What is reactive programming

1. It is new programming paradigm

* 1. Asynchronous and non-blocking
  2. Event and Message Driven
  3. Functional code style
  4. Back Pressure on Data Stream

REACTIVE CORE

Reactor is **fully non-blocking** and provides efficient demand management. It directly interacts with Java's functional API, CompletableFuture, Stream, and Duration.

TYPED [0|1|N] SEQUENCES

Reactor offers **two reactive and composable APIs**, [Flux [N]](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html)and [Mono [0|1]](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html), which extensively implement [Reactive Extensions](http://reactivex.io/).

NON-BLOCKING IO

Well-suited for a **microservices** architecture, Reactor offers **backpressure-ready network engines** for HTTP (including Websockets), TCP, and UDP.

Reactive Stream Specification

Publisher --- database or server the data store which willl send you the data

Subscriber -- It is our application which is subscribe to the datasource

Subscription -- It Connect Publisher and Subscriber

Processor -- It Extends Publisher and Subscriber

Processor can behave as a Publisher and Subscriber

Project Reactor

Implementation of reactive stream specification

Reactive Library

Spring WebFlux uses Project reactor

Flux and Mono

These are reactive types that implementation Reactive Stream Specification

Part of reactive-core Module

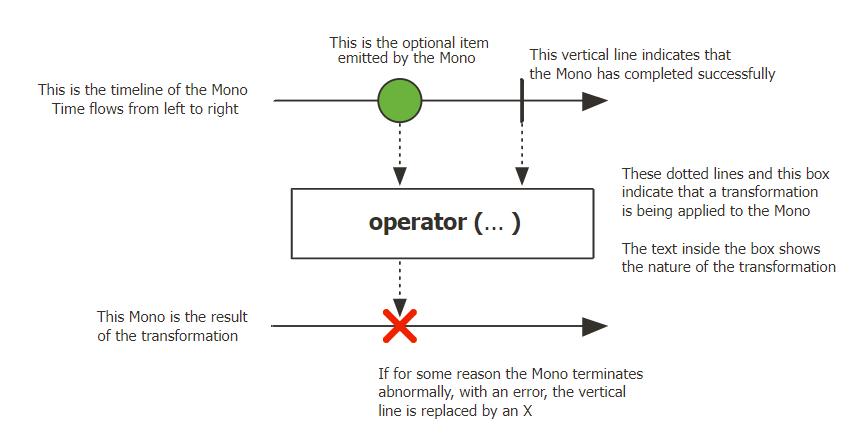
Flux represents 0 to N elements

Mono represents 0 to 1 elemets

* 1. Flux representation

This is the timeline of the Flux 
Time flows from left to right 
This Flux is the result 
of the transformation 
These are the items emitted by the Flux 
operator (...) 
This vertical line indicates that 
the Flux has completed successfully 
These dotted lines and this box 
indicate that a transformation 
is being applied to the Flux 
The text inside the box shows 
the nature of the transformation 
If for some reason the Flux terminates 
abnormally, with an error, the vertical 
line is replaced by an X 

* 1. Mono Representation



Operator -- map, filter, flatMap..

Note :- When we want run or execute any of the mono or flux we need to subscribe first.

After that subscription the publisher will emit that data.

Example:

Flux.fromIterable(Itrable) --> this method create the flux object from list

Operator

* 1. Map

It is used to map the data that you are get in to a differenet form of data, but with the same type of flux. Its converting the one flux into another flux with different format.

Flux.fromIterable(List.of("Mango","Orange","Banana"))

.map(String::toUpperCase);

* 1. Filter

This method filter out the data on predication.

Flux.fromIterable(List.of("Mango","Orange","Banana"))

.map(String::toUpperCase);

* 1. FlatMap

This method will do each and every element emmited over here it will convert that element to flux of 1 to n.

Flux.fromIterable(List.of("Mango","Orange","Banana"))

.flatMap(s -> Flux.just(s.split("")))

* 1. ConcatMap

FlatMap and ConcatMap is the same operation, but it will having the same order

Flux.fromIterable(List.of("Mango","Orange","Banana"))

.concatMap(s -> Flux.just(s.split(""))

.delayElements(Duration.ofMillis(

new Random().nextInt(1000)

)))

* 1. FlatMapMany

This method is used when you want to do the flat map opearation on mono but at the end you want return flux object.

It is convert your mono object to flux object.

Mono.just("Mango")

.flatMapMany(s -> Flux.just(s.split("")))

* 1. Transform

Transform operator is used convert one type to another type.

It will take the function interface as a parameter, where we define the input type and that perticular type will apply to that transform

Function<Flux<String>,Flux<String>> filterData

= data -> data.filter(s -> s.length() > number);

Flux.fromIterable(List.of("Mango","Orange","Banana"))

.transform(filterData)

* 1. DefaultIfEmpty

Whenever you are using transform function, and if there is no data available, if transform function not returning any data we can use defaultIfEmpty() method.

Function<Flux<String>,Flux<String>> filterData

= data -> data.filter(s -> s.length() > number);

Flux.fromIterable(List.of("Mango","Orange","Banana"))

.transform(filterData)

.defaultIfEmpty("Default")

* 1. SwitchIfEmpty

If transform method not emmitted the data from one set, then we can use this method to switch with different set of data

Function<Flux<String>,Flux<String>> filterData

= data -> data.filter(s -> s.length() > number);

Flux.fromIterable(List.of("Mango","Orange","Banana"))

.transform(filterData)

.switchIfEmpty(Flux.just("Pineapple","Jack Fruit")

.transform(filterData))

* 1. Concat

It will concat the two types available i. e. mono or flux

var fruits = Flux.just("Mango","Orange");

var veggies = Flux.just("Tomato","Lemon");

Flux.concat(fruits,veggies);

* 1. ConcatWith

It is instance method.

var fruits = Flux.just("Mango","Orange");

var veggies = Flux.just("Tomato","Lemon");

fruits.concatWith(veggies);

* 1. Merge

This method is subscribe to the publisher eagerly and it will emit the data in the eager fashion/ Asynchronously

* 1. MergeWithSequentials

It will subscribe to the both the publisher in the eager fashion, it will emiting the data in sequentials forms

Flux.mergeSequential(fruits,veggies)

* 1. Zip

This operator is very usefull for when we work with different object.

Zip two sources together, that is to say wait for all the sources to emit one element and combine these elements once into an output value (constructed by the provided combinator). The operator will continue doing so until any of the sources completes. Errors will immediately be forwarded. This "Step-Merge" processing is especially useful in Scatter-Gather scenarios.

Flux.zip(fruits,veggies, (first,second) -> first+second)

* 1. zipWith

Zip this [Flux](eclipse-javadoc:%E2%98%82=reactive-programming-examle/C:%5C/Users%5C/AnkushG2%5C/.m2%5C/repository%5C/io%5C/projectreactor%5C/reactor-core%5C/3.4.13%5C/reactor-core-3.4.13.jar=/maven.pomderived=/true=/=/maven.pomderived=/true=/=/maven.groupId=/io.projectreactor=/=/maven.artifactId=/reactor-core=/=/maven.version=/3.4.13=/=/maven.scope=/compile=/%3Creactor.core.publisher(Flux.class%E2%98%83Flux~zipWith~Lorg.reactivestreams.Publisher%5C%3C+TT2;%3E;~Ljava.util.function.BiFunction%5C%3C-TT;-TT2;+TV;%3E;%E2%98%82Flux) with another [Publisher](eclipse-javadoc:%E2%98%82=reactive-programming-examle/C:%5C/Users%5C/AnkushG2%5C/.m2%5C/repository%5C/io%5C/projectreactor%5C/reactor-core%5C/3.4.13%5C/reactor-core-3.4.13.jar=/maven.pomderived=/true=/=/maven.pomderived=/true=/=/maven.groupId=/io.projectreactor=/=/maven.artifactId=/reactor-core=/=/maven.version=/3.4.13=/=/maven.scope=/compile=/%3Creactor.core.publisher(Flux.class%E2%98%83Flux~zipWith~Lorg.reactivestreams.Publisher%5C%3C+TT2;%3E;~Ljava.util.function.BiFunction%5C%3C-TT;-TT2;+TV;%3E;%E2%98%82Publisher) source, that is to say wait for both to emit one element and combine these elements using a combinator [BiFunction](eclipse-javadoc:%E2%98%82=reactive-programming-examle/C:%5C/Users%5C/AnkushG2%5C/.m2%5C/repository%5C/io%5C/projectreactor%5C/reactor-core%5C/3.4.13%5C/reactor-core-3.4.13.jar=/maven.pomderived=/true=/=/maven.pomderived=/true=/=/maven.groupId=/io.projectreactor=/=/maven.artifactId=/reactor-core=/=/maven.version=/3.4.13=/=/maven.scope=/compile=/%3Creactor.core.publisher(Flux.class%E2%98%83Flux~zipWith~Lorg.reactivestreams.Publisher%5C%3C+TT2;%3E;~Ljava.util.function.BiFunction%5C%3C-TT;-TT2;+TV;%3E;%E2%98%82BiFunction) The operator will continue doing so until any of the sources completes. Errors will immediately be forwarded. This "Step-Merge" processing is especially useful in Scatter-Gather scenarios.

fruits.zipWith(veggies, (first,second) -> first+second)

Flux.zip(fruits,veggies,moreVeggies).map(objects -> objects.getT1() + objects.getT2() + objects.getT3())

1. doOnNext

This method is called when the onNext() method invoked.

1. doOnSuscriebe

This method is called once the subscribe() method invoked.

Flux.fromIterable(List.of("Mango","Orange","Banana"))

.filter(s -> s.length() > number)

.doOnNext(s -> {

System.out.println("s = " + s);

})

.doOnSubscribe(subscription -> {

System.out.println("subscription.toString() = " + subscription.toString());

})

.doOnComplete(() -> System.out.println("Completed!!!"))

Exception Handling

Whenever there is any error occour the reactive programming will stop the emmiting of data.

It will send the error as data. So we need to handle this data.

There are two different ways to handle the exception

1. OnErrorReturn operator

This will send the default value when the exceptoin is occurred

Flux.just("Apple","Mango")

.concatWith(Flux.error(

new RuntimeException("Exception Occurred")

))

.onErrorReturn("Orange")

1. OnErrorContinue

It will continue the opration and it will drop the element or drop the data is causing the error

Flux.just("Apple","Mango","Orange")

.map(s -> {

if (s.equalsIgnoreCase("Mango"))

throw new RuntimeException("Exception Occurred");

return s.toUpperCase();

})

.onErrorContinue((e,f) -> {

System.out.println("e = " + e);

System.out.println("f = " + f);

})

1. OnErrorMap

This is used for to use custom exception.

Flux.just("Apple","Mango","Orange")

.checkpoint("Error Checkpoint1")

.map(s -> {

if (s.equalsIgnoreCase("Mango"))

throw new RuntimeException("Exception Occurred");

return s.toUpperCase();

})

.checkpoint("Error Checkpoint2")

.onErrorMap(throwable -> {

System.out.println("throwable = " + throwable);

return new IllegalStateException("From onError Map");

})