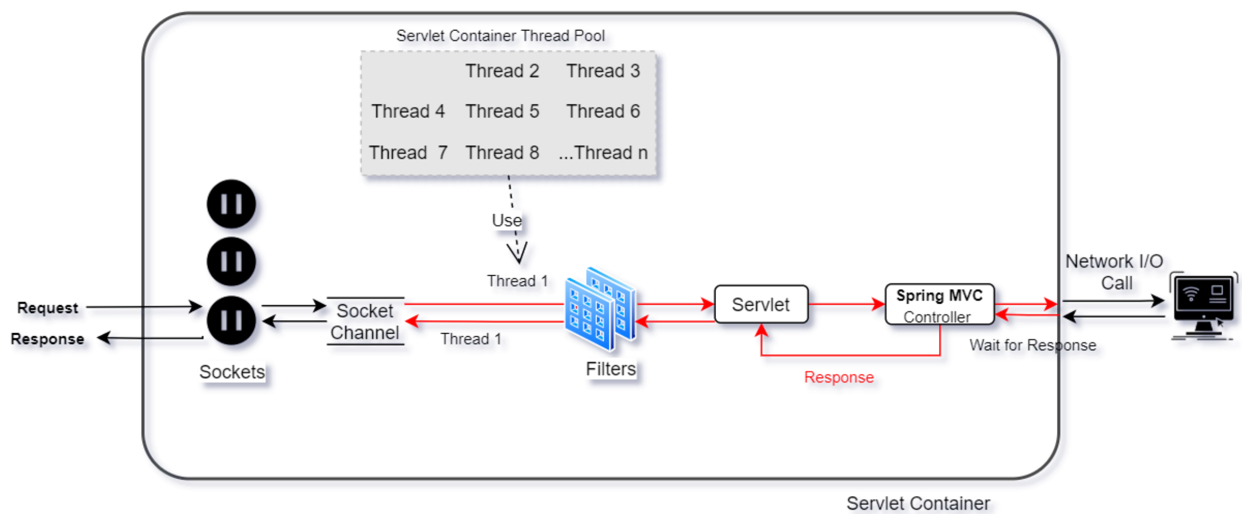
WebFlux vs. Spring MVC (Blocking vs. Non-Blocking)

Imagine a Restaurant (Spring MVC = Blocking Model)

Think of your server as a restaurant.

- **Customers = HTTP Requests**
- **Waiters = Threads**
- **Kitchen / DB / External API = Network I/O Calls**

In Spring MVC, **each customer gets one waiter**.



Step-by-Step Flow

1. Client Sends Request

A user opens a browser:

GET /users/1

This request reaches the **server socket**.

In the diagram → **Sockets**

---

2. Servlet Container Picks a Thread

servlet container (Tomcat/Jetty) has a **thread pool**:

Thread 1, Thread 2, Thread 3 ... Thread N

One free thread is assigned.

In diagram → **Servlet Container Thread Pool**

Like assigning a waiter to a customer.

---

3. Request Passes Through Filters

Before reaching your controller, request goes through:

- Security checks
- Logging
- Authentication
- CORS etc.

In diagram → **Filters**

Like checking reservation / ID at restaurant entrance.

---

4. Servlet Receives Request

DispatcherServlet handles routing.

In diagram → **Servlet**

Like restaurant manager deciding which chef handles order.

---

5. Spring MVC Controller Executes

Your controller runs:

```
@GetMapping("/users/{id}")
public User getUser() {
   return userService.findById(id);
}
```

In diagram → **Spring MVC Controller**

---

6. Blocking Happens Here (Important Part)

If controller calls DB:

User user = repository.findById(id);

The **thread WAITS** until DB responds.

Waiter stands idle in kitchen waiting for food.

Thread is blocked

Why Blocking is a Problem

If:

- **DB takes 2 seconds**
- **1000 users hit server**

You may need 1000 threads.

Threads = Memory + CPU overhead.

Too many → Server slows / crashes.

---

7. Response Returned

After DB returns:

- Controller builds response
- Servlet writes response
- Thread released

Waiter finally serves food & becomes free.

Real-Time Example (Very Practical)

Suppose API:

GET /weather

Controller calls:

- External weather API (3 seconds)

In Spring MVC:

- Thread blocked for 3 seconds
- Cannot serve other requests

[If traffic spikes → Thread pool exhausted → Requests rejected]

How WebFlux Differs (Mental Contrast)

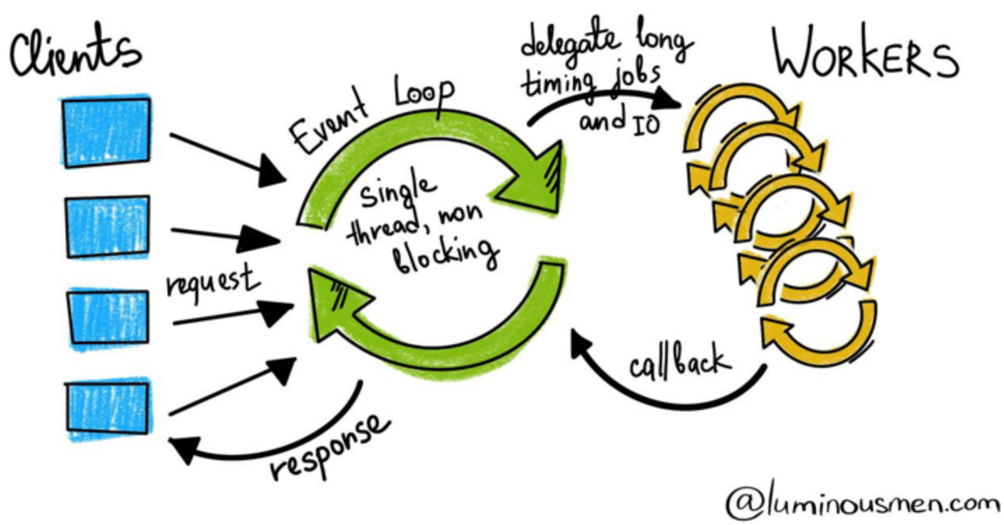
Spring WebFlux:

One Request → No Dedicated Waiting Thread

Thread starts work → Leaves → Comes back when data ready.

Waiter takes order → Doesn't stand idle → Serves others.

Much more scalable.



This diagram represents the **WebFlux / Non-Blocking / Event Loop model**.

Imagine a Smart Restaurant (WebFlux Model)

But this time the restaurant works very differently.

Instead of **one waiter per customer**, we have:

- **Event Loop = Super-efficient manager**
- **Workers = Kitchen staff / helpers**
- **Customers = HTTP Requests**

What Happens in WebFlux (Step-by-Step)

---

1. Multiple Clients Send Requests

Many users hit your API:

GET /users/1
GET /users/2
GET /users/3
GET /users/4

In diagram → **Clients**

Many customers entering restaurant.

---

2. Event Loop Receives ALL Requests

There is **NOT one thread per request**.

There is a small number of threads running an **Event Loop**.

In diagram → Single Thread Non-Blocking

Like one smart manager handling everyone.

Important idea:

The event loop **never waits**.

It only:

- **Accepts request**
- **Delegates work**
- **Moves to next request**

---

3. Long Tasks Are Delegated to Workers

If request needs:

- **DB query**
- **External API call**
- **File read**

Event loop immediately hands it off.

In diagram → **delegate long running jobs and I/O → Workers**

Manager gives order to kitchen staff and does not stand idle**.**

Example:

return userRepository.findById(id);

No blocking. No waiting.

---

4. Event Loop Continues Serving Others

While DB is working...

Event loop is free to process:

Next HTTP request
Next response
Next network event

Manager keeps talking to other customers.

---

5. Worker Finishes Job → Callback Happens

Once DB / API completes:

- Worker signals Event Loop
- Event Loop resumes processing

In diagram → **callback**

Kitchen rings bell → Food ready.

---

6. Response Sent Back

Event loop sends response to correct client.

Manager serves correct dish to correct customer.

Why This Model Is Powerful

Unlike Spring MVC:

No thread sitting idle
No thread-per-request explosion

Only few threads → Handle thousands of requests.

Real-Time Example (Very Practical)

Suppose your API calls:

- Payment Gateway (3 seconds)
- Slow DB (2 seconds)

Spring MVC (Blocking)

Request → Thread assigned → Thread waits 3 sec

1000 users → Need ~1000 threads

Server stress

WebFlux (Non-Blocking)

Request → Event loop delegates → Moves on

1000 users → Same few threads

Efficient

Mental Model

Spring MVC:

Wait until work finishes

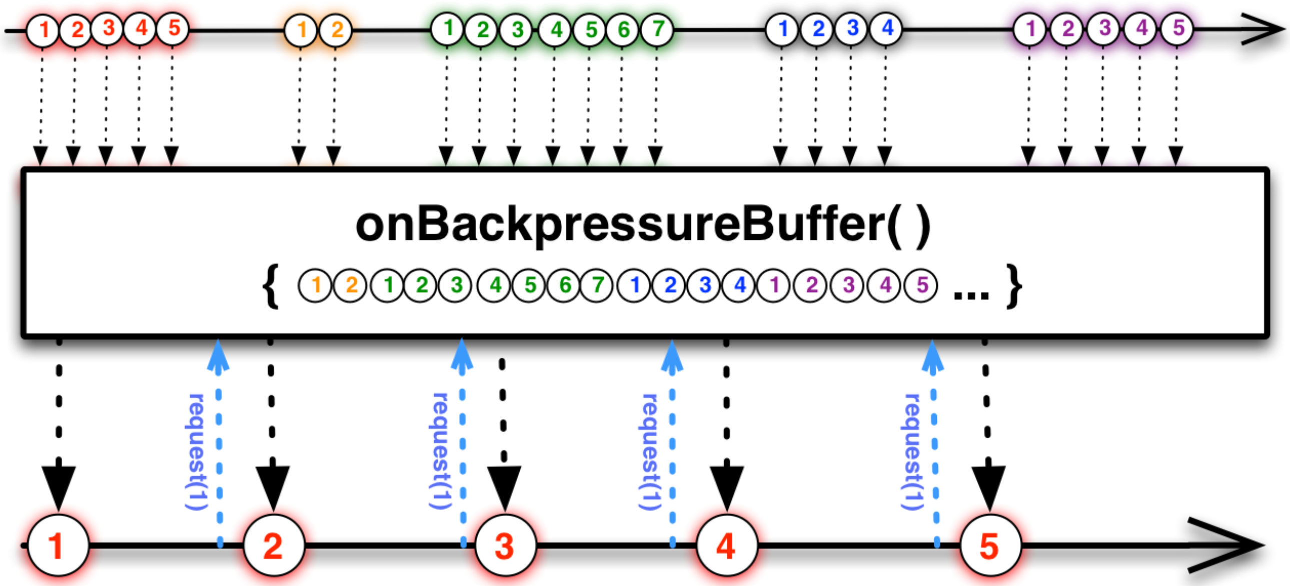
WebFlux:

Start work → Continue doing other things → Resume later

Key Concept

Event Loop = Brain
Workers = Muscles

Brain never sleeps, muscles do heavy work.



This diagram is showing one of the **most important Reactor concepts**:

Backpressure + onBackpressureBuffer()

Let’s explain it in a **real-time,**

Imagine a Water Pipe System

- **Top flow (colored circles) = Producer (fast source of data)**
- **Bottom flow (1 → 5) = Consumer (slow processor)**
- **Buffer box = Temporary storage (queue)**

Problem scenario:

Producer sends data **too fast**
Consumer processes data **slowly**

What Problem Is Happening?

Producer emits:

1 2 3 4 5 6 7 8 9 ...

Consumer can handle only:

1 item at a time

Without backpressure control:

Consumer overwhelmed
Memory issues
Data loss or crash

What onBackpressureBuffer() Does

It tells Reactor:

“If consumer is slower, store extra items in a buffer instead of failing.”

Diagram middle box:

onBackpressureBuffer()
{ 1 2 1 2 3 4 5 6 7 1 2 3 4 1 2 3 4 5 … }

This is the queue of waiting items**.**

Step-by-Step Flow

---

1. Producer Emits Quickly

Events arrive continuously.

Top colored circles = Incoming items.

Producer **does NOT slow down**.

---

2. Consumer Requests Data Slowly

See blue arrows:

request(1)

This means:

Consumer asks for only ONE item at a time

Very important concept in Reactive Streams.

Consumer controls speed.

3. Extra Data Goes Into Buffer

Since producer is faster:

Items cannot go directly to consumer
They are placed inside buffer

Like people waiting in a queue.

---

4. Consumer Processes at Its Own Pace

Consumer flow:

1 → 2 → 3 → 4 → 5

Each time consumer finishes:

request(1)

Next buffered item delivered.

Real-Time Example (Very Practical)

Imagine:

- Kafka / Event Stream → Very fast
- API / DB → Slower

Example:

```
Flux<Event> events = kafkaFlux();

events
  .onBackpressureBuffer()
  .flatMap(this::saveToDatabase)
```

If DB slow:

Events buffered
System stays stable

Without buffer:

Errors / dropped events / overflow

Why Backpressure Exists

Reactive systems must prevent:

Fast producer crashing slow consumer

Backpressure = **Flow control mechanism**

Consumer says:

"Send only what I can handle."

Important Warning

onBackpressureBuffer() is powerful BUT...

If producer is **too fast for too long**:

Buffer grows
Memory risk

Safer variants:

```
.onBackpressureBuffer(1000)       // limit size
.onBackpressureDrop()             // drop extras
.onBackpressureLatest()           // keep newest only
```

Summary

Problem:

Producer fast, Consumer slow

Solution:

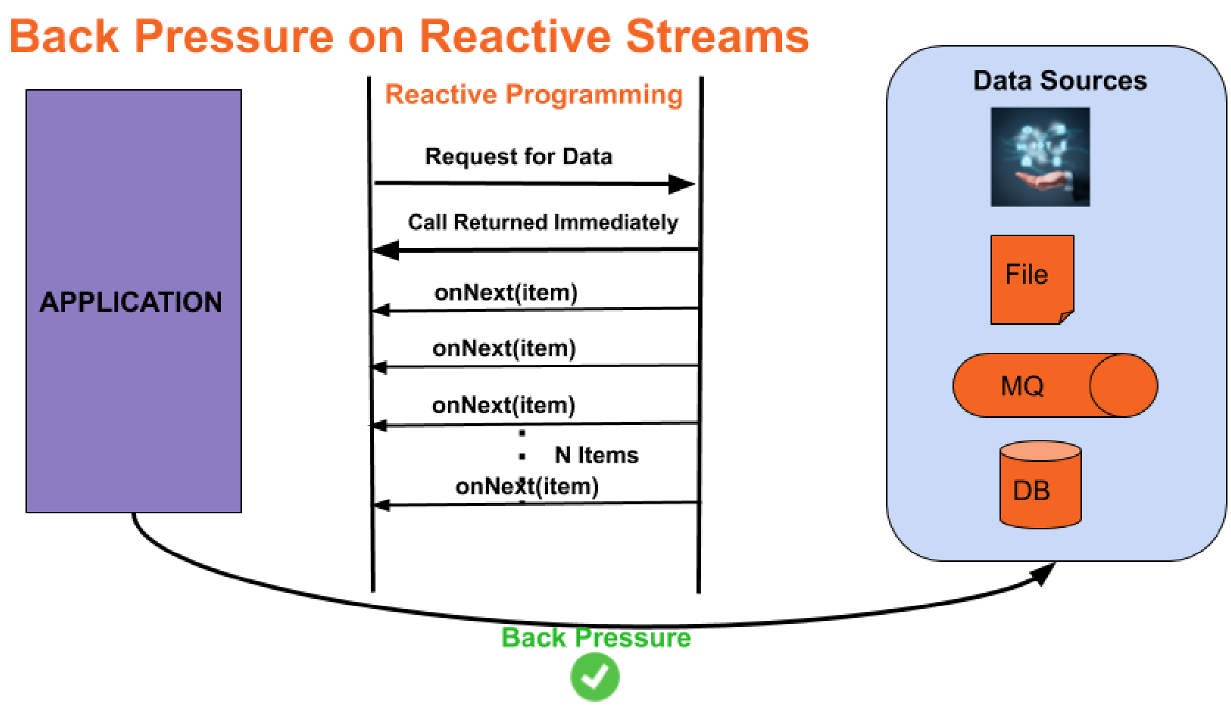
Buffer stores extra data temporarily

Consumer:

Pulls data using request(n)

Goal:

Prevent overload
Maintain stability
Avoid crashes



Spring MVC (Blocking Model)

How it works

- Each HTTP request is handled by a dedicated thread
- Thread waits (blocks) for DB calls, HTTP calls, file I/O, etc.

Characteristics

- Simple mental model
- Works great for traditional CRUD apps
- Threads sit idle during I/O → wastes resources
- Limited scalability under high concurrency

Summary

- Spring MVC = **Blocking / Thread per request**
- Easy to understand
- Traditional model
- Threads wait during DB/API calls

Spring WebFlux (Non-Blocking / Reactive Model)

How it works

- Small number of threads (event loop style)
- Threads never wait; operations run asynchronously
- Uses Reactive Streams + backpressure

Characteristics

- Excellent for high-throughput & I/O-heavy systems
- Efficient resource usage

- Requires reactive mindset
- Debugging & learning curve higher

Summary:

WebFlux = **Non-Blocking / Event driven**

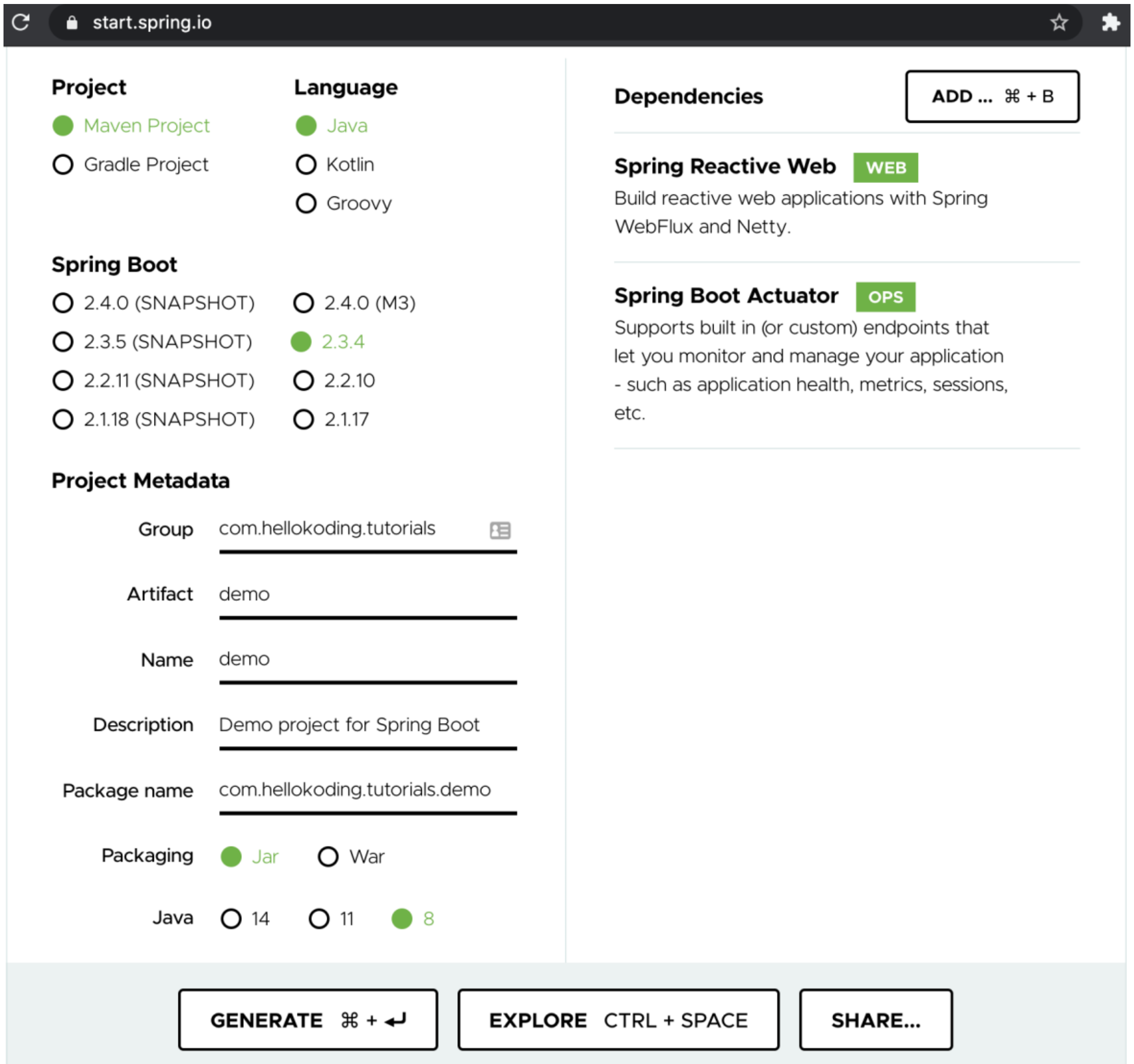
- Better scalability
- Threads never sit idle

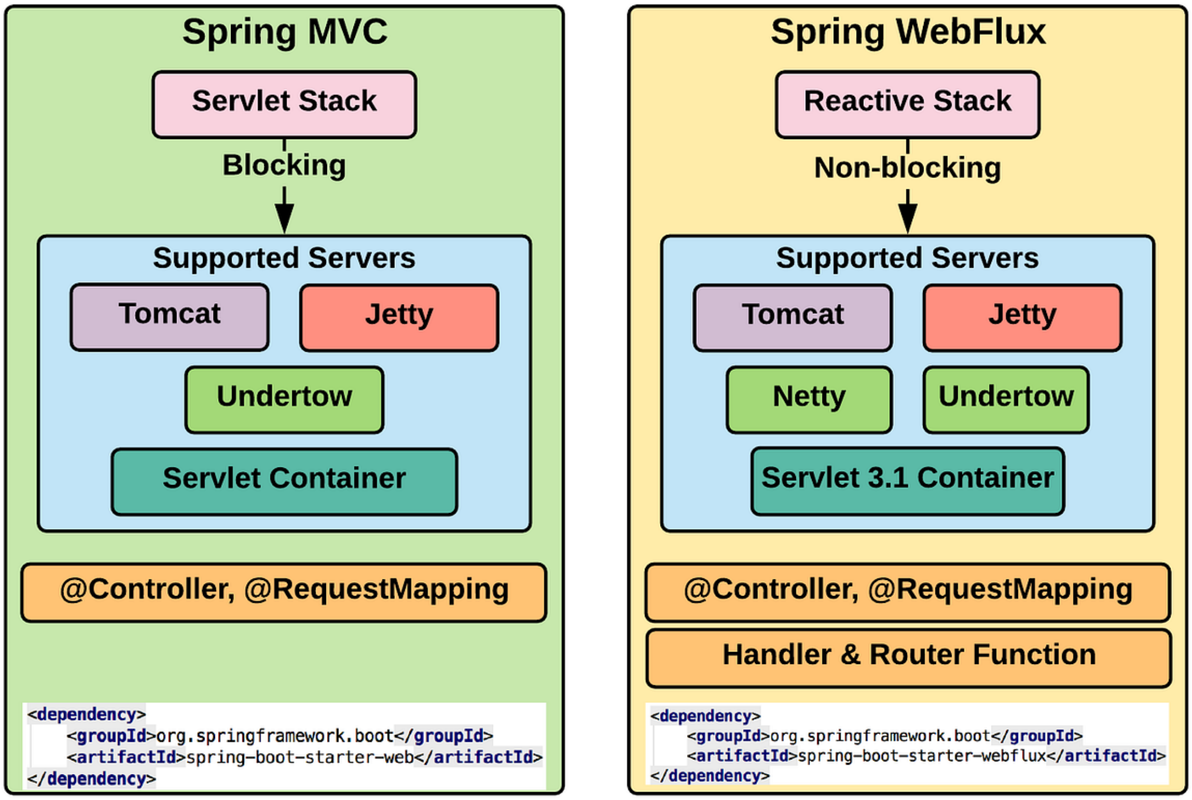
When to Choose What

Use Case	Recommended
Simple CRUD / low concurrency	Spring MVC
High concurrency / streaming / microservices	WebFlux
Heavy blocking dependencies	MVC

Creating a Spring Boot WebFlux Project

You can generate a project via **Spring Initializr**.


start.spring.io
Project
Maven Project
Gradle Project
Language
Java
Kotlin
Groovy
Spring Boot
2.4.0 (SNAPSHOT)
2.4.0 (M3)
2.3.5 (SNAPSHOT)
2.3.4
2.2.11 (SNAPSHOT)
2.2.10
2.1.18 (SNAPSHOT)
2.1.17
Project Metadata
Group com.hellokoding.tutorials
Artifact demo
Name demo
Description Demo project for Spring Boot
Package name com.hellokoding.tutorials.demo
Packaging Jar War
Java 14 11 8
Dependencies
ADD ... ⌘ + B
Spring Reactive Web WEB
Build reactive web applications with Spring WebFlux and Netty.
Spring Boot Actuator OPS
Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.
GENERATE ⌘ + ↵
EXPLORE CTRL + SPACE
SHARE...



Using Maven

Dependencies

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Key Difference from MVC

If both dependencies exist:

- spring-boot-starter-web → MVC
- spring-boot-starter-webflux → Reactive

MVC **wins by default** if both present.

Understanding Reactor Operators

WebFlux relies on **Project Reactor**.

Core types:

- **Mono** → 0..1 item
- **Flux** → 0..N items

map() — Transform Data (Sync)

```java
Mono<String> name = Mono.just("Venkat");

Mono<String> upper = name.map(n -> n.toUpperCase());
```

Used for **simple synchronous transformations**

flatMap() — Async Transformation

```java
Mono<User> userMono = userRepository.findById(id);

Mono<Order> orderMono =
   userMono.flatMap(user -> orderService.findOrders(user));
```

Used when function returns **another Mono/Flux**

Rule of thumb

- returns plain object → map
- returns reactive type → flatMap

filter() — Conditional Emission

```
Flux<Integer> numbers = Flux.just(1, 2, 3, 4);

Flux<Integer> even = numbers.filter(n -> n % 2 == 0);
```

Drops unwanted elements

Visual Mental Model

Operator	Purpose
map	Change value
flatMap	Switch async stream
filter	Keep/remove items

Writing Reactive REST APIs with @RestController

Good news: Programming model looks **almost identical** to MVC.

Simple Reactive Endpoint

```
@RestController
@RequestMapping("/hello")
public class HelloController {

    @GetMapping
    public Mono<String> hello() {
        return Mono.just("Hello Reactive World");
    }
}
```

Returning Multiple Values

```
@GetMapping("/numbers")
public Flux<Integer> numbers() {
   return Flux.just(1, 2, 3, 4, 5);
}
```

WebFlux automatically streams response.

Reactive Service Example

```
@GetMapping("/{id}")
public Mono<User> getUser(@PathVariable String id) {
   return userService.findById(id);
}
```

No blocking allowed

Avoid:

```
userRepository.findById(id).block(); // BAD
```

Reactive Composition Example

```
@GetMapping("/{id}/orders")
public Flux<Order> getOrders(@PathVariable String id) {
   return userService.findById(id)
          .flatMapMany(user -> orderService.findByUser(user));
}
```

Important WebFlux Rules

Never block (block(), sleep, JDBC without reactive driver)
Use reactive DB drivers (R2DBC, reactive Mongo, etc.)
Think in **pipelines**, not steps
