# 🌟 Core principle — Synchronous vs Asynchronous (simple & exact)

**Synchronous (blocking):**

* When you call an API synchronously, your code **waits** for the response before continuing.
* The thread that made the call is **blocked** (does nothing else) until the reply arrives.
* Simple model, easy to reason about, but **does not scale** well when many requests are outstanding (each call consumes one thread while waiting).

**Asynchronous (non-blocking):**

* When you call an API asynchronously, your code **does not wait**. It sends the request and continues doing other work.
* The call returns a **promise** (a handle) — e.g., Mono<T> or Future<T>.
* When the response arrives, a callback or reactive pipeline handles it. The thread that initiated the call is **free** to do other work.
* Better scalability: fewer threads needed for many concurrent calls; higher throughput under load.

**Key difference (one line):**

* Sync = I wait here until you return.
* Async = I send request, keep working, and handle reply when it comes.

# ⚙️ Why this matters in real projects

* If a service will make **hundreds or thousands** of outgoing calls concurrently (microservices, external APIs), **blocking** ties up many threads and memory — becomes slow or crashes under load.
* For **simple admin scripts** or small apps where calls are rare, blocking is okay and simpler.
* Async enables **high concurrency** with fewer threads (good for cloud, microservices, streaming).

# 🔁 Thread behavior (short, precise)

* **Blocking (RestTemplate)**: each call → one thread waits. If 1000 simultaneous calls → ~1000 waiting threads.
* **Non-blocking (WebClient + Reactor)**: uses an **event loop / small thread pool**; many IO operations handled by few threads. When data arrives, reactor schedules callbacks — threads are reused efficiently.

# ✅ When to use which

* Use **synchronous** for: simple apps, scripts, quick prototypes, admin tools.
* Use **asynchronous** for: microservices, high-load systems, streaming, real-time dashboards, APIs that call many other services.

# Example 1 — Synchronous call (RestTemplate)

This is how a typical blocking call looks. The calling thread waits.

// Blocking example with RestTemplate (runs inside a Spring Boot CommandLineRunner or main)

RestTemplate rest = new RestTemplate();

String url = "http://localhost:8080/api/student/101";

// This call blocks until the HTTP response arrives

ResponseEntity<Student> resp = rest.exchange(url, HttpMethod.GET, null, Student.class);

System.out.println("Thread after call: " + Thread.currentThread().getName());

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

System.out.println("Student: " + resp.getBody());

**Behavior:** the exchange() call does not return until the remote server responds. The thread is idle while waiting.

# Example 2 — Asynchronous call (WebClient, non-blocking)

This is the non-blocking style using Reactor’s Mono. The call returns immediately; result handled later.

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

// Prepare async request (returns immediately)

Mono<Student> mono = client.get()

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

.retrieve()

.bodyToMono(Student.class);

// Attach what to do when response arrives

public class StudentCallback implements Consumer<Student> {

@Override

public void accept(Student student) {

System.out.println("Callback thread: " + Thread.currentThread().getName());

System.out.println("Got student: " + student);

}

}

public class ErrorCallback implements Consumer<Throwable> {

@Override

public void accept(Throwable err) {

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

}

}

// Then subscribe:

mono.subscribe(new StudentCallback(), new ErrorCallback());

**🌍 Background: What’s Happening Here?**

In normal (synchronous) code, when you call an API (like get()), your program **waits** until the server sends back the response.  
But in **asynchronous mode**, your program **doesn’t wait** — it sends the request and continues doing other work.  
Later, when the response arrives, **a callback function** runs automatically to handle it.

**🧩 Code Breakdown**

Mono<Student> mono = client.get()

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

.retrieve()

.bodyToMono(Student.class);

Let’s go line-by-line 👇

**1️⃣ client.get()**

You are creating a **GET request** using WebClient.

Here,  
client is an object of WebClient, already created like this:

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

This means all requests will start from the base URL http://localhost:8080.

**2️⃣ .uri("/api/student/{id}", 101)**

Here you are defining **the endpoint URL** and **path variable**.

* "/api/student/{id}" → endpoint pattern
* {id} → is replaced with 101 dynamically

So the final URL becomes:

http://localhost:8080/api/student/101

**3️⃣ .retrieve()**

This line tells the WebClient:

“Send the request and wait for a response — but don’t block the thread yet.”

It starts the HTTP request and prepares the response pipeline.

**4️⃣ .bodyToMono(Student.class)**

This converts the response body (which will come as JSON) into a Student Java object.

* Mono<Student> → means **“a future object”** that will **eventually contain one Student object** when the response arrives.

Think of Mono as a **“promise”** of a value that will come later.

**🧩 Code Breakdown**

**✅ 1️⃣ StudentCallback Class**

public class StudentCallback implements Consumer<Student> {

@Override

public void accept(Student student) {

System.out.println("Callback thread: " + Thread.currentThread().getName());

System.out.println("Got student: " + student);

}

}

**🔍 Explanation**

* implements Consumer<Student> → means this class defines **what to do when a Student object is received** from the API response.
* The accept(Student student) method is automatically called **by the Mono** when the async response arrives.
* Inside it:
  + Thread.currentThread().getName() prints the **thread name** that executed the callback.
    - This helps confirm it’s running in a **different thread** (not the main thread), because async operations use worker threads.
  + System.out.println("Got student: " + student) → prints the received data (the API response body).

🧠 Think of this class as:

“What should I do when the student data is successfully received?”

**✅ 2️⃣ ErrorCallback Class**

public class ErrorCallback implements Consumer<Throwable> {

@Override

public void accept(Throwable err) {

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

}

}

**🔍 Explanation**

* implements Consumer<Throwable> → means this class defines **what to do when an error occurs** during the async call.
* The accept(Throwable err) method is automatically called **if the request fails** (e.g., invalid URL, server down, 404, etc.).
* System.err.println(...) prints the error message on the console (in red usually).

🧠 Think of this class as:

“What should I do if something goes wrong while fetching the student?”

**✅ 3️⃣ Subscription**

mono.subscribe(new StudentCallback(), new ErrorCallback());

**🔍 Explanation**

* mono.subscribe() is how you **attach the callbacks** to your reactive pipeline.

| **Argument** | **Meaning** | **Type** |
| --- | --- | --- |
| new StudentCallback() | Called when data (Student) is successfully received | Consumer<Student> |
| new ErrorCallback() | Called when any exception/error happens | Consumer<Throwable> |

So when the WebClient async call finishes:

* ✅ If **successful** → StudentCallback.accept() runs
* ❌ If **error occurs** → ErrorCallback.accept() runs

**⚙️ Execution Flow (Step-by-Step)**

1️⃣ WebClient.get()...retrieve().bodyToMono(Student.class)  
→ Starts an async call to fetch a Student object.

2️⃣ The program **does not wait** here (it continues executing other code).

3️⃣ When the response comes back later:

* If successful → StudentCallback.accept(student) is invoked.
* If failed → ErrorCallback.accept(error) is invoked.

4️⃣ These callbacks run in a **different thread** (managed by Reactor framework).

**🧾 Example Output**

Assume the API returned:

{ "id": 101, "name": "Pavan", "city": "Hyderabad" }

Output might be:

Callback thread: reactor-http-nio-3

Got student: Student{id=101, name='Pavan', city='Hyderabad'}

If the API endpoint was invalid, then:

Error: 404 NOT\_FOUND from GET http://localhost:8080/api/student/101

**💬 In Simple Words**

* StudentCallback → what to do **when success** happens.
* ErrorCallback → what to do **when error** happens.
* subscribe() → **connects** your WebClient’s async result to your callback logic.

This pattern makes your code **non-blocking**, meaning your app stays free to do other work instead of waiting.

**How subscribe() and accept() Work Together**

Let’s recall your doubt:

“We didn’t call accept() anywhere, so how did it run automatically when response came? And what does subscribe() really do?”

To understand that, let’s break it like a small story 👇

**🏗️ Step 1: What is Mono?**

When you call this:

Mono<Student> mono = client.get()

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

.retrieve()

.bodyToMono(Student.class);

* You **did not** make a network call *yet* ❌
* You just **created a pipeline** that *describes*:

“Hey Spring, when someone subscribes to me,  
I will make a GET request, receive JSON, convert it into Student.”

💬 Think of it like a **recipe card** — it tells what to do, but it doesn’t start cooking until someone asks for food. 🍳

So mono is **cold** — meaning *no request is fired yet*.

**⚡ Step 2: What Happens When You Call subscribe()**

When you write:

mono.subscribe(new StudentCallback(), new ErrorCallback());

✅ At this exact moment:

* The pipeline becomes **active** (hot).
* The actual HTTP request is sent by WebClient (asynchronously).
* When the response (or error) arrives:
  + If success → it calls StudentCallback.accept(student)
  + If error → it calls ErrorCallback.accept(error)

💬 You can think:

subscribe() = “Start the process and tell me what to do when it finishes or fails.”

Without subscribe(), the WebClient won’t even send the request.

**🔄 Step 3: Who Calls accept()?**

Here’s the magic part ✨

When you call subscribe(),  
WebClient registers your Consumer objects (the callbacks) inside the **reactor framework** (Project Reactor’s internal engine).

Then:

* When the HTTP response is received, the reactor engine automatically **invokes** the accept() method of your success callback.
* If any exception occurs (like timeout, 404, etc.), the reactor engine automatically **invokes** the accept() method of your error callback.

So you never call accept() manually —  
the **reactor library** calls it for you when data is ready.

**🔍 Analogy — “Pizza Order Example” 🍕**

Let’s imagine Mono as a **Pizza order pipeline**:

| **Action** | **Analogy** |
| --- | --- |
| Create Mono | You place a pizza order, but it’s not made yet. |
| subscribe() | You confirm the order → kitchen starts cooking 🍕 |
| Success callback (StudentCallback) | The delivery boy calls when pizza arrives ✅ |
| Error callback (ErrorCallback) | The restaurant calls if the oven broke 🔥 |

You don’t bake the pizza yourself (accept());  
the restaurant (Reactor engine) does it **and calls you back** when it’s done.

**⚙️ Step 4: What Actually Happens Inside Reactor**

1. subscribe() tells the reactor:
   * “Start executing this async task (HTTP call).”
2. Reactor:
   * Spawns or uses a worker thread (like reactor-http-nio-3).
   * Sends the request.
3. When response arrives:
   * Deserializes JSON → Java object (Student).
   * Invokes the accept(student) of your success consumer.
4. If an error happens:
   * Invokes the accept(error) of your error consumer.

All this happens **behind the scenes** using **Publisher–Subscriber pattern**.

**🧩 Publisher–Subscriber Relationship**

| **Role** | **Interface** | **Who in our code** |
| --- | --- | --- |
| Publisher | Mono<Student> | The WebClient call (produces data) |
| Subscriber | subscribe() | The consumer that consumes the data |
| Data | Student | The response |
| Error | Throwable | Any exception |

So the Mono is the **Publisher** — it *publishes data* when ready.  
The Consumer (via subscribe()) is the **Subscriber** — it *reacts* to data when it comes.

**⚡ Summary Table**

| **Concept** | **Purpose** | **Who triggers it** |
| --- | --- | --- |
| Mono<Student> | Describes the async work to be done | You create it |
| subscribe() | Starts the async process | You call it |
| accept() (in Consumer) | Handles result or error when ready | Automatically called by Reactor |

**🧩 Simple Example Summary**

Mono<Student> mono = client.get()

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

.retrieve()

.bodyToMono(Student.class);

mono.subscribe(new StudentCallback(), new ErrorCallback());

| **Step** | **What Happens** |
| --- | --- |
| 1️⃣ | Mono created (no request yet) |
| 2️⃣ | subscribe() called → starts HTTP request |
| 3️⃣ | Response arrives later |
| 4️⃣ | Reactor calls StudentCallback.accept() automatically |
| 5️⃣ | Or if failed → Reactor calls ErrorCallback.accept() |