## WebClient — Core Concept

### 🌍 What is WebClient?

➡️ WebClient is a **non-blocking, reactive HTTP client** provided by Spring to call REST APIs.

It is the **modern replacement** for the older RestTemplate.

Think of it like this:

| **Type** | **Tool** | **Nature** | **How it works** |
| --- | --- | --- | --- |
| Old | RestTemplate | Blocking | Waits until response arrives |
| New | WebClient | Non-blocking | Works asynchronously (reactive) |

### 🧠 The Core Principle

The heart of WebClient is **reactive and event-driven** programming.

That means —  
it **doesn’t wait** for the response to come.  
Instead, it **creates a pipeline** (like a promise) that says:

“When the response arrives, please call this code.”

So, when you use WebClient, you are not directly getting the response right away.  
You are **defining what should happen** when the response arrives later — that’s why it returns Mono<T> or Flux<T>.

### 💡 Real-world Analogy — “Pizza Order System 🍕”

Imagine you call a restaurant to order pizza.

* **RestTemplate way (Blocking)**:  
  You stay on the call until the pizza is made and delivered. You waste your time waiting.
* **WebClient way (Non-blocking)**:  
  You place the order and hang up.  
  When the pizza is ready, they call you back (asynchronous callback).  
  You are free to do other work in between.

That’s the beauty of WebClient —  
you don’t block the main thread, so your app stays responsive and fast.

### 🔩 How WebClient Works Internally

When you use WebClient:

1. It builds a **pipeline (Mono/Flux)** describing:
   * Which URL to call
   * What method to use (GET/POST/PUT/DELETE)
   * What headers and body to send
   * What type of data to expect in response
2. It **doesn’t call** the API immediately.  
   It stays cold (just like you learned with Mono).
3. When you call **subscribe()**,  
   it **starts the network request** in the background (on a worker thread).
4. When the response arrives:
   * If successful → it converts JSON → Java object → calls your success callback (accept()).
   * If failed → it calls your error callback.

## 🚀 Step-by-Step Example Explanations

Let’s now look at each HTTP method — one by one.

### 🟢 1️⃣ GET Request — Fetching Data

#### 🧠 Core Idea

You use GET to **read/fetch data** from a REST API.  
In WebClient, it is completely asynchronous — you tell it what to fetch and what to do when data comes.

#### 🧩 Example

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

Mono<Student> mono = client.get()

.uri("/api/student/{id}", 101)

.retrieve()

.bodyToMono(Student.class);

mono.subscribe(student -> System.out.println("Received: " + student),

error -> System.out.println("Error: " + error.getMessage()));

#### 🔍 Explanation

* WebClient.create(...) → creates the client with base URL.
* .get() → tells it this is a GET request.
* .uri(...) → defines the endpoint and path variables.
* .retrieve() → tells WebClient to actually fetch the response body.
* .bodyToMono(Student.class) → converts JSON → Java object (Student).
* .subscribe(...) → starts the async request and waits for callback when data arrives.

💬 Until you call subscribe(), nothing is sent.  
Once you call it, WebClient sends the request asynchronously.

### 🟠 2️⃣ POST Request — Sending Data

#### 🧠 Core Idea

Use POST when you want to **send data** (like a JSON object) to create something on the server.

#### 🧩 Example

Student student = new Student(101, "John", "john@gmail.com");

Mono<Student> mono = client.post()

.uri("/api/student")

.header(HttpHeaders.CONTENT\_TYPE, MediaType.APPLICATION\_JSON\_VALUE)

.body(Mono.just(student), Student.class)

.retrieve()

.bodyToMono(Student.class);

mono.subscribe(response -> System.out.println("Created: " + response),

error -> System.out.println("Error: " + error.getMessage()));

#### 🔍 Explanation

* .post() → tells WebClient to make a POST request.
* .header(...) → sets the content type to JSON.
* .body(Mono.just(student), Student.class) → sends our object as JSON in body.
* .retrieve() → fetches the response.
* .bodyToMono(Student.class) → converts response JSON → Student.
* .subscribe(...) → triggers the request and waits for async callback.

💬 The POST call sends data to the server, but the thread doesn’t wait.  
The response is handled asynchronously when it arrives.

### 🔵 3️⃣ PUT Request — Updating Data

#### 🧠 Core Idea

Use PUT when you want to **update** existing data on the server.

#### 🧩 Example

Student updated = new Student(101, "John Updated", "john.new@gmail.com");

Mono<Student> mono = client.put()

.uri("/api/student/{id}", 101)

.header(HttpHeaders.CONTENT\_TYPE, MediaType.APPLICATION\_JSON\_VALUE)

.body(Mono.just(updated), Student.class)

.retrieve()

.bodyToMono(Student.class);

mono.subscribe(res -> System.out.println("Updated: " + res),

err -> System.out.println("Error: " + err.getMessage()));

#### 🔍 Explanation

* .put() → tells WebClient to make a PUT request.
* .uri("/api/student/{id}", 101) → points to the record to be updated.
* .body(...) → sends updated student data as JSON.
* .bodyToMono(Student.class) → maps response JSON → Student.
* .subscribe(...) → starts the process asynchronously.

💬 The PUT request modifies existing resource — once it’s complete, WebClient calls your success callback automatically.

### 🔴 4️⃣ DELETE Request — Removing Data

#### 🧠 Core Idea

Use DELETE when you want to **remove** an existing resource.

#### 🧩 Example

Mono<Void> mono = client.delete()

.uri("/api/student/{id}", 101)

.retrieve()

.bodyToMono(Void.class);

mono.subscribe(v -> System.out.println("Deleted successfully."),

err -> System.out.println("Error: " + err.getMessage()));

#### 🔍 Explanation

* .delete() → creates DELETE request.
* .uri(...) → identifies which record to delete.
* .retrieve() → prepares to receive response.
* .bodyToMono(Void.class) → means there is no body in response.
* .subscribe(...) → sends the delete request asynchronously.

💬 Once deletion is done, your success callback executes automatically — just like in the pizza delivery example 🍕.

## ⚡ Summary Table

| **HTTP Method** | **Use** | **Example API** | **Body Required?** | **Typical Response** |
| --- | --- | --- | --- | --- |
| GET | Fetch data | /api/student/{id} | ❌ | Student object |
| POST | Create new data | /api/student | ✅ | Created Student |
| PUT | Update existing data | /api/student/{id} | ✅ | Updated Student |
| DELETE | Remove data | /api/student/{id} | ❌ | Void / Message |

## 🧠 Final Recap

| **Concept** | **Explanation** |
| --- | --- |
| WebClient | Reactive (non-blocking) HTTP client |
| .get(), .post(), .put(), .delete() | Define the HTTP method |
| .uri() | Defines endpoint and path variables |
| .header() | Adds headers like Content-Type |
| .body() | Sends request body (for POST/PUT) |
| .retrieve() | Tells WebClient to fetch response |
| .bodyToMono() | Converts JSON → Java Object |
| .subscribe() | Starts async call and registers callbacks |
| Consumer.accept() | Automatically called when response/error arrives |

Execution flow:

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

execute Eg: WebClientConsumer

**🔹 What is CommandLineRunner?**

In Spring Boot, **CommandLineRunner** is a **special interface** that lets you **run some code automatically** **after the application starts** — that is, **just after Spring Boot finishes loading all beans** and the **application context is ready**.

You can think of it like:

“When my Spring Boot app is fully ready, please run this piece of code automatically once.”

**🔹 Why we need CommandLineRunner?**

In **real-time microservice consumer applications**, we often need to **trigger some tasks automatically** right after startup.  
For example:

* A consumer microservice may need to **call another microservice’s API** (producer) to get initial data.
* Or it may need to **register itself** to a service registry.
* Or it may **preload cache**, **initialize database tables**, or **start message consumers**.

Without CommandLineRunner, you would have to **manually call** such setup code somewhere else — which is error-prone and not clean.

Hence, Spring provides CommandLineRunner as a **startup hook**.

**🔹 When does CommandLineRunner run?**

It runs **immediately after Spring Boot finishes**:

1. Loading the application context
2. Creating all beans
3. Finishing dependency injection
4. But **before** the application fully goes into idle state (waiting for requests)

So, at this point, **all your components and services are ready** to be used safely.

**🔹 Where do we use it?**

We typically use it inside:

* **Main class** (where @SpringBootApplication is placed), or
* **Any @Component / @Service class**

That means you can place it **anywhere** inside your Spring project where you want **startup logic** to run.

**🔹 How do we implement it?**

We just make a class implement the CommandLineRunner interface and override its single method:

@Override

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

This run() method automatically executes after the app starts.

**🧩 Example: Triggering REST API call automatically**

Let’s say we have a **consumer microservice** that should **call a producer API** when it starts — maybe to fetch product details.

package in.orcas.runner;

import org.springframework.boot.CommandLineRunner;

import org.springframework.stereotype.Component;

import org.springframework.web.client.RestTemplate;

@Component

public class MyStartupRunner implements CommandLineRunner {

@Override

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

System.out.println("🚀 Application started. Running startup logic...");

// Step 1: Create RestTemplate object

RestTemplate rt = new RestTemplate();

// Step 2: Define API URL (producer microservice endpoint)

String url = "http://localhost:8081/products";

// Step 3: Call the API and get response

String response = rt.getForObject(url, String.class);

// Step 4: Print or process the data

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

}

}

**🔍 Explanation:**

* This class is annotated with @Component → Spring automatically detects and runs it.
* When the app starts:
  1. Spring loads all beans.
  2. Spring finds this CommandLineRunner bean.
  3. It automatically executes the run() method.
* Inside run(), we used RestTemplate to **call an external API automatically**.

So without any manual trigger, your microservice **acts like a consumer** — fetching or initializing data right when it boots up.

**🔹 Real-Time Use Cases**

Here are **common real-time scenarios** where CommandLineRunner is used:

| **Use Case** | **Description** |
| --- | --- |
| **API Preload** | Automatically call another microservice to fetch startup data |
| **Cache Warm-up** | Load some data into cache (like Redis) on startup |
| **Health Check** | Ping dependent services to ensure connectivity |
| **Consumer Registration** | Register this service to API gateway or service registry |
| **Database Initialization** | Create initial data or validate table structure |
| **Kafka/RabbitMQ listener setup** | Start listening to queues as soon as the app starts |

**🔹 Difference between CommandLineRunner and ApplicationRunner**

Both are almost same — both run at startup.  
The only difference is:

* CommandLineRunner gives **arguments as String array**
* ApplicationRunner gives **arguments as ApplicationArguments object** (more structured)

So most developers prefer **CommandLineRunner** because it’s simple.

**🔹 Summary (Easy Recap)**

| **Concept** | **Explanation** |
| --- | --- |
| **Purpose** | To run custom code automatically when the Spring Boot app starts |
| **Interface** | CommandLineRunner |
| **Method** | public void run(String... args) |
| **Execution Time** | After all beans are created, before app is fully up |
| **Used For** | Auto API calls, preload, initial setup, etc. |
| **Where Defined** | Inside any @Component or main class |

Execution flow:

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

execute Eg: CommandLineRunnerConsumer

**🌱 Spring Boot Application Startup Flow (Step-by-Step)**

**🧩 Step 1️⃣ — main() executes first**

When you run your Spring Boot app, Java starts from the **main() method** in your main class (for example WebClientConsumerApplication.java).

public static void main(String[] args) {

SpringApplication.run(WebClientConsumerApplication.class, args);

}

➡️ Here, the **main()** method doesn’t directly execute your logic.  
Instead, it calls SpringApplication.run() — which is like saying

“Hey Spring, start the container and get everything ready.”

**⚙️ Step 2️⃣ — Spring Boot engine starts**

When SpringApplication.run() is executed, Spring Boot internally performs several startup tasks:

| **Task** | **Description** |
| --- | --- |
| 1️⃣ | Loads application.properties / application.yml |
| 2️⃣ | Scans your project for components (@Component, @Service, @Repository, @Controller, etc.) |
| 3️⃣ | Creates Spring Beans for those components |
| 4️⃣ | Injects dependencies using @Autowired |
| 5️⃣ | Configures the Web environment (port, Tomcat, etc.) |
| 6️⃣ | Prepares everything before the app officially becomes "ready" |

At this point, your Spring application is **fully initialized**, but your own business logic hasn’t executed yet.

**🚀 Step 3️⃣ — Spring searches for CommandLineRunner beans**

Now Spring Boot internally asks:

“Do I have any beans that implement CommandLineRunner or ApplicationRunner?”

If **yes**, Spring automatically calls their run() method **after** all beans are ready and the context is initialized.

**🧠 Step 4️⃣ — The run() method of CommandLineRunner executes**

Example:

@Component

public class StartupRunner implements CommandLineRunner {

@Autowired

private StudentWebClientService service;

@Override

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

System.out.println("✅ Consumer started — calling Producer APIs...");

service.getAllStudents();

service.saveStudent(...);

}

}

➡️ Now your actual logic runs automatically —  
like calling Producer APIs, checking database, or performing any startup task.

**🪄 Step 5️⃣ — Application fully running**

After the CommandLineRunner.run() method completes:

* The API server (Tomcat) is already listening on the configured port.
* The console prints startup logs and your business logic outputs.
* Your app is now 100% ready to accept external requests.

**🧭 Final Summary — Full Execution Order**

| **Step** | **What Happens** | **Who Runs It** |
| --- | --- | --- |
| 1️⃣ | JVM starts app | main() |
| 2️⃣ | Spring Boot starts container | SpringApplication.run() |
| 3️⃣ | Beans are created and injected | Spring |
| 4️⃣ | Spring finds CommandLineRunner beans | Spring |
| 5️⃣ | Executes run() of those beans | Spring |
| 6️⃣ | Application fully running | Spring |