable String department,
        @RequestBody EmployeeDTO employee) {
    employee.setDepartment(department);
    return employee;
}
```

👉 `POST /employees/HR` with body:

json Copy code

```json
{ "id": 102, "name": "Bob" }
```

👉 Response:

json Copy code

```json
{ "id": 102, "name": "Bob", "department": "HR" }
```

🔹 When to use `@PostMapping`

- When you want to **create** a new resource.
- When you need to send **sensitive or large data** in the request body.
- Example real-world APIs:
  - `POST /orders` → place a new order on Swiggy.
  - `POST /users/register` → create a new user.
  - `POST /employees` → add an employee in HR system.

🔹 Difference from `@GetMapping`

- `@GetMapping` → Fetch data (idempotent, safe, no side-effects).
- `@PostMapping` → Submit data / create new resource (has side-effects).

✅ Summary

- `@PostMapping` maps **HTTP POST** requests.
- Often combined with `@RequestBody` for JSON input.
- Used for **creating resources** in REST APIs.

🔹 What is `@PutMapping`?

- `@PutMapping` is a Spring MVC annotation that handles **HTTP PUT requests**.
- It's a **specialized shortcut** for:

```java
@RequestMapping(method = RequestMethod.PUT)
```

- In REST APIs, **PUT is used to update an existing resource** (replace it entirely).
- It is **idempotent** → calling it multiple times with the same data will always produce the same result.

🔹 Example 1: Basic PUT Mapping

java Copy code

```java
@RestController
@RequestMapping("/employees")
public class EmployeeController {

    @PutMapping("/{id}")
    public String updateEmployee(@PathVariable Long id) {
        return "Employee with ID " + id + " updated successfully!";
    }
}
```

👉 Request:

`PUT /employees/101`

👉 Response:

csharp Copy code

```
Employee with ID 101 updated successfully!
```

🔹 Example 2: Updating with JSON body

Usually, you pass new values in the **request body** with `@RequestBody`.

java Copy code

```java
@PutMapping("/{id}")
public EmployeeDTO updateEmployee(
        @PathVariable Long id,
        @RequestBody EmployeeDTO updatedEmployee) {

    updatedEmployee.setId(id);
    // Normally, save this updatedEmployee to the DB
    return updatedEmployee;
}
```

👉 Request:

`PUT /employees/101`

Body:

json Copy code

```json
{
  "name": "Alice Johnson",
  "department": "Finance"
}
```

👉 Response:

json

Copy code

```
{
  "id": 101,
  "name": "Alice Johnson",
  "department": "Finance"
}
```

🔹 Example 3: Difference between `@PutMapping` and `@PatchMapping`

- **PUT** → Replaces the **entire resource**.
  - If fields are missing in request, they may get overwritten with `null`.
- **PATCH** → Updates **partially** (only given fields).

Example:

- Employee = `{id:101, name:"Alice", dept:"IT"}`
- `PUT` request with body: `{name:"Bob"}` → may overwrite dept to `null`.
- `PATCH` request with body: `{name:"Bob"}` → only changes the name, dept remains `"IT"`.

🔹 Real-world examples

- Swiggy/Zomato: `PUT /users/123/address` → update your saved address.
- Banking app: `PUT /accounts/456` → update account details.
- HR system: `PUT /employees/789` → replace full employee info.

✅ Summary

- `@PutMapping` → maps HTTP **PUT** requests.
- Used to **update/replace existing resources**.
- Works with `@PathVariable` (to identify the resource) and `@RequestBody` (new data).
- **Idempotent** → multiple identical calls = same result.

🔹 What is Lombok?

- **Project Lombok** is a Java library that helps reduce **boilerplate code** like getters, setters, constructors, `toString()`, `equals()`, `hashCode()`, and builders.
- Instead of writing them manually, Lombok uses **annotations** to generate them **at compile time**.

👉 This makes your code **cleaner, shorter, and easier to maintain**.

🔹 How to use Lombok?

1. Add Lombok dependency in your `pom.xml` (Maven):

xml Copy code

```xml
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.32</version> <!-- latest version -->
    <scope>provided</scope>
</dependency>
```

For Gradle:

gradle Copy code

```
compileOnly 'org.projectlombok:lombok'
annotationProcessor 'org.projectlombok:lombok'
```

🔹 What is Lombok?

- **Project Lombok** is a Java library that helps reduce **boilerplate code** like getters, setters, constructors, `toString()`, `equals()`, `hashCode()`, and builders.
- Instead of writing them manually, Lombok uses **annotations** to generate them **at compile time**.

👉 This makes your code **cleaner, shorter, and easier to maintain**.

🔹 How to use Lombok?

1. Add Lombok dependency in your `pom.xml` (Maven):

xml Copy code

```xml
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.32</version> <!-- latest version -->
    <scope>provided</scope>
</dependency>
```

For Gradle:

gradle Copy code

```
compileOnly 'org.projectlombok:lombok'
annotationProcessor 'org.projectlombok:lombok'
```

2. Install the **Lombok plugin** in IntelliJ or Eclipse (so your IDE understands the generated code).

🔹 Common Lombok Annotations

1. `@Getter` and `@Setter`

Generates getters and setters automatically.

java Copy code

```java
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class Employee {
    private Long id;
    private String name;
}
```

👉 No need to manually write `getId()`, `setId()`, etc.

2. `@Data`

Generates:

- Getters
- Setters
- `toString()`
- `equals()` and `hashCode()`
- `RequiredArgsConstructor`

java Copy code

```java
import lombok.Data;

@Data
public class Employee {
    private Long id;
    private String name;
}
```

3. `@NoArgsConstructor`, `@AllArgsConstructor`, `@RequiredArgsConstructor`

Generates constructors automatically.

java Copy code

```java
import lombok.*;

@AllArgsConstructor
@NoArgsConstructor
public class Employee {
    private Long id;
    private String name;
}
```

---

4. `@Builder`

Creates a builder pattern for objects.

java Copy code

```java
import lombok.Builder;
import lombok.Data;

@Data
@Builder
public class Employee {
    private Long id;
    private String name;
    private String department;
}
```

👉 Usage:

java Copy code

```java
Employee emp = Employee.builder()
                       .id(101L)
                       .name("Alice")
                       .department("IT")
                       .build();
```

5. `@Value`

Makes a class **immutable** (all fields `final`, private, getters only).

java Copy code

```java
import lombok.Value;

@Value
public class Employee {
    Long id;
    String name;
}
```

🔹 Real-world use in Spring Boot

- DTOs (Data Transfer Objects)
- Entities (if not overriding equals/hashCode manually)
- Request/Response models in REST APIs
- Builder pattern for test data or complex objects

is during JSON body given to dispatcher during using @RequestBody ,
JSON body is converted into the java object by lombok

Ahh ⚡ great catch! This is where people often get confused between **Lombok** and **Jackson** in Spring Boot. Let's clear it step by step 👇

🔹 What happens with `@RequestBody`?

When you send a request like:

```http
POST /employees
Content-Type: application/json

{
  "id": 101,
  "name": "Alice",
  "department": "IT"
}
```

and your controller is:

```java
@PostMapping("/employees")
public Employee createEmployee(@RequestBody Employee employee) {
    return employee;
}
```

👉 **Process:**

1. The HTTP request comes into **DispatcherServlet**.
2. **Spring uses Jackson (a JSON library)** to **deserialize JSON → Java object (** `Employee` **).**
3. For deserialization, Jackson needs:
   - A **default constructor** (no-args constructor).
   - **Setters** (or direct field access).
4. Once the object is created, Spring injects it into the `employee` parameter.

🔹 Where does Lombok come in?

- Lombok itself **does not convert JSON**.
- Lombok only generates boilerplate code (getters, setters, constructors) at **compile-time**.
- Jackson **uses those generated setters & constructors** to do the conversion.

So effectively:

- **Jackson = does the JSON ↔ Java conversion**
- **Lombok = just provides setters/getters/constructors automatically** so Jackson can use them

---