Reactive Programming in Modern Java using Project Reactor

Dilip

About Me

- Dilip
- Building Software’s since 2008
- Teaching in **UDEMY** Since 2016

What's Covered ?

- Need for Reactive Programming
- What is Reactive Programming
- Reactive Streams
- Write Reactive Programming using **Project Reactor**
- Build Nonblocking rest client using **Spring Webclient**
- JUnit test cases using **JUnit5**

Targeted Audience

- Any developer interested in learning Reactive Programming
- Any developer who has the requirement to write fast performing code under heavy load
- Any developer who is curious to understand the internals of Reactive Programming

Source Code

Thank You!

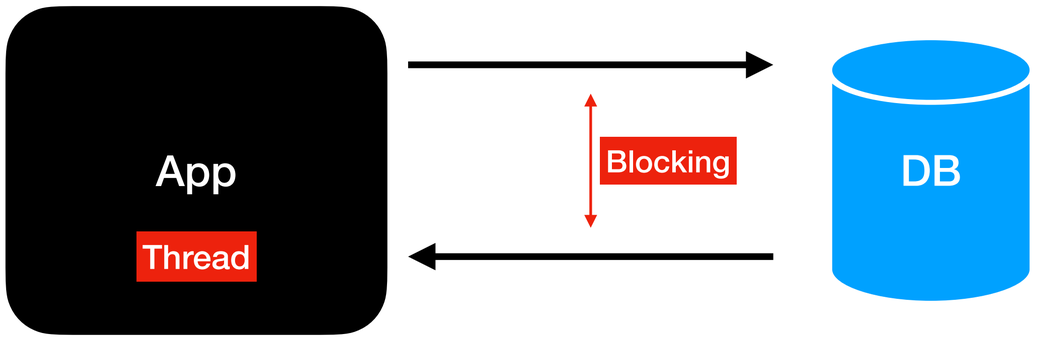
Prerequisites

Prerequisites

- Java 11 or Higher is needed
- Prior Java Experience is a must
- Experience writing **Lambdas**, **Streams** and **Method References**
- Experience Writing JUnit tests
- **Intellij** or any other IDE

Why Reactive Programming ?

Traditional Programming



Synchronous or Blocking style of writing code

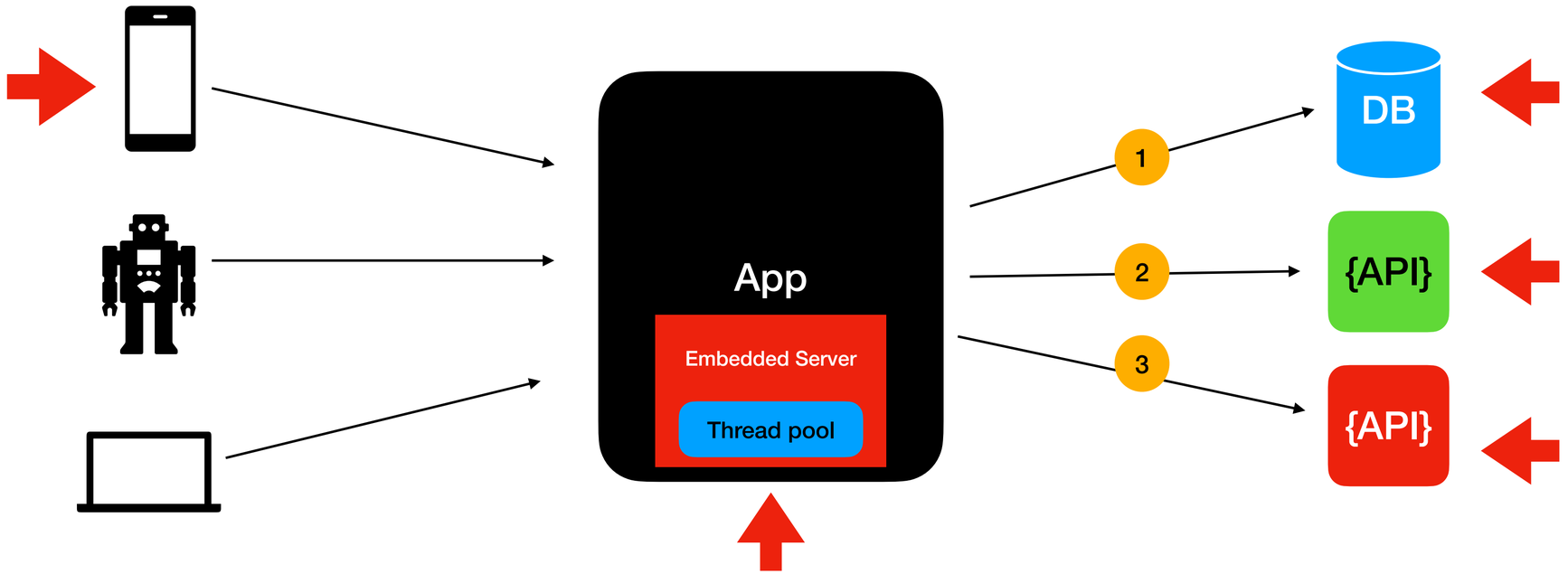
Blocking code won’t scale well to fulfill today’s customer needs

What has changed?

What has Changed ?

- Internet usages spiked up
  - Network interactions are pretty common
- **MicroServices Architectures** are pretty much everywhere
- Applications are deployed in cloud environments
- Response times are expected in **milliseconds**
- No downtime

Today’s Architecture (Backend RestFul API)



- Support 100000 users

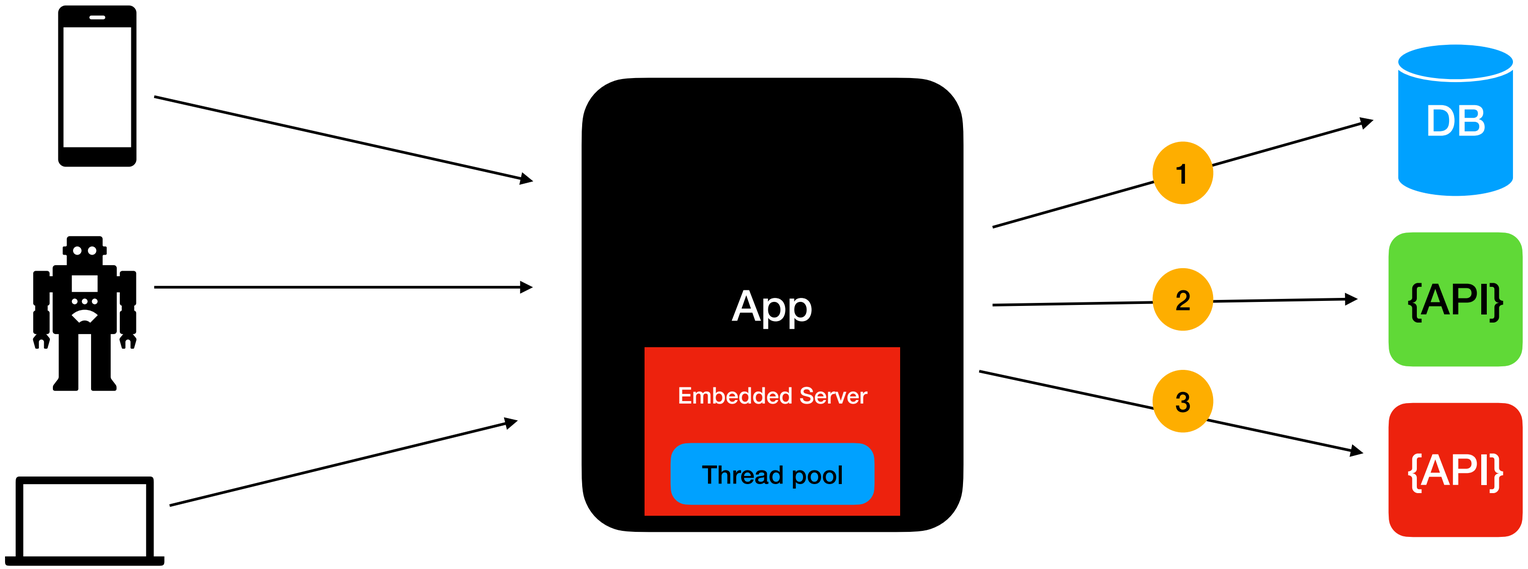
Latency = Summation of (DB + API + API) response times

- Cannot have a Thread pool size of 100000

Threads and its side effects

- Thread is an expensive resource
    - It takes up to 1 MB of heap space
- More threads leads to more heap space which leads to shortage of JVM memory for handling the request

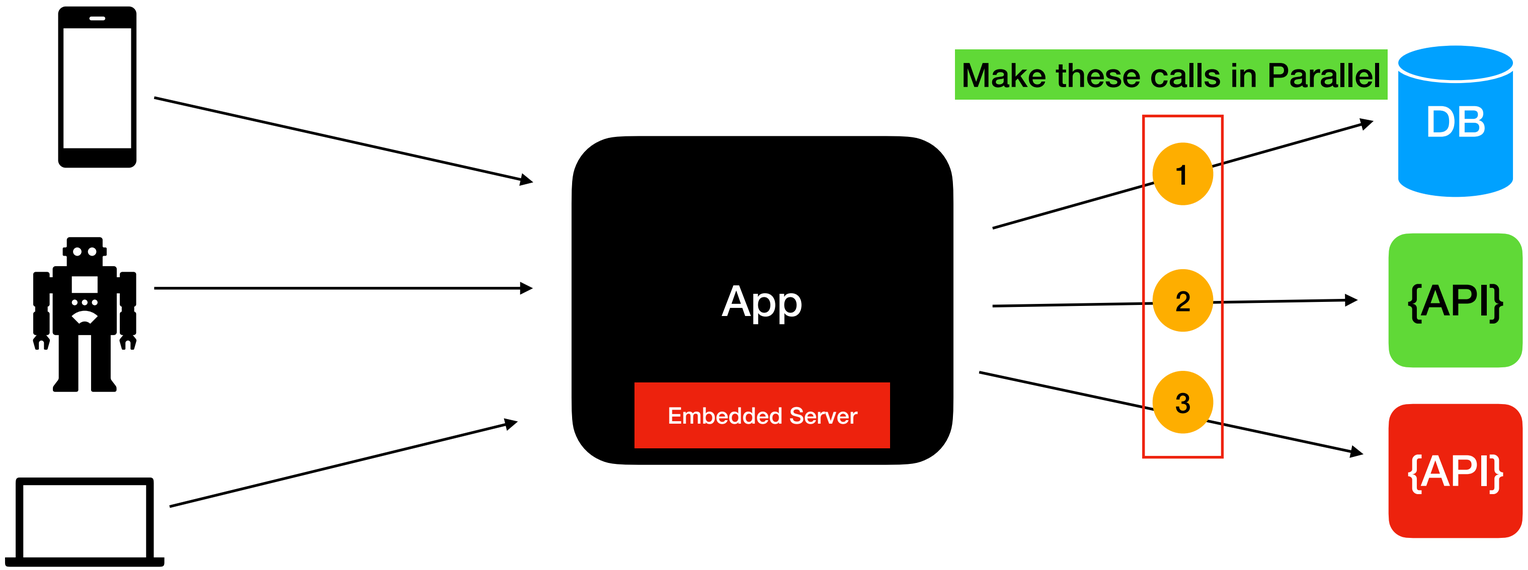
Today’s Architecture (Backend RestFul API)



- Blocking Code leads to In-efficient usage of Threads
- Wont scale to meet todays customer needs

Can we do better ?

Today’s Architecture (Backend RestFul API)



What are the tools/api that’s available in Java ?

Asynchronous/Concurrency APIs in Java

- CallBacks

- Future

Callbacks

Callbacks

- Asynchronous methods that accept a callback as a parameter and invokes it when the blocking call completes.
- Writing code with Callbacks are hard to compose and difficult to read and maintain
- **Callbackhell**

Future

Concurrency APIs in Java

Future

- Released in Java 5
- Write Asynchronous Code
- **Disadvantages:**
- No easy way to combine the result from multiple futures
- Future.get()
  - This is a blocking call

CompletableFuture

- Released in Java8
- Write Asynchronous code in a functional style
- Easy to compose/combine MultipleFutures
- **Disadvantages**:
  - Future that returns many elements
  - Eg., CompletableFuture<List<Result> will need to wait for the whole collection to built and readily available
  - CompletableFuture does not have a handle for infinite values

Flow API

- Release as part of Java 9

- This holds the contract for reactive streams but no implementation is available as part of JRE

Summary

Options in Java to solve the **Effective Thread Utilization** , **Latency** and **Handling heavy load** problem partially

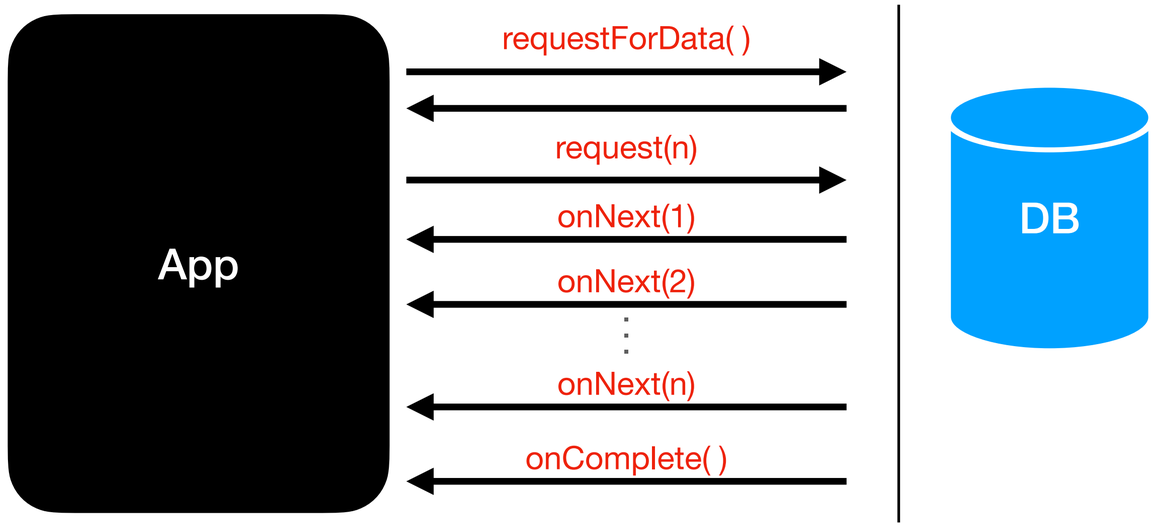
Are there better options available ?

What is Reactive Programming ?

What is Reactive Programming ?

- Reactive Programming is a new programming paradigm
- Asynchronous and non blocking
- Data flows as an **Event/Message** driven stream

Reactive Programming



- This is not a blocking call anymore

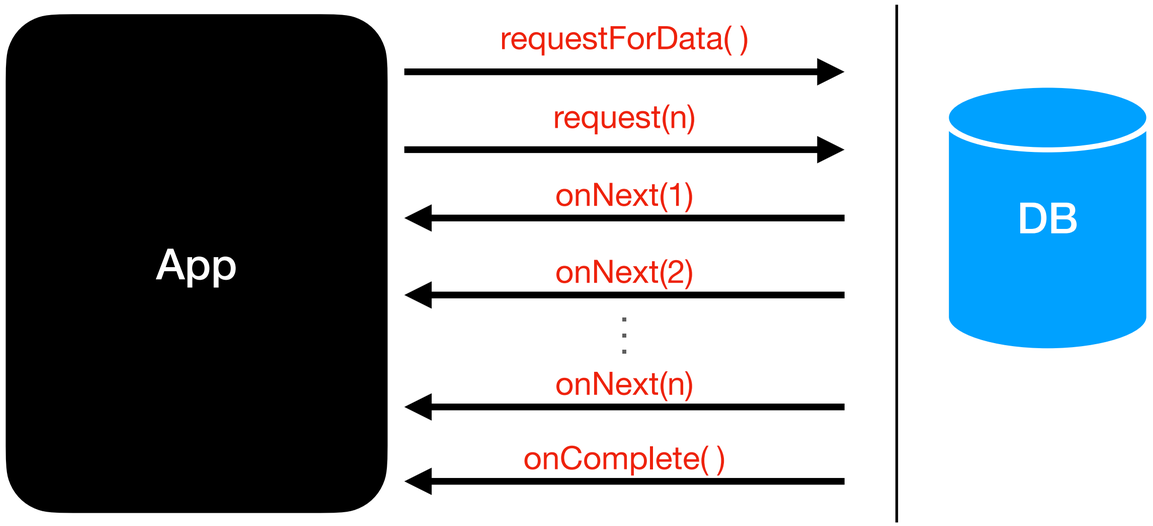
Push Based data streams model

- Calling thread is released to do useful work

What is Reactive Programming ?

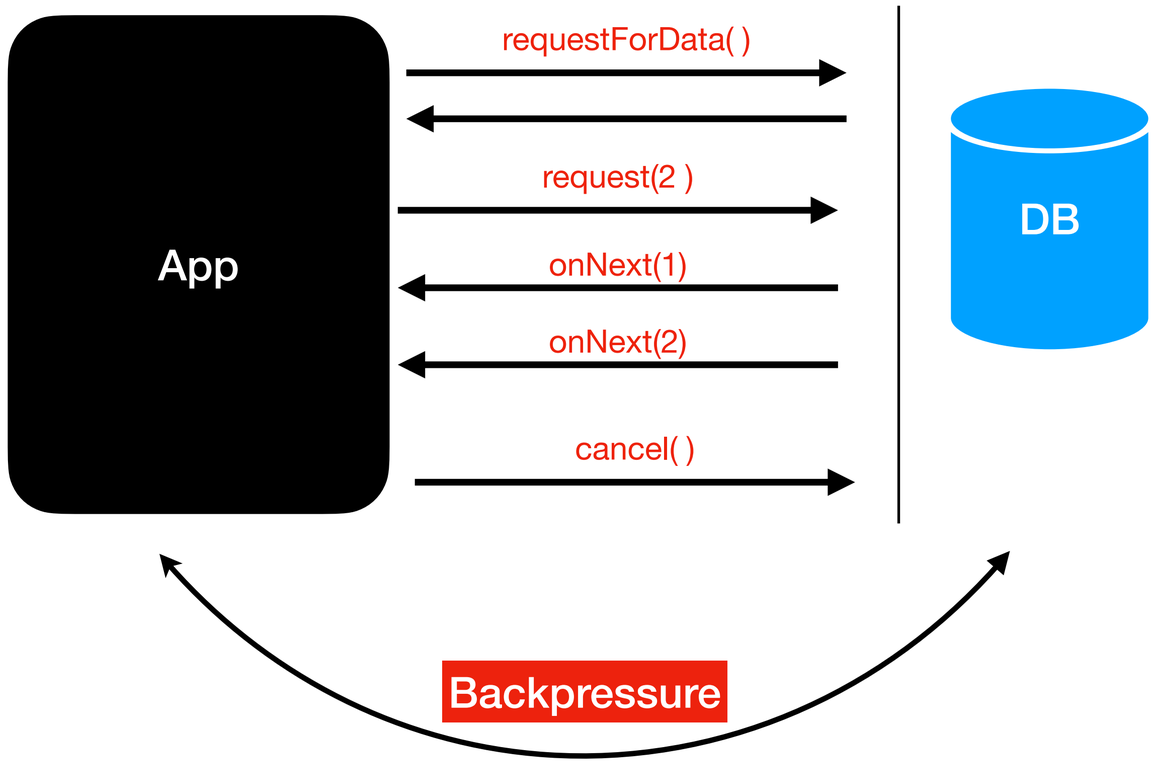
- Reactive Programming is a new programming paradigm
- Asynchronous and non blocking
- Data flows as an **Event/Message** driven stream
- Functional Style Code
- BackPressure on Data Streams

Backpressure



Overwhelm the app with more data

Backpressure



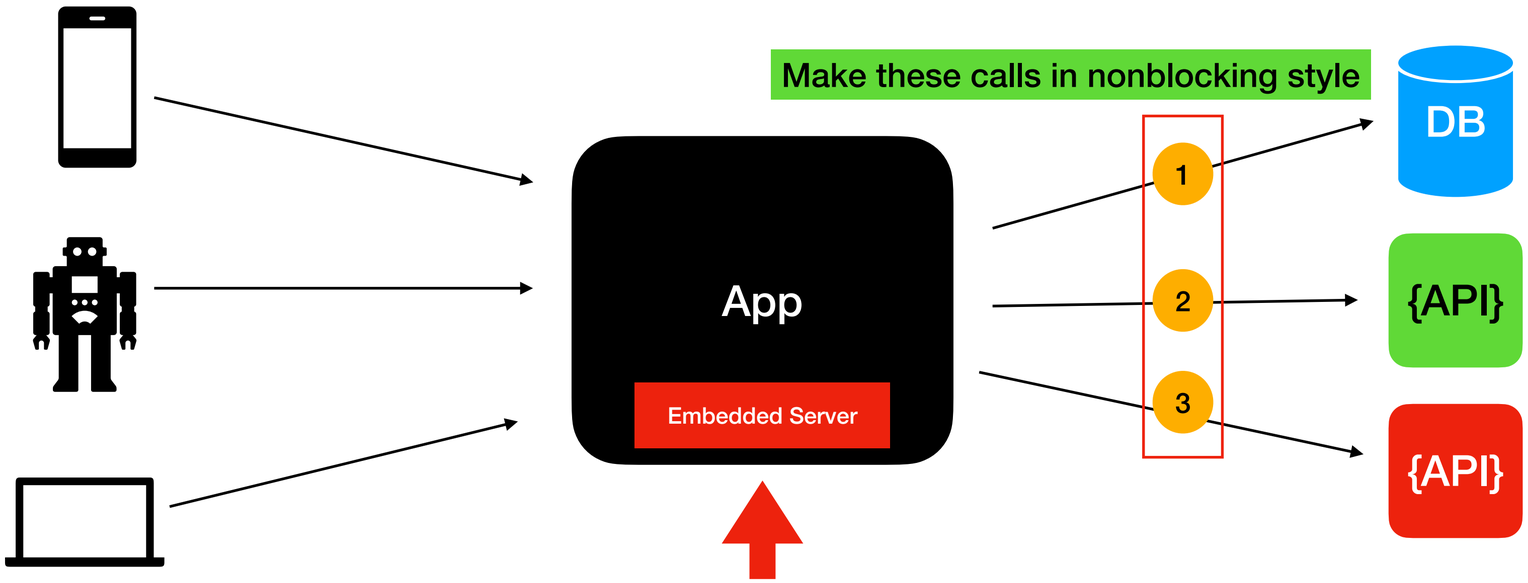
Push-based data flow model

Push-Pull based data flow model

When to use Reactive Programming ?

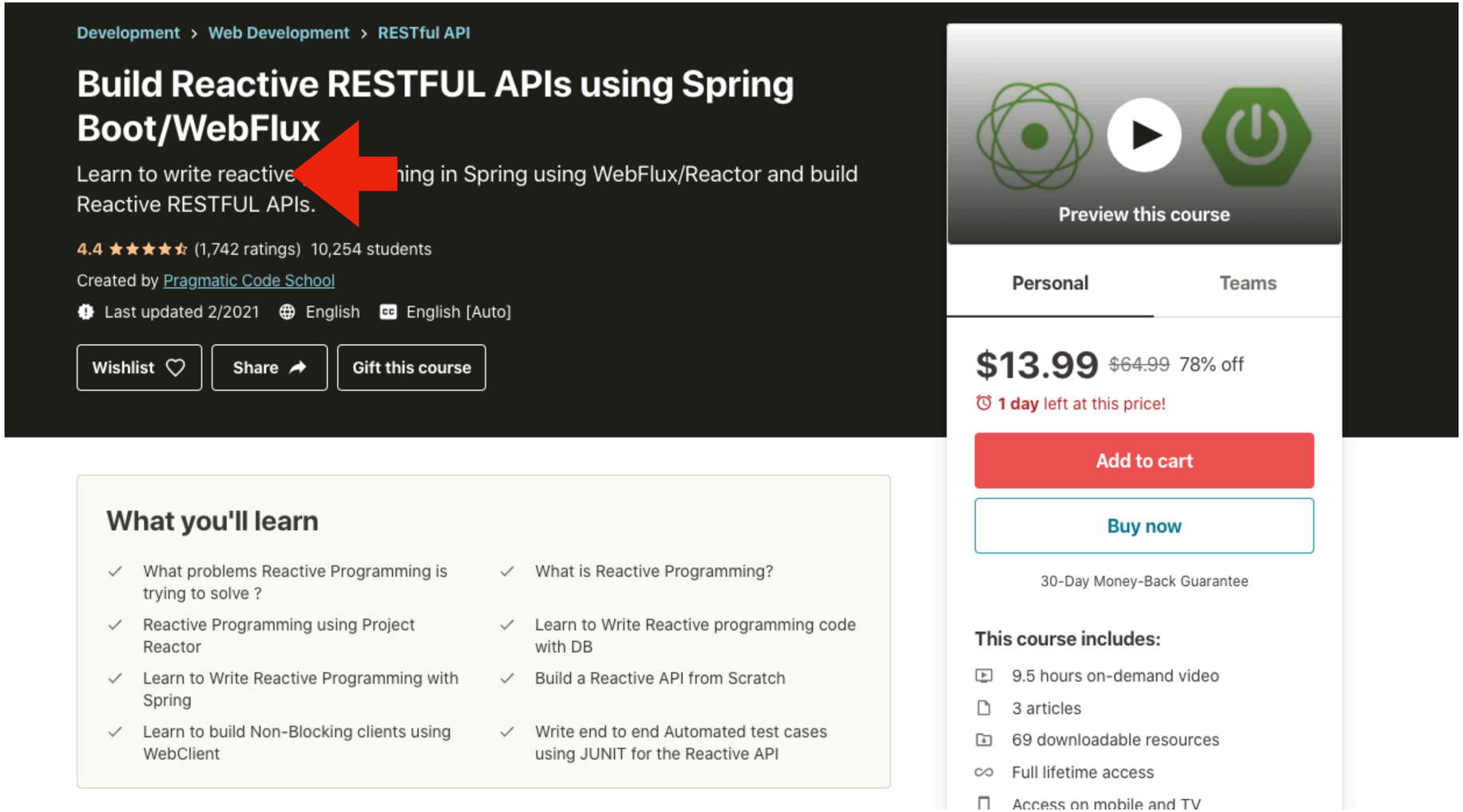
Use Reactive Programming when there is need to build and support app that can handle high load

Reactive App Architecture



- Handle request using non blocking style
  - Netty is a non blocking Server uses Event Loop Style
- Using Project Reactor for writing non blocking code
- Spring WebFlux uses the Netty and Project Reactor for building non blocking or reactive APIs

Spring WebFlux using Project Reactor



Reactive Streams

Reactive Streams are the foundation for Reactive programming.

Reactive Streams

- Reactive Streams Specification is created by engineers from multiple organizations:
  - Lightbend
  - Netflix
  - VmWare (Pivotal)

Reactive Streams Specification

- Reactive Streams Specification:
  - Publisher
  - Subscriber
  - Subscription
  - Processor

Publisher

```
public interface Publisher<T> {

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

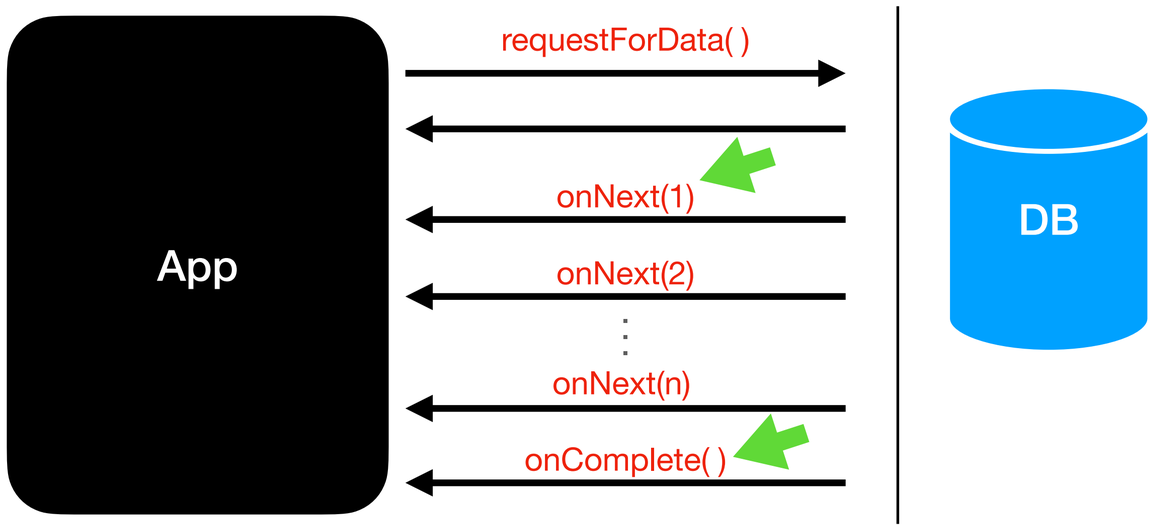
1

- Publisher represents the DataSource
  - Database
  - RemoteService etc.,

Subscriber

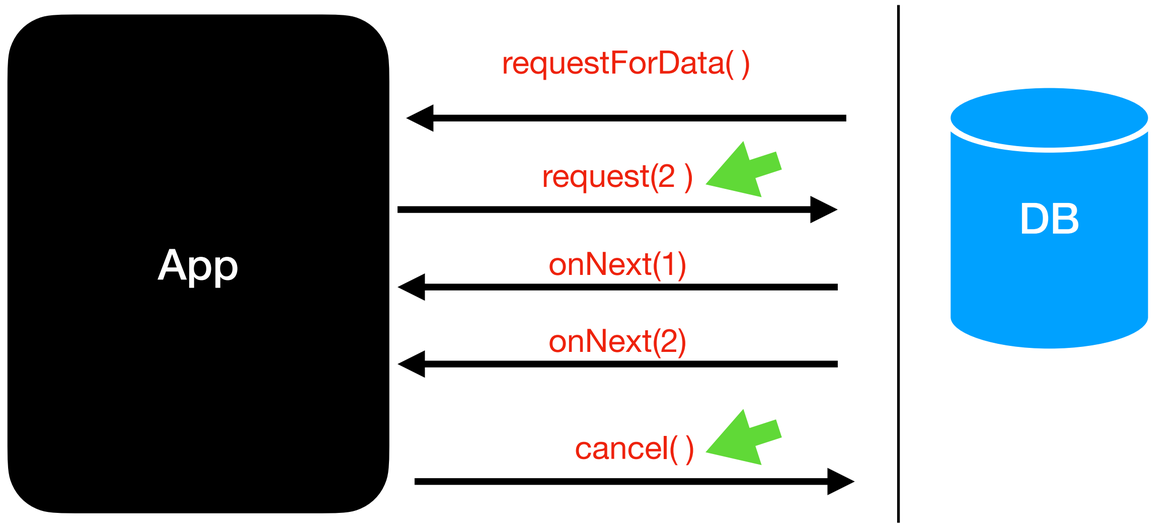
```
public interface Subscriber<T> {

    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

1 2 3 4



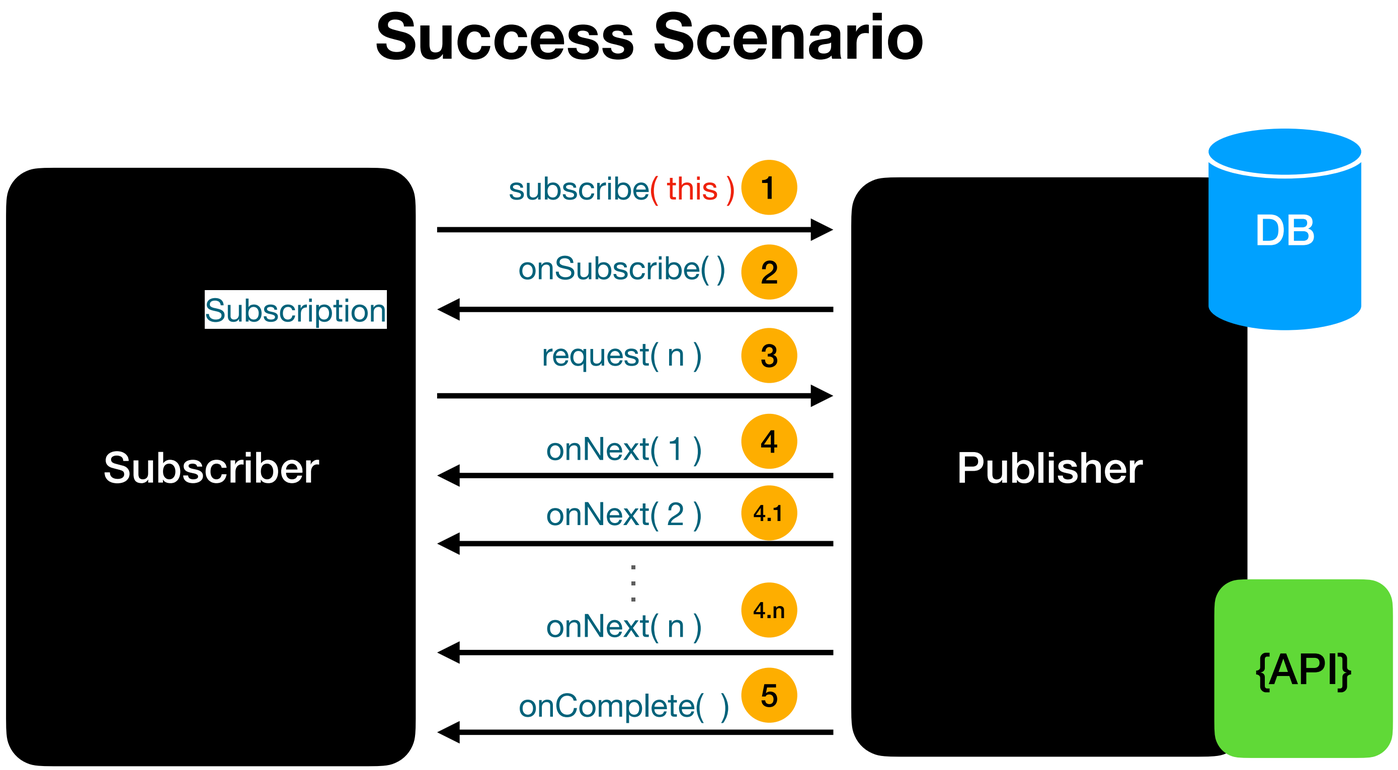
Subscription

```
public interface Subscription {

    public void request(long n);

    public void cancel();
}
```



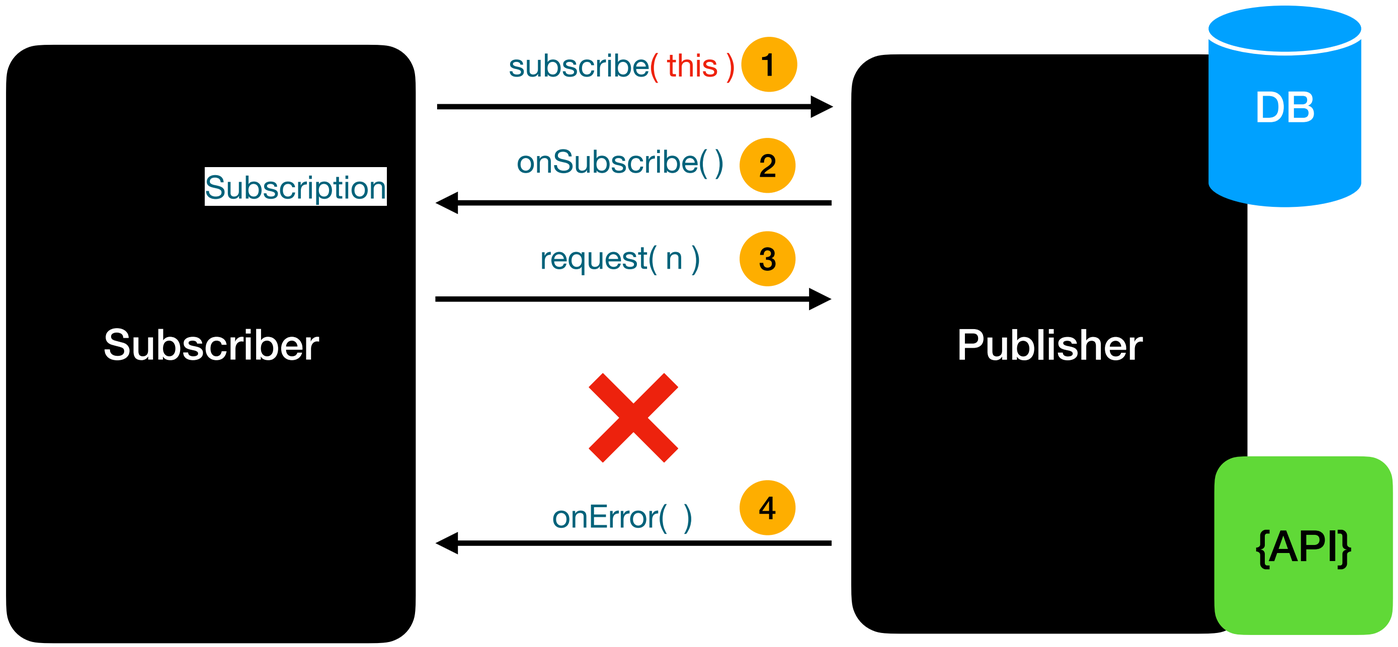
Subscription is the one which connects the app and datasource

Reactive Streams - How it works together ?



Reactive Streams - How it works together ?

Error/Exception Scenario



- Exceptions are treated like the data
- The Reactive Stream is dead when an exception is thrown

Processor

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

}
```

- Processor extends Subscriber and Publisher
  - Processor can behave as a Subscriber and Publisher
  - Not really used this on a day to day basis

Introduction to Project Reactor

Project Reactor

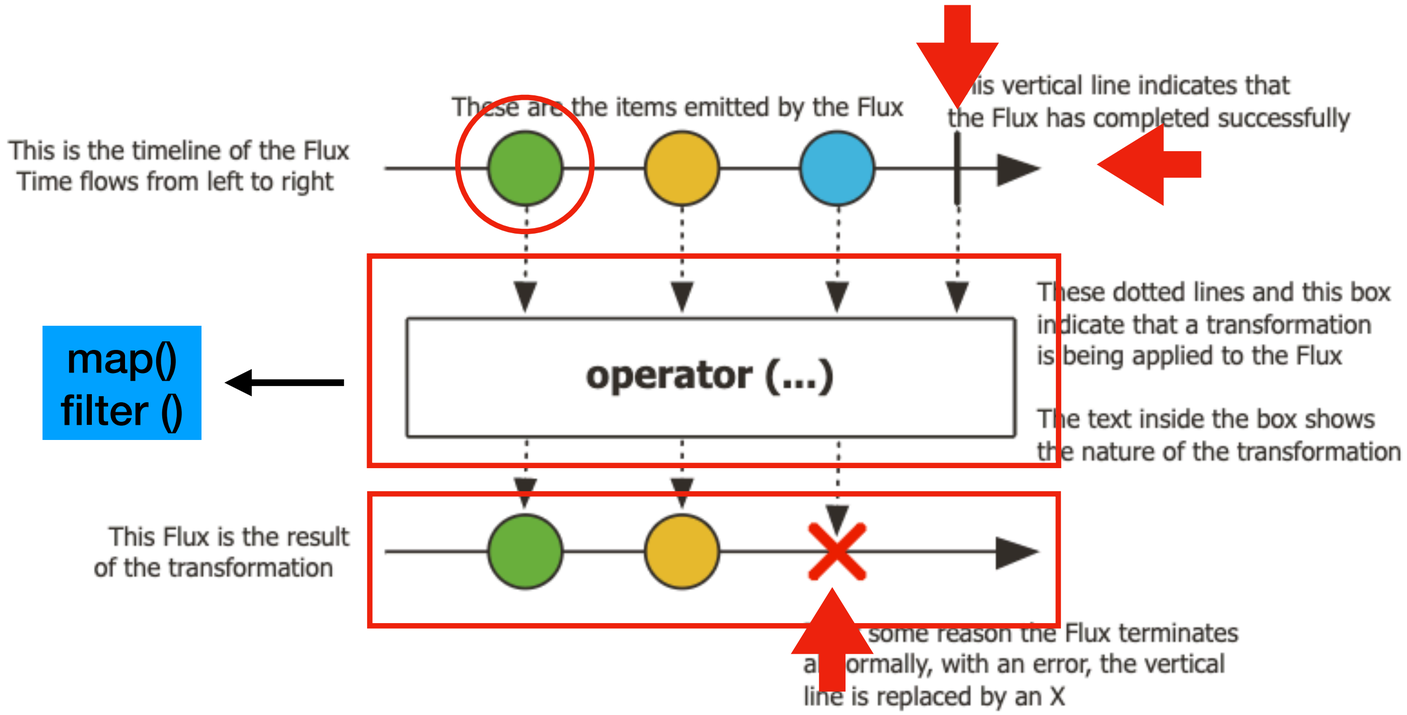
- Project Reactor is an implementation of Reactive Streams Specification
- Project Reactor is a Reactive Library
- Spring WebFlux uses Project Reactor by default

Flux
&
Mono

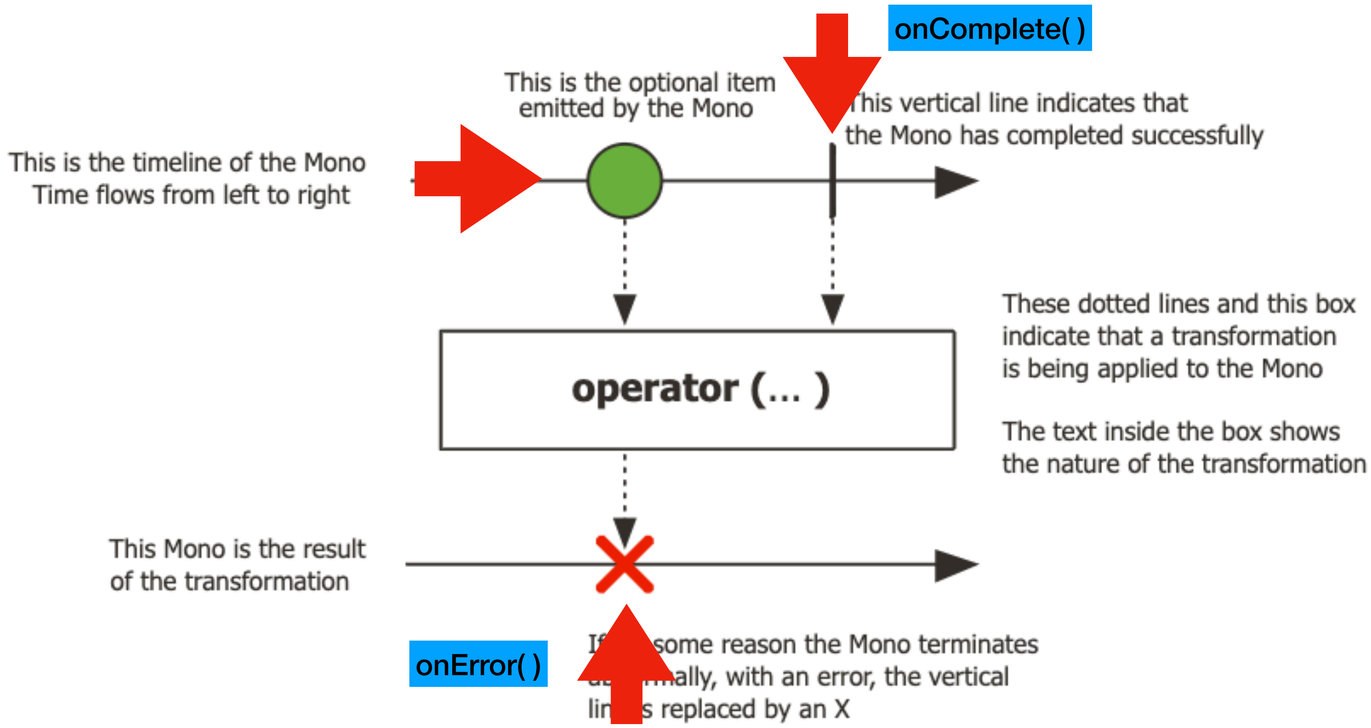
Flux & Mono

- Flux and Mono is a reactive type that implements the Reactive Streams Specification
- Flux and Mono is part of the **reactor-core** module
- Flux is a reactive type to represent 0 to N elements
- Mono is a reactive type to represent 0 to 1 element

Flux - 0 to N elements



Mono - 0 to 1 Element



Project Setup

Functional Programming In Modern Java

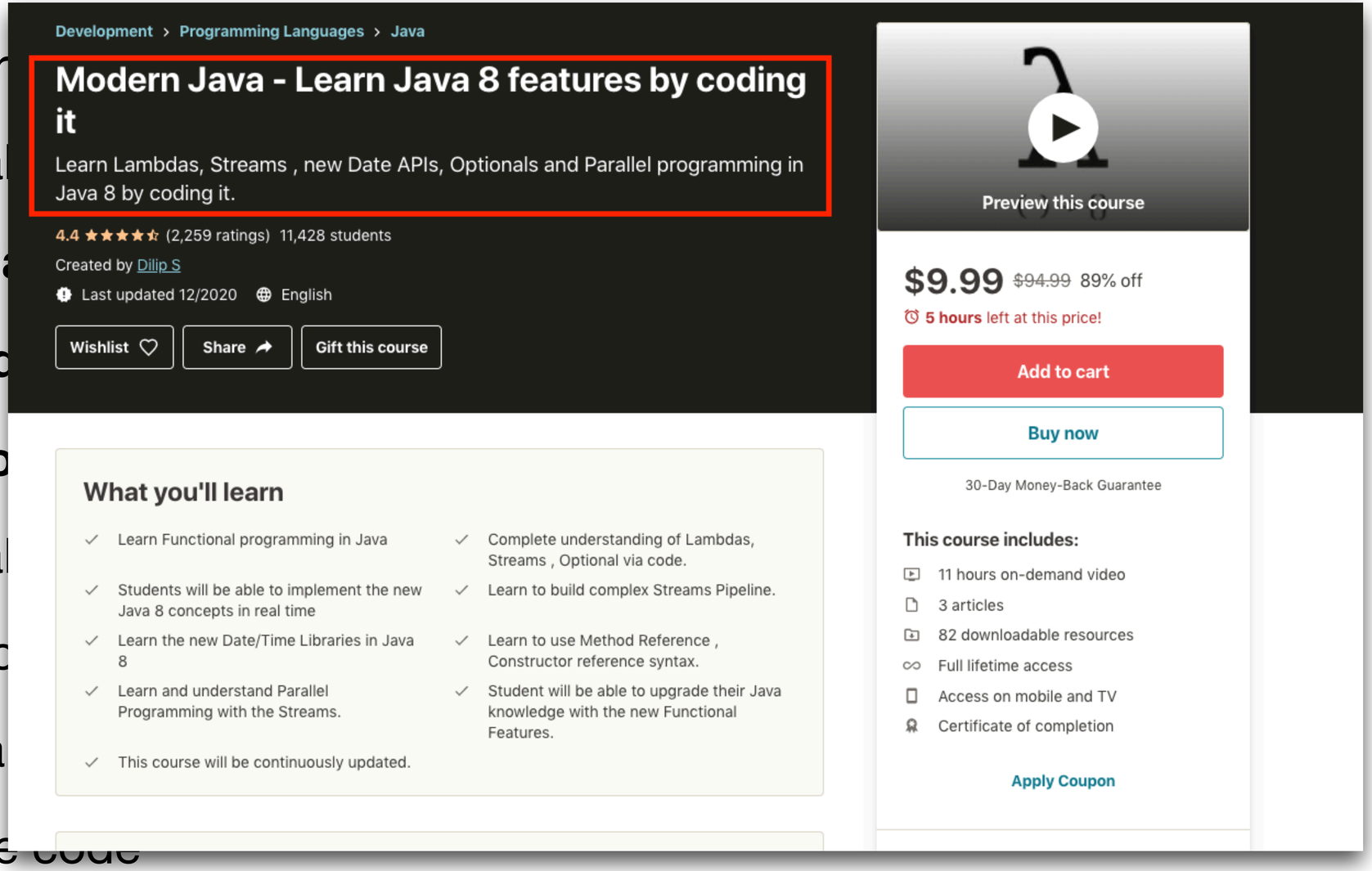
Why Functional Programming ?

Why Functional Programming ?

- Reactive programming uses **Functional Programming** style of code
  - Eg., Code similar to **Streams API**
- Reactive Programming is an extension to Functional Programming

What is Functional Programming ?

- This progr
- Functional
  - **Lambda**
  - **Method**
  - **Functio**
- Functional
  - Behavic
  - Immuta
  - Concise code

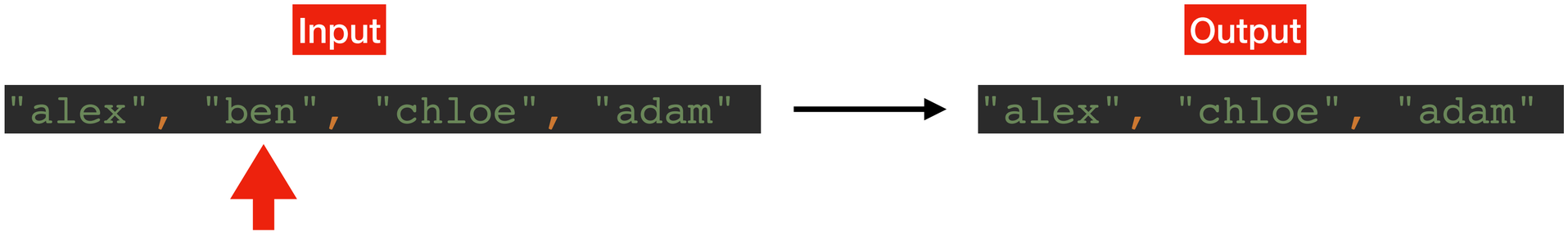


What’s the style of code that’s written before Java 8 ?

Imperative Style of Code

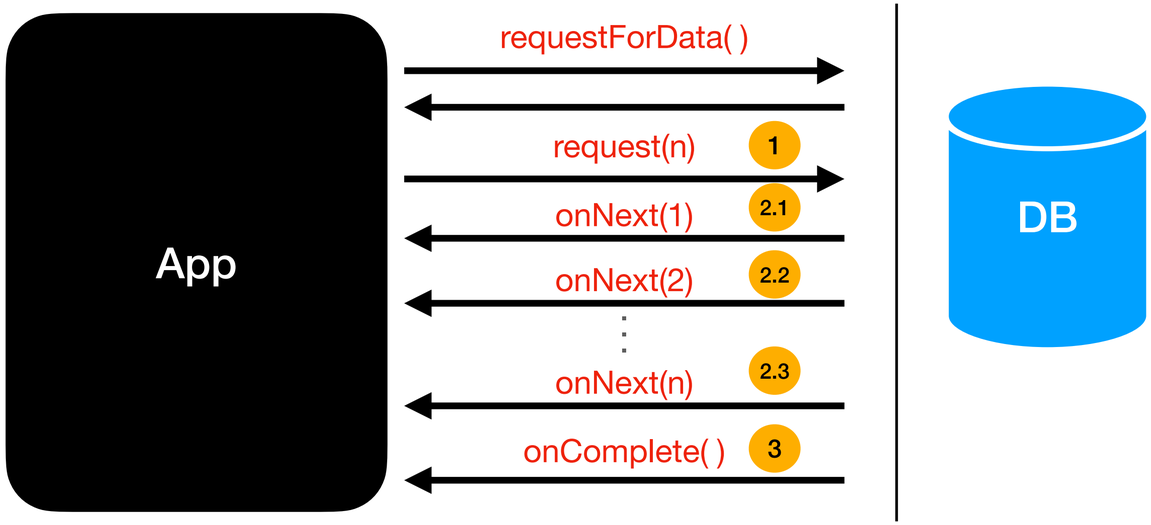
Use-case

Filter the list of Strings whose length is greater than 3



Reactive Stream Events

Reactive Streams Events



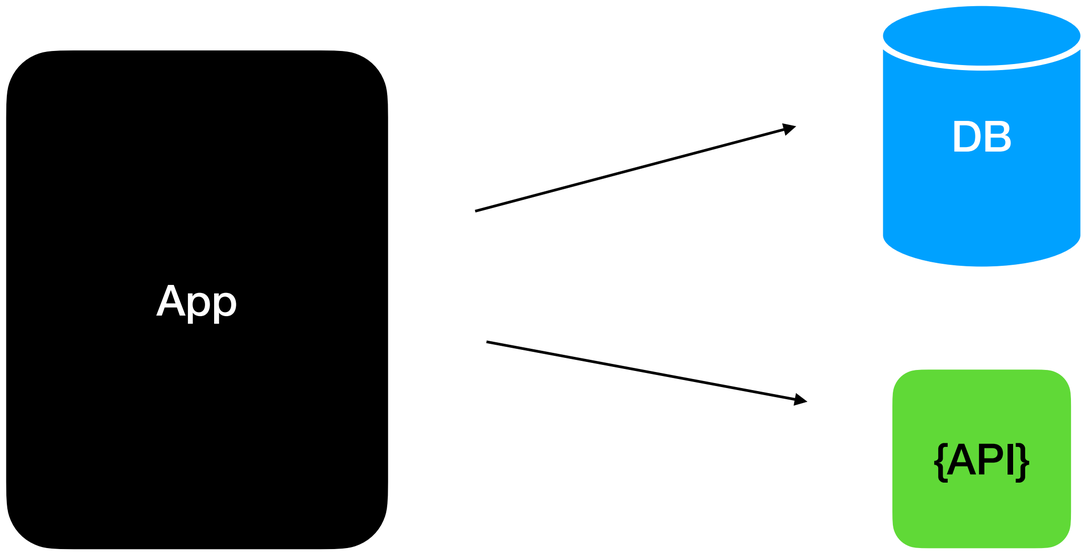
Testing Flux and Mono using StepVerifier & JUnit5

Transforming Data Using Operators in Project Reactor

Why Transform Data ?

- It is pretty common for apps to transform data from its original form

Why Transform Data ?



map()

map() Operator

- Used to transform the element from one form to another in a Reactive Stream

- Similar to the map() operator in Streams API

filter()

filter() Operator

- Used to filter elements in a Reactive Stream

- Similar to the filter() operator in Streams API

flatMap()

flatMap()

- Transforms one source element to a Flux of 1 to N elements

```
"ALEX" -> Flux.just("A", "L", "E", "X")
```

- Use it when the transformation returns a Reactive Type (Flux or Mono)

- Returns a Flux<Type>

map()

- One to One Transformation
- Does the simple transformation from **T** to **V**
- Used for simple synchronous transformations
- Does not support transformations that returns Publisher

flatmap()

- One to N Transformations
- Does more than just transformation. Subscribes to Flux or Mono that’s part of the transformation and then flattens it and sends it downstream
- Used for asynchronous transformations
- Use it with transformations that returns Publisher

concatMap()

concatMap()

- Works similar to **flatMap()**

- Only difference is that **concatMap()** preserves the ordering sequence of the Reactive Streams.

Use concatMap() if ordering matters

flatMap in Mono

flatMap in Mono

- Use it when the transformation returns a Mono

- Returns a Mono<T>

```
private Mono<List<String>> splitStringMono(String s) {
    var charArray = s.split("");
    return  Mono.just(List.of(charArray))
            .delayElement(Duration.ofSeconds(1));
}
```

- Use flatMap if the transformation involves making a REST API call or any kind of functionality that can be done asynchronously

flatMapMany( )
in
Mono

flatMap in Mono

- Works very similar to flatMap( )

```
private Flux<String> splitString_withDelay(String name) {
    var delay = new Random().nextInt(1000);
    var charArray = name.split("");
    return Flux.fromArray(charArray)
            .delayElements(Duration.ofMillis(delay));
}
```

transform()

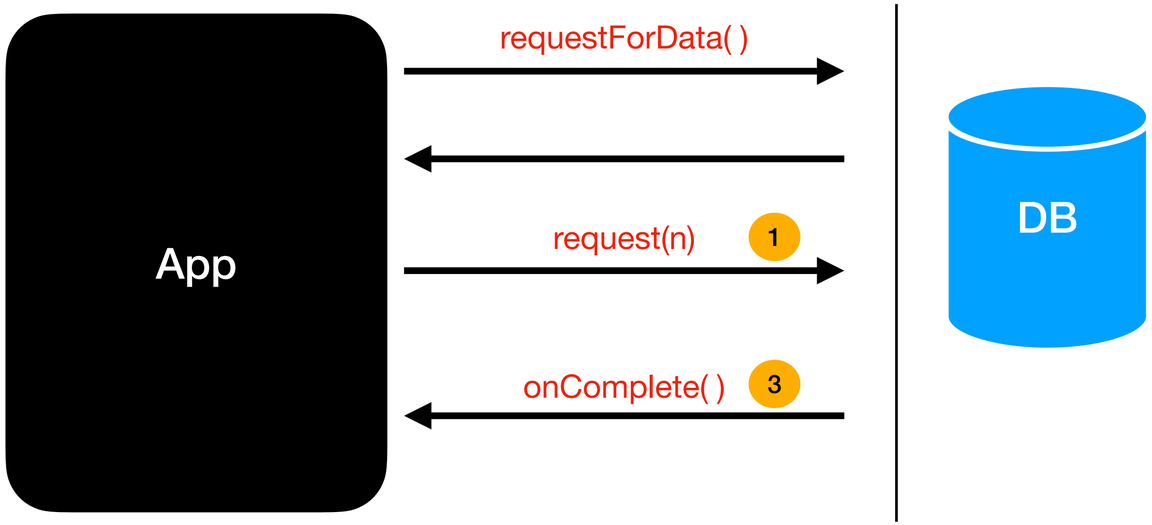
transform()

- Used to transform from one type to another
- Accepts **Function Functional Interface**
  - Function Functional Interface got released as part of Java 8
  - Input - Publisher (Flux or Mono)
  - Output - Publisher (Flux or Mono)

defaultIfEmpty()
&
switchIfEmpty()

defaultIfEmpty( ) & switchIfEmpty( )

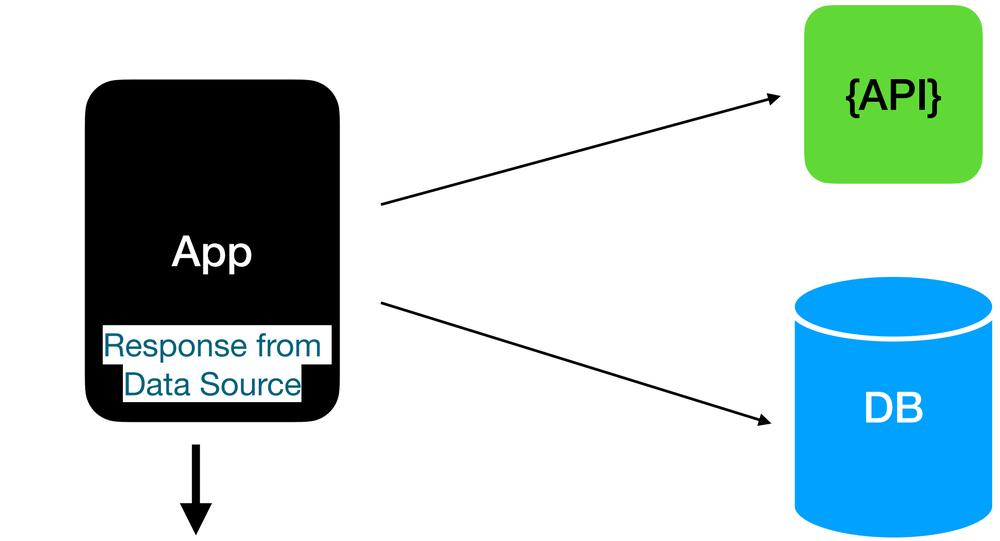
- Its not mandatory for a data source to emit data all the time



- We can use the defaultIfEmpty() or switchIfEmpty() operator to provide default values

Combining Flux & Mono

Why combining Flux and Mono ?

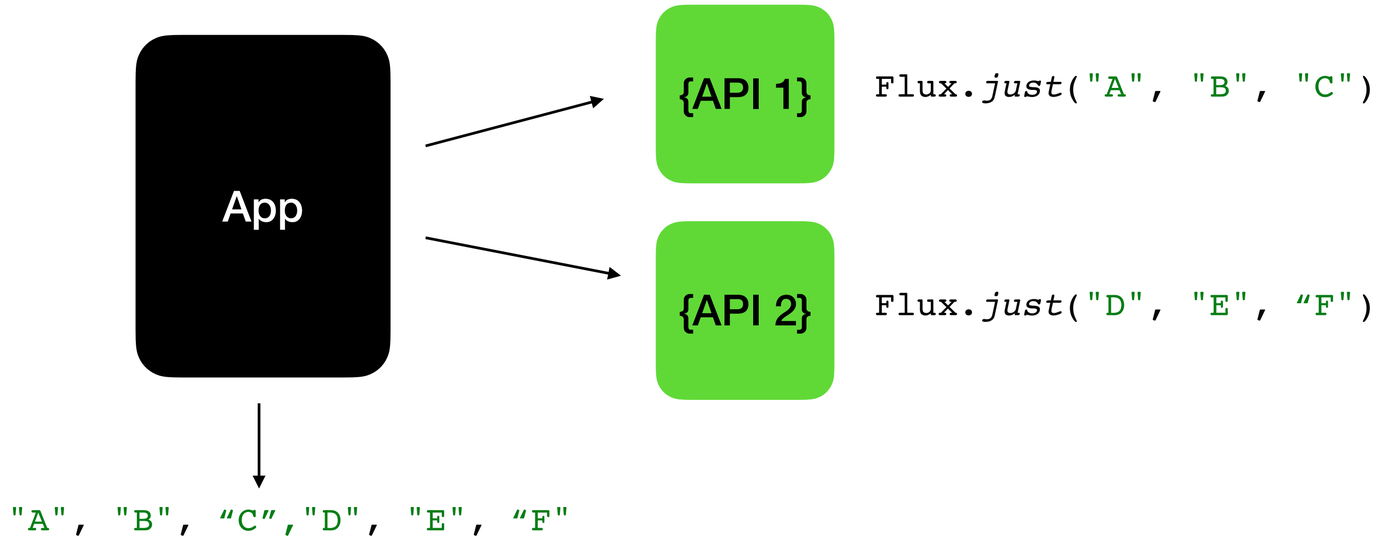



```
{
    "response": "output"
}
```

concat()
&
concatWith()

concat() & concatWith()

- Used to combine two reactive streams in to one



concat() & concatWith()

- Used to combine two reactive streams in to one
- Concatenation of Reactive Streams happens in a sequence
    - First one is subscribed first and completes
    - Second one is subscribed after that and then completes
- **concat( )** - static method in Flux
- **concatWith( )** - instance method in Flux and Mono
- Both of these operators works similarly

merge()
&
mergeWith()

merge( ) and mergeWith( ) are used to combine two publishers in to one.

merge()

- merge() operator takes in two arguments

```
// "A", "D", "B", "E", "C", "F"
// Flux is subscribed early
public Flux<String> explore_merge() {

    var abcFlux = Flux.just("A", "B", "C")
            .delayElements(Duration.ofMillis(100))
            ;

    var defFlux = Flux.just("D", "E", "F")
            .delayElements(Duration.ofMillis(125))
            ;

    return Flux.merge(abcFlux,defFlux).log();


}
```

1

2

merge() & mergeWith()

- Both the publishers are subscribed at the same time
    - Publishers are subscribed eagerly and the merge happens in an interleaved fashion
    - concat() subscribes to the Publishers in a sequence
- **merge( )** - static method in Flux
- **mergeWith( )** - instance method in Flux and Mono
- Both of these operators works similarly

mergeSequential()

mergeSequential()

- Used to combine two Publishers (Flux) in to one
- Static method in Flux
- Both the publishers are subscribed at the same time
  - Publishers are subscribed eagerly
  - Even though the publishers are subscribed eagerly the merge happens in a sequence

zip()
&
zipWith()

zip()

- Zips two publishers together in this example

```
// AD, BE, CF
public Flux<String> explore_zip() {

    var abcFlux = Flux.just("A", "B", "C");

    var defFlux = Flux.just("D", "E", "F");

    return Flux.zip(abcFlux, defFlux, (first, second) ->first + second );

}
```

AD BE CF

zip() & zipWith()

- zip()
  - Static method that’s part of the Flux
  - Can be used to merge up-to 2 to 8 Publishers (Flux or Mono) in to one
- zipWith()
  - This is an instance method that’s part of the Flux and Mono
  - Used to merge two Publishers in to one
- Publishers are subscribed eagerly
- Waits for all the Publishers involved in the transformation to emit one element
  - Continues until one publisher sends an OnComplete event

zip()

- Zips four publishers together in this example

```
// AD14, BE25, CF36
public Flux<String> explore_zip_1() {

    var abcFlux = Flux.just("A", "B", "C");
    var defFlux = Flux.just("D", "E", "F");
    var flux3 = Flux.just("1", "2", "3");
    var flux4 = Flux.just("4", "5", "6");

    return Flux.zip(abcFlux, defFlux,flux3,flux4)
            .map(t4 -> t4.getT1()+t4.getT2()+t4.getT3()+t4.getT4());


}
```

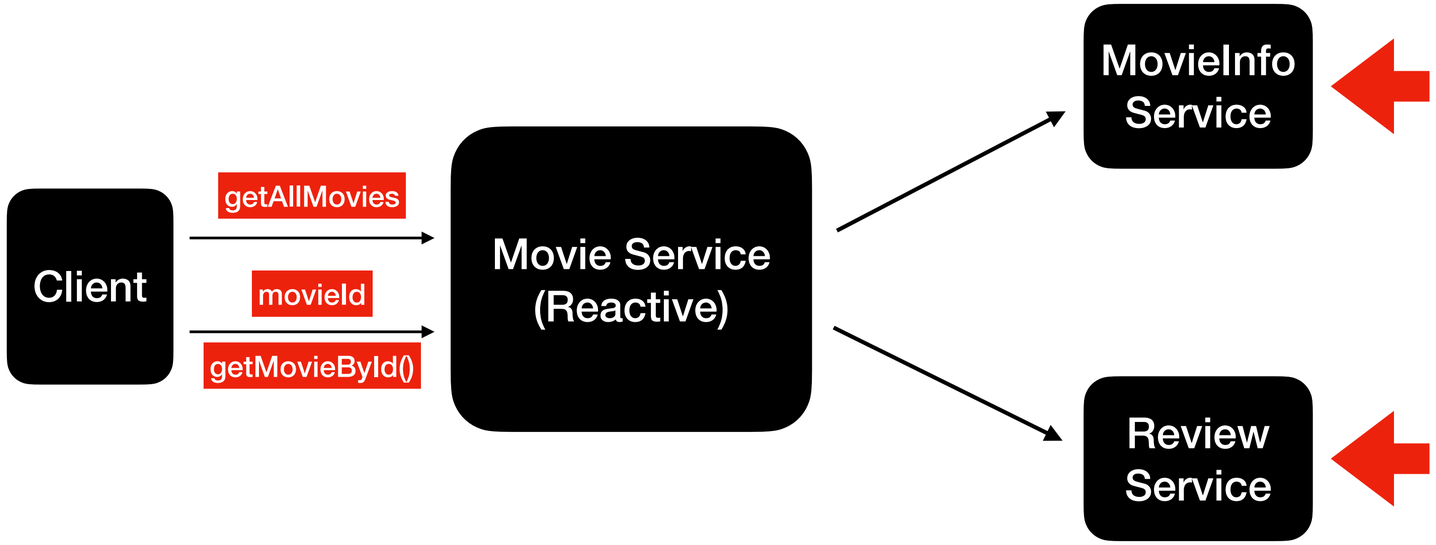
Tuple4

zip() can be used to combine 2 to 8 Publishers

zipWith() can be used to combine 2 Publishers

Reactive MovieService

Reactive Movie Service



doOn CallBacks

doOn* CallBacks

- These operators allow you to peek in to the events that are emitted by the Publisher(Flux or Mono)
- These are also called side effect operators.
  - They don’t change the original data at all
- There are many different callback operators that are available in **Project Reactor**

DoOn* CallBack operators

DoOn CallBack Functions	Usage
doOnSubscribe( )	Invoked for every new subscription from the Subscriber
doOnNext( )	Invoked for every element that's emitted from the publisher
doOnComplete( )	Invoked when the Completion signal is sent from the publisher
doOnError ( )	Invoked when an exception signal is sent from the publisher
doFinally ( )	Invoked in a successful or error scenario

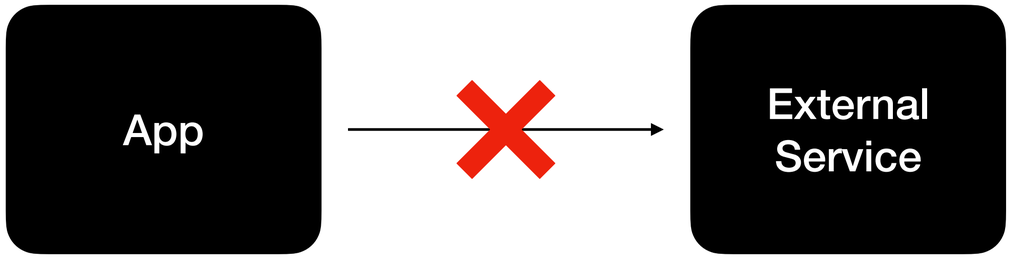
When to use doOn* CallBack operators ?

- Used for debugging an issue in your local environment

- Send a notification when the reactive sequence completes or errors out

Exception
In
Reactive Streams

Exception in Reactive Streams



Any Exception will terminate the Reactive Stream

Exception Handling In Project Reactor

Exception Handling in Project Reactor

- Two Categories of Operators:
  - **Category 1** : Recover from an Exception
  - **Category 2** : Take an action on the exception and re-throw the exception

Exception Handling in Project Reactor

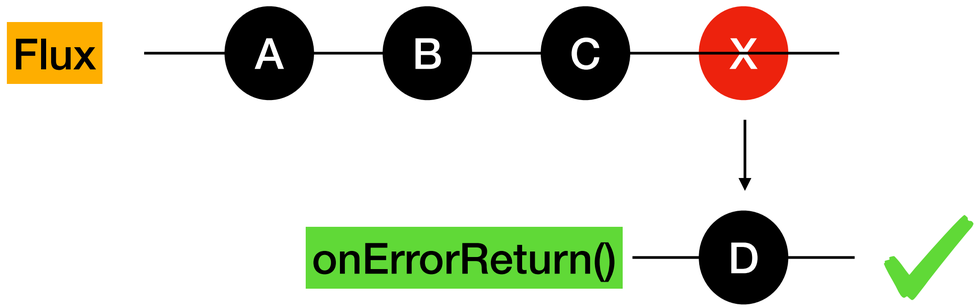
Recover From an Exception (Category 1)	Take an Action and throw the Exception (Category 2)
• onErrorReturn()	• onErrorMap()
• onErrorResume()	• doOnError()
• onErrorContinue()	

onErrorReturn()

onErrorReturn()

- Catch the exception

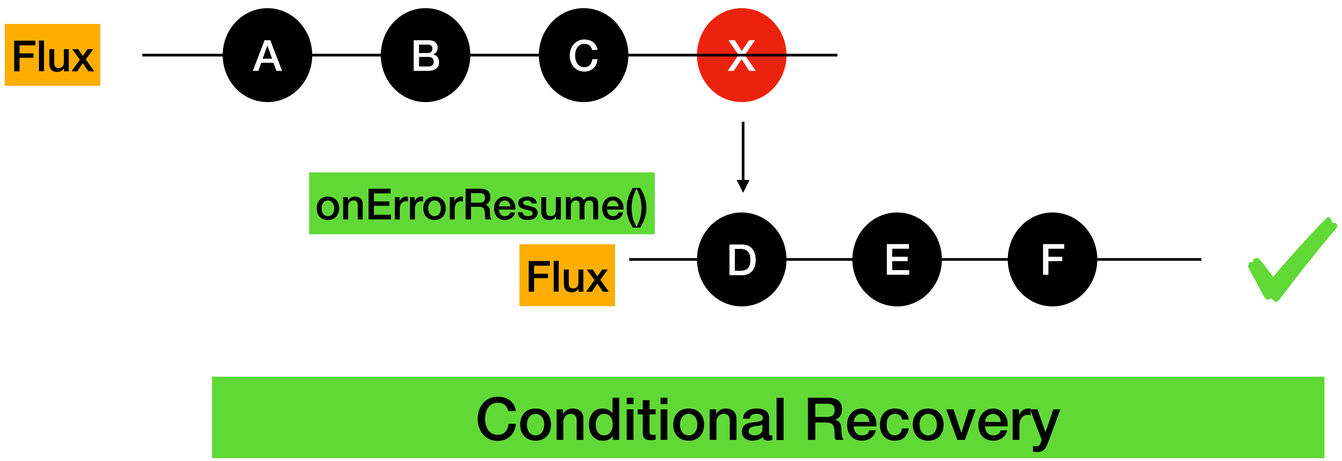
- This also provides a single default value as a fallback value



onErrorResume()

onErrorResume()

- Catch the exception
- This provides a fallback stream as a recoverable value

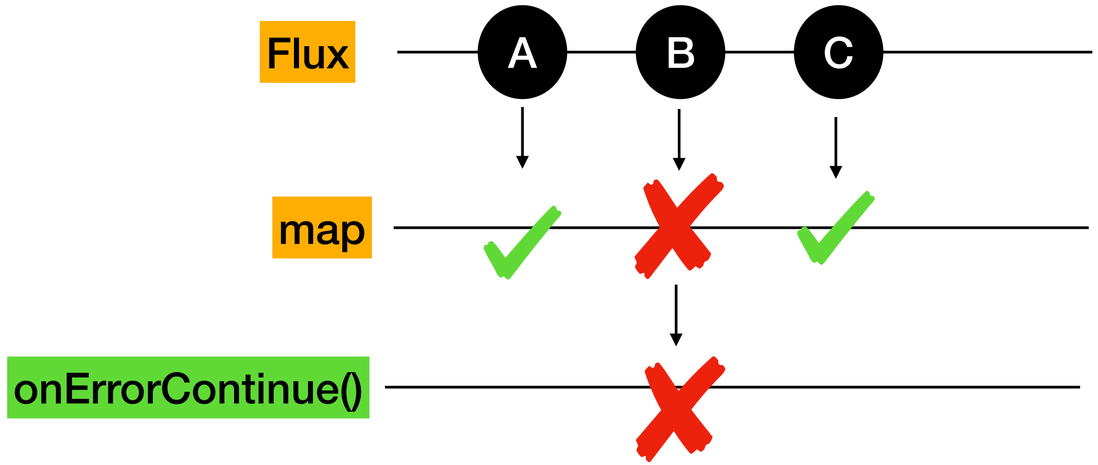


onErrorContinue()

onErrorContinue()

- Catches the exception

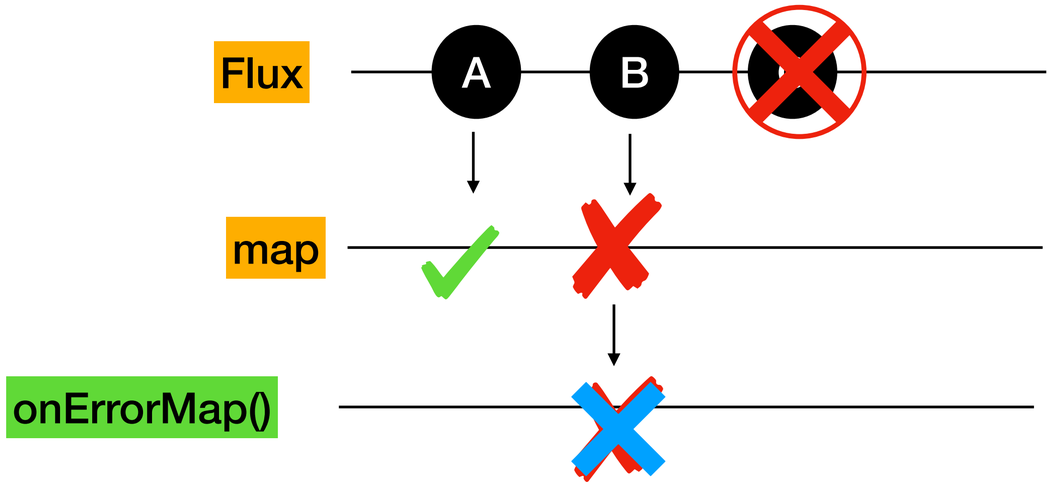
- This drops the element that caused the exception and continue emitting the remaining elements



onErrorMap()

onErrorMap()

- Catches the exception
- Transforms the exception from one type to another
  - Any RunTimeException to BusinessException
- Does not recover from the exception



doOnError()

doOnError()

- Catches the exception
- Take an action when an Exception occurs in the pipeline
- Does not modify the Reactive Stream
  - Error still gets propagated to the caller

```
public void exception(){
    try{
        // code statements
    }catch (Exception e){
        //log the exception
        throw  e;
    }
}
```

Exception Handling in Project Reactor

Recover From an Exception (Category 1)

- **onErrorReturn()**
  - Catches the exception and provides a recoverable single default value
  - Stream that caused the error will be terminated
- **onErrorResume()**
  - Catches the exception and provides another dynamic reactive stream as a fallback value
  - Stream that caused the error is terminated
  - Conditional Recovery
- **onErrorContinue()**
  - Catches the exception and allows the reactive stream to continue emitting elements

Take an Action and throw the Exception (Category 2)

- **onErrorMap()**
  - Catches the exception and transform to some Custom Exception type
- **doOnError()**
  - Catch the exception and propagate it down stream

Exception Handling Operators In Mono

Mono has the support for all the exception handling operators that we coded until now for **Flux**

Exception Handling in Mono

Recover From an Exception (Category 1)

- **onErrorReturn()**
  - Catches the exception and provides a recoverable single default value
  - Stream that caused the error will be terminated
- **onErrorResume()**
  - Catches the exception and provides another dynamic reactive stream as a fallback value
  - Stream that caused the error is terminated
  - Conditional Recovery
- **onErrorContinue()**
  - Catches the exception and allows the reactive stream to continue emitting elements

Take an Action and throw the Exception (Category 2)

- **onErrorMap()**
  - Catches the exception and transform to some Custom Exception type
- **doOnError()**
  - Catch the exception and propagate it down stream

retry()

retry()

- Use this operator to retry failed exceptions
- When to use it ?
  - Code interacts with external systems through network
    - Examples are : RestFul API calls, DB Calls
  - These calls may fail intermittently

retry()

- **retry**()
  - Retry the failed exception indefinitely
- **retry**(N)
  - N is a long value
  - Retry the failed exception “n” number of times

retryWhen()

retryWhen( )

- **retryWhen**( ) is more advanced compared to retry( )

- Conditionally perform retry on specific exceptions

repeat()

repeat( )

- Used to repeat an existing sequence
- This operator gets invoked after the onCompletion( ) event from the existing sequence
- Use it when you have an use-case to subscribe to same publisher again
- This operator works as long as **No Exception** is thrown

repeat( )

- **repeat**( )
  - Subscribes to the publisher indefinitely
- **repeat**(n)
  - Subscribes to the publisher “N” times

repeatWhen()

repeatWhen( )

- This is an advanced operator compared to **repeat( )/repeat(N)**
- Everything about repeat( )/repeat(n) is true for repeatWhen( )
- You have more control on repeating a sequence
  - Example , You can introduce a delay before repeating a sequence
- This only works when there is **No Exception** is thrown

retry( ) vs repeat( )

- **retry**( )
  - This operator gets invoked when there is an Exception
  - This operator makes our code more resilient to Exception/Errors
- **repeat**( )
  - This operator gets invoked when there is an OnComplete() event from the original sequence

Reactor Execution Model

Reactor Execution Model

- Reactor Execution model is determined by **Scheduler**
- This is an interface which is part of the Project Reactor
- Similar to **ExecutorService** in Java which takes care of scheduling and executing tasks

By default, the data flows in the thread where the subscribe() was invoked.

Reactor Execution Model

```
private Flux<String> splitString_withDelay(String name) {
    var delay = new Random().nextInt(1000);
    var charArray = name.split("");

    return Flux.fromArray(charArray)

            .delayElements(Duration.ofMillis(delay));
}
```

Switched the thread to “parallel”

No of threads = No of CPU cores in the machine

Can we instruct the project-reactor to use a different Scheduler ?

Scheduler Options

- **Schedulers** is a factory class that can be used to switch the threads in the reactive pipeline execution
- **Schedulers.parallel()**
  - It has a fixed pool of workers. No of threads is equivalent to no of CPU cores
  - The time based operators use this by default (**delayElements(), interval()**)
- **Schedulers.boundElastic()**
  - It has a bounded elastic thread pool of workers
  - The no of threads can grow based on the need. It can increase up to 10 X no of CPU cores
  - This is ideal for making Blocking IO calls
- **Schedulers.single()**
  - A single reusable thread for executing the tasks

publishOn(Scheduler s)

publishOn(Scheduler s)

- This operator is used to hop the Thread of execution of the reactive pipeline from one to another.

- When to use **publishOn(Scheduler s)** ?

  - Never block the thread in reactive programming

  - Blocking operation in the reactive pipeline can be performed after **publishOn** operator.

  - The thread of execution is determined by the Scheduler that passed to it

Operators after publishOn( ) call will use the same thread that’s part of the Scheduler that’s passed on to the publishOn( )

publishOn( ) influences the Thread downstream.

subscribeOn(Scheduler s)

subscribeOn(Scheduler s)

- This operator is used to hop the Thread of execution of the reactive pipeline from one to another.

- subscribeOn( ) is used to influence the thread upstream( )

  - It influences the operators above the subscribeOn() to switch the thread

subscribeOn(Scheduler s)

```
var namesFlux1 = flux2()
    1   .map((s) -> {
            log.info("Value of s is {}", s);
            return s;
        })
    2   .subscribeOn(Schedulers.boundedElastic())
        .log();
```

Thread of execution will be part of boundedElastic

Switch the thread of execution to boundedElastic

subscribeOn(Scheduler s)

```
var namesFlux1 = flux2()
        .map((s) -> {
            log.info("Value of s is {}", s);
            return s;
        })
        .subscribeOn(Schedulers.boundedElastic())
        .map((s) -> {
            log.info("Value of s after boundedElastic is {}", s);
            return s;
        })
        .log();
```

Thread of execution will be part of boundedElastic

Thread of execution will be part of boundedElastic

subscribeOn( ) impacts the whole reactive pipeline

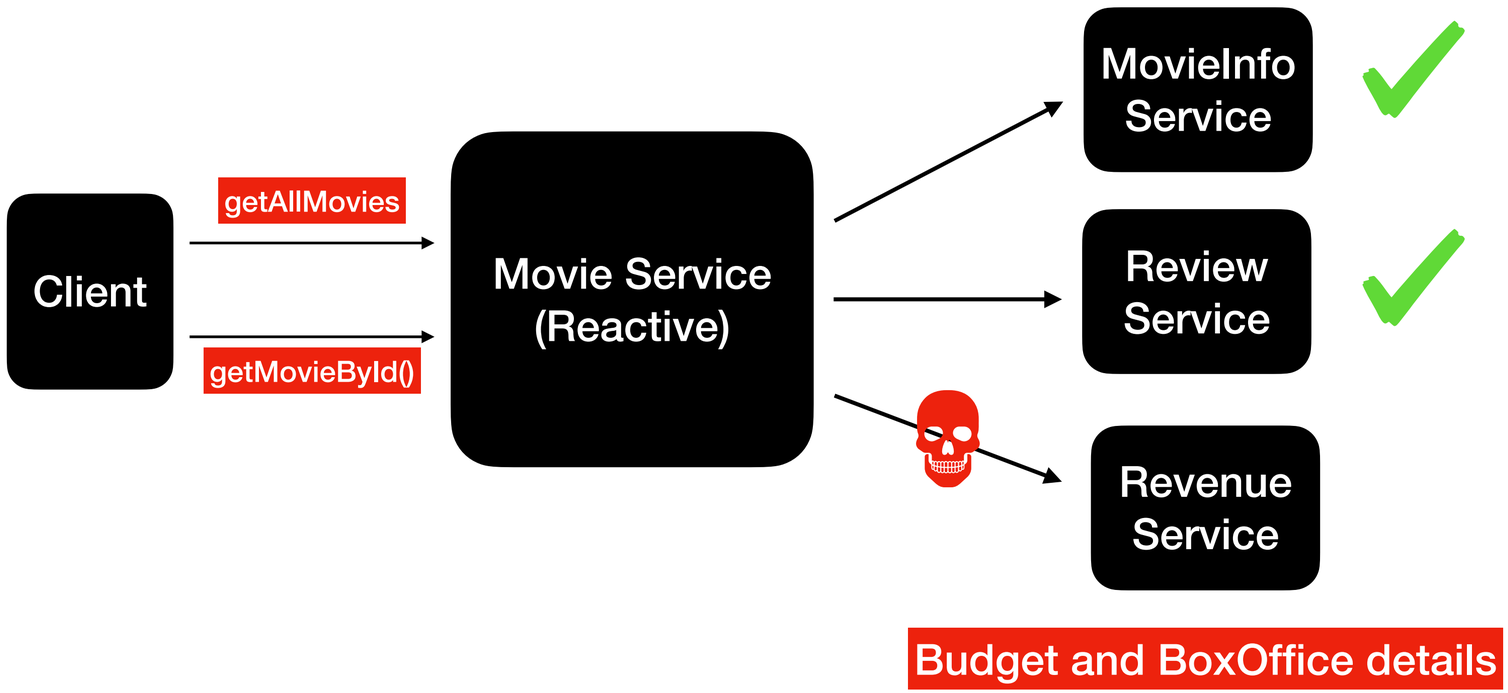
subscribeOn() - When to use it ?

subscribeOn() - When to use it ?

- Blocking code is part of the library where publishOn() is not added to it
  - Use subscribeOn() to influence the upstream to switch the thread

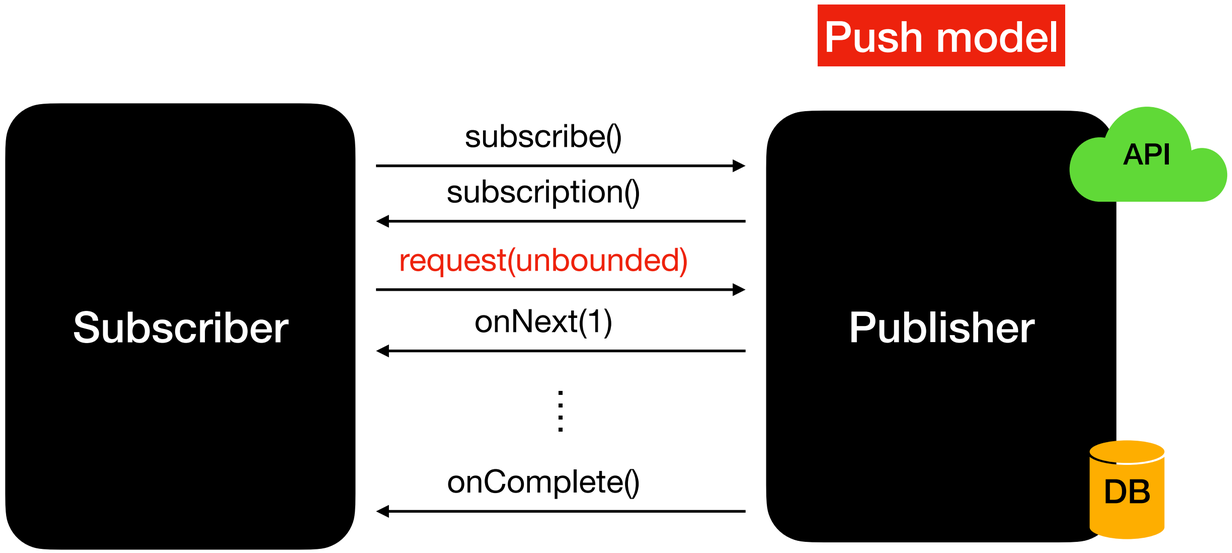
Blocking Calls in MovieReactive Service

Movie Service



Backpressure

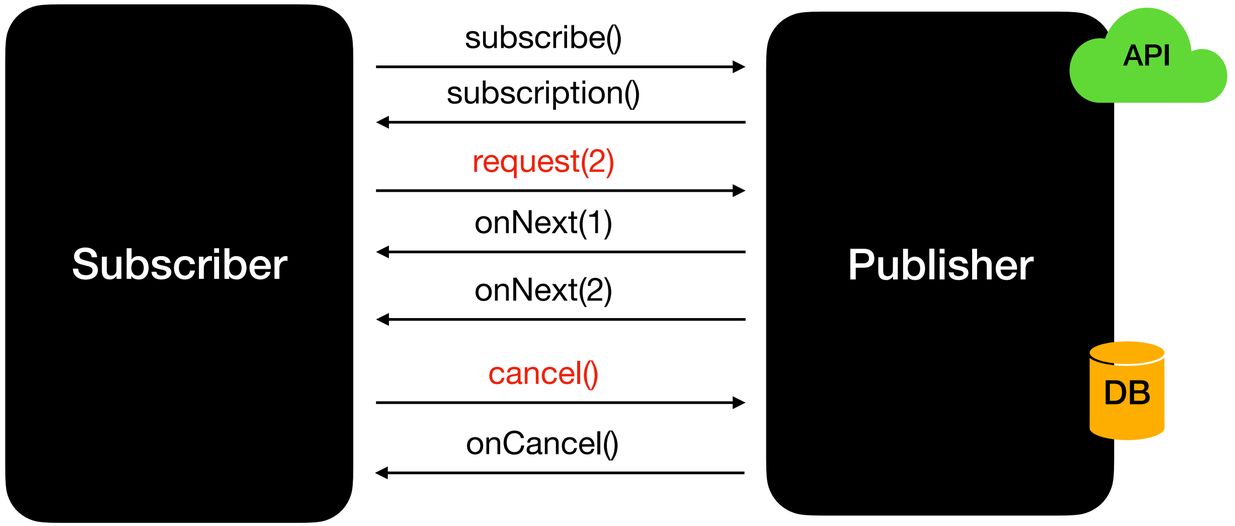
Reactive Programing - Recap and Issues



1. Overwhelmed with more data than the subscriber can handle

2. Data might be pushed at a faster rate than the subscriber can handle

BackPressure



onBackPressureDrop( )

onBackPressureDrop( )

- Overrides the subscribers request and requests for unbounded data
- It stores all the data in an internal queue
- Drops the remaining elements that are not needed by the subscriber
- This operator helps to track the items that are not needed by the subscriber

onBackPressureBuffer( )

onBackPressureBuffer( )

- Overrides the subscribers request and requests for unbounded data
- It stores all the data in an internal queue
- Buffers the remaining elements that are not needed by the subscriber
- The advantage is that the following requests after the initial request for data from the subscriber does not need to go all the way to the Publisher

onBackPressureError( )

onBackPressureError( )

- Overrides the subscribers request and requests for unbounded data
- It stores all the data in an internal queue
- Throws an **OverflowException** when the publisher sends more data than the subscriber’s requested amount

Processing Data in Parallel in Project Reactor

Reactive Flow is sequential by default

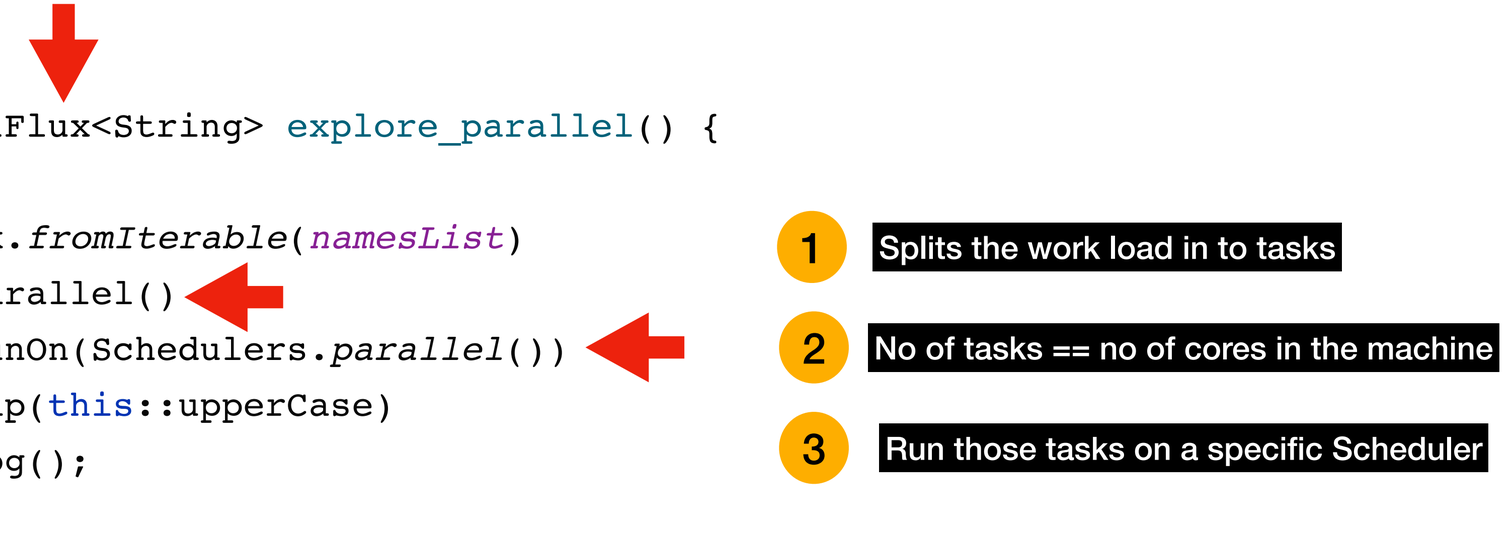
ParallelFlux

ParallelFlux

- The idea behind **ParallelFlux** is to leverage the multi-core processors that we have in today's hardware

- MultiCore = Process multiple things at the same time

ParallelFlux



No of elements that can be processed in parallel is equal to the no of cores in your machine

Parallelism using flatMap( )

flatMapSequential()

flatMapSequential( )

This operator helps to achieve concurrency and maintain the ordering of elements at the same time

Cold & Hot Streams

Cold Streams

- Cold Stream is a type of Stream which emits the elements from beginning to end for every new subscription

```
@Test
public void coldPublisherTest() throws InterruptedException {

    Flux<Integer> stringFlux = Flux.range(1, 10)
            .delayElements(Duration.ofSeconds(1));

1   stringFlux.subscribe(s -> System.out.println("Subscriber 1 : " + s)); 1 .. 10
    Thread.sleep(2000);

2   stringFlux.subscribe(s -> System.out.println("Subscriber 2 : " + s)); 1 .. 10
    Thread.sleep(10000);
}
```

Cold Streams

- Examples of Cold Streams
  - HTTP Call with similar request
  - DB call with similar request

Hot Streams

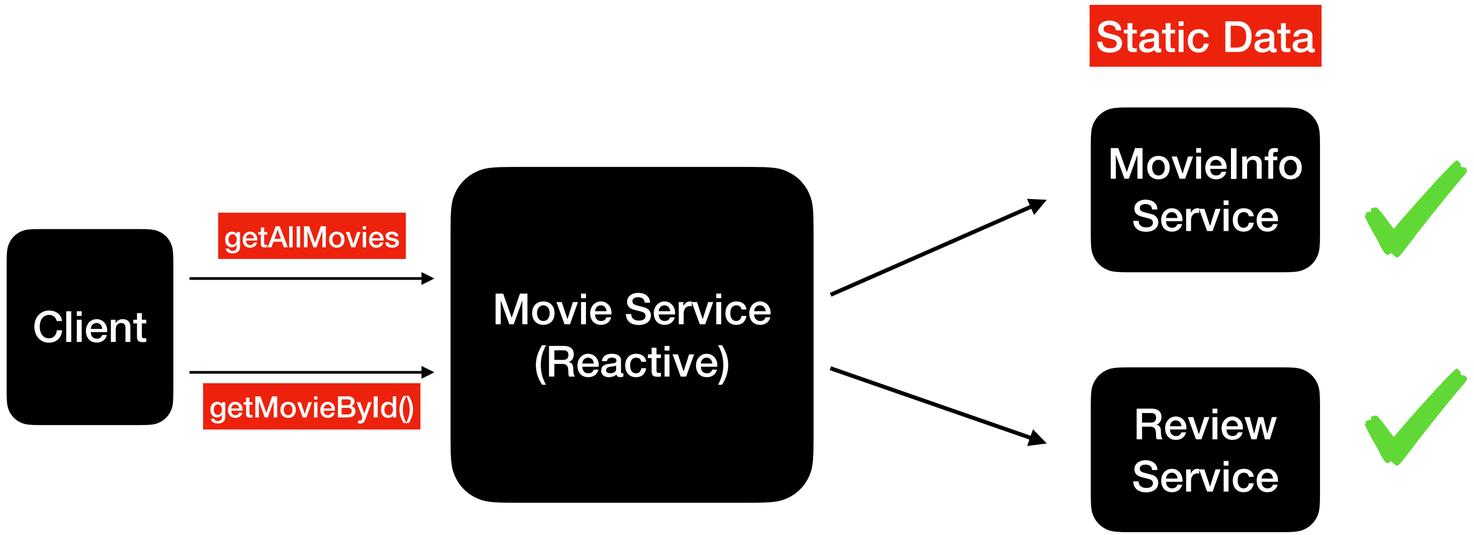
- Data is emitted continuously
- Any new subscriber will only get the current state of the Reactive Stream
  - Type 1: Waits for the first subscription from the subscriber and emits the data continuously
  - Type 2: Emits the data continuously without the need for subscription

Hot Streams

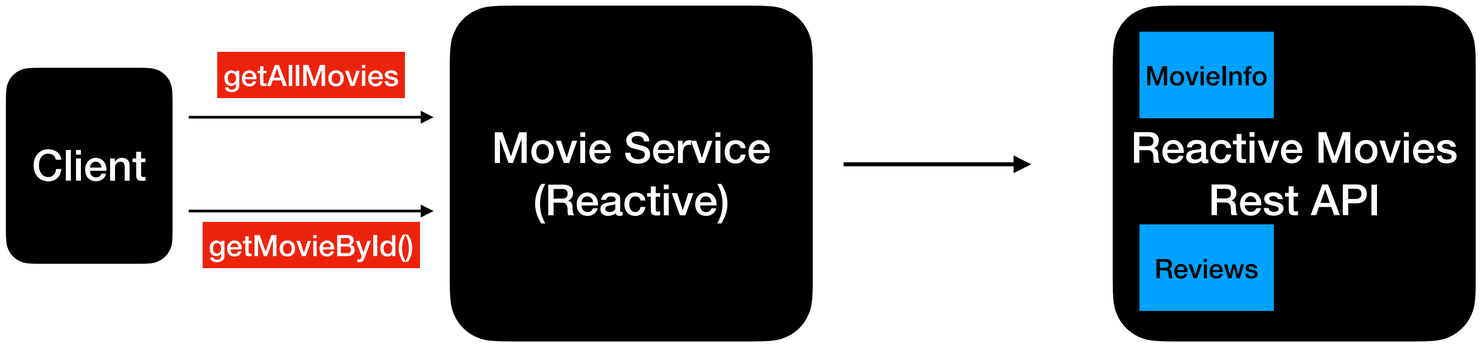
- Examples:
  - Stock Tickers - Emits stock updates continuously as they change
  - Uber Driver Tracking -> Emits the of the current position of the Driver Continuously

Building Non Blocking Client Using Spring WebClient

Movie Service



Movie Service



- Spring WebFlux
- Uses H2 InMemory DB

Programmatically Creating Flux/Mono

Creating Flux/Mono Until Now

Flux

```
Flux.just("A", "B", "C");
```

```
Flux.fromIterable(namesList)
```

```
Flux.fromArray(charArray)
```

```
Flux.range(0, max)
```

```
webClient.get().uri(uri)
        .retrieve()
        .bodyToFlux(Review.class)
```

Mono

```
Mono.just("alex");
```

```
Mono.empty()
```

```
Mono.fromCallable()
```

```
webClient.get().uri("/v1/movie_infos/{id}",
                    movieInfoId)
        .retrieve()
        .bodyToMono(MovieInfo.class)
```

Any External System will have its own Reactive Adapters to build Flux and Mono

Creating Flux/Mono Programmatically

Flux	Mono
`Flux.generate();`	`Mono.create()`
`Flux.create()`	
`Flux.push()`	
`Flux.handle()`	

We need to explicitly emit the OnNext, OnComplete and onError events from our code.

Flux.generate( )

Flux.generate( )

- This operator takes a initial value and a generator function as an input and continuously emit values
- This is also called Synchronous generate
- We will be able to generate the OnNext, OnComplete and onError events using the **SynchronousSink** class
- Use this operator, if you have a use case to emit values from a starting value until a certain a certain condition is met. (Similar to **for** loop)

Generate a sequence from 1 to 10 and Multiply each element by 2

1,2,3,4,5,6,7,8,9,10

2,4,6,8,10,12,14,16,18,20

Flux.create( )

Flux.create( )

- Used to bridge an existing API in to the Reactive World
- This is Asynchronous and Multithreaded
  - We can generate/emit these events from multiple threads
- We will be able to generate the OnNext, OnComplete and onError events using the **FluxSink** class
- MultipleEmissions in a single round is supported

Mono.create( )

Mono.create( )

- Programmatically create a Mono

- We will be able to generate the OnNext, OnComplete and onError events using the **MonoSink** class

Flux.push( )

Flux.push( )

- Used to bridge an existing API in to the Reactive World
- This is Asynchronous and Single Threaded
- We will be able to generate the OnNext, OnComplete and onError events using the **FluxSink** class
- MultipleEmissions in a single round is supported

Hooks.onOperatorDebug()

Hooks.onOperatorDebug()

- Gives you the visibility on which operator caused the problem
- This feature captures the stack-trace each operator
- This feature needs to be activated during the start up of the application

Hooks.onOperatorDebug() is not recommended for prod as it may slow down the performance of the app.

Production-ready Global Debugging using "ReactorDebugAgent"

ReactorDebugAgent

- This is recommended option for debugging exceptions in project Reactor
- Java Agent that runs alongside your app
- It collects the stack trace information of each operator without any performance overhead
- **reactor-tools**

Using ReactorDebugAgent in SpringBoot

```
public static void main(String[] args) {

    ReactorDebugAgent.init();

    SpringApplication.run(Application.class, args);
}
```

Next Steps

Next Steps:

- Build Reactive APIS using Spring Webflux and Project Reactor