# **Intro to the Spring Boot Reactive Stack**

Spring Boot’s Reactive Stack—anchored by **Spring WebFlux** and **Project Reactor**—lets you build **non-blocking**, **asynchronous**, event-driven applications that can handle very high levels of concurrency with a minimal thread count.

**Concept & Origins**

* **Reactive Streams & Project Reactor**  
  Reactive Streams (2016) defined a standard for asynchronous stream processing with back-pressure. Spring WebFlux (introduced in Spring Framework 5.0, 2017) implements this via Reactor’s two core types—**Mono<T>** (0–1 element) and **Flux<T>** (0–N elements).
* **Non-Blocking I/O Model**  
  Instead of “one thread per request,” reactive apps use an **event loop** (Netty by default) so that threads never block on I/O; they register callbacks and move on to other work.
* **Back-Pressure**  
  Consumers explicitly signal demand to producers, preventing fast producers from overwhelming slower consumers.

**🌐 Background: Traditional Spring MVC (Blocking Model)**

Spring Boot originally shipped with Spring MVC built on top of the Servlet API (Servlet 3.1+). It’s a thread-per-request, blocking I/O model:

* Each HTTP request is handled by a servlet container (Tomcat, Jetty, etc.).
* A request is tied to a thread until the response is return.
* If that thread blocks (e.g., waiting on DB, external service, file I/O), it consumes server resources.

👉 This works very well for CRUD-style apps with limited concurrency, but it doesn’t scale efficiently when apps have to handle high concurrency with slow/blocking I/O.

**🚨 Problems with Traditional Spring Boot (MVC)**

1. Thread per request model is resource-heavy
   * Each thread has ~1 MB stack memory + context switching overhead.
   * 1000s of concurrent requests = 1000s of threads → memory + CPU pressure.

Example:  
A web app making many calls to a payment API. If API latency is 500 ms and 1000 clients hit simultaneously, you need 1000 threads just waiting idle.

1. Blocking I/O causes scalability bottlenecks
   * When your service waits on DB/HTTP call/file read, the thread is blocked.
   * No reuse possible until that I/O finishes.

Example:  
An e-commerce app where product details API fetches data from multiple microservices (pricing, stock, shipping). If one service is slow, the entire request is blocked.

1. Limited scalability in microservices era
   * Modern apps often do service-to-service communication.
   * Traditional blocking model struggles when one service fans out to many other services (API aggregation, GraphQL, BFFs).

Example:  
A mobile backend that needs to fetch user profile, order history, and recommendations from 3 services. Each call is blocking → user sees slow response + high infra cost.

1. Inefficient for streaming use-cases
   * Handling server-sent events (SSE), WebSockets, or continuous data streams is inefficient with thread-per-request.
   * A WebSocket connection holds a thread open for its lifetime in blocking model.

Example:  
A live stock-ticker app with 100K users → impossible with blocking MVC (100K threads).

**✅ Why Spring WebFlux Came**

**Spring team introduced WebFlux (Spring 5, 2017) as a reactive, non-blocking web framework.**

* Based on Reactive Streams standard (Publisher, Subscriber, Flux, Mono).
* Works with Netty (event-loop model, no thread-per-request).
* Supports backpressure (don’t overwhelm slow consumers).
* Enables functional + annotation-based programming style.

**🌍 Real-World Examples & Impact**

**1. High-concurrency APIs**

* Problem: A social media app API has 50K concurrent connections fetching timelines.
* Spring MVC: Needs 50K threads → impossible.
* WebFlux: Uses event loop + reactive I/O → 50K connections can be handled with a few hundred threads.
* Impact: Lower infra cost, higher scalability.

**2. API Gateway / Aggregator Service**

* Problem: A BFF (Backend For Frontend) service aggregates data from 5 downstream microservices.
* Spring MVC: Each downstream call blocks → total latency adds up.
* WebFlux: Calls made in parallel, async response composed with Mono.zip() / Flux.merge().
* Impact: Reduced latency (e.g., 1.2s → 400ms), better user experience.

**3. Streaming Data / Real-time Updates**

* Problem: Stock trading platform streaming live prices to clients.
* Spring MVC: Threads blocked waiting on stream, cannot scale beyond a few 1000s.
* WebFlux: Non-blocking SSE/WebSocket, handles 100K+ live clients.
* Impact: Huge scalability with same hardware.

**4. IoT Data Ingestion**

* Problem: Millions of IoT sensors sending telemetry every second.
* Spring MVC: Thread-per-request model collapses under load.
* WebFlux: Netty-based reactive stack handles high-throughput ingestion.
* Impact: Lower infra footprint, stable ingestion at scale.

**5. Slow external dependencies**

* Problem: Travel booking system relies on external airline APIs (sometimes 1-2 sec latency).
* Spring MVC: Blocks threads → thread pool exhaustion → timeouts.
* WebFlux: Non-blocking, other requests not impacted while waiting.
* Impact: More resilient under high latency conditions.

**⚖️ Trade-offs**

* ✅ Pros: Scalability, async composition, efficient streaming.
* ❌ Cons: Steeper learning curve, debugging harder, reactive libraries everywhere, CPU-bound tasks don’t benefit much.

**🔴 The Blocking Problem Recap**

In Spring MVC (Servlet):

* Each HTTP request → tied to a dedicated thread.
* That thread does everything: read request, call DB/service, prepare response, write response.
* If any operation blocks (DB query, REST call, file read), the thread is stuck.
* Meanwhile, that thread can’t serve other requests.

👉 So with 10,000 concurrent requests waiting on slow I/O, you need 10,000 threads.

**🟢 How WebFlux Solves It (Non-Blocking I/O)**

WebFlux is built on Reactive Streams + Netty’s event loop model.

🔑 Key Ideas:

1. Event Loop Instead of Thread-Per-Request
   * Netty runs a small number of event loop threads (like Node.js).
   * Threads are not tied to requests. They react to I/O events (data ready, socket open/close).

Example:

* + A request arrives → event loop parses headers.
  + While waiting for DB → thread is free to handle other requests.
  + When DB result is ready → event loop resumes processing.

⚡ This is why few threads can handle 1000s of requests.

1. **Reactive Programming (Publisher–Subscriber model)**
   * Calls are represented as Mono (0/1 result) or Flux (0..N results).
   * Instead of blocking to "get" a value, you subscribe to it.
   * Processing continues only when data arrives.

Example (pseudo-code):

// Blocking style (Spring MVC)

User user = userRepository.findById("123"); // blocks thread

// Reactive style (WebFlux)

Mono<User> user = userRepository.findById("123"); // non-blocking

user.subscribe(u -> System.out.println(u.getName()));

**👉 The call immediately returns a promise-like publisher, and the thread is free.**

1. **Backpressure Handling**
   * Reactive Streams protocol allows consumers to say:  
     *“Send me only 10 items for now”* instead of overwhelming them.
   * This avoids OutOfMemoryErrors in high-throughput systems.

Example: A Kafka consumer built with WebFlux can process 10 messages/sec, and tell the producer not to send more until it’s ready.

1. **Non-Blocking Drivers/Libraries**
   * WebFlux shines when combined with non-blocking DB drivers (R2DBC, Reactive MongoDB, Reactive Redis) and non-blocking HTTP clients (WebClient).
   * This ensures end-to-end async I/O.

⚠️ If you call a blocking library inside WebFlux (e.g., JDBC), you lose the benefit because the thread still blocks.

**🛠️ Real-World Example: API Aggregator Service**

Let’s say a travel booking app needs to fetch:

* Flight info (200 ms)
* Hotel info (300 ms)
* Car rental info (400 ms)

In Spring MVC (blocking):

Flight f = flightService.getFlights(); // blocks 200 ms

Hotel h = hotelService.getHotels(); // blocks 300 ms

Car c = carService.getCars(); // blocks 400 ms

return combine(f, h, c);

* Total = 200 + 300 + 400 = 900 ms latency.
* Thread is idle 90% of the time.
* 1000 concurrent users → 1000 threads needed.

In Spring WebFlux (non-blocking):

Mono<Flight> flight = flightService.getFlights();

Mono<Hotel> hotel = hotelService.getHotels();

Mono<Car> car = carService.getCars();

return Mono.zip(flight, hotel, car) // run in parallel

.map(tuple -> combine(tuple.getT1(), tuple.getT2(), tuple.getT3()));

* All calls fire concurrently.
* Total latency = max(200, 300, 400) = 400 ms.
* Same event loop threads can handle thousands of requests.

👉 Impact: Faster response + smaller infra footprint.

**🚀 Why It Works**

* Threads are not wasted waiting → event loop handles many requests.
* Latency is reduced by parallel non-blocking composition.
* Server can scale to 10x or 100x more concurrent connections with same hardware.

**Core Rules & Conventions**

1. **Return Reactive Types at Boundaries**
   * Controllers, repositories, and handlers should expose **only** Mono/Flux.
2. **Avoid Blocking in Pipelines**
   * If you must call a blocking API (e.g., legacy JDBC or file I/O), wrap it in Schedulers.boundedElastic() to keep the main event-loop threads free.
3. **Enforce Back-Pressure**
   * Use operators such as .limitRate(), .onBackpressureBuffer(), or .onBackpressureDrop() when composing multiple sources to handle varying production speeds.
4. **Error Handling Within Streams**
   * Don’t let exceptions bubble up and kill the pipeline: use .onErrorResume(), .onErrorMap(), or .doOnError() to translate errors into meaningful responses.
5. **Functional vs. Annotated Routing**
   * **Annotated**: @RestController + reactive return types.
   * **Functional**: RouterFunction<ServerResponse> + HandlerFunction, giving you programmatic control over routing and filter chains.

**Examples of Rules in Action**

1. **Reactive Repository with Bounded Elastic Scheduler**

public Flux<User> getAllUsers() {

return Mono.fromCallable(() -> userJdbcRepository.findAll())

.flatMapMany(Flux::fromIterable)

.subscribeOn(Schedulers.boundedElastic());

}

1. **Back-Pressure Buffering**

Flux.interval(Duration.ofMillis(10))

.onBackpressureBuffer(100) // buffer up to 100 events

.subscribe(consumer::processEvent);

1. **Stream-Level Error Mapping**

Flux.fromIterable(fileLines)

.map(this::parseRecord)

.onErrorResume(ParseException.class, ex ->

Flux.empty()

);

**Where & When to Use**

* **High-Concurrency HTTP APIs**  
  When you expect thousands of simultaneous, long-lived connections (e.g., chat servers, live feeds).
* **Event-Driven Data Pipelines**  
  Ingest → transform → forward streams (Kafka, MQTT, WebSocket) with end-to-end back-pressure.
* **IoT & Telemetry Platforms**  
  Process millions of small sensor events per second, filtering and aggregating in reactive streams.
* **Real-Time Dashboards**  
  Push live metrics (stock prices, server health) via WebFlux SSE or WebSocket support.
* **Resource-Constrained Environments**  
  Cloud-native or serverless deployments where minimizing thread count reduces memory and CPU usage.

**Real-World Usage Examples**

1. **Live Sports Scoreboard**  
   Uses WebFlux to consume a WebSocket feed of game events, processes them as a Flux<GameEvent>, and broadcasts updates to thousands of browser clients via SSE.
2. **Industrial IoT Hub**  
   Connects to hundreds of MQTT topics, each producing Flux<SensorReading>, filters anomalies, aggregates per-minute averages, and writes to a time-series database without ever blocking.
3. **Reactive API Gateway**  
   Built on Spring Cloud Gateway (WebFlux), it performs JWT validation reactively, applies rate-limits via .limitRate(), and routes requests to downstream services—all on a small fixed thread pool.

**Why It Came to Be**

1. **Scalability Limits of the Servlet Model**  
   Traditional Spring MVC (Servlet API) ties up one thread per request, leading to thread exhaustion under high concurrency.
2. **Formalizing Asynchronous Streams**  
   The Reactive Streams spec provided a standard for back-pressure. Spring WebFlux embraces it to offer a cohesive, non-blocking programming model.
3. **Rise of Event-Driven & Cloud-Native Architectures**  
   Behaviors like WebSocket, Server-Sent Events, and microservices demanded low-latency, high-throughput, event-loop–driven servers.

**Motivating Scenarios**

1. **Breaking a Monolith**  
   As a large retail monolith split into microservices, blocking calls to a shared session DB became a bottleneck—migration to reactive services eliminated that contention.
2. **Mobile & Web Convergence**  
   A financial services firm needed identical real-time push logic for its SPA and mobile apps—reactive streams unified the backend logic for both.
3. **Cost-Sensitive Cloud Deployments**  
   Running on Kubernetes with strict resource quotas, adopting a small-thread-count reactive stack reduced CPU and memory costs by over 50%.

# **Reactive Streams & Project Reactor**

**Concept & Origins**

**Reactive Streams** is a specification (first released in 2016) that standardizes asynchronous stream processing with non-blocking back-pressure. It defines four main interfaces:

* **Publisher<T>**: emits values according to downstream demand.
* **Subscriber<T>**: consumes values, signaling demand via a **Subscription** and handling completion or errors.
* **Subscription**: bridges Publisher and Subscriber—controls request(n) and cancellation.
* **Processor<T,R>**: both a Subscriber and a Publisher, enabling processing stages in the stream.

**Project Reactor** (part of Spring since 2017) is a fully compliant Reactive Streams library providing two core types:

* **Mono<T>** – 0 or 1 element
* **Flux<T>** – 0 … N elements

It builds on Netty for networking, integrates with WebFlux, and offers hundreds of operators for composing asynchronous pipelines.

**Core Rules & Conventions**

1. **Demand Must Drive Production**
   * A Publisher must not emit more items than requested by the Subscriber.
2. **Signals Sequence**
   * onSubscribe → zero or more onNext → onError or onComplete.
3. **Single Terminal Signal**
   * Only one of onError or onComplete may be called, and no onNext after that.
4. **No Concurrent Calls**
   * onNext, onError, and onComplete must not be called concurrently; order is preserved.
5. **Cancellation Semantics**
   * Calling cancel() must stop further signals.
6. **Error Handling**
   * Errors are fatal—once onError is invoked, no further signals.
7. **Cold vs. Hot Publishers**
   * **Cold**: data produced per subscriber (e.g., Flux.range)
   * **Hot**: data produced independently, shared among subscribers (e.g., ConnectableFlux)
8. **Schedulers**
   * Offload blocking or CPU-bound work to appropriate thread pools (parallel(), elastic(), single()).
9. **Operator Fusion & Optimization**
   * Reactor minimizes allocation and context-switch overhead via fusion where possible.

**Where & When to Use**

Reactive Streams & Reactor shine when you need:

* **High‐Concurrency I/O**
  + e.g., web servers handling thousands of long-lived connections without one-thread-per-connection overhead.
* **Back-Pressured Pipelines**
  + e.g., ingesting data from Kafka or MQTT, transforming, and writing to a database while preventing overload.
* **Asynchronous Composability**
  + e.g., fetching from multiple external services in parallel and combining results in a single non-blocking flow.
* **Event-Driven Architectures**
  + e.g., WebSocket servers, Server-Sent Events, real-time dashboards.
* **Resource‐Constrained Environments**
  + e.g., serverless functions or containers where thread count and memory must be tightly controlled.

**Why It Came to Be**

1. **Unmanageable Callback Hell**
   * Early async code used nested callbacks or raw Futures, leading to tangled, hard-to-maintain logic.
2. **Lack of Standard Back-Pressure**
   * Streaming systems (Kafka, TCP, JDBC) didn’t agree on how to signal overwhelm, causing dropped data or resource exhaustion.
3. **Need for End-to-End Non-Blocking**
   * Traditional servlet/JDBC stacks block threads during I/O; scaling required heavy hardware.
4. **Interoperability Across Libraries**
   * A common spec lets different components (web, messaging, DB) plug into one reactive pipeline.

**Real-World Examples**

**Reactive Streams Spec in Action**

1. **Reactive Kafka Client**
   * KafkaReceiver (Project Reactor Kafka) implements Publisher<ConsumerRecord>, honoring downstream demand and avoiding unbounded queueing.
2. **R2DBC (Reactive Relational DB Connectivity)**
   * Executes SQL queries non-blocking over Netty, returning a Flux<Row> that streams results respecting back-pressure.
3. **Reactive RabbitMQ**
   * Reactor RabbitMQ exposes Receiver, a Publisher<Delivery>, so consumers request exactly as many messages as they can process.

**Project Reactor Usage**

1. **Spring WebFlux End-to-End**

@GetMapping("/events")

public Flux<ServerSentEvent<MyEvent>> stream() {

return Flux.interval(Duration.ofSeconds(1))

.map(seq -> ServerSentEvent.builder(new MyEvent(seq)).build());

}

* + Streams live events to browsers via SSE with zero blocking threads.

1. **Parallel External Service Fan-Out**

Flux.zip(

webClient.get().uri("/svcA").retrieve().bodyToMono(A.class),

webClient.get().uri("/svcB").retrieve().bodyToMono(B.class),

(a, b) -> combine(a, b)

);

* + Fetches from two services concurrently, merging results in a non-blocking pipeline.

1. **Back-Pressure in File Processing**

Flux.using(

() -> Files.lines(path),

Flux::fromStream,

Stream::close

)

.limitRate(100)

.flatMap(line -> processLineAsync(line))

* + Reads a large file lazily, processes lines asynchronously, but never more than 100 in flight.

# **Spring WebFlux**

Spring WebFlux is the reactive, non-blocking web framework introduced in Spring 5.0, built on Reactor and the Reactive Streams API. It provides two programming models—annotated controllers and functional routing—and runs on servers such as Netty, Undertow, or any Servlet 3.1+ container using non-blocking I/O.

**1. Concept & Origins**

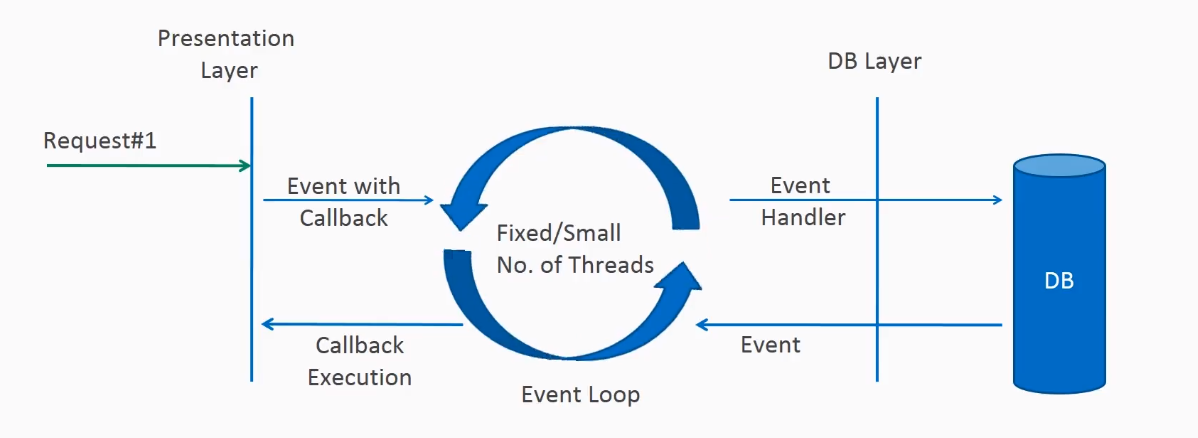
* **Reactive, Non-Blocking I/O**  
  Uses event loops (Netty by default) instead of dedicating one thread per request, allowing a small thread pool to serve thousands of connections concurrently.
* **Reactive Streams Compliance**  
  Leverages Reactor’s Mono<T> (0–1 element) and Flux<T> (0–N elements), honoring back-pressure so consumers can signal how much data they’re ready to process.
* **Dual Programming Models**
  1. **Annotated**: @RestController returning Mono/Flux.
  2. **Functional**: RouterFunction<ServerResponse> and HandlerFunction<ServerRequest> for explicit routing and filter chains.

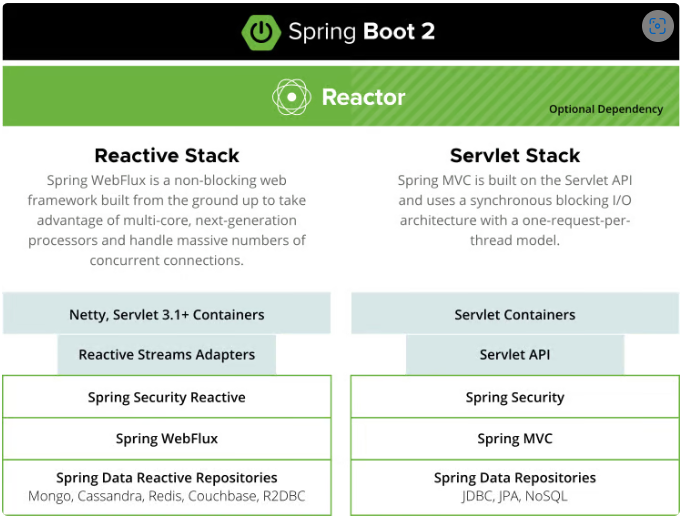
**Real-World Concept Illustrations**

1. **Live Chat Server**  
   A single Netty event loop handles WebSocket connections from thousands of users, broadcasting chat messages as a Flux<Message> without blocking threads.
2. **Streaming Database Queries**  
   Using R2DBC, a service issues a SQL query and returns a Flux<Row>, streaming millions of rows directly to the client with back-pressure.
3. **Parallel API Aggregation**  
   A façade service calls multiple downstream REST APIs via WebClient, merges their responses with Flux.merge(), and returns a unified stream.

**2. Core Rules & Conventions**

1. **Return Only Reactive Types at Entry Points**  
   Controllers and handlers must return Mono<T> or Flux<T>, never raw values or blocking Futures.
2. **Avoid Blocking Calls on Event Loops**  
   Wrap any legacy, blocking I/O (e.g., JDBC, file access) in Schedulers.boundedElastic() to offload to a separate thread pool.
3. **Honor Back-Pressure**  
   Use operators like .limitRate(), .onBackpressureBuffer(), or .onBackpressureDrop() when upstream producers outpace consumers.
4. **Error Handling in Pipelines**  
   Translate exceptions into meaningful HTTP statuses using .onErrorResume(), .onErrorMap(), or ResponseStatusException.
5. **Choose Programming Model Wisely**
   * **Annotated** for typical REST controllers.
   * **Functional** for dynamic routing or when you need fine-grained control over filters and routes.





**Examples of Rules in Action**

1. **Offloading Blocking I/O**

Flux<User> users = Flux.fromIterable(jdbcRepo.findAll())

.subscribeOn(Schedulers.boundedElastic());

1. **Back-Pressure Buffering**

Flux.interval(Duration.ofMillis(20))

.onBackpressureBuffer(50)

.subscribe(this::processEvent);

1. **Error Translation**

@GetMapping("/items/{id}")

public Mono<Item> get(@PathVariable String id) {

return repo.findById(id)

.switchIfEmpty(Mono.error(new ResponseStatusException(NOT\_FOUND)))

.onErrorMap(DataAccessException.class,

ex -> new ResponseStatusException(INTERNAL\_SERVER\_ERROR));

}

**Motivating Use Cases**

1. **Microservices Breakup**  
   A monolith refactored into microservices saw its session-store become a bottleneck; reactive services eliminated the need for server-side session storage.
2. **Real-Time Mobile & Web Feeds**  
   A media company required identical push-notification logic for web and mobile; a single WebFlux pipeline served both via SSE and push notifications.
3. **Cost-Sensitive Cloud Deployments**  
   Running on Kubernetes with strict quotas, switching to reactive WebFlux halved the number of threads and cut CPU usage by over 40%.

## **✅ When & Where to Use WebFlux**

WebFlux shines when your system is **I/O-bound** (lots of waiting on network, DB, files) and needs **high concurrency**.

**1. API Gateway / BFF (Backend For Frontend)**

* **Use Case:** An API gateway aggregates results from 5–6 downstream services (user profile, pricing, recommendations, cart, payments).
* **Why WebFlux:** Calls can be fired **in parallel**, not sequentially. Latency is reduced to the *slowest* call instead of the *sum*.
* **Example:** Netflix Zuul 2 and Spring Cloud Gateway (built on WebFlux).
* **Impact:** Lower latency for user apps, efficient infra usage.

**2. Streaming & Real-time Data**

* **Use Case:** Chat app, stock ticker, live dashboards, IoT telemetry.
* **Why WebFlux:** WebSockets, Server-Sent Events (SSE), and continuous streaming work much better without blocking threads.
* **Example:** A cryptocurrency exchange streaming live market prices to 100K+ clients.
* **Impact:** Fewer servers needed, smooth real-time updates.

**3. High-Concurrency, I/O-Heavy APIs**

* **Use Case:** A travel booking system making slow airline/hotel API calls.
* **Why WebFlux:** Threads aren’t wasted while waiting → can handle 10x+ concurrent users.
* **Example:** Skyscanner-style aggregator where latency of external APIs varies from 500 ms to 2 seconds.
* **Impact:** Resilient under load, fewer request timeouts.

**4. IoT / Event Ingestion Pipelines**

* **Use Case:** Millions of IoT sensors sending data every second.
* **Why WebFlux:** Efficient ingestion without thread explosion. Works nicely with Kafka, RabbitMQ, RSocket.
* **Example:** Smart home company ingesting data from 1M devices.
* **Impact:** Sustains high throughput on modest infra.

**5. When Using Reactive DB/Clients**

* **Use Case:** Reactive MongoDB, Reactive Redis, R2DBC (Reactive SQL).
* **Why WebFlux:** End-to-end non-blocking stack avoids thread starvation.
* **Example:** A financial risk system running analytics queries while streaming results reactively to clients.
* **Impact:** Lower memory usage, no thread pool exhaustion.

## **❌ When & Where to Avoid WebFlux**

WebFlux **is not a silver bullet**. It adds complexity. If your app is **CPU-bound** or depends on **blocking libraries**, WebFlux won’t help.

**1. Traditional CRUD Applications**

* **Use Case:** Internal HR app with classic Create/Read/Update/Delete operations on relational DB.
* **Why Avoid:** Most work is DB CRUD via JDBC (blocking). No benefit from reactive model.
* **Example:** Payroll or Leave Management system.
* **Impact if forced:** More complex code, same or worse performance.

**2. Apps with Blocking Libraries**

* **Use Case:** Heavy reliance on JDBC, legacy SOAP clients, or libraries without reactive support.
* **Why Avoid:** Even in WebFlux, blocking JDBC calls will block Netty event loop → performance collapse.
* **Example:** Banking app using Oracle DB with JDBC drivers only.
* **Impact if forced:** Thread starvation, worse scalability than MVC.

**3. CPU-Intensive Workloads**

* **Use Case:** Image processing, ML model inference, video encoding.
* **Why Avoid:** WebFlux helps with I/O, not CPU. Heavy CPU tasks still block threads.
* **Example:** Video streaming service transcoding files.
* **Impact if forced:** Complexity with no performance gain. Use worker pools instead.

**4. Small/Medium Monolithic Apps**

* **Use Case:** Simple CMS, ERP, e-commerce backend with <1000 concurrent users.
* **Why Avoid:** MVC is simpler, debugging is easier, devs are familiar.
* **Example:** A bookstore website running on Tomcat + JDBC.
* **Impact if forced:** Over-engineering, steeper learning curve for dev team.

**5. Teams New to Reactive Programming**

* **Use Case:** Project with strict deadlines and developers not trained in Reactor/Reactive Streams.
* **Why Avoid:** Steep learning curve, debugging async code is harder, error handling more complex.
* **Example:** Startup building MVP for B2B SaaS.
* **Impact if forced:** Slower development, more bugs, frustrated team.

**⚖️ Decision Matrix (Quick Guide)**

| **Scenario** | **MVC (Blocking)** | **WebFlux (Reactive)** |
| --- | --- | --- |
| CRUD over JDBC (blocking DB) | ✅ Best choice | ❌ Overkill |
| High concurrency API gateway | ❌ Thread explosion | ✅ Scales with event loop |
| Real-time streaming (chat, SSE, WebSocket) | ❌ Bad fit | ✅ Ideal |
| IoT data ingestion (100K+ devices) | ❌ Heavy | ✅ Efficient |
| CPU-heavy workloads | ✅ OK (with worker pools) | ❌ No benefit |
| External slow APIs (latency-prone) | ❌ Poor scalability | ✅ Parallel async |

👉 **Rule of Thumb**:

* If your app is **I/O-heavy & high-concurrency → WebFlux**.
* If your app is **CRUD/CPU-heavy/legacy blocking stack → Stick to MVC**.

# **✅ Spring WebFlux Quiz**

**1. What is the primary purpose of Spring WebFlux?**  
A. To manage SQL databases  
B. To create synchronous REST APIs  
C. To build reactive, non-blocking web applications  
D. To serve static files only

**Answer:** C. To build reactive, non-blocking web applications

**2. Which programming paradigm is at the core of Spring WebFlux?**  
A. Object-oriented programming  
B. Imperative programming  
C. Reactive programming  
D. Procedural programming

**Answer:** C. Reactive programming

**3. Which of the following is a key dependency used in Spring WebFlux?**  
A. spring-boot-starter-web  
B. spring-boot-starter-data-jpa  
C. spring-boot-starter-webflux  
D. spring-boot-starter-batch

**Answer:** C. spring-boot-starter-webflux

**4. What reactive types does Spring WebFlux use from Project Reactor?**  
A. Stream and Optional  
B. Mono and Flux  
C. CompletableFuture and Stream  
D. Promise and Async

**Answer:** B. Mono and Flux

**5. What does Mono represent in Spring WebFlux?**  
A. A stream of infinite data  
B. A stream of optional integers  
C. A sequence of 0 or 1 element  
D. A fixed batch of rows

**Answer:** C. A sequence of 0 or 1 element

**6. What does Flux represent in WebFlux?**  
A. A single value that never emits  
B. A collection of key-value pairs  
C. A sequence of 0 to N elements  
D. An SQL connection

**Answer:** C. A sequence of 0 to N elements

**7. Which of the following is true about WebFlux vs Spring MVC?**  
A. WebFlux uses blocking I/O, Spring MVC is non-blocking  
B. Both are synchronous  
C. WebFlux is reactive and non-blocking, Spring MVC is blocking  
D. WebFlux doesn't support annotations

**Answer:** C. WebFlux is reactive and non-blocking, Spring MVC is blocking

**8. Which annotation is commonly used in both Spring MVC and WebFlux controllers?**  
A. @Controller  
B. @Async  
C. @WebService  
D. @RestService

**Answer:** A. @Controller

**9. What is the reactive counterpart of RestTemplate in Spring WebFlux?**  
A. AsyncRestTemplate  
B. FluxRestClient  
C. WebClient  
D. ReactiveRestTemplate

**Answer:** C. WebClient

**10. Which of the following is NOT a benefit of using WebFlux?**  
A. Efficient resource utilization  
B. Non-blocking I/O  
C. Simple debugging of thread flow  
D. Scalability under high load

**Answer:** C. Simple debugging of thread flow

**11. How does Spring WebFlux handle backpressure?**  
A. Ignores it  
B. Uses bounded queues  
C. Relies on Reactive Streams specification  
D. Uses threading

**Answer:** C. Relies on Reactive Streams specification

**12. Which class is used to define functional routing in Spring WebFlux?**  
A. RouterFunctions  
B. WebRouter  
C. FluxRouter  
D. RouteBuilder

**Answer:** A. RouterFunctions

**13. What does flatMap() do in the context of a Flux or Mono?**  
A. Transforms the elements synchronously  
B. Filters null values  
C. Maps each element to another reactive type and flattens it  
D. Creates an exception

**Answer:** C. Maps each element to another reactive type and flattens it

**14. What server implementations can be used with Spring WebFlux?**  
A. Tomcat only  
B. Undertow only  
C. Netty, Tomcat, Jetty, and Undertow  
D. Oracle WebLogic only

**Answer:** C. Netty, Tomcat, Jetty, and Undertow

**15. In Spring WebFlux, what does subscribe() do?**  
A. Cancels the publisher  
B. Blocks the thread  
C. Starts the execution of a Publisher  
D. Adds a value to the stream

**Answer:** C. Starts the execution of a Publisher