**🌍 Core Principle of RestTemplate and its Methods**

**1️⃣ Core Idea — Communication Between Two Applications**

Let’s imagine you have **two Spring Boot applications**:

* **App A (Client App)** — wants to call some API (it sends the request)
* **App B (Server App)** — has REST endpoints (it receives and responds)

Normally, if you were testing App B, you might use **Postman** to call its APIs.  
But when another application (App A) wants to call it **programmatically**,  
you need some tool **inside App A’s Java code** to send HTTP requests.

That tool is 👉 **RestTemplate**.

**2️⃣ What is RestTemplate?**

**🔹 Definition:**

RestTemplate is a **Spring class** used to **consume RESTful web services** (call APIs) from within another Spring application.

In simple words:

RestTemplate is like a built-in Postman for your Java code.

You use it when you want your application to talk to another REST API —  
by sending **GET**, **POST**, **PUT**, or **DELETE** requests.

**3️⃣ Why We Need It?**

Because REST APIs work through **HTTP** —  
to fetch data, insert, update, or delete,  
you need to send **HTTP requests** and receive **HTTP responses**.

Manually handling HTTP (using sockets or HttpURLConnection) is **hard and repetitive**,  
so Spring created RestTemplate —  
it **hides all the low-level networking code** and gives you **simple Java methods** to perform these operations easily.

**4️⃣ Internally How It Works**

When you call a method like:

restTemplate.getForEntity("http://localhost:8080/student/101", Student.class);

This happens internally:

1. RestTemplate builds an HTTP **GET request** with the given URL.
2. Sends that request over the network.
3. Waits for the response from the target API.
4. Converts the JSON (or XML) response body into a **Java object** (like Student),  
   using an internal **message converter** (e.g., MappingJackson2HttpMessageConverter).
5. Wraps it into a ResponseEntity that contains:
   * status code (200, 404, etc.)
   * headers (like Content-Type)
   * body (converted Java object)

**5️⃣ How We Send HTTP Requests (Methods)**

To match the four HTTP operations, RestTemplate provides specific methods:

| **HTTP Method** | **RestTemplate Method** | **Used For** | **Returns** |
| --- | --- | --- | --- |
| GET | getForEntity() or getForObject() | Fetching data | ResponseEntity / Object |
| POST | postForEntity() or postForObject() | Creating data | ResponseEntity / Object |
| PUT | put() | Updating data | void |
| DELETE | delete() | Deleting data | void |

**⚙️ Detailed Working of Each Method (with Core Explanation)**

**🧩 1. getForEntity()**

**Core Principle:**

Used to **retrieve data** from another API using an HTTP GET request.

It is like asking the server:

“Please give me the resource with this ID.”

This method **does not send a request body**, only a **URL**.

**Example:**

ResponseEntity<Student> response = restTemplate.getForEntity(

"http://localhost:8080/student/101", Student.class);

**What Happens Internally:**

* Sends an HTTP GET request to /student/101
* Waits for the response
* Converts JSON → Student object
* Returns all HTTP details inside a ResponseEntity

**Output:**

Student s = response.getBody(); // Access the actual data

HttpStatus status = response.getStatusCode(); // Access HTTP status

**🧩 2. postForEntity()**

**Core Principle:**

Used to **send new data** to the server via an HTTP POST request.

It’s like saying:

“Here’s a new record — please insert it on your side.”

It includes both **URL** and **request body** (usually JSON).

**Example:**

Student s = new Student(101, "Ravi", "Hyd");

ResponseEntity<String> response = restTemplate.postForEntity(

"http://localhost:8080/addStudent", s, String.class);

**What Happens Internally:**

* Converts Student object → JSON
* Sends POST request with JSON body
* Receives the response (maybe a success message)
* Returns it inside a ResponseEntity

**🧩 3. put()**

**Core Principle:**

Used to **update** an existing resource on the server.

It’s like saying:

“Here’s the updated data for this record — replace the old one.”

**Example:**

Student s = new Student(101, "Ravi Kumar", "Bangalore");

restTemplate.put("http://localhost:8080/updateStudent", s);

**What Happens Internally:**

* Converts Student → JSON
* Sends a PUT request with JSON body
* Server updates the record
* RestTemplate doesn’t return anything (no ResponseEntity)

**🧩 4. delete()**

**Core Principle:**

Used to **remove** an existing resource from the server.

It’s like saying:

“Please delete this record permanently.”

**Example:**

restTemplate.delete("http://localhost:8080/deleteStudent/101");

**What Happens Internally:**

* Sends DELETE request to that URL
* Server deletes the record
* No response body is returned

**🧩 The Problem with These Methods**

Each method can only handle **one specific HTTP type**:

* getForEntity() → GET only
* postForEntity() → POST only
* put() → PUT only
* delete() → DELETE only

If you need to dynamically choose HTTP type at runtime (e.g., based on a condition),  
you would need multiple if-else blocks or duplicate code.

Also, if you want to **add custom headers** (like Authorization, Tokens, etc.),  
these methods are limited.

**🧭 The Solution — exchange() Method**

**🔹 Core Principle of exchange()**

exchange() is the **universal, all-in-one method** of RestTemplate.  
It is built to overcome the limitations of the above methods.

Think of it like:

“A single method that can send any type of HTTP request, with full control over headers, body, and response.”

It’s flexible and powerful —  
you just tell it:

1. The URL
2. The HTTP method (GET, POST, PUT, DELETE)
3. The full request (headers + body) as an HttpEntity
4. The expected response type

**⚙️ Syntax:**

exchange(String url, HttpMethod method, HttpEntity<?> requestEntity, Class<T> responseType)

**🧩 Example:**

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

Student s = new Student(103, "Kiran", "Chennai");

HttpEntity<Student> entity = new HttpEntity<>(s, headers);

ResponseEntity<String> response = restTemplate.exchange(

"http://localhost:8080/addStudent",

HttpMethod.POST,

entity,

String.class

);

**🔍 What Happens Internally:**

1. HttpEntity combines both **headers** and **body** into one object.
2. You call exchange() and specify:
   * Which URL to hit
   * Which HTTP method to use (GET, POST, PUT, DELETE)
   * What request data to send (HttpEntity)
   * What type of response to expect
3. It executes the request accordingly and returns a ResponseEntity.

**⚡ Advantages of exchange()**

| **Feature** | **Description** |
| --- | --- |
| ✅ Universal | Works with all HTTP methods |
| ✅ Flexible | You can pass custom headers (Auth tokens, content-type, etc.) |
| ✅ Full control | You control the entire request (headers + body) |
| ✅ Returns ResponseEntity | Contains status, headers, and body |
| ⚡ Best for production | Preferred in real-world projects where APIs are secure and dynamic |

**🧭 Difference Between @RequestMapping and @GetMapping / @PostMapping**

These are **on the server side**, used inside your RestController.

**🧩 @RequestMapping**

**Core Principle:**

Used to **map** a specific URL to a method in your controller,  
and can handle **any HTTP method** (GET, POST, PUT, DELETE) depending on method= attribute.

**Example:**

@RequestMapping(value="/student", method=RequestMethod.GET)

public Student getStudent() {

...

}

* @RequestMapping is **generic**.
* You must **manually specify** which HTTP method it supports.
* Used in older Spring versions or when you need multiple methods on same path.

**🧩 @GetMapping, @PostMapping, etc.**

**Core Principle:**

They are **shortcuts (specialized forms)** of @RequestMapping.

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

public Student getStudent(@PathVariable Integer id) {

...

}

@PostMapping("/student")

public String addStudent(@RequestBody Student s) {

...

}

* Cleaner and easier to read.
* Automatically assume HTTP method from the annotation name.
* Recommended in modern Spring Boot projects.

**🧩 Complete Example (Client + Server Flow)**

**🎯 Step 1: Server (Provider) — StudentController**

@RestController

@RequestMapping("/student")

public class StudentController {

@GetMapping("/{id}")

public Student getStudent(@PathVariable Integer id) {

return new Student(id, "Ravi", "Hyderabad");

}

@PostMapping

public String addStudent(@RequestBody Student s) {

return "Student added successfully: " + s.getName();

}

}

**🎯 Step 2: Client (Consumer) — Using RestTemplate**

@SpringBootApplication

public class RestTemplateClientApp implements CommandLineRunner {

@Autowired

private RestTemplate restTemplate;

public static void main(String[] args) {

SpringApplication.run(RestTemplateClientApp.class, args);

}

@Bean

public RestTemplate getTemplate() {

return new RestTemplate();

}

@Override

public void run(String... args) throws Exception {

// Using getForEntity()

ResponseEntity<Student> response = restTemplate.getForEntity(

"http://localhost:8080/student/101", Student.class);

System.out.println("GET Response: " + response.getBody());

// Using exchange()

Student s = new Student(105, "Kiran", "Chennai");

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

HttpEntity<Student> entity = new HttpEntity<>(s, headers);

ResponseEntity<String> postResponse = restTemplate.exchange(

"http://localhost:8080/student",

HttpMethod.POST,

entity,

String.class);

System.out.println("POST Response: " + postResponse.getBody());

}

}

**🧾 Final Summary**

| **Concept** | **Side** | **Purpose** | **Example** |
| --- | --- | --- | --- |
| RestTemplate | Client | To consume APIs | restTemplate.getForEntity() |
| xxxForEntity() | Client | For specific HTTP method | One method per operation |
| exchange() | Client | For any HTTP method | Flexible & powerful |
| @RequestMapping | Server | Generic mapping | Needs method attribute |
| @GetMapping, @PostMapping | Server | Simplified, modern | Method implied by name |

# 🧠 Execution Order

1. **Run Eg: RestTemplateStudentProvider first**  
   → It starts the REST API provider.
2. **Run Eg: RestTemplateConsumerApplication second**  
   → It automatically sends requests using RestTemplate and prints results.

**Use different port numbers for two applications , don’t run both the applications in the same port number.**

## Understanding HttpEntity, HttpHeaders, and ResponseEntity

### 🎯 Core Principle

When we send a request or receive a response using REST APIs,  
two main parts are always involved:

1. **Headers** – metadata (information about the request or response)
2. **Body** – actual data (the content we send or receive)

Spring provides **special wrapper classes** to manage these parts in an organized way.

Those wrappers are:

* **HttpHeaders** → used to represent and manage HTTP header information.
* **HttpEntity** → used to wrap both headers and body together when sending requests.
* **ResponseEntity** → used to wrap both headers, body, and status code when sending responses.

Let’s understand each of them step by step in simple language.

### 🧩 1. HttpHeaders

Think of **HttpHeaders** like a “cover letter” attached to your main data.

It doesn’t hold the actual content (like JSON or XML) —  
instead, it holds instructions about how to handle that content.

Example headers:

Content-Type: application/json

Accept: application/json

Authorization: Bearer <token>

💡 **In Spring Boot**, HttpHeaders is a class used to:

* Set these headers in requests.
* Read headers from responses.

👉 **Example**

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

headers.set("Authorization", "Bearer abc123token");

So, now your header says — “I’m sending JSON data, and here’s my authorization token.”

### 🧩 2. HttpEntity

Now that you have headers, you also need to attach your **data (body)**.

That’s where HttpEntity comes in.

It acts like an **envelope** that holds both:

* The **body** (actual data to send)
* The **headers** (instructions about the data)

🧠 You can imagine it as:

“A complete request packet that contains both message content and metadata.”

👉 **Example**

Student student = new Student(101, "Pavan", "Hyderabad");

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

HttpEntity<Student> requestEntity = new HttpEntity<>(student, headers);

Now requestEntity has both:

* Headers → Content type = JSON
* Body → Student object data

This HttpEntity can now be sent using RestTemplate.exchange().

### 🧩 3. ResponseEntity

This is what we use on the **server side (Producer/Controller)** to send back responses.

It’s like a **complete response packet** that contains:

* **Body** → actual response data
* **Headers** → response headers
* **Status code** → success/failure status (e.g., 200 OK, 404 Not Found)

👉 **Example**

@GetMapping("/student")

public ResponseEntity<Student> getStudent() {

Student s = new Student(101, "Pavan", "Hyderabad");

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

}

So when the client receives this:

* Body = { "id": 101, "name": "Pavan", "city": "Hyderabad" }
* Status = 200 OK

💡 If there was an error, you could send something like:

return new ResponseEntity<>(null, HttpStatus.NOT\_FOUND);

### 🧠 Quick Summary

| **Concept** | **Purpose** | **Used In** | **Contains** |
| --- | --- | --- | --- |
| **HttpHeaders** | To define metadata/instructions | Request + Response | Header info only |
| **HttpEntity** | To send both body + headers | Client (Consumer) side | Body + Headers |
| **ResponseEntity** | To send both body + headers + status | Server (Producer) side | Body + Headers + Status code |

### 💻 Example — Producer & Consumer with all 3 classes

#### ✅ Producer (Server App)

package in.orcas.controller;

import org.springframework.http.\*;

import org.springframework.web.bind.annotation.\*;

import in.orcas.model.Student;

@RestController

@RequestMapping("/student")

public class StudentController {

@PostMapping("/add")

public ResponseEntity<String> addStudent(@RequestBody Student student) {

// Normally we’d save this student in DB, but here just return success message

HttpHeaders headers = new HttpHeaders();

headers.set("Custom-Header", "Student-Added-Successfully");

return new ResponseEntity<>(

"Student saved: " + student.getName(),

headers,

HttpStatus.CREATED

);

}

}

#### ✅ Consumer (Client App)

package in.orcas.runner;

import org.springframework.boot.CommandLineRunner;

import org.springframework.http.\*;

import org.springframework.stereotype.Component;

import org.springframework.web.client.RestTemplate;

import in.orcas.model.Student;

@Component

public class RestTemplateRunner implements CommandLineRunner {

@Override

public void run(String... args) throws Exception {

String url = "http://localhost:8080/student/add";

// 1️⃣ Prepare Body

Student student = new Student(101, "Pavan", "Hyderabad");

// 2️⃣ Prepare Headers

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

headers.setAccept(List.of(MediaType.TEXT\_PLAIN));

// 3️⃣ Combine both using HttpEntity

HttpEntity<Student> requestEntity = new HttpEntity<>(student, headers);

// 4️⃣ Send request using exchange()

RestTemplate rt = new RestTemplate();

ResponseEntity<String> response = rt.exchange(url, HttpMethod.POST, requestEntity, String.class);

// 5️⃣ Print Response Details

System.out.println("Status Code: " + response.getStatusCode());

System.out.println("Headers: " + response.getHeaders());

System.out.println("Body: " + response.getBody());

}

}

### ⚙️ Output

When you run the **Consumer** app:

Status Code: 201 CREATED

Headers: [Custom-Header:"Student-Added-Successfully", Content-Type:"text/plain;charset=UTF-8"]

Body: Student saved: Pavan

### ✅ ****Execution Flow****

1️⃣ Run Eg: Stundet**ProviderApp** → starts on port 9090  
2️⃣ Run Eg: **StudentConsumerApp** → starts on port 9091  
3️⃣ Watch both consoles:

### 💬 In Simple Words

* **HttpHeaders** = The “instructions” for how to send or read the message.
* **HttpEntity** = The “complete request envelope” with data and instructions.
* **ResponseEntity** = The “complete reply envelope” with data, instructions, and status.

**⚙️ In Real-Time Industry Projects**

In actual enterprise projects (microservices, REST APIs, large-scale apps),  
all three — HttpHeaders, HttpEntity, and ResponseEntity — are used,  
but **each one has its own role and importance** depending on **whether you are on the Producer side (backend service)** or **Consumer side (calling another API)**.

**🏗️ 1️⃣ ResponseEntity — Most Common on Producer Side (Server/Backend)**

**Used almost everywhere in backend REST APIs.**

✅ **Why:**

* Every REST API endpoint must return:
  + Body (the actual data or message)
  + Status code (to tell success/failure)
  + Optional headers (extra info)
* ResponseEntity wraps all of these neatly.

✅ **Real Use:**

return ResponseEntity

.status(HttpStatus.CREATED)

.header("operation", "insert")

.body("Employee Created Successfully");

💬 In big companies (like Amazon, Infosys, or Deloitte):

* All controllers in REST APIs generally return ResponseEntity<T>.
* This helps standardize responses and integrate with tools like **Swagger**, **API Gateways**, and **Exception Handlers**.

So, in real-time **producer-side (backend)** code —  
👉 **ResponseEntity is mandatory and used heavily.**

**🌐 2️⃣ HttpEntity & HttpHeaders — Mostly Used on Consumer Side (when calling external APIs)**

**Used in services or microservices that consume other APIs**, for example:

* Calling Payment Gateway API
* Calling Authentication API
* Calling 3rd-party REST services

✅ **Why:**

* You often need to send authentication tokens, content type, or custom headers.
* HttpEntity allows you to combine **body + headers** neatly in one object before sending.
* This becomes necessary when using RestTemplate.exchange() or WebClient.

✅ **Real Use:**

HttpHeaders headers = new HttpHeaders();

headers.set("Authorization", "Bearer " + token);

headers.setContentType(MediaType.APPLICATION\_JSON);

HttpEntity<Order> entity = new HttpEntity<>(order, headers);

ResponseEntity<String> response = restTemplate.exchange(

"https://api.payments.com/create",

HttpMethod.POST,

entity,

String.class

);

💬 In microservice-based architectures (common in AWS or Kubernetes setups),  
many services call other internal APIs — that’s where this combo (HttpEntity + HttpHeaders) is used regularly.

**🧠 Quick Recap (Industry Summary Table)**

| **Concept** | **Used Mostly In** | **Purpose** | **Common Real-Time Scenario** |
| --- | --- | --- | --- |
| **ResponseEntity** | Backend (Producer / REST API Controller) | Send complete response with status code | Returning data from your service (e.g., UserService, OrderController) |
| **HttpHeaders** | Consumer (Client calling API) | Set header info like Content-Type, Authorization | Calling Payment Gateway, Token API |
| **HttpEntity** | Consumer (Client calling API) | Combine Body + Headers for outgoing requests | Used with RestTemplate.exchange() |

**💬 Real Project Flow Example**

Imagine you are working on an **E-commerce platform** with two microservices:

1. **OrderService** → creates orders
2. **PaymentService** → handles payments

🔹 **OrderService (Producer)**  
Uses ResponseEntity to send responses:

@PostMapping("/create")

public ResponseEntity<String> createOrder(@RequestBody Order order) {

// save order

return ResponseEntity.status(HttpStatus.CREATED)

.body("Order Created Successfully");

}

🔹 **PaymentService (Consumer)**  
Uses HttpEntity + HttpHeaders to call OrderService:

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

HttpEntity<Order> entity = new HttpEntity<>(order, headers);

ResponseEntity<String> response = restTemplate.exchange(

"http://ORDER-SERVICE/create",

HttpMethod.POST,

entity,

String.class

);

So both are used **side by side** — one for sending, one for receiving.

**💬 In Short (Industry Reality)**

* 🏢 ResponseEntity → **Used everywhere** in REST APIs (backend side).
* 🌐 HttpEntity + HttpHeaders → **Used frequently** in integration or microservices (consumer side).
* Together, they make your REST communication **clean, structured, and professional**.