**🔹 1. Core Concept — What are Path and Query Parameters?**

When we send data **inside the URL**, it can be done in **two common ways**:

**🧩 a) Path Parameters**

* These are part of the **URL path itself**.
* Used to identify a specific **resource**.
* Written directly **inside the URL** using {} placeholders.

Example:

GET /students/101

Here 101 is a **path variable** — it identifies the student with roll no 101.  
So, this request says: “Give me details of student number 101.”

**🧩 b) Query Parameters**

* These are sent **after a “?”** in the URL.
* Used to send **filtering, sorting, or searching data**.
* Written as key–value pairs joined with &.

Example:

GET /students/search?course=java&city=hyd

This means:  
“Search students who are in course = java and city = hyd.”

**🔹 2. Why Learn This in WebClient?**

Earlier, with RestTemplate, we used:

getForObject("http://localhost:8081/students/{id}", Student.class, 101);

But RestTemplate is **old (synchronous and blocking)**.  
In **modern Spring Boot microservices**, we prefer **WebClient** because it is:

* ✅ **Non-blocking** and **reactive** (better performance)
* ✅ Works well with **Spring WebFlux**
* ✅ **Future replacement** of RestTemplate

So, everything we did with RestTemplate can now be done in **WebClient style**, just more flexible and modern.

**🔹 3. How WebClient Works (in simple terms)**

You can think of WebClient as:

“A modern HTTP client used to send GET, POST, PUT, DELETE requests — same like RestTemplate, but smarter and non-blocking.”

To use it:

1. You create a **WebClient** object (like a RestTemplate object).
2. You specify:
   * The **base URL** (like http://localhost:8081)
   * The **path and query parameters**
3. You make a request (GET/POST/etc.)
4. You call .retrieve() to fetch the response.
5. You convert the response into a Java type using .bodyToMono() or .bodyToFlux().

**🔹 4. Passing Path Parameters with WebClient**

Let’s say the Producer API is:

GET /students/{id}

**🧠 Core idea:**

We’ll replace {id} with an actual number (like 101) dynamically using .uri() method.

**💻 Example 1: Path Parameter with WebClient**

package in.orcas.runner;

import org.springframework.boot.CommandLineRunner;

import org.springframework.stereotype.Component;

import org.springframework.web.reactive.function.client.WebClient;

import in.orcas.model.Student;

@Component

public class PathParamWebClientRunner implements CommandLineRunner {

@Override

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

System.out.println("🚀 Calling Producer API using Path Parameter...");

// Step 1: Create WebClient object with base URL

WebClient webClient = WebClient.create("http://localhost:8081");

// Step 2: Make GET call using Path Parameter

Student student = webClient.get()

.uri("/students/{id}", 101) // Replace {id} with 101

.retrieve()

.bodyToMono(Student.class)

.block(); // block() to wait for result (used in CommandLineRunner)

// Step 3: Print the response

System.out.println("✅ Received Student: " + student);

}

}

**🔍 Explanation**

* .uri("/students/{id}", 101) → replaces {id} with 101 automatically.
* .retrieve() → sends the request and waits for the response.
* .bodyToMono(Student.class) → converts JSON response into Student object.
* .block() → since we are not in a reactive stream, it blocks until result comes.

So here, WebClient automatically calls:

GET http://localhost:8081/students/101

**🔹 5. Passing Query Parameters with WebClient**

Let’s say the Producer API is:

GET /students/search?course=java&city=hyd

We’ll dynamically build this URL using .uri() with a lambda expression.

**💻 Example 2: Query Parameters with WebClient**

package in.orcas.runner;

import org.springframework.boot.CommandLineRunner;

import org.springframework.stereotype.Component;

import org.springframework.web.reactive.function.client.WebClient;

@Component

public class QueryParamWebClientRunner implements CommandLineRunner {

@Override

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

System.out.println("🚀 Calling Producer API using Query Parameters...");

WebClient webClient = WebClient.create("http://localhost:8081");

// Step 1: Build URL with query parameters

String response = webClient.get()

.uri(uriBuilder -> uriBuilder

.path("/students/search")

.queryParam("course", "java")

.queryParam("city", "hyd")

.build())

.retrieve()

.bodyToMono(String.class)

.block();

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

}

}

**🔍 Explanation:**

* .uri(uriBuilder -> uriBuilder.path("/students/search").queryParam(...).build())  
  → This dynamically builds a URL like:
* http://localhost:8081/students/search?course=java&city=hyd
* .retrieve() → sends the GET request.
* .bodyToMono(String.class) → takes the API response as a String.

So we are successfully sending **query parameters** in WebClient style.

**🔹 6. Real-Time Uses**

| **Type** | **Real-time Use Case** | **Example** |
| --- | --- | --- |
| **Path Parameter** | Fetch one record by ID | /orders/555, /users/10 |
| **Query Parameter** | Filter or search | /orders?status=delivered&month=oct |
| **Combined** | Both together | /users/10/orders?status=pending |

In real consumer microservices, these parameterized URLs are **very common** because they make the API flexible and dynamic.

**🔹 7. Summary (Quick Recap)**

| **Concept** | **Description** |
| --- | --- |
| **Path Parameter** | Sent inside the URL (like /students/101) |
| **Query Parameter** | Sent after ? (like /students/search?course=java) |
| **WebClient** | Modern, non-blocking alternative to RestTemplate |
| **Retrieve** | Executes the HTTP call |
| **bodyToMono() / block()** | Converts and waits for the response |
| **Usage** | Used heavily in microservice-to-microservice communication |

Execution flow:

First execute Eg: WebClientPathAndQueryParametersProducer on one port, then on another port

execute Eg: WebClientPathAndQueryParametersConsumer

# HATEOAS (Hypermedia as the Engine of Application State)

## 🔹 1. What Is the Core Idea?

Let’s start simple 👇

When a REST API sends data to the client (for example, a Student object), normally it just sends the **data only** — like:

{

"id": 101,

"name": "Ravi",

"course": "Java",

"city": "Hyd"

}

That’s fine — but imagine you’re a client (or a mobile app) using this response.  
How would you know what to do next?  
For example:

* Where to update this student?
* Where to delete it?
* Where to get all students?

You’d need to **remember or hardcode** all URLs in your frontend — which makes the system **tightly coupled** and hard to maintain.

### 💡 Enter HATEOAS

HATEOAS means **Hypermedia as the Engine of Application State**.

👉 It simply means:  
**Every response from the server should include not only data but also navigation links about possible next actions.**

So the API becomes **self-descriptive** — clients don’t need to “remember” routes; they can follow the links provided by the API itself.

### 🧠 Simple Analogy

Think of HATEOAS like **Google Maps navigation**:

You start at one location (current resource) → the map automatically shows possible next paths (links).  
You don’t memorize roads; you just click “next turn”.

Similarly, HATEOAS adds **links in JSON** so the client can directly follow them.

### 💬 Example Without HATEOAS

{

"id": 101,

"name": "Ravi",

"course": "Java",

"city": "Hyd"

}

### 💬 Example With HATEOAS

{

"id": 101,

"name": "Ravi",

"course": "Java",

"city": "Hyd",

"\_links": {

"self": { "href": "http://localhost:8080/students/101" },

"update": { "href": "http://localhost:8080/students/update/101" },

"delete": { "href": "http://localhost:8080/students/delete/101" },

"all": { "href": "http://localhost:8080/students" }

}

}

Now the response itself tells what can be done next with this student.

## 🔹 2. Why Use HATEOAS?

| **Problem Without It** | **Benefit With It** |
| --- | --- |
| Clients must remember endpoint URLs | API becomes self-explaining |
| Changing endpoint breaks clients | Links are dynamic in responses |
| Harder to navigate | Easy navigation through embedded links |
| No “next step” information | Clients can discover available actions |

That’s why it’s widely used in **enterprise-level REST APIs** to make them **discoverable** and **evolvable**.

## 🔹 3. How Spring Boot Supports HATEOAS

Spring provides a library — **Spring HATEOAS** — which gives helper classes to build such links easily.

📦 Dependency:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-hateoas</artifactId>

</dependency>

Main classes:

* EntityModel<T> → wraps a single resource and adds links
* CollectionModel<T> → wraps multiple resources
* Link → represents a hyperlink

## 🔹 4. Step-by-Step Real Example

Let’s build a **StudentService** that returns student data **with HATEOAS links**.

### 🧩 Step 1: Model — Student.java

package in.orcas.model;

public class Student {

private Integer id;

private String name;

private String course;

private String city;

public Student(Integer id, String name, String course, String city) {

this.id = id;

this.name = name;

this.course = course;

this.city = city;

}

public Student() {}

// Getters, Setters

public Integer getId() { return id; }

public void setId(Integer id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getCourse() { return course; }

public void setCourse(String course) { this.course = course; }

public String getCity() { return city; }

public void setCity(String city) { this.city = city; }

}

### 🧩 Step 2: Controller — StudentController.java

package in.orcas.controller;

import java.util.List;

import org.springframework.hateoas.\*;

import org.springframework.hateoas.server.mvc.WebMvcLinkBuilder;

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

import in.orcas.model.Student;

@RestController

@RequestMapping("/students")

public class StudentController {

// Simulate student data

private List<Student> list = List.of(

new Student(101, "Ravi", "Java", "Hyd"),

new Student(102, "Kiran", "Python", "Chn"),

new Student(103, "Asha", "Java", "Pune")

);

// 1️⃣ Get Student by ID (with HATEOAS links)

@GetMapping("/{id}")

public EntityModel<Student> getStudent(@PathVariable Integer id) {

Student s = list.stream()

.filter(st -> st.getId().equals(id))

.findFirst()

.orElse(null);

// Wrap Student in EntityModel

EntityModel<Student> model = EntityModel.of(s);

// Add navigation links

model.add(WebMvcLinkBuilder.linkTo(

WebMvcLinkBuilder.methodOn(StudentController.class)

.getStudent(id)).withSelfRel());

model.add(WebMvcLinkBuilder.linkTo(

WebMvcLinkBuilder.methodOn(StudentController.class)

.getAllStudents()).withRel("all-students"));

model.add(WebMvcLinkBuilder.linkTo(

WebMvcLinkBuilder.methodOn(StudentController.class)

.deleteStudent(id)).withRel("delete-student"));

return model;

}

// 2️⃣ Get All Students

@GetMapping

public CollectionModel<Student> getAllStudents() {

return CollectionModel.of(list);

}

// 3️⃣ Delete Student (just for demonstration)

@DeleteMapping("/{id}")

public String deleteStudent(@PathVariable Integer id) {

return "Student with ID " + id + " deleted (demo only)";

}

}

### 🧩 Step 3: Main Class — HateoasApp.java

package in.orcas;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class HateoasApp {

public static void main(String[] args) {

SpringApplication.run(HateoasApp.class, args);

}

}

## 🔹 5. Output (API Response)

**When you hit**  
👉 GET http://localhost:8080/students/101

**You’ll get:**

{

"id": 101,

"name": "Ravi",

"course": "Java",

"city": "Hyd",

"\_links": {

"self": {

"href": "http://localhost:8080/students/101"

},

"all-students": {

"href": "http://localhost:8080/students"

},

"delete-student": {

"href": "http://localhost:8080/students/101"

}

}

}

## 🔹 6. What Happened Internally

* Spring HATEOAS automatically adds \_links section in JSON.
* Each link represents a **possible next action**.
* This makes the API **navigable** — a client can “follow” links like a browser follows hyperlinks.

## 🔹 7. Real-Time Use Cases of HATEOAS

| **Use Case** | **Description** |
| --- | --- |
| **API Navigation** | Clients can discover related resources dynamically |
| **Version Tolerance** | If endpoint URLs change, clients just follow updated links |
| **HAL Browsers** | Tools like HAL Explorer can automatically display linked APIs |
| **Microservice Chains** | Each service provides links to related services or operations |

## 🔹 8. Summary (Quick Recap)

| **Concept** | **Explanation** |
| --- | --- |
| **Full Form** | Hypermedia as the Engine of Application State |
| **Purpose** | Make REST APIs self-descriptive and discoverable |
| **Key Classes** | EntityModel, CollectionModel, Link |
| **Benefit** | Clients navigate dynamically without hardcoding URLs |
| **Spring Support** | Provided through spring-boot-starter-hateoas |
| **Real Output** | JSON response includes \_links with navigation |

## 🧠 Key Takeaway

Traditional REST gives you **data only**.  
HATEOAS gives you **data + navigation**.

It’s like a REST API that **teaches clients how to explore itself**.

Execution flow:

First execute Eg: HateosProducer on one port, then on another port

execute Eg: HateosConsumer