# REST Architecture – Two Main Actors

In REST (Representational State Transfer) architecture, **two important actors** always participate in communication:

## 1. **Resource**

✅ **What it means:**

* A **resource** is something that represents a piece of information or a service in the application.
* It can be data (like a customer record, product details, or order history) or a business operation (like placing an order or updating a profile).
* In REST, every resource is usually represented using a **URL (Uniform Resource Locator)**.

✅ **Role:**

* The resource **provides business services** that other applications or systems can use.
* It acts like a **service provider**.

✅ **Example Scenario:**

* Suppose you are building an online shopping system.
  + https://shop.com/products → Resource for list of products
  + https://shop.com/orders → Resource for orders
  + https://shop.com/customers/101 → Resource for customer with ID 101

Each of these URLs points to a **resource** which can be used by others.

✅ **Simple Summary of Example:**  
The **resource** is like a “restaurant menu.” It lists the services (dishes) that can be ordered by the client.

## 2. **Client**

✅ **What it means:**

* A **client** is an application or system that wants to use the services or data provided by the resource.
* It could be a web browser, a mobile app, another backend system, or even a command-line tool like curl.

✅ **Role:**

* The client **accesses the resources** using HTTP methods (GET, POST, PUT, DELETE).
* It acts like a **service consumer**.

✅ **Example Scenario:**

* In the same online shopping system:
  + A **mobile app** (client) requests product details → GET https://shop.com/products/201
  + A **browser** (client) submits an order → POST https://shop.com/orders
  + A **third-party payment service** (client) checks order status → GET https://shop.com/orders/999/status

✅ **Simple Summary of Example:**  
The **client** is like the “customer in a restaurant” who reads the menu (resources) and places an order (request).

# 🔑 Final Big Picture

* **Resource** = Service provider (it owns data and business operations).
* **Client** = Service consumer (it uses the resource to do something).

👉 REST is basically about **clients requesting resources** and **resources responding with representations** (usually JSON or XML).

# 6. REST Components and JAX-RS Implementation

## REST Components

In REST architecture, we usually deal with the following **main components**:

### 1. **Resource (REST Resource)**

* A **resource** is the central concept in REST.
* It represents data (like Customer, Product, Order) or a service (like Place Order, Update Profile).
* Resources are always identified by a **URL**.
  + Example:
    - http://shop.com/customers/101 → Customer with ID 101
    - http://shop.com/orders/555 → Order with ID 555

### 2. **WADL / Swagger**

* **WADL (Web Application Description Language)** and **Swagger (OpenAPI Specification)** are tools used to **describe REST APIs**.
* They act like a **blueprint** or **API documentation** so that other developers know:
  + Which resources are available,
  + What request methods to use (GET, POST, PUT, DELETE),
  + What input/output format to expect.

👉 Swagger is widely used today because it also provides a UI for testing.

### 3. **Data Formats (XML / JSON / Text / YAML)**

* REST does not force a specific data format.
* Commonly used ones are:
  + **XML** → Older but still used in some enterprise systems.
  + **JSON** → Most popular today, lightweight and easy to use.
  + **Text / YAML** → Sometimes used for configuration or simple responses.

👉 These formats are **language independent** (can be used in Java, Python, .NET, etc.) and **platform independent** (work across Windows, Linux, Mac).

### 4. **Client**

* A **client** is any application or system that consumes REST services.
* Examples:
  + Browser
  + Mobile App
  + Backend Service
  + API Testing Tool like **Postman**

👉 **Postman** is very popular for **testing REST APIs** because it allows you to send requests (GET, POST, PUT, DELETE) and see responses easily.

## History and JAX-RS

### Roy Fielding’s Contribution

* **Roy Fielding** introduced REST architectural principles in his PhD thesis.
* His principles were already **inspired by how the Internet itself works** (using HTTP and resources).

### SUN Microsystems and JAX-RS

* SUN Microsystems supported Roy Fielding’s REST idea.
* They released a **standard API** called **JAX-RS** (Java API for XML – RESTful Services).
* At that time, **JSON was not yet popular**, so the name focused on XML.

### Why API Needs Implementation

* An **API is only a set of interfaces and abstract classes**.
* We cannot directly use an API; we need a **concrete implementation**.
* Example:
  + **JDBC API** → Needs implementation JARs from vendors like MySQL, Oracle, PostgreSQL.
  + **JAX-RS API** → Needs implementation like **Jersey** or **RestEasy**.

### JAX-RS Implementations

1. **Jersey** – Implementation provided by SUN.
2. **RestEasy** – Implementation provided by JBoss.

👉 Both support:

* Developing **REST Resources** (service side).
* Developing **REST Clients** (consumer side).

## Spring Alternative

* Instead of JAX-RS, we can also use **Spring REST module** (part of Spring MVC).
* **Spring MVC JARs are sufficient** to build RESTful services → No extra JAX-RS libraries are required.
* This is why many modern projects prefer **Spring Boot with Spring MVC** to build REST APIs.

# 🔑 Final Summary

* **Resource** → The service/data exposed via URL.
* **WADL/Swagger** → Documentation/blueprint for REST APIs.
* **Data Formats** → XML, JSON, Text, YAML (independent of language & platform).
* **Client** → Application consuming services (Postman, Browser, Mobile App).
* **Roy Fielding** → Father of REST principles.
* **JAX-RS** → Java API for RESTful services (initially XML-focused).
* **Implementations** → Jersey (SUN), RestEasy (JBoss).
* **Spring Alternative** → Spring MVC (no extra JAX-RS needed).

👉 In industry today, **Spring Boot + Spring MVC** or **Spring WebFlux** are most common for REST APIs, while **JAX-RS with Jersey/RestEasy** is still used in Java EE environments.

# 7. REST Principles (1–3)

## 1. **Unique Address**

✅ **What Happens:**

* In REST, **every operation or resource has a unique address (URL)**.
* This makes it very easy to identify and access the resource without confusion.
* The idea is: **One URL = One Resource (or operation)**.

✅ **Example Scenario:**

* Online shopping system:
  + https://shop.com/products → List of all products
  + https://shop.com/products/101 → Product with ID 101
  + https://shop.com/orders/555 → Order with ID 555

Each of these has a **different unique address**.

✅ **Example Detailed Note:**

* In Spring MVC, each **controller method** is mapped to a **unique URL pattern**.
* Example:
* @RestController
* public class ProductController {
* @GetMapping("/products")
* public List<Product> getAllProducts() { ... }
* @GetMapping("/products/{id}")
* public Product getProduct(@PathVariable int id) { ... }
* }
* Here:
  + /products → fetches all products
  + /products/{id} → fetches specific product
* Both are **unique addresses**.

✅ **Simple Summary of Example:**  
Unique Address is like having **different house addresses** in a city. Each house (resource) can be reached directly if you know its address (URL).

## 2. **Uniform Constraint Interfaces**

## ✅ What Happens (Simple Explanation)

* REST always talks using the **same fixed set of HTTP methods** (like GET, POST, PUT, DELETE, PATCH).
* Because the methods are **standard and limited**, anyone who knows HTTP can quickly understand how to use a REST API.
* The word **“constraint”** here means → REST does not allow you to invent your own random methods; you must use only the existing standard ones.

👉 Example:

* If you want to **fetch data** → use **GET**
* If you want to **add new data** → use **POST**
* If you want to **update data** → use **PUT/PATCH**
* If you want to **delete data** → use **DELETE**

This rule keeps REST APIs **simple, consistent, and predictable**.

✅ **Example Scenario:**

* If you see /products/101:
  + GET /products/101 → Read product details
  + PUT /products/101 → Update product details
  + DELETE /products/101 → Remove product
* No need for special documentation like SOAP’s WSDL. Anyone familiar with HTTP can **guess how to interact**.

✅ **Example Detailed Note (Spring MVC):**

@RestController

public class ProductController {

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

public Product getProduct(@PathVariable int id) { ... }

@PostMapping("/products")

public String addProduct(@RequestBody Product product) { ... }

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

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

@DeleteMapping("/products/{id}")

public String deleteProduct(@PathVariable int id) { ... }

}

* Each method is bound to a **standard HTTP method**.
* This makes REST **predictable** and **uniform**.

✅ **Simple Summary of Example:**  
Uniform Constraint Interfaces are like **traffic signals**—no matter which road you are on, the same colors (red, green, yellow) apply everywhere. Similarly, REST uses the same HTTP methods everywhere.

## 3. **Media Representation**

✅ **What Happens:**

* REST allows resources to be represented in **different formats**.
* The client and server can decide **which format** to use during communication.
* This enables **real interoperability** (unlike SOAP which was mostly XML-based).

✅ **Example Scenario:**

* Client requests GET /products/101.
* The server can return:
  + JSON → { "id": 101, "name": "Laptop" }
  + XML → <product><id>101</id><name>Laptop</name></product>
  + YAML →
  + id: 101
  + name: Laptop
* Same resource, **different representations**.

✅ **Example Detailed Note (Spring MVC):**

@RestController

public class ProductController {

@GetMapping(value = "/products/{id}", produces = {"application/json", "application/xml"})

public Product getProduct(@PathVariable int id) {

return new Product(id, "Laptop");

}

}

* Here the method supports **both JSON and XML output**.
* The client decides what it wants by setting **HTTP Header** → Accept: application/json or Accept: application/xml.

✅ **Simple Summary of Example:**  
Media Representation is like **ordering coffee in different cups**—glass, paper, or steel. The coffee (data) is the same, but the container (format) changes based on client’s choice.

# 🔑 Final Wrap-Up

1. **Unique Address** → Every resource has its own unique URL.
2. **Uniform Constraint Interfaces** → Same limited HTTP methods everywhere (GET, POST, PUT, DELETE).
3. **Media Representation** → Data can be exchanged in multiple formats (JSON, XML, YAML, Text).

👉 Together, these principles make REST **simple, predictable, and interoperable**.

# 8. REST Principle 4 – Communication Stateless

## ✅ What Happens

* In REST, **communication between client and server is stateless**.
* This means the **server does not remember anything** about the client after sending the response.
* Every request from the client is treated by the server as a **new request**, even if it comes from the same client.
* If the client needs the server to know something (like login credentials or session info), it must **send that information with every request**.

## ✅ Example Scenario

Imagine you are shopping online:

1. You log in and then request product details:
   * GET /products/101 with header Authorization: Bearer <token>
   * The server responds with product info.
   * After sending the response, the server **forgets everything** about you.
2. If you want to add this product to the cart:
   * POST /cart with the **same Authorization token** again.
   * The server processes it but does not keep any memory of your previous product request.

👉 Each request must carry its own **context information**, because the server is **stateless**.

## ✅ Example Detailed Note (Spring MVC Example)

@RestController

public class CartController {

@PostMapping("/cart")

public String addToCart(

@RequestHeader("Authorization") String token,

@RequestBody Product product) {

// Validate the token each time (stateless check)

if (isValidToken(token)) {

return "Product " + product.getName() + " added to cart!";

} else {

return "Unauthorized request!";

}

}

}

* Here, the **Authorization token** must be sent with **every request**.
* The server does not maintain session state; it only validates based on the request it receives.

## ✅ Simple Summary of Example

Communication Stateless is like **ordering food at a restaurant where the waiter never remembers you**.

* Every time you order, you must tell your **table number and dish** again.
* The waiter (server) treats each order (request) as if you are a **new customer**.

# 🔑 Final Wrap-Up for Principle 4

* REST is **stateless** → Server does not remember client details between requests.
* Each request must carry **all the required information** (like tokens, IDs, parameters).
* This improves **scalability**, because the server does not need to store client session data.

# 9. REST Principle 5 – HATEOAS

## ✅ What Happens (Concept in Simple Words)

* **HATEOAS** stands for **Hypermedia As The Engine Of Application State**.
* It simply means:
  + When the server sends a response to the client, it should **not just send raw data**, but also include **links (hyperlinks)** that tell the client what actions can be performed next.
* This way, the **server guides the client** step by step.
* The client doesn’t need to guess or hardcode URLs.
* The client application just **follows the links given by the server**.

👉 Without HATEOAS → The client needs prior documentation to know what to do next.  
👉 With HATEOAS → The server itself provides hints (hyperlinks) in responses.

## ✅ Example Scenario (Detailed Step-by-Step)

### Step 1: Client requests course details

GET http://www.ineuron.in/get/CID-100

### Step 2: Server responds with JSON (including links)

{

"course-id": "CID-100",

"course-name": "Spring Boot & Microservices",

"start-date": "19-Jul-2022",

"timings": "7:30 to 8:30 AM",

"trainer": "Mr. Naveen Reddy",

"course-content": "http://www.ineuron.ai/get/CID-100/course-content",

"enroll-link": "http://www.ineuron.ai/get/CID-100/enroll",

"feedback-link": "http://www.ineuron.ai/get/CID-100/feedback"

}

👉 Notes on this response:

* The client **doesn’t need to guess URLs** for content, enrollment, or feedback.
* The server itself says:
  + If you want **course content**, go here → course-content link
  + If you want to **enroll**, go here → enroll-link
  + If you want to **give feedback**, go here → feedback-link

👉 This makes the system **self-descriptive and discoverable**.

## ✅ Example Detailed Note (Spring Boot Example with Links)

@RestController

public class CourseController {

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

public Map<String, Object> getCourse(@PathVariable String id) {

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

response.put("course-id", id);

response.put("course-name", "Spring Boot & Microservices");

response.put("start-date", "19-Jul-2022");

response.put("timings", "7:30 to 8:30 AM");

response.put("trainer", "Mr. Naveen Reddy");

// Adding HATEOAS links

response.put("course-content", "http://localhost:8080/courses/" + id + "/course-content");

response.put("enroll-link", "http://localhost:8080/courses/" + id + "/enroll");

response.put("feedback-link", "http://localhost:8080/courses/" + id + "/feedback");

return response;

}

}

### How it works:

1. Client sends request:
2. GET http://localhost:8080/courses/CID-100
3. Server responds with JSON including **hyperlinks**.
4. Client does not need hardcoded URLs. It simply reads the response and follows links.

## ✅ Simple Analogy (Everyday Example)

Think of **HATEOAS like an ATM machine**:

* When you insert your card, the ATM does not expect you to memorize codes.
* Instead, it shows you **next possible actions** as options on screen:
  + “Check Balance”
  + “Withdraw Money”
  + “Deposit Money”
  + “Print Mini Statement”

👉 Similarly, the server (like ATM) tells the client what to do next by giving **hyperlinks**.

# 🔑 Extra Notes on HATEOAS

* REST without HATEOAS → Client must know documentation or hardcode URLs.
* REST with HATEOAS → Client can **navigate dynamically** just by following server-provided links.
* Advantage:
  + Reduces dependency on static documentation.
  + Makes REST services more **flexible and adaptable** to changes.
  + Easier for **B2B communication**, because machines can just follow links.

# 🛠 Spring Boot Notes for REST Development

1. **Spring Boot Starter Needed** →
   * spring-boot-starter-web
   * This includes everything:
     + Spring MVC for web + REST
     + Embedded Tomcat for running the app
   * No need to add Jersey, RestEasy, or JAX-RS libraries.
2. **Development Types with Spring Web MVC Module**
   * Web application with JSP/Thymeleaf.
   * REST application with @RestController.
   * Both work with the same starter (spring-boot-starter-web).
3. **REST APIs in Spring Boot** → use annotations:
   * @GetMapping → Fetch resource
   * @PostMapping → Create resource
   * @PutMapping → Update resource
   * @DeleteMapping → Remove resource

# 🔑 Final Wrap-Up

* **HATEOAS**: Server provides hyperlinks in responses to guide the client about next actions.
* **Example**: Course API response includes links for course content, enroll, and feedback.
* **Analogy**: Like an ATM machine showing next available options.
* **Spring Boot REST**: Only needs spring-boot-starter-web; same module handles web apps, REST APIs, and embedded Tomcat.