Spring WebFlux tutorial: how to build a reactive web app

The Educative Team · Follow
Published in Javarevisited · 8 min read · Mar 5, 2021

Image Source: Author

Reactive systems allow for the **unparalleled responsiveness and scalability** that we need in our high data flow world. However, reactive systems need

tools and developers specially trained to implement these unique program architectures. Spring WebFlux with Project Reactor is a framework specially built to meet the reactive needs of modern companies.

Today, we'll help you get started with **WebFlux** by explaining how it fits with other reactive stack tools, how it's different, and how to make your first app.

Here's what we'll cover today:

- What is a Reactive System?
- What is Project Reactor?
- What is Spring WebFlux?
- Salient Features of Spring WebFlux
- Get Started with Spring WebFlux
- Next steps for your learning

What is a Reactive System?

Reactive Systems are systems designed with a reactive architectural pattern that prioritizes the use of loosely coupled, flexible, and scalable components. They're also designed with **failure resolution in mind** to ensure most of the system will still operate even if one fails.

Reactive systems focus on:

- **Reactiveness**: Above all else, reactive systems should respond quickly to any user input. Reactive systems advocates argue that reactiveness helps

to optimize all other parts of the system from data collection to user experience.

- **Resilience:** Reactive systems should be designed to anticipate system failures. Reactive systems expect that components will fail eventually and design loosely-coupled systems that can remain active even if several individual parts stop working.
- **Elasticity**: Reactive systems should adapt to the size of the workload by scaling up or down to meet demands. Many reactive systems will also use predictive scaling to anticipate and prepare for sudden shifts. The key to implementing elasticity is to remove any bottlenecks and build systems that can shard or replicate components as required.
- **Message-driven communication:** All components of a reactive system are loosely coupled with hard boundaries between each. Your system should communicate across these boundaries with explicit message passing. These messages keep different components informed about failures and help them delegate workflow to components that can handle it.

The most notable difference between reactive and other web patterns is that reactive systems can execute multiple unblocked calls at once rather than having some calls wait for others. Reactive systems, therefore, speed up performance and responsiveness because each part of the web application can do its part sooner than if it had to wait for another part.

> In short, reactive systems use loosely coupled, unblocked components to increase performance, user experience, and error-handling.

What is Project Reactor?

Project Reactor is a framework built by Pivotal and powered by Spring. It implements reactive API patterns, most notably the Reactive Streams specification.

If you're familiar with Java 8 Streams, you'll quickly find many similarities between a Stream and a Flux (or its single-element version, Mono). The main difference between them is that Fluxes and Monos follow a `publisher-subscriber` pattern and implement backpressure, while the Stream API does not.

> **Backpressure** is a way for data endpoints to signal to the data producer that it is receiving too much data. This allows for better flow management and distribution as it prevents individual components from getting overworked.

The main advantage of using a Reactor is that you're in total control of the data flow. You can rely on the subscriber's ability to ask for more information when it's ready to process it, or buffer some results on the publisher's side, or even use a full push approach without backpressure.

In our reactive stack, it sits below Spring Boot 2.0 and above WebFlux:

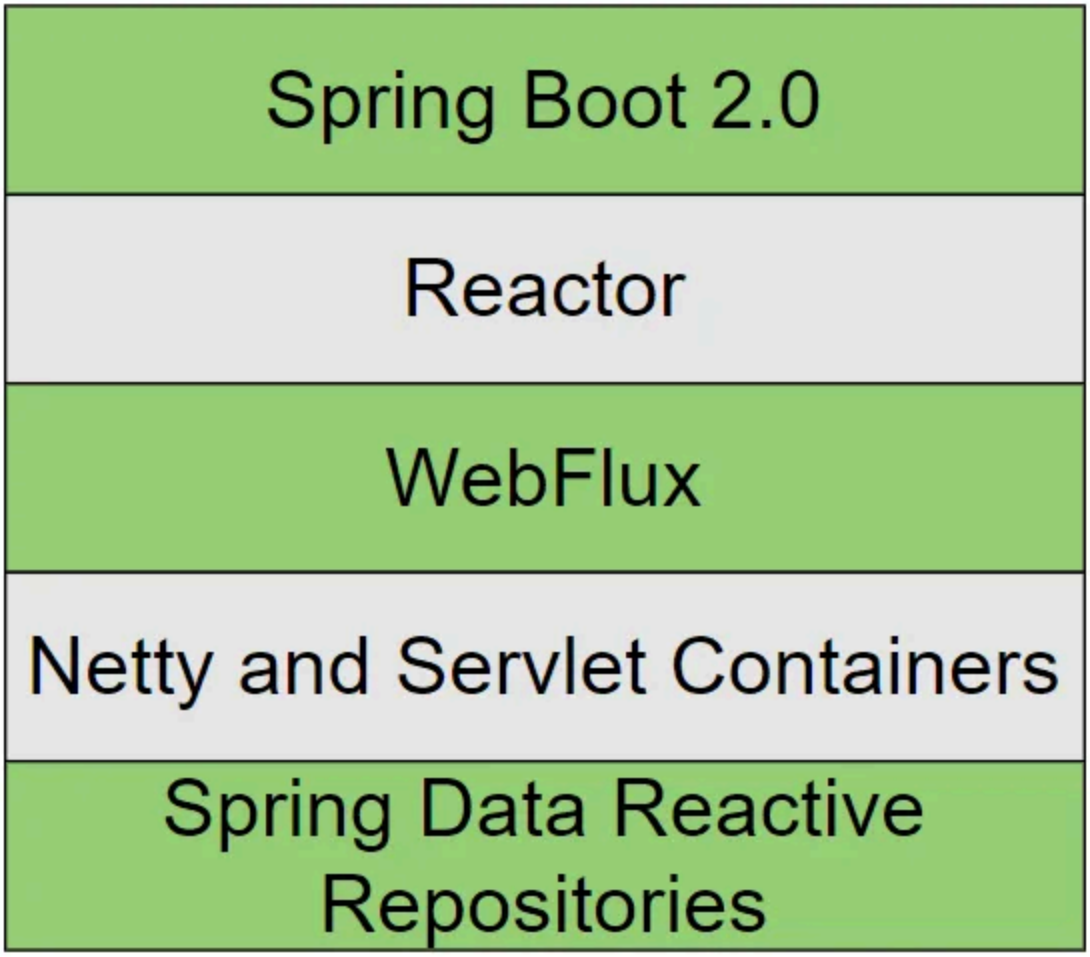


Image Source: Author

> **Stack:** A tech stack is a combination of software tools and programming languages used to create a web or mobile application. The reactive stack is the same but for creating reactive applications.

What is Spring WebFlux?

Spring WebFlux is a fully non-blocking, annotation-based web framework built on Project Reactor that makes it possible to build reactive applications on the HTTP layer. WebFlux uses a new **router functions** feature to apply functional programming to the web layer and bypass declarative controllers and RequestMappings. WebFlux requires you to import Reactor as a core dependency.

WebFlux was added in Spring 5 as a reactive alternative to Spring MVC, with added support for:

- **Non-blocking threads:** Concurrent threads that complete their designated task without waiting for previous tasks to be completed.
- **Reactive Stream API:** A standardized tool that includes options for asynchronous stream processing with non-blocking backpressure.
- **Asynchronous data processing:** When data is processed in the background and the user can continue using normal app functionality without interruption.

Ultimately, WebFlux does away with SpringMVCs thread-per-request model and instead uses a multi-EventLoop non-blocking model to enable reactive, scalable apps. With support for popular servers like Netty, Undertow, and Servlet 3.1+ containers, WebFlux has come in as a key part of a reactive stack.

Salient Features of Spring WebFlux

Router functions

`RouterFunction` is a functional alternative to the `@RequestMapping` and `@Controller` annotation style used in standard Spring MVC.

We can use it to route requests to the handler functions:

```
@RestController
public class ProductController {
    @RequestMapping("/product")
    public List<Product> productListing() {
```

```
            return ps.findAll();
        }
    }

    @Bean
    public RouterFunction<ServerResponse> productListing(ProductService
    ps) {
        return route().GET("/product", req -> ok().body(ps.findAll()))
          .build();
    }
```

You can use the `RouterFunctions.route()` to create routes instead of writing a complete router function. Routes are registered as Spring beans and therefore can be created in any configuration class.

Router functions avoid potential side effects caused by the multi-step process of request mapping and instead streamline it to a direct router/handler chain. This allows for functional programming implementations of reactive programming.

> `RequestMapping` and `Controller` annotation styles are still valid in WebFlux if you are more comfortable with the old style, `RouterFunctions` is just a new option for your solutions.

WebClient

WebClient is WebFlux's reactive web client built from the well-known `RestTemplate`. It is an interface that represents the main entry point for web requests and supports both synchronous and asynchronous operations. WebClient is mostly used for reactive backend-to-backend communication.

You can build and create a WebClient instance by importing standard WebFlux dependencies with Maven:

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

WebClient client = WebClient.create();
```

Reactive Steam API

Reactive Stream API is an imported collection of functions that allow for smarter stream data flow. It has built-in support for back-pressure and asynchronous processing that ensures the application makes the most efficient use of both computer and component resources.

There are four major interfaces in Reactive Stream API:

- **Publisher**: Emits events to linked `Subscribers` based on their demands. Acts as a central linking point that `subscribers` can watch for events.
- **Subscriber**: Receives and processes events emitted by `Publisher`. Multiple `Subscribers` can link to a single `Publisher` and respond differently to the same event. Subscribers can be set to react:
- `onNext`, when it receives the next event.
- `onSubscribe`, when a new subscriber is added
- `onError`, when an error occurs with another subscriber
- `onComplete`, when another subscriber finishes its task
- **Subscription**: Defines a relationship between selected `Publisher` and `Subscriber`. Each `Subscriber` can only be linked to a single `Publisher`.
- **Processor**: Represents the processing stage of the `Subscriber`

Servers

WebFlux is supported on Tomcat, Jetty, Servlet 3.1+ containers, as well as on non-Servlet runtimes such as Netty and Undertow. Netty is most commonly used for **async and non-blocking designs**, so WebFlux will default to that. You can easily switch between these server options with a simple change to your Maven or Gradle build software.

This makes WebFlux highly versatile in what technologies it can work with and allows you to easily implement it with existing infrastructure.

Concurrency Model

WebFlux is built with non-blocking in mind and therefore uses a different concurrent programming model from Spring MVC.

Spring MVC assumes threads will be blocked and uses a large thread pool to keep moving during instances of blocking. This larger thread pool makes MVC more resource-intensive as the computer hardware must keep more threads spun up at once.

WebFlux instead uses a **small thread pool** because it assumes you'll never need to pass off work to avoid a blocker. These threads, called **event loop workers**, are fixed in number and cycle through incoming requests quicker than MVC threads. This means WebFlux uses computer resources more efficiently because active threads are always working.

Spring WebFlux Security

WebFlux uses Spring Security to implement authentication and authorization protocols. Spring Security uses `WebFilter` to check requests

against an authenticated list of users, or it can be set to automatically refuse requests that fit criteria like origin or request type.

```
@EnableWebFluxSecurity
public class HelloWebFluxSecurityConfig {
    @Bean
    public MapReactiveUserDetailsService userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("user")
            .roles("USER")
            .build();
        return new MapReactiveUserDetailsService(user);
    }
}
```

This is a minimal implementation that sets all settings to default. Here we can see that the user has a `username`, a `password`, and one or more `roles` tags that allow them a certain level of access.

Get Started with Spring WebFlux

Now let's get hands-on with WebFlux. First, we'll need to set up a project. We'll use `Spring Initializr` to generate a Maven build with the `Spring Reactive Web` dependency.

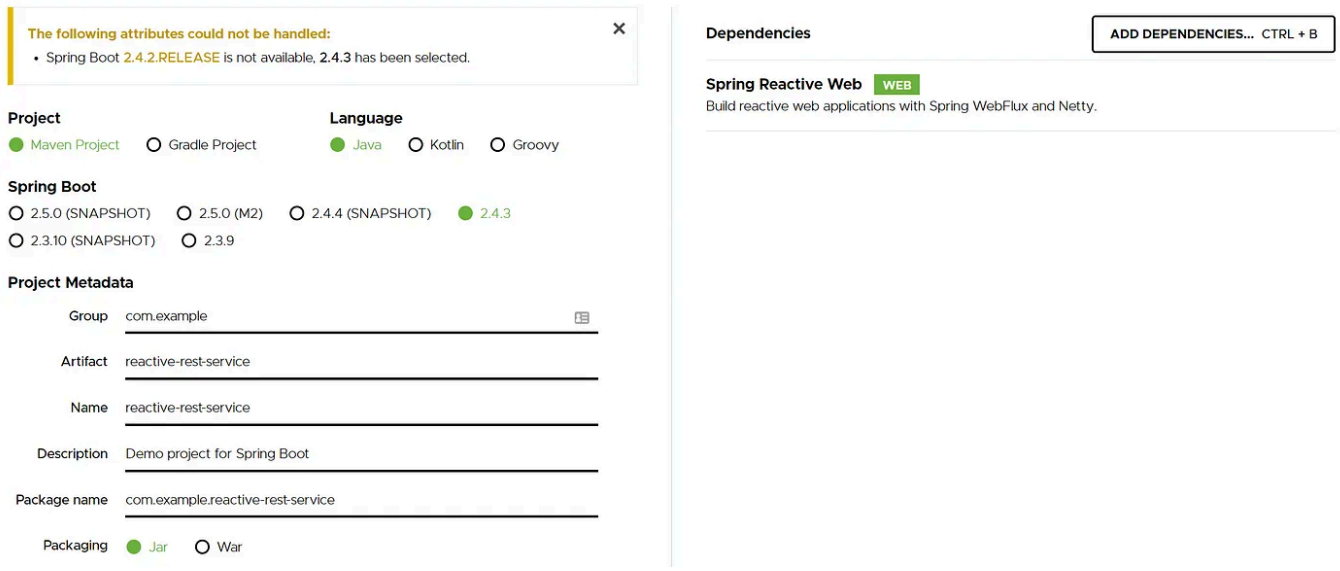


Image Source: Author

This will generate a `pom.xml` file that looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.4.2</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>reactive-rest-service</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>reactive-rest-service</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-webflux</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>io.projectreactor</groupId>
            <artifactId>reactor-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Now you have everything you need to continue! From here we'll add some components to make an `Hello-World` app. We'll just add a router and a handler, which are the minimum requirements to create our basic WebFlux app.

Router

First, we'll create an example router to show our text once at the URL `http://localhost:8080/example`. This defines how the user can request the data that we'll define in our handler.

```
@Configuration
public class ExampleRouter {
  @Bean
  public RouterFunction<ServerResponse> routeExample (ExampleHandler
exampleHandler) {
    return RouterFunctions

.route(RequestPredicates.GET("/example").and(RequestPredicates.accept
(MediaType.TEXT_PLAIN)), exampleHandler::hello);
```

```
    }
}
```

Handler

Now we'll add a handler that listens for any user that requests the `/example` route. Once the router recognizes that the requested path matches, it sends the user to the handler. Our handler receives the message and brings the user to a page with our greeting.

```
@Component
public class ExampleHandler {
  public Mono<ServerResponse> hello(ServerRequest request) {
    return ServerResponse.ok().contentType(MediaType.TEXT_PLAIN)
        .body(BodyInserters.fromObject("Hello, Spring WebFlux
Example!"));
  }
}
```

Run the application

Now we'll run our application by executing the Maven target `spring-boot:run`. You'll now be able to visit `http://localhost:8080/example` in your browser to find:

```
Hello, Spring WebFlux Example!
```

Next steps for your learning

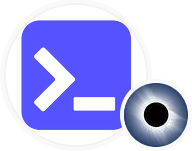
Now that you've finished your first basic app, you're ready to move onto some more advanced topics and applications.

Some next concepts to explore are:

- Reactive Controllers
- Repository Layer
- Angular with WebFlux
- MongoDB and WebFlux
- Classic and functional endpoints

Happy learning!

Spring Boot | Webflux | Java | Programming | Software Development

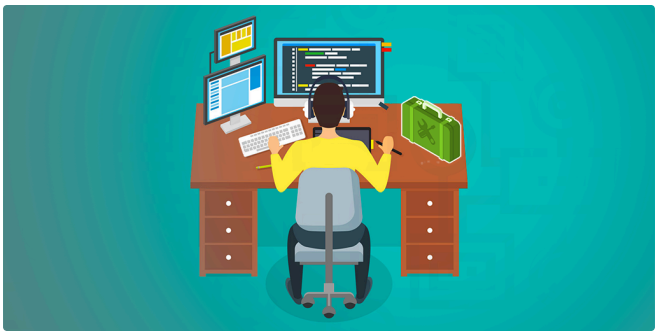


Written by The Educative Team

16.1K Followers · Writer for Javarevisited

Master in-demand coding skills with Educative's hands-on courses & tutorials.

More from The Educative Team and Javarevisited



The Educative Team in Dev Learning Daily

The 7 Most Important Software Design Patterns

For a comprehensive deep-dive into the subject of Software Design Patterns, check...

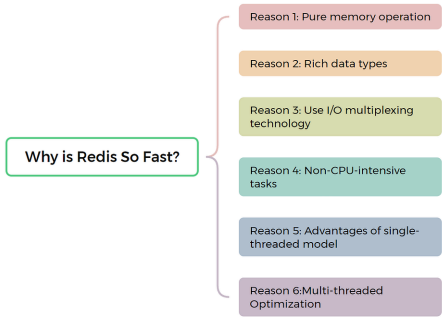
Nov 8, 2018 3K 8

Dylan Smith in Javarevisited

Interview: How to Check Whether a Username Exists Among One...

My articles are open to everyone; non-member readers can read the full article by...

Aug 18 2.8K 47



Dylan Smith in Javarevisited

Interview: Why is Redis so fast even though it is single-threaded?...

My articles are open to everyone; non-member readers can read the full article by...

Aug 31 1K 6

The Educative Team in Grokking the Tech Interview

Top 25 System Design Interview questions in 2024

Over my 10+ years as a systems engineer and hiring manager at Microsoft and Facebook, I...

Aug 21 113

See all from The Educative Team

See all from Javarevisited

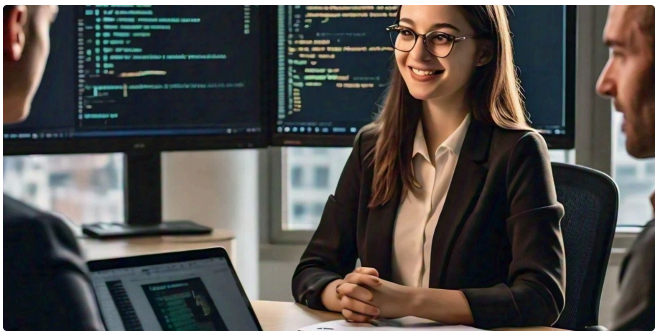
Recommended from Medium

Rabinarayan Patra

Why 1==1 is true but 128==128 is false in Java

Ever wondered why comparing 1==1 returns true, but 128==128 returns false in Java? Let'...

Sep 11 219 6



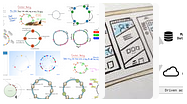
Sanjay Singh

Top Spring Boot and Microservices Interview Questions: No Question...

Zero to Hero: Ace your Java interview with our comprehensive guide to Java Stack. Boost...

Aug 23 157 5

Lists



General Coding Knowledge

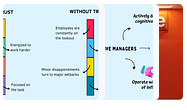
20 stories · 1579 saves

Coding & Development

11 stories · 815 saves

Stories to Help You Grow as a Software Developer

19 stories · 1368 saves



Leadership

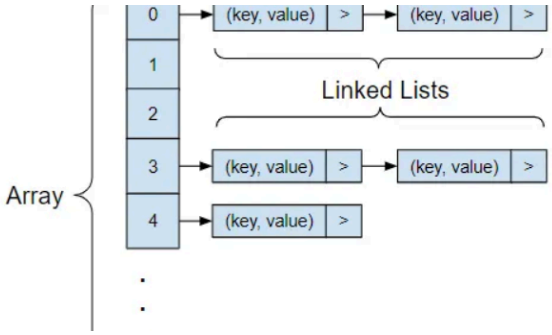
56 stories · 438 saves

Hùng Trần in Javarevisited

Best practices for using Java Stream

When working with Java, understanding and knowing how to use streams to optimize wor...

Apr 17 60



Ajay Rathod

Barclays Java Spring-Boot Micro-service Interview Question with...

Hello folks welcome to another real life interview transcript from a banking giant. Th...

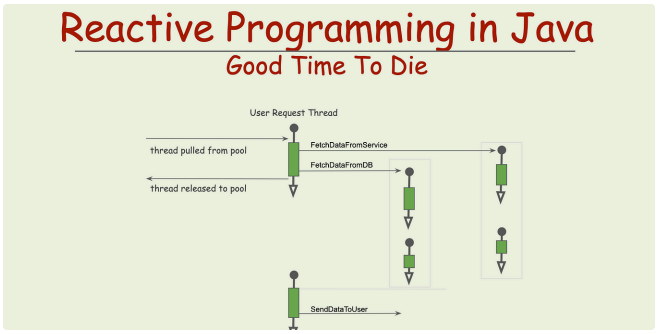
Mar 30 431 5

Raksmey Koung

When should you choose Spring WebFlux over @Async for buildin...

Choosing between Spring WebFlux and asynchronous operations with @Async...

May 7 3



Viraj Shetty

Reactive Programming in Java — Good Time to Die

This article explains the reason for Reactive Programming, why it is not popular with...

Oct 12, 2023 997 11

See more recommendations