Day 1/90

Date : 18-April-2023

Introduction

Problem with Thread Request (traditional thread/request)

The problem with this approach is that, one single thread is allocated for a request. Let's say we want to make a db call. Get the cells, and do some operations on those cells, or write the data cells on the file system.

Db calls, File system, etc ... are I/O task (Input/Output). These tasks are time consuming. During execution, the thread will be blocked in will stay in waiting state.

> Each thread is assigned to 1MB of stack memory. If you are expecting 500 concurrent request, you'd need 500MB of ram your ram will be used for the request alone without including the memory needed for the actual work!!!

IO Models

There are 4 types of IO models :

1. **Synchronous + blocking** : this is the model that we've been using by default. We make a request, and wait for that request to give a response.

   REAL WORLD ANALOGY : I called an insurance company, the automated phone service tells me to click this button, that button, until I reach the manager's office. They say manager is in meeting so please wait for 5 minutes. I'm going to wait for the meeting to end and then I can talk to the manager. Here, I'm the thread and Insurance company is the remote server. In this whole time, they've blocked, and I'm waiting for the response.

2. **Asynchronous** : when we make request, we don't wait for the response instead, we leave the responsibility to another thread and busy ourselves (requested thread) with another logic. It sounds like a better solution but it's a bit difficult to deal with the threads.

   From previous analogy : I give the phone to my friend and I go get myself busy with something else. My friend will do the work.

3. **Non-blocking** (event-driven model) : It is more difficult than asynchronous 🙄😂😵

   From previous example : Once I call the company, press all those options, It will make a note of my phone number and will call you later.

4. **Non-blocking + Asynchronous :** mixture of 2 and 3.

   From previous example : My friend calls the company, and he don't have to wait for the response either. They'll call him.

The levels of I/O Models is :

easy < medium < hard < very hard obviously.

This is where Reactive programming helps us. It provides us nice abstraction that, we write code like synchronous but behind the scenes it follows non-blocking + async model. All the complexities are taken care of for us.

Reactive Streams

In a microservice pattern, we have lots and lots of services. For instance, when a user wants to buy a product. The *UserService* has to make a request to *OrderService*, this service has to make a request to *PaymentService*. Also, payment service needs to know to the user-id to ship the product using *ShippingService*.

As you can see, there are a lot of I/O calls done in a sequential way bcuz we cannot call the PaymentService if we do not know the order-id. Also we need to do that for every user.

I/O-as-a-service

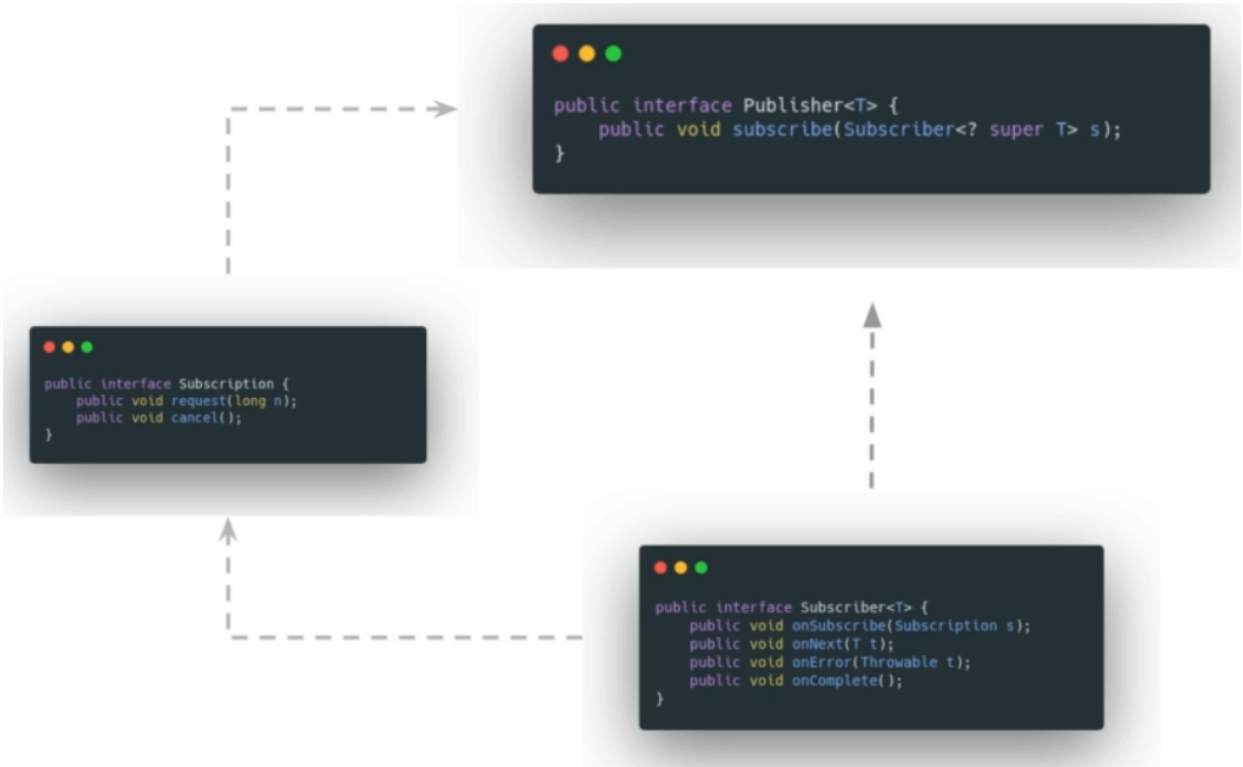
From previous example, our friend will act as a service. He will take requests and make calls to company. Since it is a non-blocking + async model, he won't wait for the response and instead, he'll take infinitely another request.

From his perspective, it's unbounded never ending stream of requests. He gets the requests continuously and he need to handle them efficiently.

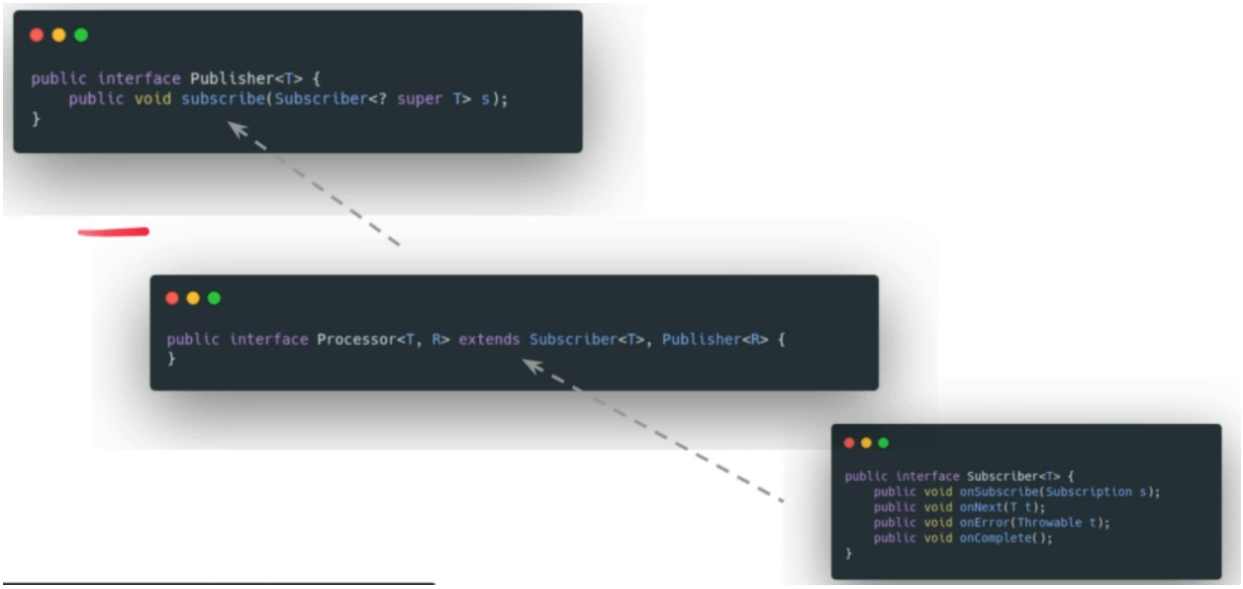
So for it to be solved, peoples like Netflix, Twitter, LinkedIn, they all came up with a specification for processing potentially unbounded streaming data. It is called **Reactive Stream Specification** like JPA.

This follows *Observer Pattern*. Like for instance twitter, we follow a highly skilled developer. We also have some followers. When that skilled developer posts a tweet, we may or may not like or comment or share on it. Our followers will also see that tweet.

The person on top (developer) is the publisher, and the followers are observer.



for our example, where we are both a follower and publisher, (publisher and subscriber) :



Reactive programming

With all the introduction, Reactive programming is a subset of event-driven, asynchronous programming in which you register a set of callbacks (listeners), where data goes through the pipeline (chain).

We can see reactive programming as these 3 pillars :

- Asynchronous data processing
- Non-blocking
- Functional style / Declarative

We have these implementations for reactive programming :

- Akka
- rxJava2
- Reactor

Publisher & Subscriber Communication

publisher subscriber communication steps :

- step 1 : Subscriber wants to connect

  There will be 2 instances, Publisher and subscriber. Subscriber wants to get updates from publisher.

  ```
  -----------             ------------
  |publisher| <-----------|subscriber|
  -----------             ------------
  ```

- step 2 : Publisher calls onSubscriber

When the publisher accepts the subscriber, he hands over the subscription object to the subscriber.

```
-----------              ------------
|publisher| <-----------|subscriber|
----------- ====> [ ]   ------------
```

- step 3 : Subscription

```
-----------                                    ------------
|publisher|       --------------------         |subscriber|
----------- ====> |SubscriptionObject| <====   ------------
                  --------------------
```

- step 4 : Publisher pushes data via onNet

  When the subscriber requests let's say 4 for an object from the publisher using subscription object, the publisher will respond with 4 items and calling `onNext()` method 4 times.

```
-----------                                          ------------
|publisher|       -------o--o--o------               |subscriber|
----------- ====> |SubscriptionObject| <==3requests==  ------------
                  --------------------
```

- step 5 : onComplete

  When the publisher has already emitted all the items to the subscriber, the publisher can call the `onComplete` method to notify subscriber.

```
-----------                                  ------------
|publisher|       ----------✅----------      |subscriber|
-----------                                  ------------
```

- step 6 : onError

  If an error occurs, then publisher will notify the subscriber with the error details.

```
-----------                                  ------------
|publisher|       ----------❌----------      |subscriber|
-----------                                  ------------
```

Terminologies

- Publisher
  o Source
  o Observable
  o Upstream
  o Producer

- Subscriber
  o Sink
  o Observer
  o Downstream
  o Consumer

Done

- [ ] Day 1/90 ===> 0000000

Day 2/90

Date : 19-April-2023

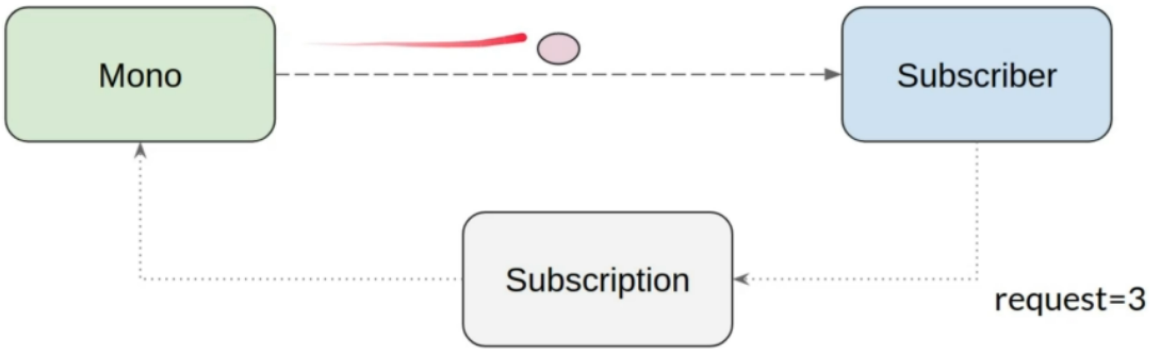
Mono

Project Reactor - Introduction

As we learned earlier, Reactive Stream is the specification, and the Reactor is the library, the implementation (something like hibernate).

Publisher Interface : It is one of the interfaces t that reactor provides 2 different implementation :

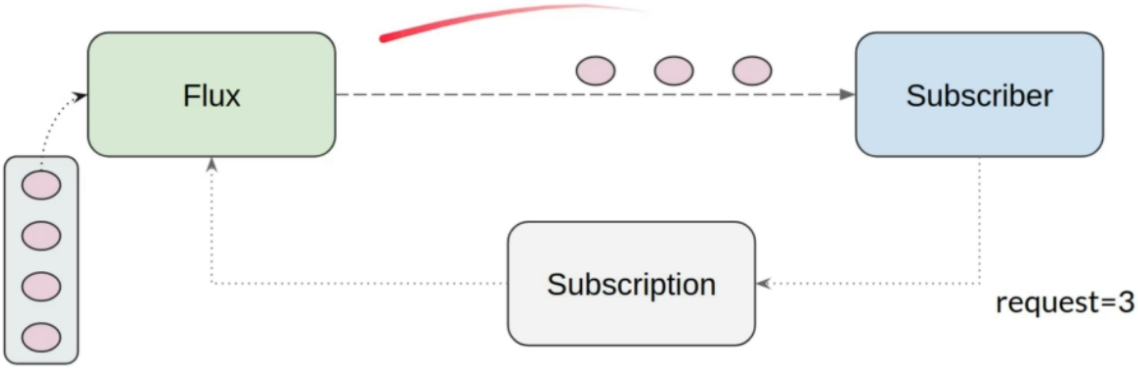
- *Mono* : it can emit 0 or 1 item, followed by an onComplete / or onError signal.



  subscriber can request for 15 calls, but publisher will give just 1 and call the onComplete method to close the call.

  It can emit 0 also meaning publisher has the freedom to emit or to not emit.

- *Flux* : it can emit 0 or N item, followed by an onComplete / or onError signal.



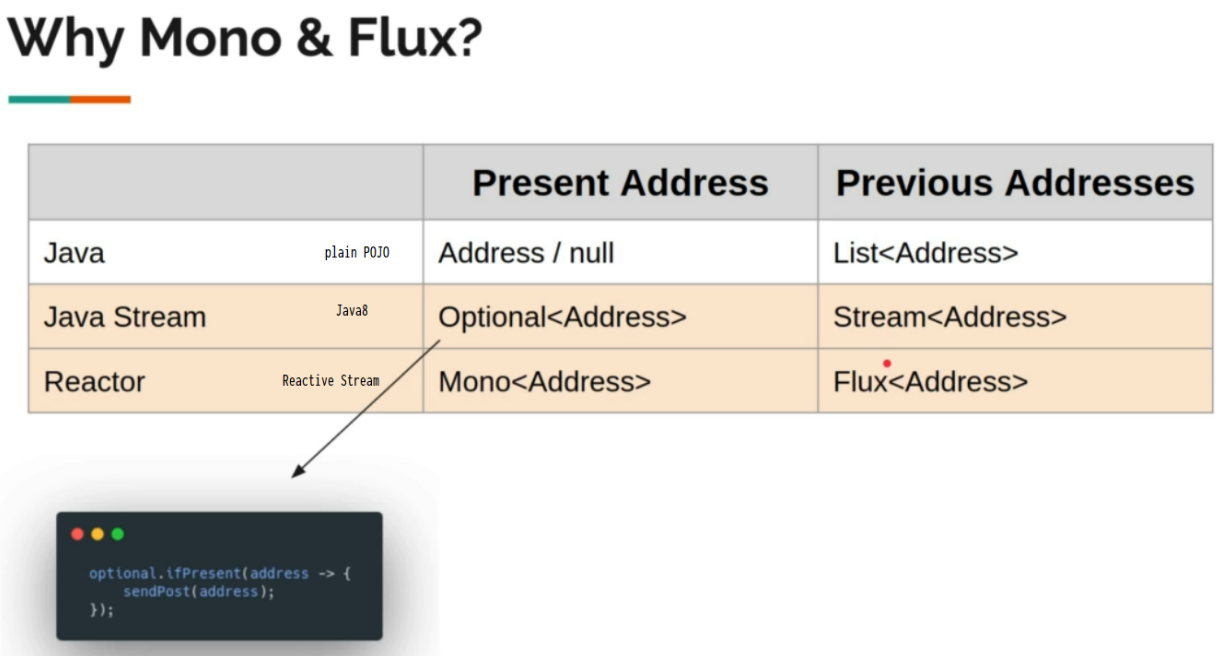
> Flux can be an infinite stream. Stream can be completed by onComplete, or canceled by the subscriber itself.

Why Mono & Flux?

Let's see an example.

Let's assume you want the count of the rows in a table in db. You'll use Mono bcuz you only want the count, single aggregated information for you.

If you want all the records from the table, you can use flux. You know for sure you can expect 0 record, 1 record, or N record.



Setup

you should add these dependencies :

```
<dependencies>
    <!-- version, is specified in dependency management -->
    <dependency>
        <groupId>io.projectreactor</groupId>
        <artifactId>reactor-core</artifactId>
    </dependency
```

```
    <!-- this is for generating test data -->
    <dependency>
        <groupId>com.github.javafaker</groupId>
        <artifactId>javafaker</artifactId>
        <version>1.0.2</version>
    </dependency

    <dependency>
        <groupId>io.projectreactor</groupId>
        <artifactId>reactor-test</artifactId>
        <scope>test</scope>
    </dependency
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.4.2</version>
        <scope>test</scope>
    </dependency
</dependencies>

<!-- we didn't specified the version of project reactor bcuz of this -->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.projectreactor</groupId>
            <artifactId>reactor-bom</artifactId>
            <version>2020.0.2</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Stream Lazy Behavior

```
public class Lec01Stream {
    public static void main(String[] args) {
        Stream<Integer> stream = Stream.of(1)
                .map(i -> {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                    return i * 2;
                });
        System.out.println("This will run without any delay bcuz of lazy behavior of
Stream.\n" + stream);

        long startTime = System.currentTimeMillis();
        System.out.println("\n\"This will run with delay :\"");
```

```
        stream.forEach(System.out::println);

        long afterTime = System.currentTimeMillis();
        LocalDateTime now = LocalDateTime.now();
        LocalDateTime afterIteration = now.plusSeconds(afterTime - startTime);

        System.out.println("Executed in : " + afterIteration.getSecond());
    }
}
```

The code uses the lazy evaluation feature of Java Streams. In general, streams are processed in a lazy manner, which means that intermediate operations (such as `map`) are not executed until a terminal operation (such as `forEach`) is called.

In this program, the first `System.out.println()` statement will execute immediately because it only accesses the Stream object and does not consume any elements from it. It just prints the address of the Stream object.

However, the `map` operation inside the Stream's pipeline is not executed until a terminal operation is called on the Stream. In this case, the terminal operation is the `forEach` method that triggers the processing of the Stream's pipeline.

Therefore, the second `System.out.println()` statement and the `forEach` loop are executed after a delay of one second, because of the `Thread.sleep(1000)` statement inside the `map` operation. The `forEach` loop consumes the only element present in the Stream (i.e., 1) and applies the `map` function to it, producing the output value of 2.

> In stream, unless you connect it to a terminal operator, nothing works.

Mono - Just

> number 1 rule in reactive programming :
>
> **Nothing works until you subscribe.**

```
public class Lec02MonoJust {
    public static void main(String[] args) {
        //publisher
        Mono<Integer> mono = Mono.just(1);//easiest way to create mono
        System.out.println(mono); //this will just print the toString

        //now, this guys a subscriber
        mono.subscribe(i -> System.out.println("hey publisher (mono), I'm a
subscriber, I want to Receive whatever the object that you will give me : " + i));
    }
}
```

In the program, a Mono is created using the `Mono.just()` method with an integer value of 1. This creates a Publisher that emits a single item (in this case, the integer value of 1) and then completes.

The `System.out.println(mono)` line just prints out the string representation of the Mono object, which doesn't actually trigger any subscription or emission of values.

Next, a subscriber is created by calling the `subscribe()` method on the Mono instance. The subscriber is represented by a lambda function that takes an integer value and prints out a message indicating that it has received the value.

When the `subscribe()` method is called, it triggers the Publisher to emit its single item (the integer value of 1) and send it to the subscriber's lambda function for processing. The lambda function prints out the message, completing the reactive stream.

Done

- [ ] Day 2/90 ===> 00000000000