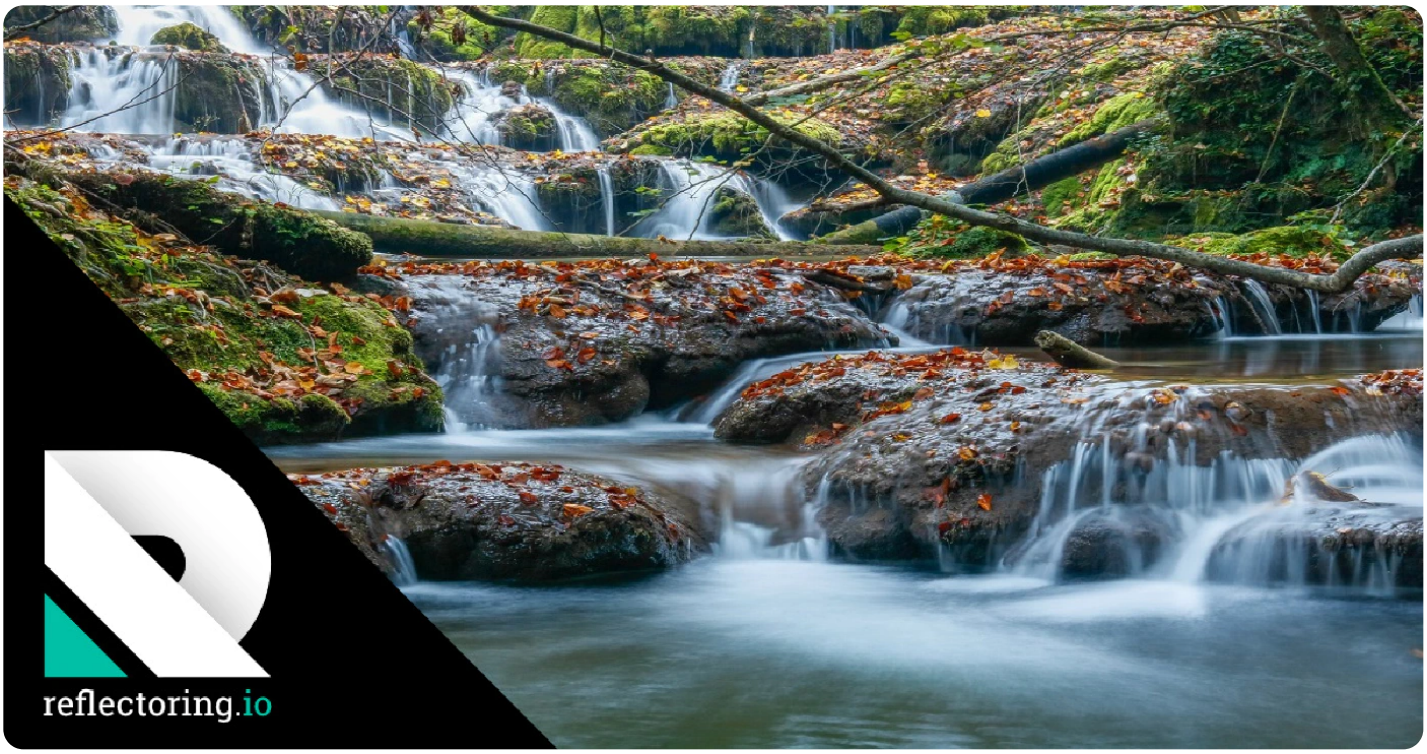


Getting Started with Spring WebFlux

March 10, 2022 Spring

Table Of Contents

Most traditional applications deal with *blocking* calls or, in other words, *synchronous* calls. This means that if we want to access a particular resource in a system with most of the threads being busy, then the application would block the new one or wait until the previous threads complete processing its requests.

If we want to process *Big Data*, however, we need to do this with immense speed and agility. That's when the software developers realized that they would need some kind of multi-threaded environment that handles asynchronous and non-blocking calls to make the best use of processing resources.

Example Code

This article is accompanied by a working code example on GitHub.

What is a Stream?

Before jumping on to the reactive part, we must understand what streams are. A *Stream* is a sequence of data that is transferred from one system to another. It traditionally operates in a blocking, sequential, and FIFO (first-in-first-out) pattern.

This blocking methodology of data streaming often prohibits a system to process real-time data while streaming. Thus, a bunch of prominent developers realized that they would need an approach to build a "reactive" systems architecture that would ease the processing of data *while streaming*. Hence, they signed a manifesto, popularly known as the Reactive Manifesto.

The authors of the manifesto stated that a reactive system must be an *asynchronous* software that deals with *producers* who have the single responsibility to send messages to *consumers*. They introduced the following features to keep in mind:

> **Responsive**: Reactive systems must be fast and responsive so that they can provide consistent high quality of service.
>
> **Resilient**: Reactive systems should be designed to anticipate system failures. Thus, they should be responsive through replication and isolation.

Elastic: Reactive systems must be adaptive to shard or replicate components based upon their requirement. They should use predictive scaling to anticipate sudden ups and downs in their infrastructure.

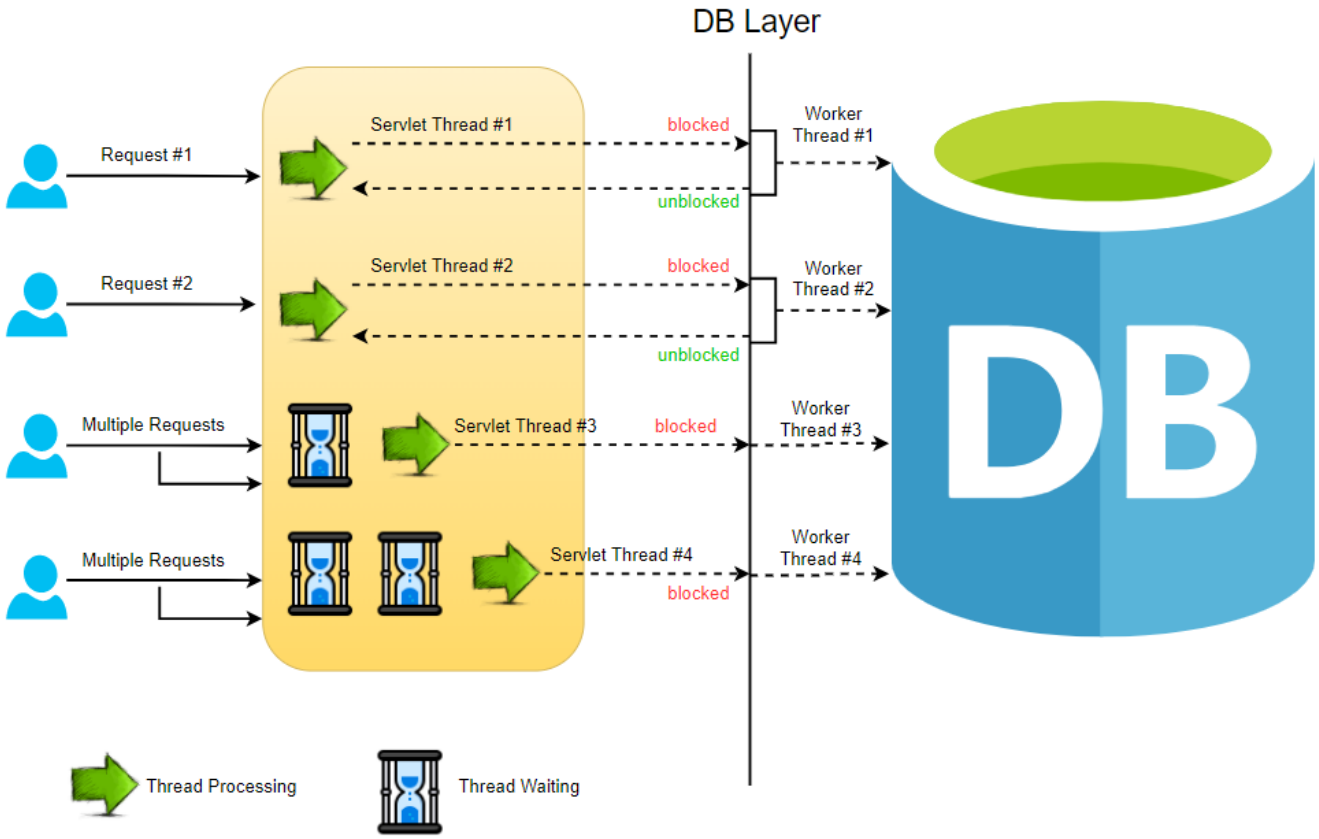
Message-driven: Since all the components in a reactive system are supposed to be loosely coupled, they must communicate across their boundaries by asynchronously exchanging messages.

Introducing Reactive Programming Paradigm

Reactive programming is a programming paradigm that helps to implement non-blocking, asynchronous, and event-driven or message-driven data processing. It models data and events as streams that it can observe and react to by processing or transforming the data. Let's talk about the differences between blocking and non-blocking request processing.

Blocking Request

In a conventional MVC application, whenever a request reaches the server, a servlet thread is being created and delegated to worker threads to perform various operations like I/O, database processing, etc. While the worker threads are busy completing their processes, the servlet threads enter a waiting state due to which the calls remain blocked. This is blocking or synchronous request processing.



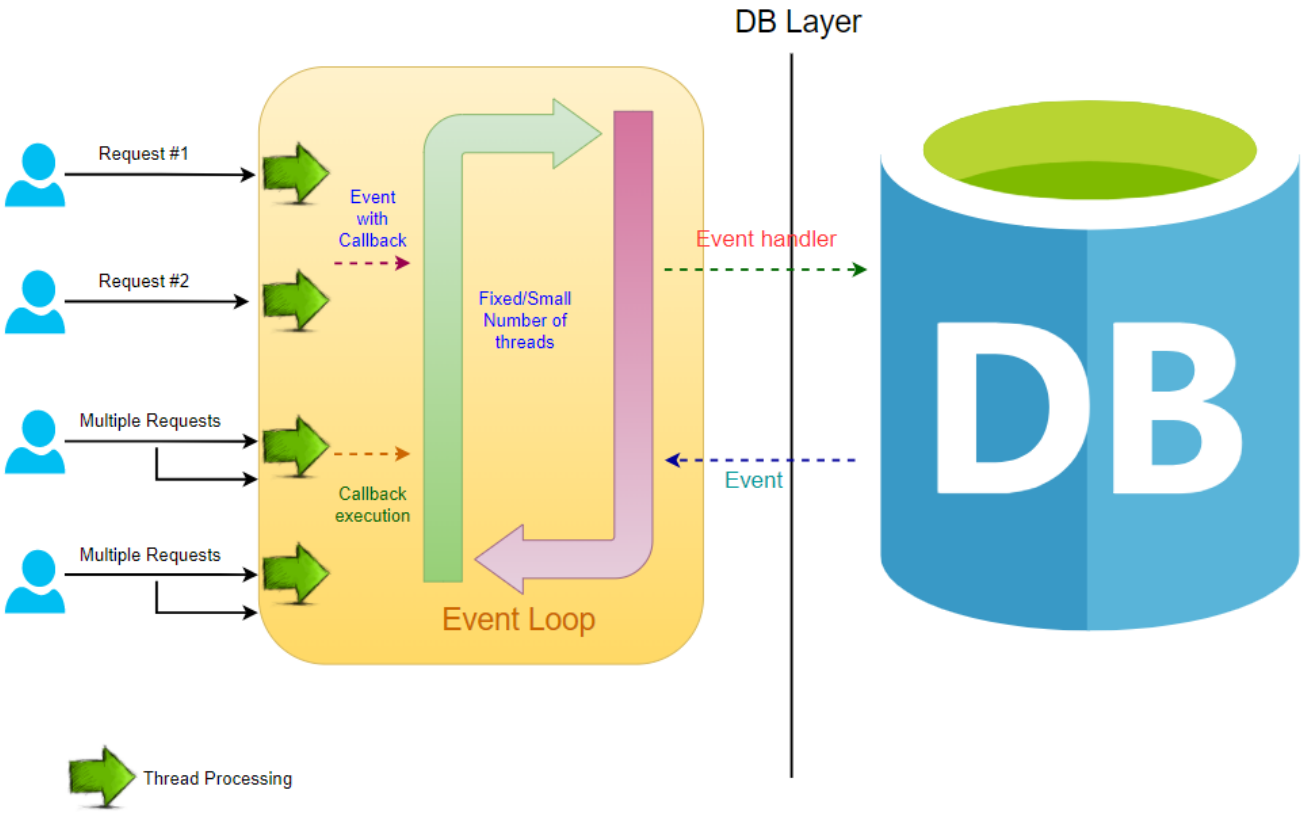
Non-Blocking Request

In a non-blocking system, all the incoming requests are accompanied by an event handler and a callback. The request thread delegates the incoming request to a thread pool that manages a pretty small number of threads. Then the thread pool delegates the request to its handler function and gets available to process the next incoming requests from the request thread.

When the handler function completes its process, one of the threads from the pool fetches the response and passes it to the callback function. Thus

the threads in a non-blocking system never go into the waiting state. This increases the productivity and the performance of the application.

A single request is potentially processed by multiple threads!



Backpressure

Working with reactive code, we often come across the term “backpressure”. It is an analogy derived from fluid dynamics which literally means the *resistance* or *force* that opposes the desired flow of data. In Reactive Streams, backpressure defines the mechanism to regulate the data transmission across streams.

Consider that server A sends 1000 EPS (events per second) to server B. But server B could only process 800 EPS and thus has a deficit of 200 EPS. Server B would now tend to fall behind as it has to process the deficit data and send it downstream or maybe store it in a database. Thus, server B deals with backpressure and soon will go out of memory and fail.

So, this backpressure can be handled or managed by the following options or strategies:

Buffer - We can easily buffer the deficit data and process it later when the server has capacity. But with a huge load of data coming in, this buffer might increase and the server would soon run out of memory.

Drop - Dropping, i.e. not processing events, should be the last option. Usually, we can use the concept of data sampling combined with buffering to achieve less data loss.

Control - The concept of controlling the producer that sends the data is by far the best option. Reactive Streams provides various options in both push and pull-based streams to control the data that is being produced and sent to the consumer.

Reactive Java Libraries

The reactive landscape in Java has evolved a lot in recent years. Before we move on further to understand the Spring Webflux component, let's take a look into the reactive core libraries written in Java today. Here are the most popular ones:

RxJava: It is implemented out of the ReactorX project which hosts implementations for multiple programming languages and platforms. *ReactiveX* is a combination of the best ideas from the *Observer* pattern, the *Iterator* pattern, and *functional programming*.

Project Reactor: Reactor is a framework built by Pivotal and powered by Spring. It is considered as one of the foundations of the *reactive stack* in the Spring ecosystem. It implements Reactive API patterns which are based on the *Reactive Streams* specification.

Akka Streams: Although it implements the Reactive Streams implementation, the Akka Streams API is completely decoupled from the Reactive Streams interfaces. It uses *Actors* to deal with the streaming data. It is considered a 3rd generation Reactive library.

Ratpack: Ratpack is a set of Java libraries used for building scalable and high-performance HTTP applications. It uses Java 8, *Netty*, and reactive principles to provide a basic implementation of Reactive Stream API. You can also use Reactor or RxJava along with it.

Vert.x: Vert.x is a foundation project by *Eclipse* which delivers a *polyglot event-driven framework* for JVM. It is similar to Ratpack and allows to use RxJava or their native implementation for Reactive Streams API.

Spring Webflux is internally built using the core components of RxJava and RxNetty.

Intro to Java 9 Reactive Streams API

The whole purpose of Reactive Streams was to introduce a standard for asynchronous stream processing of data with non-blocking backpressure. Hence, Java 9 introduced the *Reactive Streams API*. It is implemented based upon the *Publisher-Subscriber* Model or *Producer-Consumer* Model and primarily defines four interfaces:

`Publisher`: It is responsible for preparing and transferring data to subscribers as individual messages. A `Publisher` can serve multiple subscribers but it has only one method, `subscribe()`.

```
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}
```

`Subscriber`: A `Subscriber` is responsible for receiving messages from a `Publisher` and processing those messages. It acts as a terminal operation in the Streams API. It has four methods to deal with the events received:

`onSubscribe(Subscription s)`: Gets called automatically when a publisher registers itself and allows the subscription to request data.

`onNext(T t)`: Gets called on the subscriber every time it is ready to receive a new message of generic type T.

`onError(Throwable t)`: Is used to handle the next steps whenever an error is monitored.

`onComplete()`: Allows to perform operations in case of successful subscription of data.

```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

`Subscription`: It represents a relationship between the subscriber and publisher. It can be used only once by a single `Subscriber`. It has methods that allow requesting for data and to cancel the demand:

```
public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

`Processor`: It represents a processing stage that consists of both `Publisher` and `Subscriber`.

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

Introduction to Spring Webflux

Spring introduced a *Multi-Event Loop* model to enable a reactive stack known as `WebFlux`. It is a fully non-blocking and annotation-based web framework built on *Project Reactor* which allows building reactive web

applications on the HTTP layer. It provides support for popular inbuilt severs like *Netty*, *Undertow*, and *Servlet 3.1* containers.

Before we get started with Spring Webflux, we must accustom ourselves to two of the publishers which are being used heavily in the context of Webflux:

`Mono`: A `Publisher` that emits 0 or 1 element.

```
Mono<String> mono = Mono.just("John");
Mono<Object> monoEmpty = Mono.empty();
Mono<Object> monoError = Mono.error(new Exception());
```

`Flux`: A `Publisher` that emits 0 to N elements which can keep emitting elements forever. It returns a sequence of elements and sends a notification when it has completed returning all its elements.

```
Flux<Integer> flux = Flux.just(1, 2, 3, 4);
Flux<String> fluxString = Flux.fromArray(new String[]{"A", "B", "C"});
Flux<String> fluxIterable = Flux.fromIterable(Arrays.asList("A", "B",
Flux<Integer> fluxRange = Flux.range(2, 5);
Flux<Long> fluxLong = Flux.interval(Duration.ofSeconds(10));

// To Stream data and call subscribe method
List<String> dataStream = new ArrayList<>();
Flux.just("X", "Y", "Z")
    .log()
    .subscribe(dataStream::add);
```

Once the stream of data is created, it needs to be subscribed to so it starts emitting elements. The data won't flow or be processed until the `subscribe()` method is called. Also by using the `.log()` method above, we can trace and observe all the stream signals. The events are logged into the console.

Reactor also provides operators to work with `Mono` and `Flux` objects. Some of them are:

`Map` - It is used to transform from one element to another.

`FlatMap` - It flattens a list of `Publisher`s to the values that these publishers emit. The transformation is asynchronous.

`FlatMapMany` - This is a `Mono` operator which is used to transform a `Mono` object into a `Flux` object.

`DelayElements` - It delays the publishing of each element by a defined duration.

`Concat` - It is used to combine the elements emitted by a `Publisher` by keeping the sequence of the publishers intact.

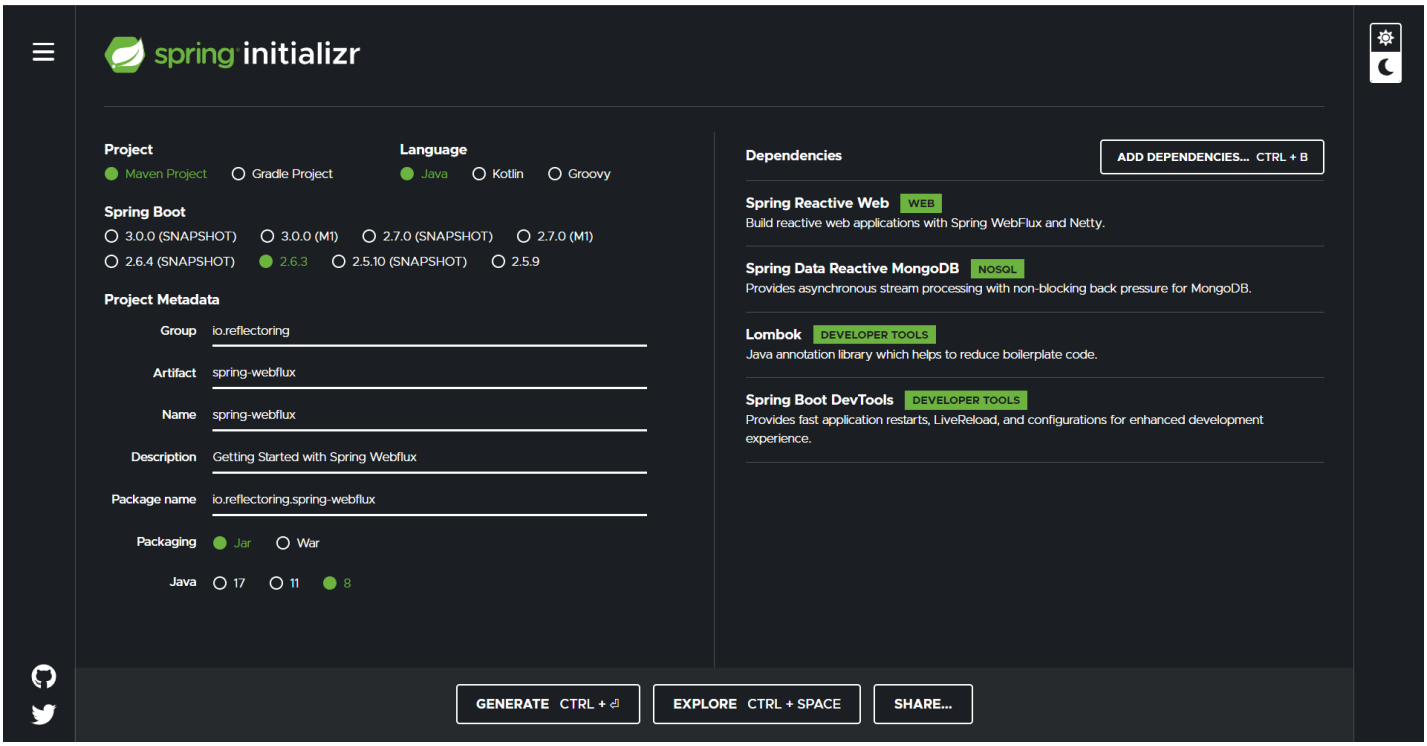
`Merge` - It is used to combine the publishers without keeping its sequence.

`Zip` - It is used to combine two or more publishers by waiting on all the sources to emit one element and combining these elements into an output value.

Sping Webflux Dependencies

Until now we have spoken a lot about Reactive Streams and Webflux. Let's get started with the implementation part. We are going to build a REST API using Webflux and we will use MongoDB as our database to store data. We will build a user management service to store and retrieve users.

Let's initialize the Spring Boot application by defining a skeleton project in Spring Initializr:



We have added the *Spring Reactive Web* dependency, *Spring Data Reactive MongoDB* to reactively connect to MongoDB, *Lombok* and *Spring DevTools*. The use of Lombok is optional, as it's a convenience library that helps us reduce boilerplate code such as getters, setters, and constructors, just by annotating our entities with Lombok annotations. Similar for Spring DevTools.

Data Model

Let's start by defining the `User` entity that we will be using throughout our implementation:

```
@ToString
@EqualsAndHashCode(of = {"id","name","department"})
@AllArgsConstructor
@NoArgsConstructor
@Data
@Document(value = "users")
public class User {

    @Id
    private String id;
    private String name;
    private int age;
    private double salary;
    private String department;
}
```

We are initially using the Lombok annotations to define Getters, Setters, `toString()`, `equalsAndHashCode()` methods, and constructors to reduce boilerplate implementations. We have also used `@Document` to mark it as a MongoDB entity.

Persistence Layer - Defining Repositories

Next, we will define our Repository layer using the `ReactiveMongoRepository` interface.

```
ository
ic interface UserRepository extends ReactiveMongoRepository<User, String>
```

Service Layer

Now we will define the Service that would make calls to MongoDB using Repository and pass the data on to the web layer:

```
    public Mono<User> updateUser(String userId,  User user){
        return userRepository.findById(userId)
                .flatMap(dbUser -> {
                    dbUser.setAge(user.getAge());
                    dbUser.setSalary(user.getSalary());
                    return userRepository.save(dbUser);
                });
    }

    public Mono<User> deleteUser(String userId){
        return userRepository.findById(userId)
                .flatMap(existingUser -> userRepository.delete(existingUs
                        .then(Mono.just(existingUser)));
    }

    public Flux<User> fetchUsers(String name) {
        Query query = new Query()
                .with(Sort
                        .by(Collections.singletonList(Sort.Order.asc("age
                );
        query.addCriteria(Criteria
                .where("name")
                .regex(name)
        );

        return reactiveMongoTemplate
                .find(query, User.class);
    }
}
```

We have defined service methods to save, update, fetch, search and delete a user. We have primarily used `UserRepository` to store and retrieve data from MongoDB, but we have also used a `ReactiveTemplate` and `Query` to search for a user given by a regex string.

Web Layer

We have covered the middleware layers to store and retrieve data, let's just focus on the web layer. Spring Webflux supports two programming models:

- Annotation-based Reactive components
- Functional Routing and Handling

Annotation-based Reactive Components

Let's first look into the annotation-based components. We can simply create a `UserController` for that and annotate with routes and methods:

```
}
```

This almost looks the same as the controller defined in Spring MVC. But the major difference between Spring MVC and Spring Webflux relies on how the request and response are handled using non-blocking publishers *Mono* and *Flux*.

We don't need to call subscribe methods in the Controller as the internal classes of Spring would call it for us at the right time.

Do Not Block!

We must make sure that we don't use any blocking methods throughout the lifecycle of an API. Otherwise, we lose the main advantage of reactive programming!

Functional Routing and Handling

Initially, the Spring Functional Web Framework was built and designed for Spring Webflux but later it was also introduced in Spring MVC. We use functions for routing and handling requests. This introduces an alternative programming model to the one provided by the Spring annotation-based framework.

First of all, we will define a `Handler` function that can accept a `ServerRequest` as an incoming argument and returns a `Mono` of `ServerResponse` as the response of that functional method. Let's name the handler class as `UserHandler`:

```
                        .body(userService.createUser(u), User.class)
                );
    }

    public Mono<ServerResponse> updateUserById(ServerRequest request) {
        String id = request.pathVariable("userId");
        Mono<User> updatedUser = request.bodyToMono(User.class);

        return updatedUser
                .flatMap(u -> ServerResponse
                        .ok()
                        .contentType(MediaType.APPLICATION_JSON)
                        .body(userService.updateUser(id, u), User.class)
                );
    }

    public Mono<ServerResponse> deleteUserById(ServerRequest request){
        return userService.deleteUser(request.pathVariable("userId"))
                .flatMap(u -> ServerResponse.ok().body(u, User.class))
                .switchIfEmpty(ServerResponse.notFound().build());
    }
}
```

Next, we will define the router function. Router functions usually evaluate the request and choose the appropriate handler function. They serve as an alternate to the `@RequestMapping` annotation. So we will define this `RouterFunction` and annotate it with `@Bean` within a `@Configuration` class to inject it into the Spring application context:

```
    @Bean
    RouterFunction<ServerResponse> routes(UserHandler handler) {
        return route(GET("/handler/users").and(accept(MediaType.APPLICATI
                .andRoute(GET("/handler/users/{userId}").and(contentType(
                .andRoute(POST("/handler/users").and(accept(MediaType.APP
                .andRoute(PUT("/handler/users/{userId}").and(contentType(
                .andRoute(DELETE("/handler/users/{userId}").and(accept(Me
    }
}
```

Finally, we will define some properties as part of `application.yaml` in order to configure our database connection and server config.

```
spring:
  application:
    name: spring-webflux-guide
  webflux:
    base-path: /api
  data:
    mongodb:
      authentication-database: admin
      uri: mongodb://localhost:27017/test
      database: test

logging:
  level:
    io:
      reflectoring: DEBUG
    org:
      springframework:
```

```
        web: INFO
        data:
          mongodb:
            core:
              ReactiveMongoTemplate: DEBUG
    reactor:
      netty:
        http:
          client: DEBUG
```

This constitutes our basic non-blocking REST API using Spring Webflux. Now this works as a Publisher-Subscriber model that we were talking about initially in this article.

Server-Sent Events

Server-Sent Events (SSE) is an HTTP standard that provides the capability for servers to push streaming data to the web client. The flow is unidirectional from server to client and the client receives updates whenever the server pushes some data. This kind of mechanism is often used for real-time messaging, streaming or notification events. Usually, for multiplexed and bidirectional streaming, we often use *Websockets*. But SSE are mostly used for the following use-cases:

> Receiving live feed from the server whenever there is a new or updated record.
>
> Message notification without unnecessary reloading of a server.
>
> Subscribing to a feed of news, stocks, or cryptocurrency

The biggest limitation of SSE is that it's unidirectional and hence information can't be passed to a server from the client. Spring Webflux allows us to define server streaming events which can send events in a given interval. The web client initiates the REST API call and keeps it open until the event stream is closed.

The server-side event would have the content type `text/event-stream`. Now we can define a Server Side Event streaming endpoint using WebFlux by simply returning a `Flux` and specifying the content type as `text/event-stream`. So let's add this method to our existing `UserController`:

```
@GetMapping(value = "/stream", produces = MediaType.TEXT_EVENT_STREAM
public Flux<User> streamAllUsers() {
    return userService
            .getAllUsers()
            .flatMap(user -> Flux
                    .zip(Flux.interval(Duration.ofSeconds(2)),
                            Flux.fromStream(Stream.generate(() -> use
                    )
                    .map(Tuple2::getT2)
            );
}
```

Here, we will stream all the users in our system every 2 seconds. This serves the whole list of updated users from the MongoDB every interval.

Webflux Internals

Traditionally, Spring MVC uses the *Tomcat* server for servlet stack applications whereas Spring WebFlux uses *Reactor Netty* by default for reactive stack applications.

Reactor Netty is an asynchronous, event-driven network application framework built out of Netty server which provides non-blocking and backpressure-ready network engines for HTTP, TCP, and UDP clients and servers.

Spring WebFlux automatically configures Reactor Netty as the default server but if we want to override the default configuration, we can simply do that by defining them under server prefix.

```
server:
  port: 9000
  http2:
    enabled: true
```

We can also define the other properties in the same way that start with the `server` prefix by overriding the default server configuration.

Conclusion

Spring WebFlux or Reactive non-blocking applications usually do not make the applications run faster. The essential benefit it serves is the ability to scale an application with a small, fixed number of threads and lesser memory requirements while at the same time making the best use of the available processing power. It often makes a service more resilient under load as they can scale predictably.

WebFlux is a good fit for highly concurrent applications. Applications which would be able to process a huge number of requests with as few resources as possible. WebFlux is also relevant for applications that need scalability or to stream request data in real time. While implementing a micro-service in WebFlux we must take into account that the entire flow uses reactive and asynchronous programming and none of the operations are blocking in nature.

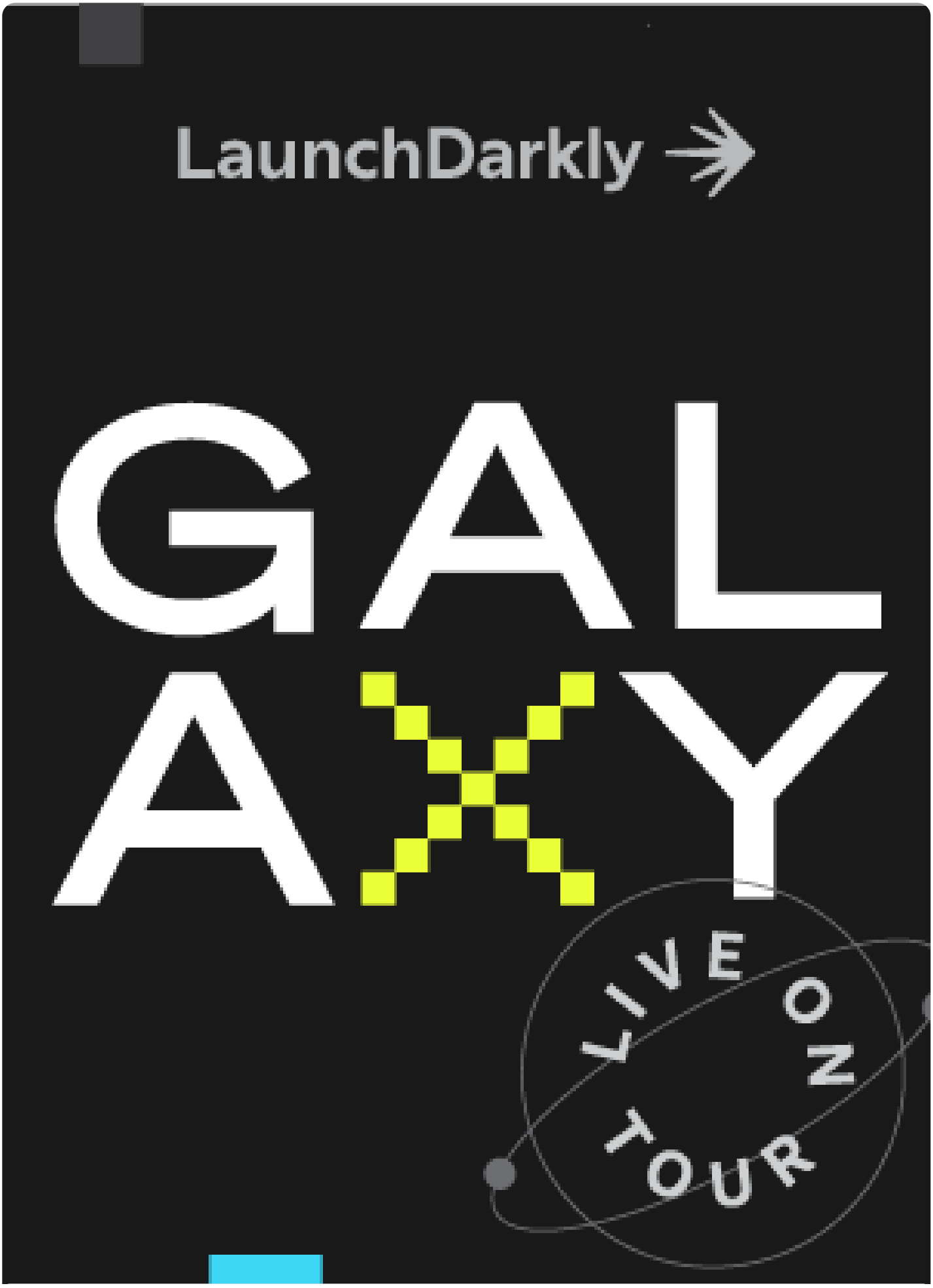
If you want to learn about how to build a client to a reactive server, have a look at our `WebClient` article.

You can refer to all the source code used in the article on Github.

Written By:

Arpendu Kumar Garai

I am a full-Stack developer with deep knowledge in Java, Microservices, Cloud Computing, Big Data, MERN, Javascript, Golang, and its relative frameworks. Besides coding and programming, I am a big foodie, love cooking, and love to travel.


LaunchDarkly
GALAXY
LIVE TOUR ON


more than 150 reviews on Amazon and Goodreads.

Save $10 by joining the Simplify! newsletter.

Get Your Hands Dirty

Recent Posts

Getting started with Spring Security and Spring Boot


Ranjani Harish | February 28, 2023

Spring Security is a framework that helps secure enterprise applications. By integrating with Spring MVC, Spring Webflux or Spring Boot, we can create a powerful and highly customizable authentication and access-control framework.

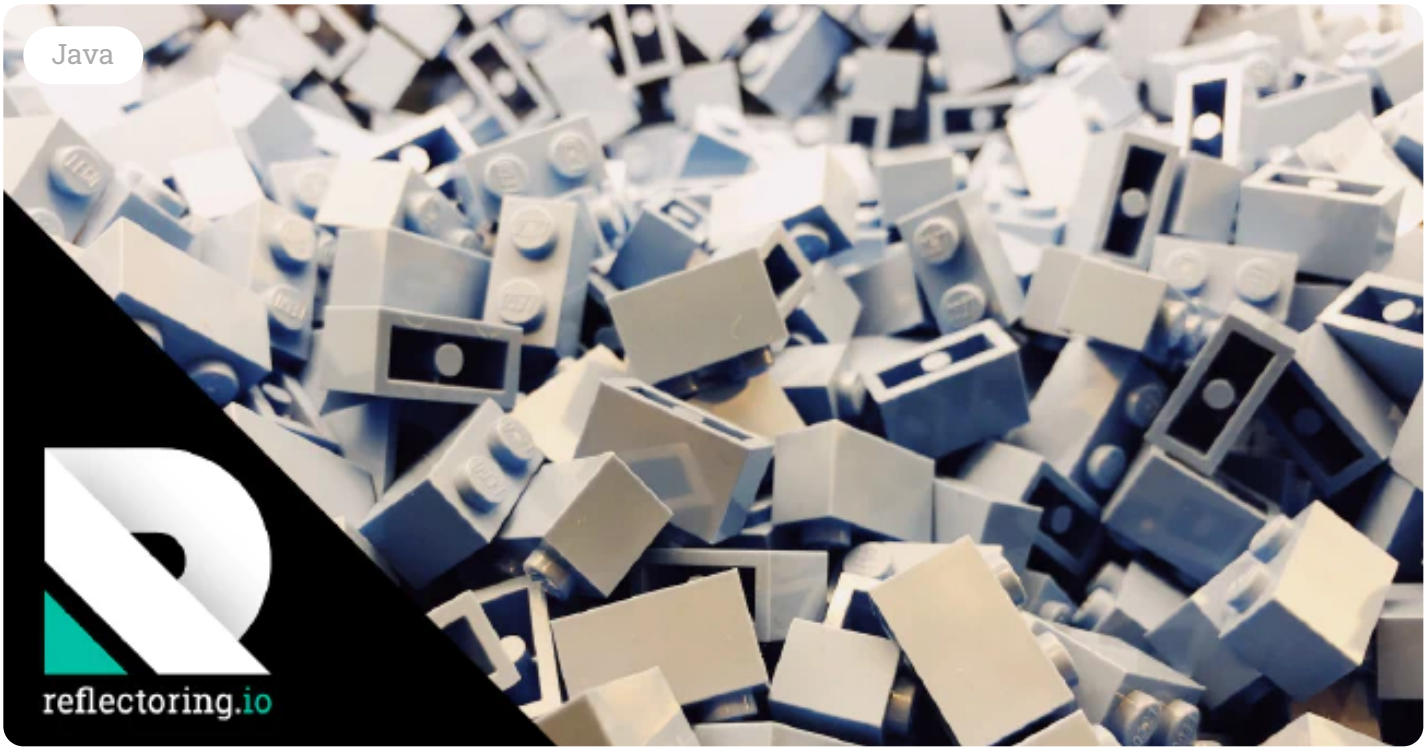
Read More

Demystifying Transactions and Exceptions with Spring

Arpendu Kumar Garai | January 31, 2023

One of the most convincing justifications for using the Spring Framework is its extensive transaction support. For transaction management, the Spring Framework offers a stable abstraction.

Read More



JUnit 5 Parameterized Tests

Pralhad Hadimani | January 29, 2023

If you're reading this article, it means you're already well-versed with JUnit. Let me give you a summary of JUnit - In software development, we developers write code which does something simple as designing a person's profile or as complex as making a payment (in a banking system).

Read More

Content

Products

Contribute

Author Workflow

Author Payment

Reflectoring

About

Atom Feed

Advertise

Book me

Privacy

Built upon Geeky Hugo theme by Statichunt