**Spring Reactor Tutorial**

# Concepts

* **Publisher** - A *Publisher* is a provider of a potentially unbounded number of elements.
* **Subscriber** - A *Subscriber* listens to that Publisher, asking for new data. Sometimes, it's also referred to as a *Consumer*.
* **Backpressure** - The ability of the *Subscriber* to let the Publisher how many requests can it handle at the time. So it's the *Subscriber* that is responsible for the flow of the data, not the *Publisher* as it just provides the data.

The Reactor Project offers 2 types of publishers. These are considered the main building blocks of *Spring Webflux*:

* **Flux** - is a publisher that produces 0 to N values. It could be unbounded. Operations that return multiple elements use this type.
* **Mono** - is a publisher that produces 0 to 1 value. Operations that return a single element use this type.

**It's advised to use Flux and Mono over Publisher. Both of these classes are implementations of the Publisher interface originating from Reactive Streams.**

Mono and Flux are implementations of the Publisher interface. A Flux will observe 0 to N items and eventually terminate successfully or not. A Mono will observe 0 or 1 item, with Mono<Void> hinting at most 0 items.

## ****Backpressure****

The next thing we should consider is backpressure. In our example, the subscriber is telling the producer to push every single element at once. This could end up becoming overwhelming for the subscriber, consuming all of its resources.

**Backpressure is when a downstream can tell an upstream to send it fewer data in order to prevent it from being overwhelmed**.

## Class Flux<T>

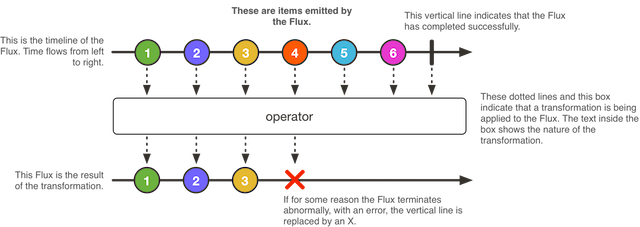
public abstract class **Flux<T>**

extends [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html?is-external=true)

implements [Publisher](http://www.reactive-streams.org/reactive-streams-1.0.2-javadoc/org/reactivestreams/Publisher.html?is-external=true)<T>

A Reactive Streams [Publisher](http://www.reactive-streams.org/reactive-streams-1.0.2-javadoc/org/reactivestreams/Publisher.html?is-external=true) with rx operators that emits 0 to N elements, and then completes (successfully or with an error).

The recommended way to learn about the [Flux](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html) API and discover new operators is through the reference documentation, rather than through this javadoc (as opposed to learning more about individual operators). See the ["which operator do I need?" appendix](http://projectreactor.io/docs/core/release/reference/docs/index.html#which-operator).



It is intended to be used in implementations and return types. Input parameters should keep using raw [Publisher](http://www.reactive-streams.org/reactive-streams-1.0.2-javadoc/org/reactivestreams/Publisher.html?is-external=true) as much as possible.

If it is known that the underlying [Publisher](http://www.reactive-streams.org/reactive-streams-1.0.2-javadoc/org/reactivestreams/Publisher.html?is-external=true) will emit 0 or 1 element, [Mono](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html) should be used instead.

Note that using state in the java.util.function / lambdas used within Flux operators should be avoided, as these may be shared between several [Subscribers](http://www.reactive-streams.org/reactive-streams-1.0.2-javadoc/org/reactivestreams/Subscriber.html?is-external=true).

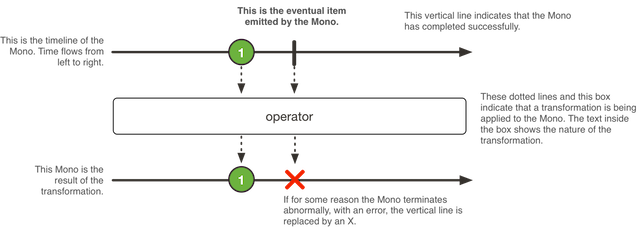
[subscribe(CoreSubscriber)](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#subscribe-reactor.core.CoreSubscriber-) is an internal extension to [subscribe(Subscriber)](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#subscribe-org.reactivestreams.Subscriber-) used internally for [Context](https://projectreactor.io/docs/core/release/api/reactor/util/context/Context.html) passing. User provided [Subscriber](http://www.reactive-streams.org/reactive-streams-1.0.2-javadoc/org/reactivestreams/Subscriber.html?is-external=true) may be passed to this "subscribe" extension but will loose the available per-subscribe @link Hooks#onLastOperator}.

## Class Mono<T>

public abstract class **Mono<T>** extends [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html?is-external=true) implements [Publisher](http://www.reactive-streams.org/reactive-streams-1.0.2-javadoc/org/reactivestreams/Publisher.html?is-external=true)<T>

A Reactive Streams [Publisher](http://www.reactive-streams.org/reactive-streams-1.0.2-javadoc/org/reactivestreams/Publisher.html?is-external=true) with basic rx operators that completes successfully by emitting an element, or with an error.

The recommended way to learn about the [Mono](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html) API and discover new operators is through the reference documentation, rather than through this javadoc (as opposed to learning more about individual operators). See the ["which operator do I need?" appendix](http://projectreactor.io/docs/core/release/reference/docs/index.html#which-operator).



The rx operators will offer aliases for input [Mono](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html) type to preserve the "at most one" property of the resulting [Mono](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html). For instance [flatMap](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html#flatMap-java.util.function.Function-) returns a [Mono](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html), while there is a [flatMapMany](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html#flatMapMany-java.util.function.Function-) alias with possibly more than 1 emission.

Mono<Void> should be used for [Publisher](http://www.reactive-streams.org/reactive-streams-1.0.2-javadoc/org/reactivestreams/Publisher.html?is-external=true) that just completes without any value.

It is intended to be used in implementations and return types, input parameters should keep using raw [Publisher](http://www.reactive-streams.org/reactive-streams-1.0.2-javadoc/org/reactivestreams/Publisher.html?is-external=true) as much as possible.

Note that using state in the java.util.function / lambdas used within Mono operators should be avoided, as these may be shared between several [Subscribers](http://www.reactive-streams.org/reactive-streams-1.0.2-javadoc/org/reactivestreams/Subscriber.html?is-external=true).

<https://stackoverflow.com/questions/1028250/what-is-functional-reactive-programming>

**What Is Reactive?**

If you want to get a feel for FRP, you could start with the old [Fran tutorial](http://conal.net/fran/tutorial.htm) from 1998, which has animated illustrations. For papers, start with [*Functional Reactive Animation*](http://conal.net/papers/icfp97/) and then follow up on links on the publications link on my home page and the [FRP](http://haskell.org/haskellwiki/FRP) link on the [Haskell wiki](http://haskell.org/haskellwiki/Haskell).

Personally, I like to think about what FRP *means* before addressing how it might be implemented. (Code without a specification is an answer without a question and thus "not even wrong".) So I don't describe FRP in representation/implementation terms as Thomas K does in another answer (graphs, nodes, edges, firing, execution, etc). There are many possible implementation styles, but no implementation says what FRP *is*.

I do resonate with Laurence G's simple description that FRP is about "datatypes that represent a value 'over time' ". Conventional imperative programming captures these dynamic values only indirectly, through state and mutations. The complete history (past, present, future) has no first class representation. Moreover, only *discretely evolving* values can be (indirectly) captured, since the imperative paradigm is temporally discrete. In contrast, FRP captures these evolving values *directly*and has no difficulty with *continuously* evolving values.

FRP is also unusual in that it is concurrent without running afoul of the theoretical & pragmatic rats' nest that plagues imperative concurrency. Semantically, FRP's concurrency is *fine-grained*, *determinate*, and *continuous*. (I'm talking about meaning, not implementation. An implementation may or may not involve concurrency or parallelism.) Semantic determinacy is very important for reasoning, both rigorous and informal. While concurrency adds enormous complexity to imperative programming (due to nondeterministic interleaving), it is effortless in FRP.

So, what is FRP? You could have invented it yourself. Start with these ideas:

* Dynamic/evolving values (i.e., values "over time") are first class values in themselves. You can define them and combine them, pass them into & out of functions. I called these things "behaviors".
* Behaviors are built up out of a few primitives, like constant (static) behaviors and time (like a clock), and then with sequential and parallel combination. *n* behaviors are combined by applying an n-ary function (on static values), "point-wise", i.e., continuously over time.
* To account for discrete phenomena, have another type (family) of "events", each of which has a stream (finite or infinite) of occurrences. Each occurrence has an associated time and value.
* To come up with the compositional vocabulary out of which all behaviors and events can be built, play with some examples. Keep deconstructing into pieces that are more general/simple.
* So that you know you're on solid ground, give the whole model a compositional foundation, using the technique of denotational semantics, which just means that (a) each type has a corresponding simple & precise mathematical type of "meanings", and (b) each primitive and operator has a simple & precise meaning as a function of the meanings of the constituents.*Never, ever* mix implementation considerations into your exploration process. If this description is gibberish to you, consult (a) [*Denotational design with type class morphisms*](http://conal.net/papers/type-class-morphisms), (b) [*Push-pull functional reactive programming*](http://conal.net/papers/push-pull-frp) (ignoring the implementation bits), and (c) the [*Denotational Semantics* Haskell wikibooks page](http://en.wikibooks.org/wiki/Haskell/Denotational_semantics). Beware that denotational semantics has two parts, from its two founders Christopher Strachey and Dana Scott: the easier & more useful Strachey part and the harder and less useful (for software design) Scott part.

If you stick with these principles, I expect you'll get something more-or-less in the spirit of FRP.

Where did I get these principles? In software design, I always ask the same question: "what does it mean?". Denotational semantics gave me a precise framework for this question, and one that fits my aesthetics (unlike operational or axiomatic semantics, both of which leave me unsatisfied). So I asked myself what is behavior? I soon realized that the temporally discrete nature of imperative computation is an accommodation to a particular style of *machine*, rather than a natural description of behavior itself. The simplest precise description of behavior I can think of is simply "function of (continuous) time", so that's my model. Delightfully, this model handles continuous, deterministic concurrency with ease and grace.

It's been quite a challenge to implement this model correctly and efficiently, but that's another story.

# Examples

**package** com.ddlab.rnd.core;

**import** java.time.Duration;

**import** java.time.Instant;

**import** java.util.function.Consumer;

**import** java.util.stream.Stream;

**import** reactor.core.publisher.Flux;

**import** reactor.core.publisher.Mono;

**import** reactor.core.scheduler.Schedulers;

**import** reactor.util.function.Tuple2;

**public** **class** Test1 {

**public** **static** Consumer<Integer> show() {

Consumer<Integer> consumer = e -> System.***out***.print(e + "\t");

**return** consumer;

}

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

Flux<Integer> flux = Flux.*just*(1, 2, 3, 4, 5);

flux.subscribe(e -> System.***out***.print(e + "\t"));

System.***out***.println("\n------------------------");

flux.subscribe(Test1.*show*());

Mono<String> emptyMono = Mono.*empty*();

emptyMono.subscribe(e -> System.***out***.println("emptyMono : " + e));

Mono<String> staticMono = Mono.*just*("e4developer");

staticMono.subscribe(e -> System.***out***.println("Static Mono : " + e));

Flux<Integer> emptyFlex = Flux.*empty*();

Flux<Integer> numbersOneToTen = Flux.*range*(1, 10);

numbersOneToTen.subscribe(e -> System.***out***.println("numbersOneToTen : " + e));

Flux<String> staticFlex = Flux.*just*("e4developer", "reactive", "reactor");

staticFlex.subscribe(e -> System.***out***.println("staticFlex : " + e));

Flux.*range*(1, 10)

.parallel(3)

.runOn(Schedulers.*parallel*())

.subscribe(i -> System.***out***.println(i));

Flux<String> stringFlux =

Flux.<String>*generate*(sink -> sink.next(**new** String("Hello")))

.take(10);

stringFlux.subscribe( e -> System.***out***.println("Generated Value =>"+e));

Flux<Long> delay = Flux.*interval*(Duration.*ofMillis*(5));

Flux<String> alphabetsWithDelay = Flux.*just*("A", "B").zipWith(delay, (s, l) -> s);

alphabetsWithDelay.subscribe( e -> System.***out***.print(e));

}

}

**package** com.ddlab.rnd.core;

**import** reactor.core.publisher.Flux;

**import** reactor.core.publisher.Mono;

**public** **class** Test2 {

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

Mono<String> mono1 = Mono.*just*("grokonez.com");

Mono<String> mono2 = Mono.*just*("|Java Technology");

Mono<String> mono3 = Mono.*just*("|Spring Framework");

System.***out***.println("=== Flux.concat(mono1, mono3, mono2) ===");

Flux.*concat*(mono1, mono3, mono2).subscribe(System.***out***::print);

System.***out***.println("\n=== combine the value of mono1 then mono2 then mono3 ===");

mono1.concatWith(mono2).concatWith(mono3).subscribe(System.***out***::print);

Flux<String> flux1 = Flux.*just*("A", "B}", "C", "D");

Flux<String> flux2 = Flux.*just*("E", "F", "G");

Flux.*zip*(flux2, flux1,

(itemFlux2, itemFlux1) -> "-[" + itemFlux2 + itemFlux1 + "]-")

.subscribe(System.***out***::print);//[EA]--[FB}]--[GC]-

}

}

# Spring WebFlux and Mono Example with Spring Boot

# Maven (pom.xml)

<project xmlns=*"http://maven.apache.org/POM/4.0.0"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>reactive-spring1</groupId>

<artifactId>reactive-spring1</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>reactive-spring1</name>

<url>http://maven.apache.org</url>

**<parent>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-parent</artifactId>**

**<version>2.1.1.RELEASE</version>**

**<relativePath />**

**</parent>**

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-webflux</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-test</artifactId>**

**<scope>test</scope>**

**</dependency>**

**<dependency>**

**<groupId>io.projectreactor</groupId>**

**<artifactId>reactor-test</artifactId>**

**<scope>test</scope>**

**</dependency>**

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

# AppConfig.java

**package** com.ddlab.rnd;

**import** org.springframework.context.annotation.ComponentScan;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.web.reactive.config.EnableWebFlux;

@EnableWebFlux

@Configuration

@ComponentScan("com.ddlab.rnd")

**public** **class** AppConfig {}

# Movie.java

**package** com.ddlab.rnd;

**public** **class** Movie {

**private** String name;

**private** **int** rating;

**public** Movie(String name, **int** rating) {

**super**();

**this**.name = name;

**this**.rating = rating;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getRating() {

**return** rating;

}

**public** **void** setRating(**int** rating) {

**this**.rating = rating;

}

@Override

**public** String toString() {

**return** "Movie [name=" + name + ", rating=" + rating + "]";

}

}

# AppController.java

**package** com.ddlab.rnd;

**import** java.time.Duration;

**import** java.util.ArrayList;

**import** java.util.List;

**import** org.springframework.http.MediaType;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.PathVariable;

**import** org.springframework.web.bind.annotation.RestController;

**import** reactor.core.publisher.Flux;

**import** reactor.core.publisher.Mono;

@RestController

**public** **class** AppController {

@GetMapping("/movie/{id}")

**public** **Mono<Movie> getMovieById**(@PathVariable("id") Integer id) {

**return** Mono.*just*(**new** Movie("Polar (2019)", 64));

}

// @GetMapping(

// path = "/allbooks11",

// produces = {MediaType.TEXT\_EVENT\_STREAM\_VALUE})

@GetMapping(

path = "/allMovies",

produces = {MediaType.***APPLICATION\_STREAM\_JSON\_VALUE***})

**public** **Flux<Movie> getAllBooks11**() {

List<Movie> movie = **new** ArrayList<>();

movie.add(**new** Movie("Polar (2019)", 64));

movie.add(**new** Movie("Iron Man (2008)", 79));

movie.add(**new** Movie("The Shawshank Redemption (1994)", 93));

movie.add(**new** Movie("Forrest Gump (1994)", 83));

movie.add(**new** Movie("Glass (2019)", 70));

**return** Flux.*fromIterable*(movie).delayElements(Duration.*ofSeconds*(2));

}

}

# Application.java

**package** com.ddlab.rnd;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

**public** **class** Application {

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

SpringApplication.*run*(Application.**class**, args);

}

}

**Note: server.servlet.context-path=/react 🡸 This property does not work in case of webflux as per Spring issues and documentation.**

**Access the following URLs:**

**GET** [**http://localhost:8080/allMovies**](http://localhost:8080/allMovies)

**GET http://localhost:8080/movie/13**