**Understanding RestTemplate Exchange Methods (Modes)**

**🔹 1️⃣ Core Concept — The Real Meaning**

When one Spring Boot application (Consumer) wants to talk to another (Producer),  
it has to send an **HTTP Request** — either:

| **Type** | **Action** | **Example** |
| --- | --- | --- |
| GET | To read data | Fetch all students |
| POST | To insert data | Add a new student |
| PUT | To update data | Update student info |
| DELETE | To remove data | Delete student by ID |

**🧠 The Problem (Before exchange())**

RestTemplate gives separate methods for each request type:

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

These work fine, but when your project grows, having **different methods for every operation** makes code:

* lengthy,
* repetitive,
* and harder to maintain.

So, instead of remembering 4–5 separate methods,  
Spring introduced **one universal method** that can handle all operations 👇

**🔹 2️⃣ The Universal Solution — exchange()**

The exchange() method can perform **all HTTP methods (GET, POST, PUT, DELETE)** with a **single unified structure**.

**✅ Syntax:**

ResponseEntity<T> exchange(

String url,

HttpMethod method,

HttpEntity<?> requestEntity,

Class<T> responseType

)

**📦 Parameters Explained**

| **Part** | **Meaning** |
| --- | --- |
| url | The API endpoint (Producer URL) |
| method | Type of HTTP request (GET, POST, PUT, DELETE) |
| requestEntity | Request body + headers (wrapped inside HttpEntity) |
| responseType | What data type we expect in response (like String.class, Student.class) |

**📥 What It Returns**

exchange() returns a **ResponseEntity<T>** object —  
which contains:

* **Body** (data),
* **Status Code** (like 200 OK),
* **Headers** (like Content-Type).

So exchange() = **One method for all modes + full response control.**

**🔄 Think of it like this:**

In a restaurant 🍴

| **You do** | **Represents** |
| --- | --- |
| Tell waiter “Get me the Menu” | GET |
| Tell waiter “Add new order” | POST |
| Tell waiter “Change my order” | PUT |
| Tell waiter “Cancel my order” | DELETE |

Now imagine there’s a **universal waiter** 🧑‍🍳 who understands all 4 commands.  
That’s exactly what **exchange()** does — it listens to all request types.

### 🧠 Summary

| **Mode** | **Method** | **API URL** | **Result** |
| --- | --- | --- | --- |
| POST | Create | /save | New student saved |
| GET | Read | /find/{id} | Fetch student |
| PUT | Update | /update/{id} | Fee updated |
| DELETE | Remove | /delete/{id} | Student deleted |

### ✅ ****Execution Flow****

1️⃣ Run Eg: RestTemplateExchangeModesProvider→ starts on port 9090  
2️⃣ Run Eg: **RestTemplateExchangeModesConsumer** → starts on port 9091  
3️⃣ Watch both consoles:

## Understanding ResponseEntity

### 🔷 Core Concept

When two applications talk to each other (Producer ↔ Consumer),  
**Producer → sends a Response**,  
**Consumer → receives that Response**.

Now, when Producer sends the response, it doesn’t just send data (body) —  
it also sends metadata like:

* **Status Code** (✅ success / ❌ failure)
* **Headers** (extra info like content type)
* **Body** (actual data)

Spring gives us a wrapper class called **ResponseEntity** to hold all these three parts together.

### 🧩 Think of ResponseEntity like a Parcel 📦

When you order something online:

| **Part of Parcel** | **Represents in REST** | **Example** |
| --- | --- | --- |
| 🏷️ Label (delivery info) | **Headers** | Content-Type: application/json |
| 💌 Delivery status slip | **Status Code** | 200 OK, 404 Not Found |
| 🎁 Actual product | **Body** | JSON data or message |

So, a **ResponseEntity** = 📦 (Status Code + Headers + Body)

### 🔸 Syntax in Spring

public class ResponseEntity<T>

Here,

* T = type of the response body (like String, Student, or any model)

Common methods:

* getBody() → returns response body
* getStatusCode() → returns HTTP status (like OK, NOT\_FOUND)
* getStatusCodeValue() → returns numeric code (like 200, 404)
* getHeaders() → returns response headers

### 🧠 Where It Is Used

#### ✅ 1. On the **Producer side**

To send a custom structured response with:

* body,
* status code,
* and optional headers.

#### ✅ 2. On the **Consumer side**

To receive the same structure from Producer and read:

* body data,
* status,
* and headers.

### 🧩 Example 1 — Producer Side (API Provider)

#### 🎯 Goal

Send a response with data + status code + header.

package in.orcas.rest;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

@RestController

@RequestMapping("/api/student")

public class StudentController {

@PostMapping("/register")

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

String msg = "Student registered successfully with name: " + student.getName();

// Send data + status

return new ResponseEntity<>(msg, HttpStatus.CREATED); // 201 Created

}

}

🧠 **Explanation:**

* ResponseEntity<String> → Response body is a String.
* HttpStatus.CREATED → tells consumer that a new record was created.
* So consumer can check 201 instead of just reading “success”.

### 🧩 Example 2 — Consumer Side (API Caller)

#### 🎯 Goal

Consume the above producer API and print the full response details.

package in.orcas.consumer;

import org.springframework.boot.CommandLineRunner;

import org.springframework.http.\*;

import org.springframework.stereotype.Component;

import org.springframework.web.client.RestTemplate;

@Component

public class StudentConsumerRunner implements CommandLineRunner {

@Override

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

RestTemplate restTemplate = new RestTemplate();

String url = "http://localhost:8080/api/student/register";

// 1️⃣ Request body

Student student = new Student(101, "Pavan", 7500.0);

// 2️⃣ Set headers

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

// 3️⃣ Combine into HttpEntity

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

// 4️⃣ Send request and get full response

ResponseEntity<String> responseEntity = restTemplate.exchange(

url,

HttpMethod.POST,

requestEntity,

String.class

);

// 5️⃣ Print response details

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

System.out.println("Status Code Value : " + responseEntity.getStatusCodeValue());

System.out.println("Status Code Name : " + responseEntity.getStatusCode().name());

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

}

}

### 🧩 Output Example

Response Body : Student registered successfully with name: Pavan

Status Code Value : 201

Status Code Name : CREATED

Response Headers : [Content-Type:"application/json"]

### ✅ Why ResponseEntity Is So Important

| **Without ResponseEntity** | **With ResponseEntity** |
| --- | --- |
| Only sends plain data | Sends full HTTP response info |
| Hard to know success/failure | Easy to identify via status code |
| Limited debugging | Rich details for logs/testing |
| Less control over response | Full control over headers, status, and body |

### 💡 In Real-Time Projects

In **industry**, ResponseEntity is almost **always used** because:

* It gives **complete control** over the HTTP response.
* It helps front-end (Angular/React) apps to **easily check status codes** and display proper messages.
* It makes REST APIs **professional, standard, and testable**.

## Steps to Run

1️⃣ first run Eg: ResponseEntityStudentProducer → port **9090**  
2️⃣ Then run Eg: ResponseEntityStudentConsumer→ port **9091**

**🧩 Step-4: exchange() Method — (Universal HTTP Request Method)**

**💡 Core Concept**

When we use RestTemplate to call other APIs, we have two categories of methods:

| **Category** | **Example** | **Limitation** |
| --- | --- | --- |
| **Shortcut methods** | getForEntity(), postForEntity(), put(), delete() | Each method is specific to one HTTP type (e.g., only GET or only POST). |
| **Universal method** | exchange() | Works for **all HTTP methods** (GET, POST, PUT, DELETE, PATCH, etc.). |

So, the **exchange() method is like the “master method”** — it can perform any HTTP operation you want.

**🧠 Why exchange()?**

In real-time projects:

* We often need to send **headers + body together** (like Authorization, Content-Type, etc.).
* We may need to **control HTTP methods dynamically** (for example, sending a DELETE or PATCH based on a condition).
* Shortcut methods (getForEntity(), postForEntity()) cannot handle such cases easily.

That’s why **developers use exchange() in production** — because it gives **full flexibility**.

**🧩 Syntax**

ResponseEntity<T> response = restTemplate.exchange(

String url,

HttpMethod method,

HttpEntity<?> requestEntity,

Class<T> responseType

);

| **Parameter** | **Meaning** |
| --- | --- |
| **url** | Endpoint URL (e.g., "http://localhost:8080/api/student/register") |
| **method** | HTTP method like HttpMethod.GET, POST, PUT, DELETE, etc. |
| **requestEntity** | Combines headers + body using HttpEntity |
| **responseType** | The Java class type expected in the response (e.g., String.class, Student.class, List.class) |

**⚙️ How it works internally**

1. We create a **Student** object (if body is needed).
2. We create **HttpHeaders** (to set content type, authorization, etc.).
3. We combine both into a **HttpEntity** (like an envelope holding both header + body).
4. Then we pass all of them into exchange() along with the URL and method type.
5. Finally, we get a **ResponseEntity** as output (which contains both status code and body).

**📦 ResponseEntity Structure Recap**

When we call:

ResponseEntity<String> response = restTemplate.exchange(...);

We can get:

response.getStatusCode(); // e.g., 200 OK

response.getBody(); // e.g., "Student saved successfully"

response.getHeaders(); // optional, full response headers

**🎯 Summary**

| **Concept** | **Shortcut Methods** | **exchange()** |
| --- | --- | --- |
| HTTP Type | Fixed | Any (GET, POST, PUT, DELETE) |
| Supports Headers | Limited | ✅ Fully Supported |
| Supports Body | Partial | ✅ Fully Supported |
| Industry Usage | For small apps | ✅ Real-time, large-scale projects |

That’s why **exchange()** is called **“universal request method”** in Spring REST Clients.

**🧩 1️⃣ Core Principle: What Actually Changed**

**🧠 REST Communication Basic Idea:**

Whenever your Spring Boot app talks to another REST API (for example, calling another microservice),  
it needs a **client** to send HTTP requests and receive responses.

Two main clients are available in Spring:

* 🏷️ **RestTemplate** → Old way
* 🚀 **WebClient** → New, modern, reactive way

**🏗️ 2️⃣ Understanding RestTemplate (Old Traditional Way)**

**💬 Think of it like this:**

RestTemplate works **like a simple phone call**.

When you call someone, you wait until they pick up and finish the conversation  
before you can make another call.

That’s exactly how **RestTemplate** works.

When you send a request:

* It **blocks** the current thread.
* It **waits** for the full response.
* Only then, your code continues.

This is called **Blocking / Synchronous** programming.

**✅ Example Flow:**

ResponseEntity<String> response =

restTemplate.exchange(url, HttpMethod.GET, null, String.class);

System.out.println(response.getBody());

Here, until the API responds, the code **stops** on this line.  
If the external API takes 5 seconds — your thread also waits for 5 seconds.

**⚠️ Problem in Modern Systems:**

In real-world enterprise systems (like AWS microservices or fintech apps),  
your service might make **thousands of concurrent API calls**.

If each call **blocks one thread**, your server will soon run out of threads, causing:

* High CPU usage
* Delayed responses
* Slow performance
* Poor scalability

That’s why RestTemplate started becoming outdated.

**⚡ 3️⃣ WebClient — The Modern, Reactive Replacement**

Spring introduced **WebClient** in **Spring 5** (WebFlux module)  
to solve the **blocking problem**.

**💬 Analogy:**

WebClient is like sending a message on WhatsApp —  
you don’t wait for the reply to continue other work.

It uses **Reactive Streams** and **Non-blocking I/O**.  
This means:

* The request is sent asynchronously.
* The thread is **freed immediately**.
* When the response arrives, it triggers a callback to handle it.

This is called **Non-blocking / Asynchronous** behavior.

**✅ Example Flow:**

WebClient client = WebClient.create("http://localhost:8080");

Mono<String> response = client.get()

.uri("/student")

.retrieve()

.bodyToMono(String.class);

response.subscribe(System.out::println);

🧠 Explanation:

* The request is sent immediately.
* Your main thread continues working.
* When data arrives, it executes the subscribe() callback.
* This helps handle **thousands of concurrent requests** efficiently.

**💡 4️⃣ Detailed Comparison Table**

| **Feature** | **RestTemplate** | **WebClient** |
| --- | --- | --- |
| Nature | Blocking | Non-blocking |
| Programming Model | Synchronous | Reactive / Asynchronous |
| Thread Usage | One thread per request | Event-loop (shared threads) |
| Performance | Slower (waits for response) | Faster (continues immediately) |
| Module | spring-web | spring-webflux |
| Supported from | Spring 3.x | Spring 5+ |
| Future Updates | ❌ Deprecated (maintenance mode) | ✅ Active development |
| Ease of Use | Simple (for small apps) | More powerful (for modern apps) |
| Use Case | Small monoliths, internal tools | Microservices, cloud apps, streaming APIs |

**🏢 5️⃣ Real-Time Industry Usage Scenarios**

| **Scenario** | **Common Choice** | **Why** |
| --- | --- | --- |
| Legacy enterprise systems (pre-2020) | RestTemplate | Easy migration, synchronous flow |
| Modern microservices (AWS, GCP, Kubernetes) | ✅ WebClient | Reactive, scalable, lightweight |
| High-load systems (banking, e-commerce, analytics) | ✅ WebClient | Handles 1000+ parallel requests easily |
| Small CRUD web apps | RestTemplate | Simple, quick to build |
| Real-time streaming / chat / notification apps | ✅ WebClient | Handles async data streams |

**⚙️ 6️⃣ Spring’s Official Position**

Spring team (Pivotal) has clearly mentioned in documentation:

“RestTemplate is now in **maintenance mode** —  
for new applications, we **recommend using WebClient**.”

So:

* RestTemplate still **works perfectly fine**.
* But it’s **no longer evolving**.
* All new Spring projects prefer **WebClient**.

**🧠 7️⃣ In Real-Time Projects (Industry Reality)**

**✅ Example 1 — Inside AWS Microservice Architecture**

* PaymentService calls OrderService, InventoryService, and ShippingService.
* If all 3 are called using RestTemplate → 3 threads will be blocked.
* Using WebClient → all 3 calls run in **parallel** on the same thread using reactive callbacks.  
  👉 This reduces memory usage and increases throughput dramatically.

**✅ Example 2 — When You Should Still Use RestTemplate**

In smaller apps or internal admin panels where:

* You make few API calls.
* You don’t need async or reactive behavior.
* Simplicity is more important than scalability.

RestTemplate is still fine here.  
(That’s why many old Spring projects still use it.)

**🧭 8️⃣ Final Summary**

| **Term** | **Description** |
| --- | --- |
| **RestTemplate** | Old, blocking client — good for simple or legacy apps. |
| **WebClient** | Modern, non-blocking, reactive client — ideal for cloud & microservices. |
| **Replacement** | ✅ Yes, WebClient is the official replacement for RestTemplate. |
| **Industry Trend** | 90% of new Spring projects now use WebClient. |