Class-21: Introduction to REST API

by Pial Kanti Samadder

What is REST?

REST is an acronym for **REpresentational State Transfer** and **an architectural style** for **distributed hypermedia systems**. Roy Fielding first presented it in 2000 in his famous **dissertation**. Since then, it has become one of the most widely used approaches for building web-based APIs (*Application Programming Interfaces*).

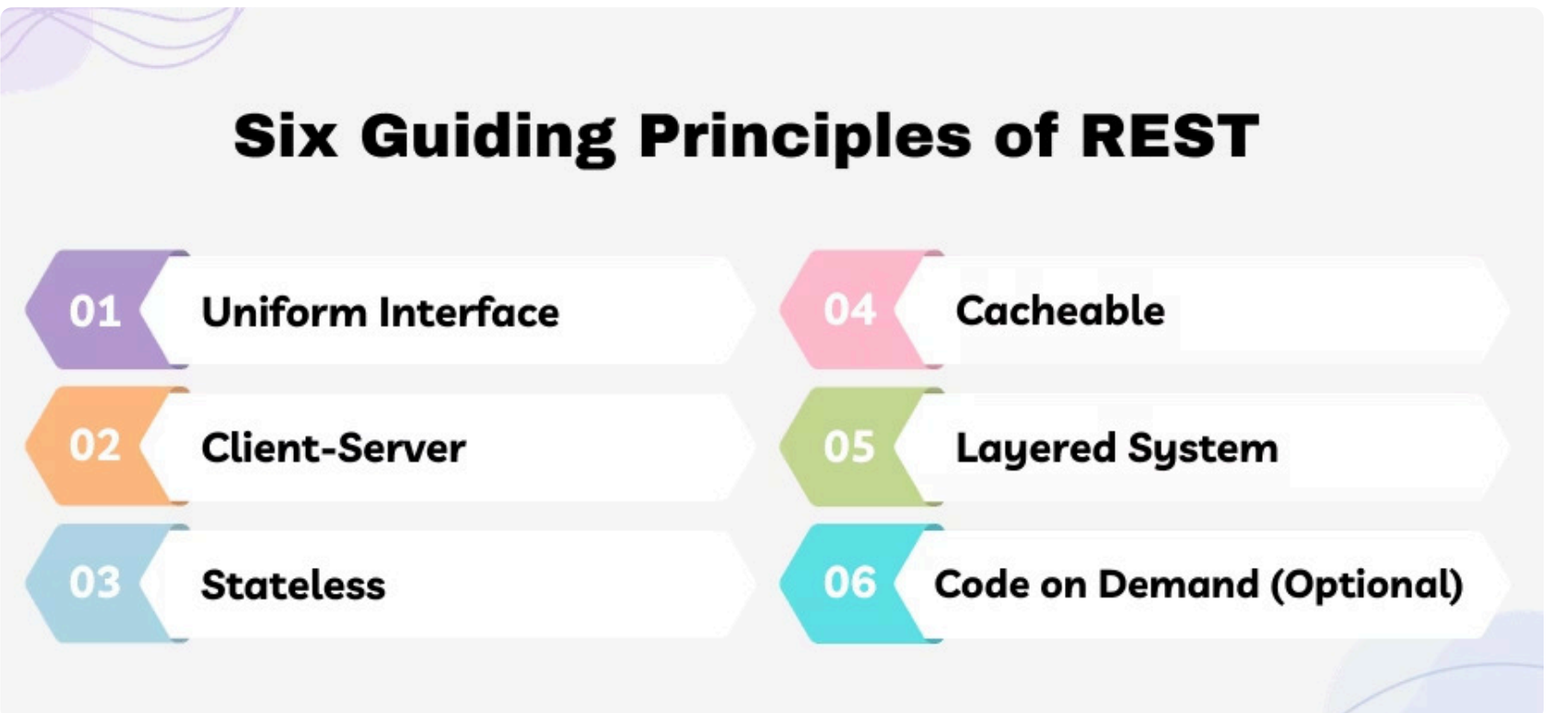
REST is not a protocol or a standard, it is an architectural style. During the development phase, API developers can implement REST in a variety of ways.

> **A Web API (or Web Service) conforming to the REST architectural style is called a *REST API* (or *RESTful API*).**

REST Architectural Constraints

REST defines **6 architectural constraints** that make any web service – a truly RESTful API.

1. Uniform interface
2. Client-server
3. Stateless
4. Cacheable
5. Layered system
6. Code on demand (optional)



1. Client-Server Architecture

An API's job is to connect two pieces of software without limiting their own functionalities. This is one of REST's core restrictions:

- The **client** (that makes requests) and the **server** (that gives responses) remain separate and independent.
- When properly designed, both can **evolve independently** without breaking communication.
- This separation is crucial when a single server serves **many different clients** — for example, a weather API sending data to multiple mobile apps.

A simple illustration:

📱 Mobile App(Client) → 🌐 REST API(Server) → 💾 Database

2. Stateless

For an API to be **stateless**, each call must be **independent** of previous ones.

- Every API request must include **all necessary data and credentials** (e.g., API key).
- The server does **not store session state** between requests.
- If one request fails, it doesn’t affect the next.

Example:

- Non-stateless: Server remembers API key from first call.
- Stateless: Each request includes the API key again.

3. Uniform Interface

Even though the client and server can evolve separately, the **interface** that connects them must remain consistent.

- REST APIs use a **uniform interface** — most commonly **HTTP**.
- Standard HTTP methods: `GET`, `POST`, `PUT`, `DELETE`.
- Data is exchanged in standard formats like **JSON** for simplicity and compatibility.

This ensures smooth communication between all connected systems, regardless of changes.

4. Layered System

RESTful architecture structures the design into **layers** that work together.

- Each layer performs a **specific function**, then passes data to the next.
- Layers interact **only with adjacent layers**, not the entire system.
- Enhances **security** and **scalability** — you can add, remove, or modify layers without affecting others.

Example Layers:

- Proxy servers
- Load balancers
- Application servers
- Databases

5. Cacheable

Stateless APIs can generate large amounts of repetitive traffic.

Caching helps reduce that by allowing clients to **store responses** temporarily.

- Cached data can be reused for repeated requests.
- Reduces **server load** and **response times**.
- Often controlled with HTTP headers like `Cache-Control` or `ETag`.

Example:

A client requests weather data once and reuses it locally instead of fetching it again immediately.

6. Code on Demand (Optional)

This is the **only optional** REST constraint.

- Allows a server to send **code or applets** (e.g., JavaScript) to be executed by the client.
- Can make clients **more dynamic and flexible** with less work on the server.

Risks & Limitations:

- Potential **security risks** if code comes from untrusted sources.
- Code must be in a **language the client can execute**.
- Best suited for **internal or trusted systems**.

Example:

Server sending `script.js` → Client executes new feature dynamically.

RESTful Verbs

REST uses standard **HTTP verbs (methods)** to define the **intended action** on a resource.
Because REST is **stateless**, each request must clearly specify what operation it performs, and responses use **HTTP status codes** to indicate success or failure.

Common HTTP Verbs and Their Purpose

HTTP Verb	Typical Use	Operation Type
GET	Retrieve a resource or a collection of resources	**Read (Safe)**
POST	Create a new resource	**Create (Non-Idempotent)**
PUT	Update or replace an existing resource	**Update (Idempotent)**
PATCH	Apply a partial update to a resource	**Partial Update (Idempotent)**
DELETE	Remove a resource	**Delete (Idempotent)**

HTTP Status Codes

HTTP operates on a **Request–Response** model.
Every request sent by the client receives a **response** from the server — including a **status code** that indicates the result of the operation.

Category	Code Range	Meaning	Common Examples
Success	200–299	Request was successfully received, understood, and accepted.	`200 OK` – Successful GET `201 Created` – Resource created `204 No Content` – Successful, no response body

Redirection	`300–399`	Further action is needed to complete the request.	`301 Moved Permanently`
			`302 Found`
			`304 Not Modified`

Client Error	`400–499`	Problem with the request sent by the client.	`400 Bad Request` – Invalid input `401 Unauthorized` – Auth required `403 Forbidden` – Access denied `404 Not Found` – Resource missing `409 Conflict` – Data conflict

Server Error	`500–599`	Server failed to fulfill a valid request.	`500 Internal Server Error`
			`502 Bad Gateway`
			`503 Service Unavailable`

REST API URI Naming Conventions & Best Practices

1. **Use Nouns for Resources:** URIs should represent *resources* (nouns), not *actions* (verbs).
   ✅ `/user-management/users`
   ✅ `/device-management/managed-devices/{device-id}`
   ❌ `/createUser`, `/getUserDetails`
   - **Document** – Single item (singular)
     `/users/{id}` `/managed-devices/{device-id}`
   - **Collection** – Group of items (plural)
     `/users`, `/users/{id}/accounts`
   - **Store** – Client-managed resources
     `/users/{id}/playlists`
2. **Consistency is Key**
   - Use forward slash (/) to indicate hierarchical relationships → `/devices/{id}/scripts`
   - Do not use trailing forward slash (/) in URIs → `/devices` ✔️
   - Use hyphens (-) to improve the readability of URIs
   - Do not use underscores (_)
   - Use lowercase letters in URIs → `/api/v1/users`
   - Avoid file extensions → `/users` ✔️ `/users.json` ❌
3. **Do not use file extensions:** File extensions look bad and do not add any advantage
   - `/device-management/managed-devices.xml  /*Do not use it*/`
   - `/device-management/managed-devices  /*This is correct URI*/`
4. **Never use CRUD function names in URIs:** Use HTTP verbs to define actions
   - `GET /devices` → Retrieve all
   - `POST /devices` → Create
   - `PUT /devices/{id}` → Update
   - `DELETE /devices/{id}` → Delete
5. **Use query component to filter URI collection**
   - `/devices?region=USA&brand=XYZ&sort=install-date`
6. **Do not Use Verbs in the URI:** It is not correct to put the verbs in REST URIs. REST uses nouns to represent resources, and HTTP methods (GET, POST, PUT, DELETE, etc.) are then used to perform actions on those resources, effectively acting as verbs.
   - ❌ `/scripts/{id}/execute`
   - ✅ `/scripts/{id}/status` (POST with `action=execute`)

Idempotent HTTP Methods in REST

An operation is **idempotent** if performing it multiple times has the **same effect** as performing it once.

Idempotent Methods

Method	Purpose	Idempotency Explanation
GET	Retrieve a resource	No data modification; calling it multiple times gives same result (unless resource changes externally).
PUT	Update or create a resource	Repeated calls with same data result in same final state.
DELETE	Remove a resource	Deleting multiple times doesn't change the outcome — resource remains deleted.
HEAD	Retrieve metadata (headers only)	Same as GET but without the body; does not modify resources.
OPTIONS	Retrieve supported methods for a resource	Only provides information; no side effects.

Non-Idempotent Example

- **POST** → Used to create a resource; each call creates a **new entry**, so it's **not idempotent**.

Understanding the Basics of Spring MVC

Before we dive into the specifics of `@Controller` and `@RestController`, it's helpful to understand the core principles of Spring MVC (Model-View-Controller). The MVC design pattern separates the application logic into three interconnected components:

- **Model:** The Model represents the application's data and the business rules to manipulate the data.
- **View:** The View is responsible for visualizing the data provided by the Model.
- **Controller:** The Controller handles the user input and interacts with the Model to perform operations on the data.

In a Spring MVC application, `@Controller` and `@RestController` are annotations used to define the Controller component.

Spring MVC *@Controller*

We can annotate classic controllers with the `@Controller` annotation. **This is simply a specialization of the `@Component` class, which allows us to auto-detect implementation classes through the classpath scanning**.

We typically use `@Controller` *r* in combination with a *@RequestMapping* annotation for request handling methods.

Let's see a quick example of the Spring MVC controller:

```
@Controller
@RequestMapping("books")
public class SimpleBookController {
    @GetMapping("/{id}", produces = "application/json")
    public @ResponseBody Book getBook(@PathVariable int id) {
        return findBookById(id);
    }

    private Book findBookById(int id) {
        // ...
    }
}
```

We annotated the request handling method with *@ResponseBody*. This annotation enables automatic serialization of the return object into the *HttpResponse*.

We can return a view from `@Controller` annotated class method:

```
@Controller
public class GreetingController {

    @RequestMapping("/greeting")
    public ModelAndView greeting() {
        ModelAndView modelAndView = new ModelAndView("greeting");
        modelAndView.addObject("message", "Hello, Spring MVC!");
        return modelAndView;
    }
}
```

The `greeting()` method handles HTTP requests to the `/greeting` URL, returning a `ModelAndView` object. The string "greeting" is the name of the view to render, and `addObject("message", "Hello, Spring MVC!")` adds an attribute to the model named "message".

When a request is made to the `/greeting` URL, the `greeting()` method is invoked, and the result is rendered using the "greeting" view.

Spring MVC *@RestController*

@RestController is a specialized version of the controller. It includes the *@Controller* and *@ResponseBody* annotations, and as a result, simplifies the controller implementation:

```
@RestController
@RequestMapping("books-rest")
public class SimpleBookRestController {

    @GetMapping("/{id}", produces = "application/json")
    public Book getBook(@PathVariable int id) {
        return findBookById(id);
    }

    private Book findBookById(int id) {
        // ...
    }
}
```

The controller is annotated with the `@RestController` annotation; therefore, the `@ResponseBody` isn't required.

Every request handling method of the controller class automatically serializes return objects into *HttpResponse*.

Choosing Between `@Controller` and `@RestController`

The choice between `@Controller` and `@RestController` depends on what you need your controller to do.

- Use `@Controller` when you want to prepare a model and use it with a view. This is typical in web applications where you're returning views (such as Thymeleaf or JSP pages).
- Use `@RestController` when you're creating a RESTful API. The `@RestController` annotation is designed for services, such as mobile services or frontend web app calls, where you're returning data directly to the caller.

In essence, `@Controller` and `@RestController` serve different purposes, but both play an integral role in Spring MVC. Understanding these annotations and when to use each one is essential for a Spring MVC developer.