Introduction to Spring WebFlux and Reactive API

This is a **Spring WebFlux Tutorial** that covers an in-depth **Introduction to Spring WebFlux and Reactive API**, their benefits and major components. Also, difference between Spring WebFlux and Spring MVC.

Spring WebFlux is a framework for building reactive and non-blocking Web Applications. It supports reactive streams back pressure and works nicely with Java Streams and Java functional programming paradigm.

First, we will have a quick overview of the Spring WebFlux project and understand the all the important concepts of Reactive Programming. Next, we will have an overview of Project Reactor (https://projectreactor.io/) API and some of its major components. After that, we will introduce Flux and Mono – two highly used publishers along with their differences. Lastly, we will discuss difference between Spring MVC and Spring WebFlux.

What is Spring WebFlux?

As I have said above, Spring WebFlux is Spring's framework for building web applications. Unlike the servlet based web applications, the **WebFlux applications use reactive streams to facilitate non-blocking communication between a publisher and a subscriber**.

In order to achieve concurrency with the blocking components, we need to create and use Threads. Doing so, we also need to take care of the threads life cycles and thread orchestration. However, such thread based model often lead to complexity and it consumes a good amount of resources. On the other hand, reactive non-blocking models like WebFlux achieve the concurrency with fewer threads and scale with fewer resources.

The initial motivation for WebFlux came from Servlet 3.1 specifications. To explain, Servlet 3.1 brought in Non-blocking IO (NIO) support on top of Servlet 3.0's asynchronous processing. Given that, with Servlet 3.1, we can read and process the request and write response in a non-blocking manner. However, the servlet 3.1 changes a lot of semantics of using servlet APIs. Thus, Spring created WebFlux which is a lot similar to

Spring MVC in terms of components and annotations support. In the later sections, we will understand the differences and similarities between these two web frameworks.

What is Reactive?

Reactive is a model of programming that is built upon the concept of *change* and *reaction*. In other words, in reactive programming components execute their functionalities in response to a change in other components. It also means, a subscriber won't react unless there is a change published by the publisher. Thus, the subscribing component doesn't need to wait and they can continue doing other work. That's exactly what non-blocking components are.

In the project reactor, the base of reactive components are reactive streams that maintain back pressure between publisher and subscriber. In order to understand the concept of back pressure, consider a blocking interaction between a server and client. Where, a server cannot publish next set of responses until the client has fully consumed the previous response. On the other hand, in asynchronous non-blocking communication a server can produce a very large number of events that a client can process. Thus the data that is being transferred sits longer in-memory and consume expensive resources.

In order to solve this problem, reactive streams maintains a consistent back pressure. Given that, a server won't produce more events unless the client is ready. In other words, it allows clients to control the rate of events publisher publishes. **For a fast clients the same servers will produce data much quicker than to slower ones**.

Blocking Vs Non-Blocking Request Processing

Let's discuss what is the different between Blocking or Synchronous Request Processing and Non-Blocking or Asynchronous Request Processing in terms of a web application.

Blocking (Synchronous) Request Processing

When a request comes, the container invokes respective servlet by assigning a servlet thread. In order to process the request, the servlet thread may create several worker threads. Such worker threads collectively work together to fulfil the request. To do so, they may

perform some computation or they may interact with external resources like database or other services etc. While the workers are busy processing the request, the servlet thread remain blocked.

However, the server has only a finite number of threads which can act as Servlet threads. Thus, it puts a limit on the number of requests an application can process concurrently.

Non-Blocking (Asynchronous) Request Processing

On the other hand, in case of a non-blocking request processing there is no thread in the waiting or blocking condition. The reactive programming model is based on observable streams and callback functions. Thus, when response or a part of the response is ready the respective subscribers receive a callback. That means, the servlet thread can invoke various workers and then it becomes free to process another requests.

Because of this, the underlying server can have a very small number of threads in pool and the application can still process a large number requests.

WebFlux Reactive API

In the beginning, when you are new to the reactive programming model, the WebFlux and Reactive API and associated terminology may sound tricky. That is why, we will cover some API basics in this section.

Remember, reactive programming is based on a communication by means of events. Thus it needs, a publisher, a subscriber, a subscription between them and a mechanism to process the subscription. This is why, the reactive API defines 4 main components – *Publisher*, *Subscriber*, *Subscription*, and *Processor*.

Publisher

A publisher provides a finite or potentially infinite sequence of events to its subscribers. In order to maintain back pressure, it emits events only when the respective subscriber needs it. The publisher has only one method – *subscribe(subscriber)*. Subscribers invoke this method to subscribe to the Publisher. It is important to note that, a Publisher can have multiple Subscribers.

```
public interface Publisher<T> {
    void subscribe(Subscriber<? super T> var1);
}
```

Subscriber

A subscriber is the receiver and controller of this reactive communication. This is because, publisher doesn't send an event until subscriber demands it. The Subscriber interface of reactive streams looks like this.

```
public interface Subscriber<T> {
    void onSubscribe(Subscription var1);
    void onNext(T var1);
    void onError(Throwable var1);
    void onComplete();
}
```

When a subscriber subscribes to a Publisher, the call back method – *onSubscribe(subscription)* is invoked.

- **Demand** – In order to receive notifications from the publisher a subscriber must call *Subscription#request(long)* method by specifying number of events it want to process.
- **Receive** – After that, the subscriber receives events through one ore more invocations of the call back method – *onNext(object)*. Where, the number of invocations are less than or equal to the requested number of requests.
- **Demand More** – Note that, the subscriber can demand more events by invoking *Subscription#request(long)* a multiple times.

- **Error** – In case of an error, the subscriber receives exception details through the *onError(Throwable)* method. As this is a terminal state, publisher won't send more events, even if the subscriber demands more.
- **Finish** – Finally, at the end of the event sequence, the subscriber receives callback to its *onComplete()* method. After this, subscriber won't receive any events, even if it invokes *Subscription#request(long)*.

Subscription

A Subscription is an active state of contract between a publisher and a subscriber. It represents a one-to-one life cycle between publisher and subscriber. That means, if a publisher has multiple subscribers there will be multiple Subscription instances – one for each subscriber.

```
public interface Subscription {
    void request(long var1);
    void cancel();
}
```

A subscriber uses Subscription instance to signal demand for events using *request(long)* or cancel an existing demand by invoking *cancel()*.

Processor

Finally, **a Processor represents the processing stage of both publisher and subscriber.** Note that, it is the processors responsibility to ensure the contact between publisher and subscriber is followed.

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

As can be seen in the snippet, the Processor extends both Subscriber and Publisher and it is responsible for providing implementations for them. As per the specifications, if there is an exception, processor must call *onError()* method on the subscriber. However, if a processor is able to recover from the exception, it must call *onComplete()* on the subscriber.

Understand Mono and Flux

Read this section to **understand the difference between Flux and Mono interfaces** of the reactor project. While working with Spring WebFlux, you will heavily use these two interfaces.

Mono

A Mono is a reactor streams publisher that publishes a single element. Given that, Mono will signal *onNext()* to deliver the event and then calls onComplete() to signal the termination. However, in case of an error it will signal *onError()* without sending any event.

For example, we can create a Mono instance by using *just()* method.

```
Mono<String> colorPublisher = Mono.just("RED");
```

Or, simply create an empty Mono using *empty()* method.

```
Mono<String> emptyPublisher = Mono.empty();
```

Flux

On the other hand, **a Flux is a reactor streams publisher that publishes 0 to N elements**. That means, a Flux will always emit 0 or up to infinity elements, or send an error signal if something goes wrong.

For example, we can create a Flux of finite elements, using *just()* method.

```
Flux<String> colorsPublisher = Flux.just("RED", "BLUE", "ORANGE");
```

Or, an empty Flux using empty() method.

```
Flux<String> emptyPublisher = Flux.empty();
```

Alternatively, we can also create a Flux instance by concatenating multiple Flux or Mono instances.

```
Flux<String> colorsPublisher = Flux.concat(
        Mono.just("RED"), Mono.just("BLUE"), Mono.just("ORANGE")
);
```

Spring WebFlux vs Spring MVC

As it is mentioned above Spring WebFlux is a Web framework that is based on reactive model of programming. While, Spring MVC is a Web Framework, which is based on imperative blocking programming model. Both of these frameworks live side by side and they will continue being that way. Most importantly, Spring WebFlux supports all of the Spring MVC annotations (e.g. @Controller) as well as basic **Spring annotations based DI and IoC (/spring-dependency-injection-inversion-control/)**.

The most commonly used servers like Tomcat and Jetty can run both Spring MVC and Spring WebFlux applications. However, it is important to understand that Spring MVC can leverage Spring Async to incorporate asynchronous processing. However, it is based on Servlet 3.0 Specifications, in which I/O operations still happen in blocking manner. That means, a server can process request asynchronously, however its communication with client will always be blocking.

On the other hand Spring WebFlux is based on Servlet 3.1 non-blocking IO. As mentioned in previous sections, Servlet 3.1 specifications support non-blocking I/O. Thus Spring WebFlux applications are fully non-blocking in nature.

Because of the thread blocking nature of Spring MVC applications, the servlet containers prepares a large thread pool to process different requests concurrently. On the other hand, for a Spring WebFlux applications the containers have a small fixed size thread pools. This is because of the non-blocking nature of Spring WebFlux applications.