**Evolution of Programming:**

**Past (10-15) years ago:**

* Monolithic Applications
* Applications are Deployed in Application Server
* Does not embrace distributed systems

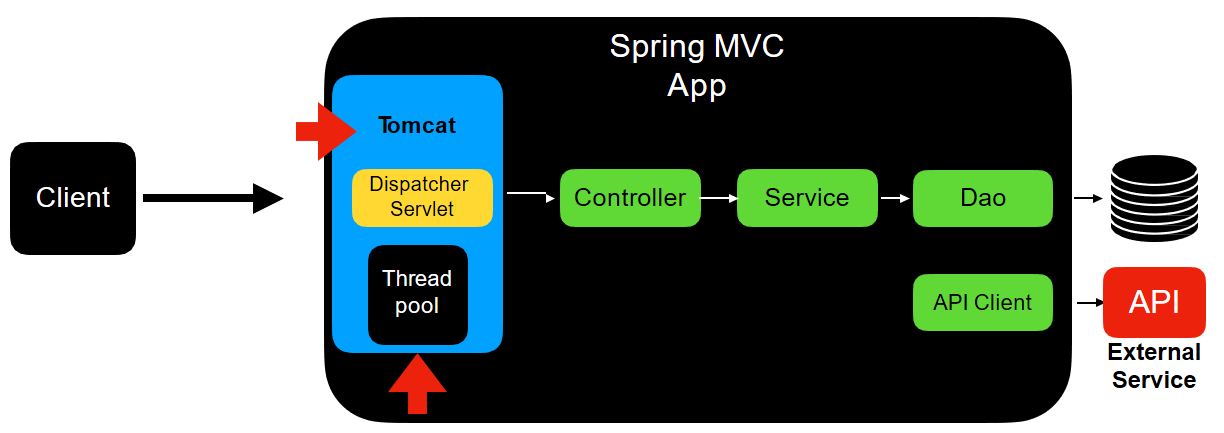
**Current:**

* Micro services Applications
* Applications are deployed in Cloud Environments
* Embraces Distributed Systems

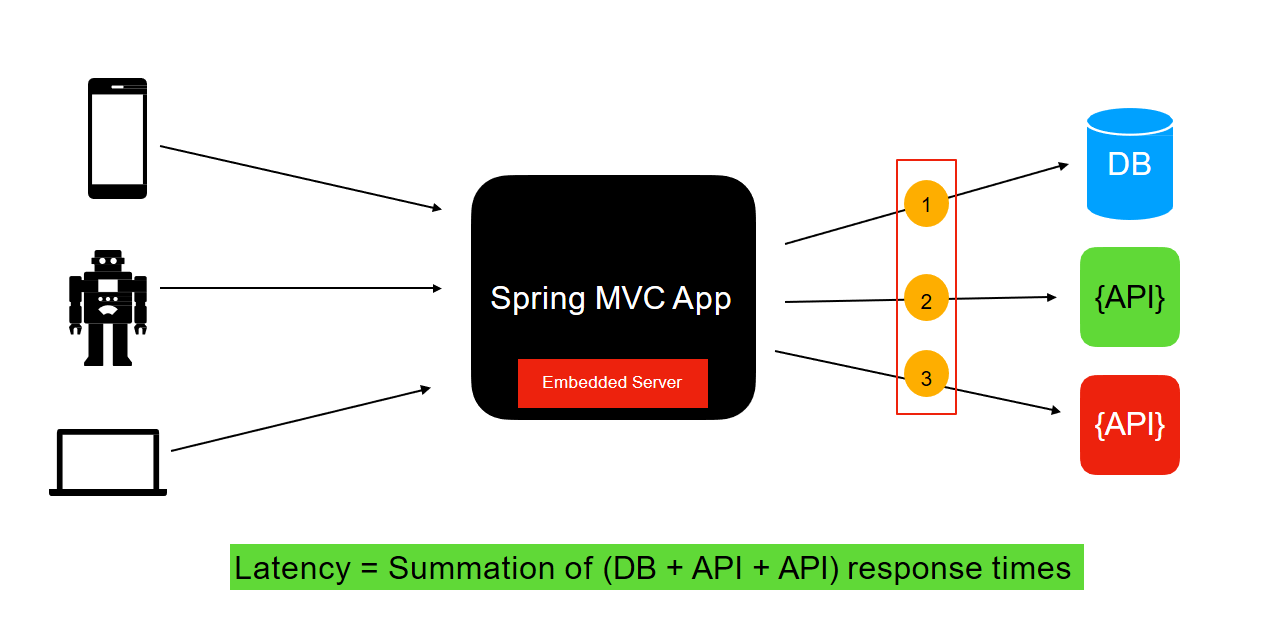
**Expectations of the Application:**

* Response times are expected in milliseconds
* No downtime is expected
* Scale up automatically based on the load

**Restful API using Spring Boot/MVC:**



* When client makes a call, a thread from thread pool is assigned to the request and the assigned thread is responsible for handling the request and sending the response to the client.
* So, the concurrency is **Thread per Request** model.
* The Apps developed today interact with multiple data sources like database, 3rd party apps, etc. in order to fulfil the client’s request. This type of API’s are called **Blocking APIs**.
* At the embedded Tomcat server level, the dispatcher servlet involved in dispatching the request to the appropriate controller is itself a blocking API by its nature.
* The thread been assigned to the client is blocked until the response is being returned.
* This Thread per request model wont scale today’s application needs. For example, let’s see the architecture shown below.



* Let’s say we have an app as shown above which interacts with three different data sources. When you have applications built in this style, you will have multiple blocking points.
* If we have an API that calls these three data sources synchronously, then

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

**Spring MVC/ Boot Limitations:**

* Thread pool size of embedded tomcat in Spring MVC’s is **200**.
* Though we can increase the thread pool size based on the thread, it is only to a certain limit.
* Let’s say you have a use case to support **10000**concurrent users.
* In that can we cannot create a thread pool of **10000** threads.

**Thread and its Limitations**:

* Thread is expensive resource.
* It can easily take up to 1MB of heap space.
* More threads means more memory consumption by the thread itself.
* Less heap space for actually processing the request.
* This might impact overall performance of the application.

**Asynchronous Options in Java:**

* In order to reduce the latency, we can call data sources asynchronously.
* The two options available are **callbacks** and **futures**.
* **Callbacks:** Asynchronous methods that accepts 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
* **Future**: it is released in Java 5. It helps in writing asynchronous Code

Disadvantages:

* No easy way to combine the result from multiple futures
* Future.get() is used to get the result from each datasource. So This results in a blocking call again.
* **CompletableFuture:** It is released in Java8. Helps Write Asynchronous code in a functional style. Easy to compose/combine MultipleFutures

Disadvantages:

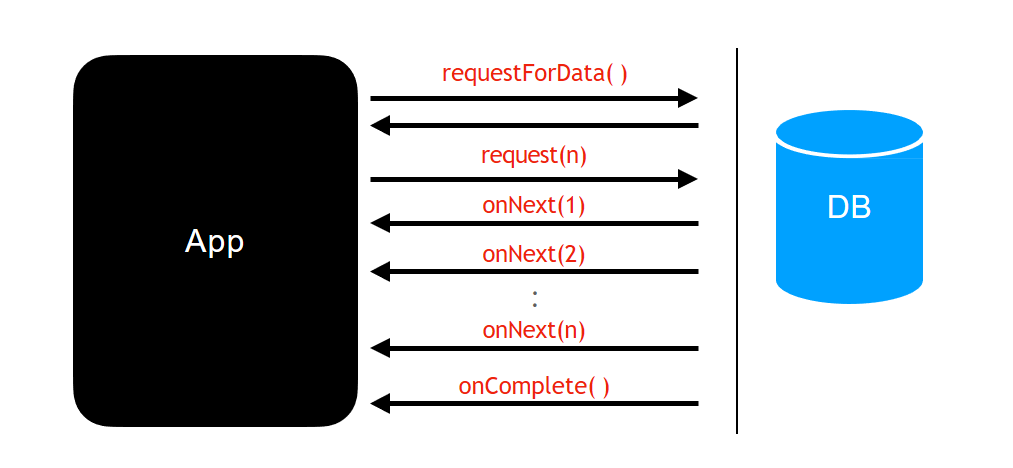
* Future that returns many elements. Eg., **CompletableFuture<List<Result>** will need to wait for the whole collection to built and readily available
* CompletableFuture does not have a handle for infinite values

**Drawbacks of Spring MVC/Boot:**

* Concurrency is limited in Spring MVC
* Blocking code leads to ineﬃcient usage of threads.
* Servlet API at the server level is a blocking one

**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



* Here, we have an app and a DB. The app makes a request for data from the DB.
* As soon as a call is received by the DB, the calling thread returns immediately.
* This is not a blocking call anymore
* Calling thread is release to do some useful work
* There will be another call that will be sent from the app behind the scenes to the DB requesting for the data providing number of data it required. This is a signal to the DB from the app that it is ready to consume the data.
* Once the query results are ready, then the data will be sent in the form of stream of events. This is called a reactive stream.
* The actual data will be sent using the onNext () function to the app. Once all the results are sent to the app, the DB sends complete event which signals the app that there is no more data and the event will be sent using the onComplete() function.
* All these callbacks, back and forth communication is taken care by the reactive library itself.

When to use Reactive Programming?

* Use reactive programming when there is need to build and support high load with available resource.

Reactive App Architecture:

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

**Reactive Streams:**

* Reactive Streams Specification is created by engineers from multiple organizations like Lighbend, Netflix and VmWare (Pivotal).
* Reactive Stream Specification has 4 interfaces:

1. Publisher
2. Subscriber
3. Subscription
4. Processor
5. **Publisher:**

* The publisher interface has just one method named subscribe and it takes parameter named Subscriber.

**public interface Publisher<T>**

**{**

**public void subscribe(Subscribe<? Super T> s);**

**}**

* Publisher represents the **Data Source**. It can be database, Remote Server or any other external system etc.,

1. **Subscriber:**

* The Subscriber interface has 4 methods

**public interface Subscriber<T> {**

**public void onSubscribe(Subscription s);**

**public void onNext(T t);**

**public void onError(Throwable t);**

**public void onComplete();**

**}**

* Subscriber is our App which requests for data

1. **Subscription:**

* The Subscriber interface has two methods.

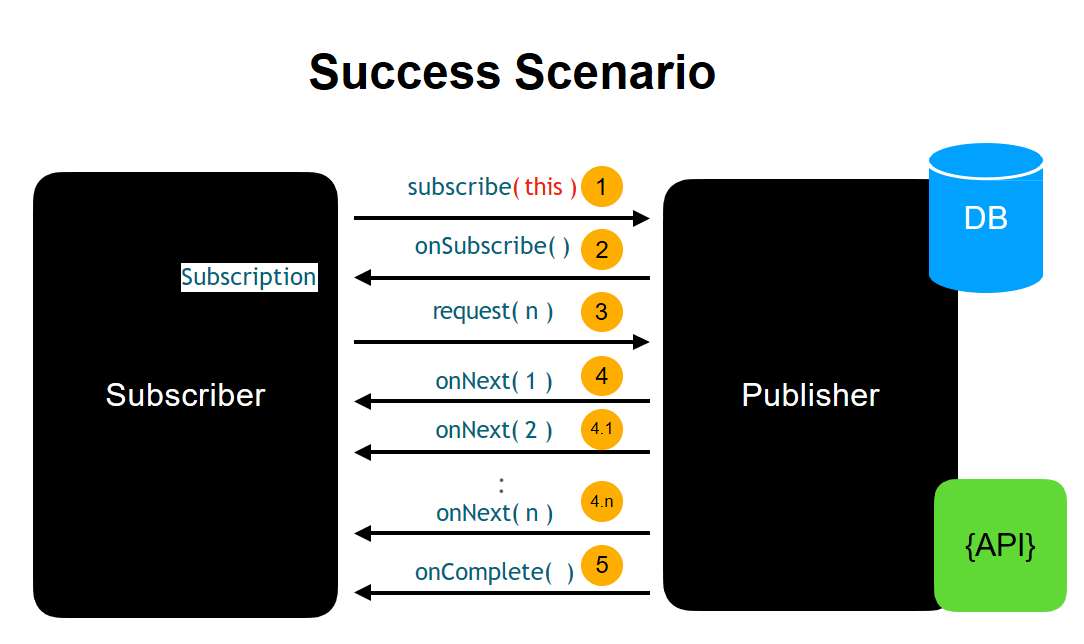
**public interface Subscription {**

**public void request(long n);**

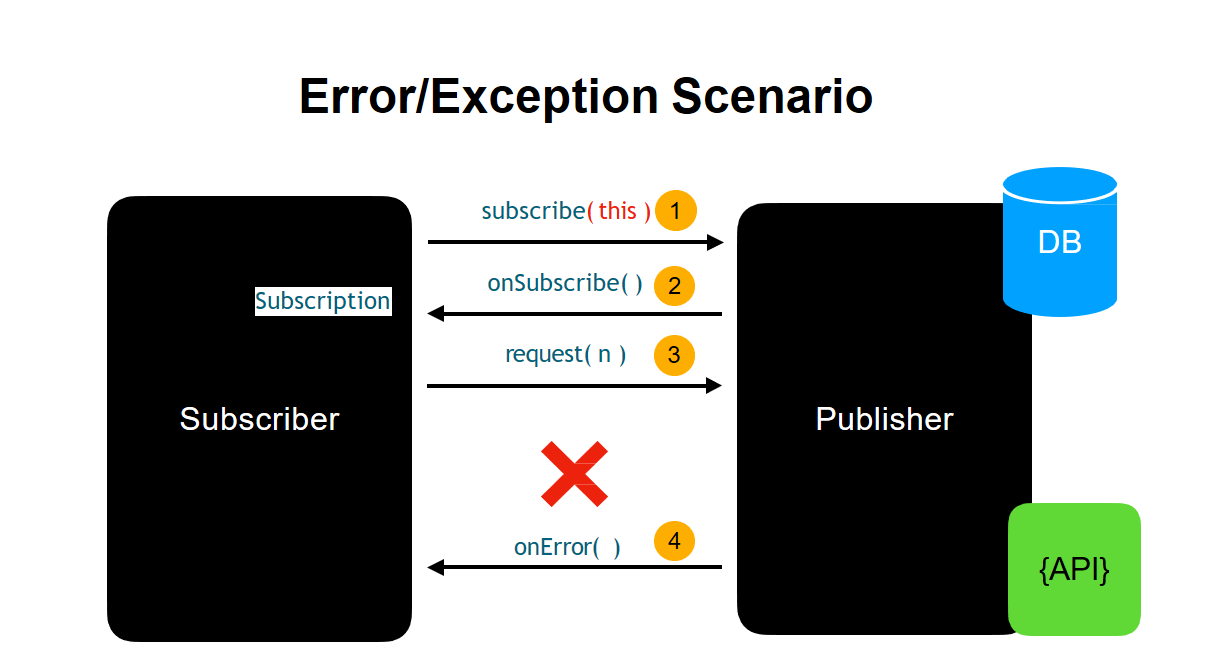
**public void cancel();**

**}**

* Subscription is the one which connects the app and data source. In other words it connects Subscriber and Publisher.



* The whole flow starts with the Subscriber initiating the request by invoking the subscribe method of the Publisher.
* The second step is Publisher handing out the Subscription object by invoking the onSubscribe() method of the Subscriber.
* Once the Subscriber has got Subscription object, the Subscriber is going to invoke the request() function of the Subscription object requesting the publisher to send all the data.
* Once a publisher receives a request to send the data, then it sends a data using the onNext() function that’s part of the Subscriber interface.
* Once all the data is sent, the last step is Publisher sending an on complete event by invoking the onComplete() function that’s part of the Subscriber.



* Nothing changes until request () call to the Publisher.
* Let’s say the code ran into a runtime exception, it could be because the DB is temporarily down or some run time exception in your code.
* In this case, the exception will be sent as an event using onError () method in the Subscriber.
* Exceptions are treated like data in reactive programming.
* The reactive stream is basically dead when an exception is thrown, which means there is no way data can flow in the same Subscription object when an error is thrown.

1. **Processor:**

* The Processor extends the Publisher and Subscriber.
* Processor can behave as a Subscriber and Publisher

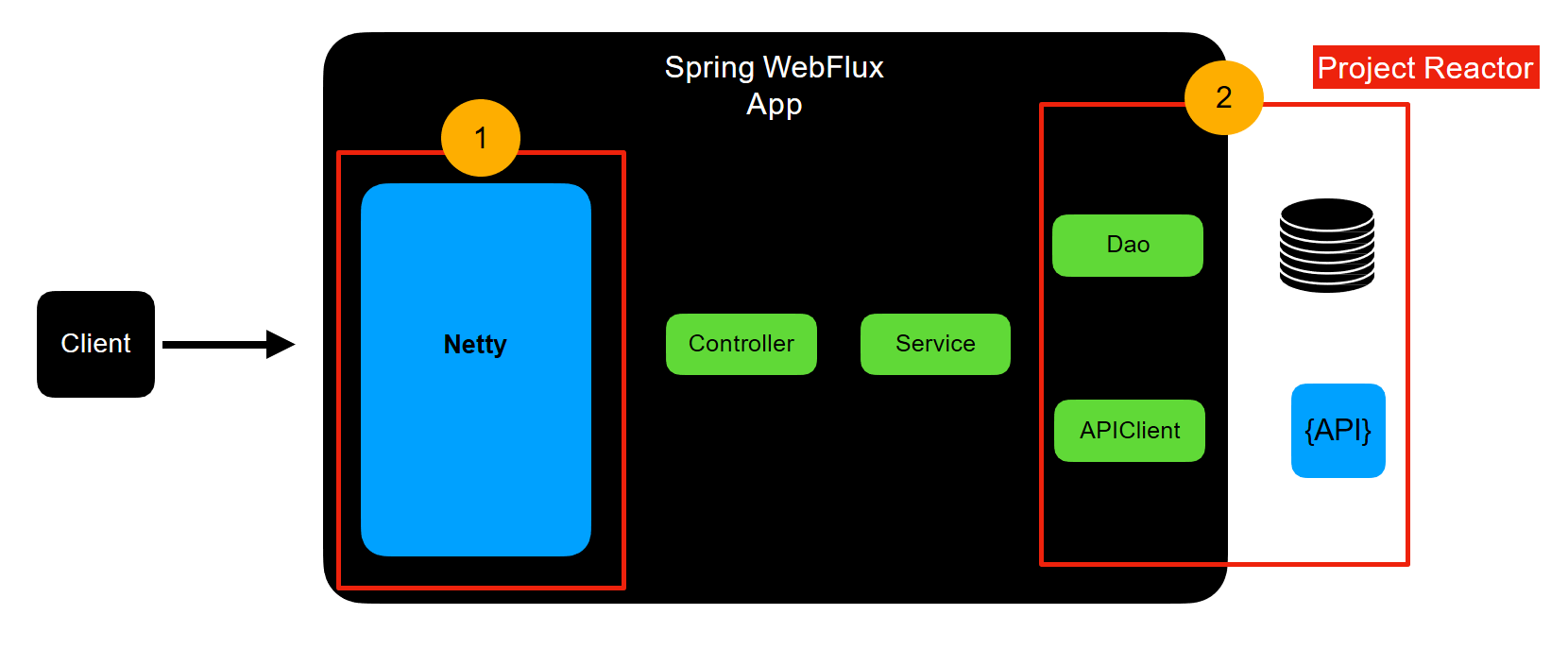
**public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {**

**}**

**What is aNon-blocking or ReactiveRestFul API?**

* A Non-Blocking or Reactive RestFul API has the behavior of providing end to end non-blocking communication between the client and service
* Non-Blocking or Reactive == Not Blocking the thread
* Thread involved in handling the **httprequest**and **httpresponse**is not blocked at all
* **Spring WebFlux**is a module that’s going to help us in achieving the **Non- Blocking** or **Reactive** behavior

**NonBlocking or Reactive API using Spring WebFlux:**

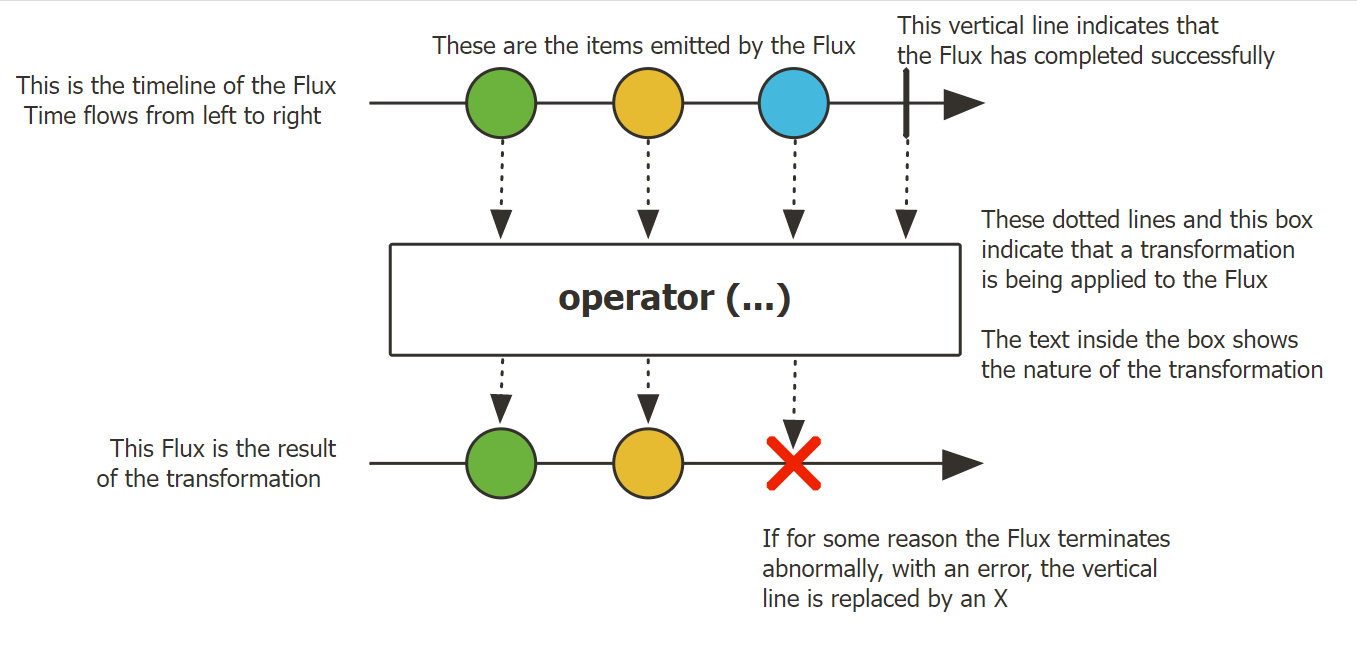


* Anytime you build spring webflux app with the Spring Boot, it’s going to come up with inbuilt server. The default embedded server is netty.
* Netty is a popular runtime that handles non-blocking IO for you behind the scenes without blocking threads.
* Project reactor is a reactive library which is the implementation of reactive streams specification. It is used to enable non-blocking interactions with the external systems.

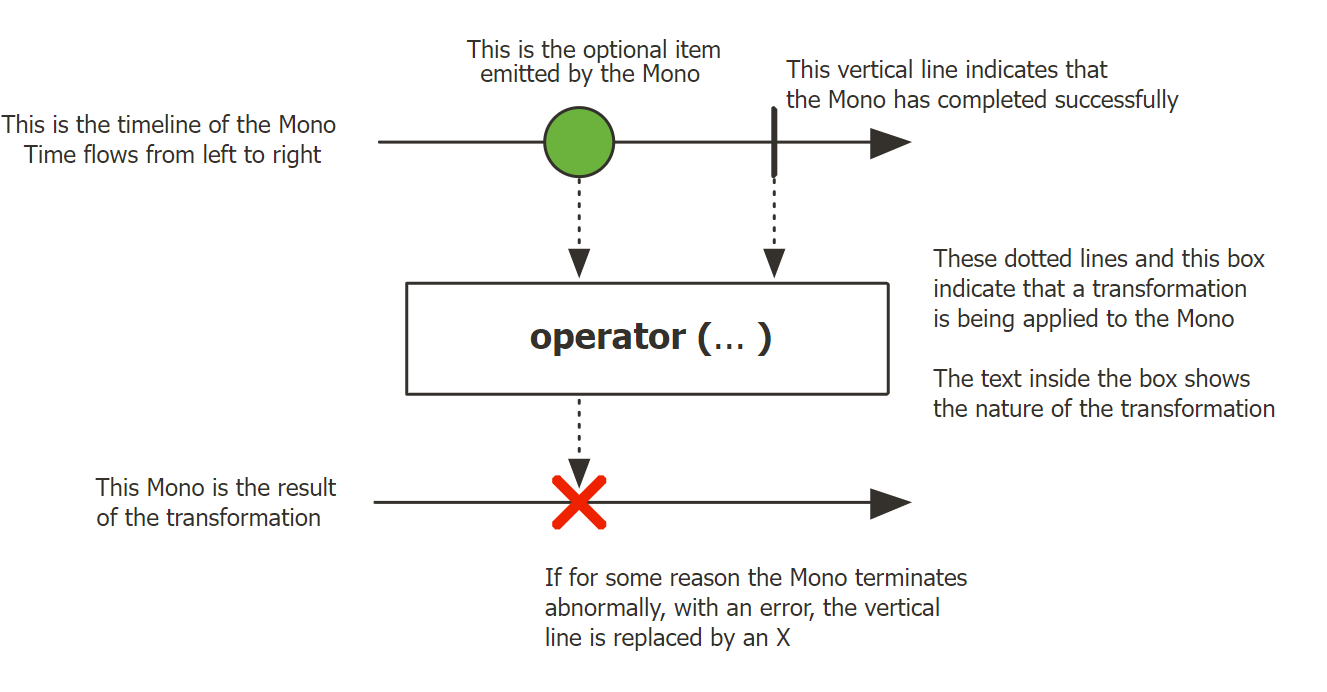
**Flux & Mono:**

* Flux and Mono is a reactive type that implements the Reactive Streams Specification.
* Flux and Mono are classes that are 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 elements:**



* Reactive Streams are immutable

flatMap():

* Transforms one source element to a Flux of 1 to N elements or Mono.
* Use it when the transformation returns a Reactive Type (Flux or Mono)
* Returns a Flux<Type> or Mono<Type>
* Use flatMap if the transformation involves making a REST API call or any kind of functionality that can be done asynchronously.

|  |  |
| --- | --- |
| **map()** | **flatMap()** |
| One to one transformation | One to N Transformations |
| Does the simple transformation from **T** to **V** | 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 simple synchronous transformations | Used for asynchronous transformations |
| Does not support transformations that returns Publisher | Use it with transformation that returns Publisher |

concatMap ():

* Works similar to flatMap()
* Only difference is that concatMap() preserves the ordering sequence of the Reactive Streams.
* It takes more time compared to flatMap()
* So, Use concatMap() if order matters.

flatMapMany():

* Works very similar to flatMap().
* Transform the item emitted by this [Mono](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html) into a Publisher, then forward its emissions into the returned [Flux](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html).

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)

defaultEmpty() &switchEmpty():

* It is not mandatory for a data source to emit data all the time
* We can use the defaultEmpty() or switchEmpty() operator to provide default values

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() and mergeWith():

* Works same as concat() and concatWith()
* Both the publishers are subscribed at the same time
* Publishers are subscribed eagerly and the merge happens in an interleaved fashion
* Whereasconcat() 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():

* 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():**
* 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 onComplete event

**Streaming Endpoint:**

* Streaming Endpoint is a kind of Endpoint which continuously sends updates to the clients as the new data arrives.
* This concept is similar to Server Sent Events(SSE)
* **Examples**: Stock Tickers, Real time updates of Sports Events

**ResponseEntity in spring webFlux:**

* ResponseEntity is like @ResponseBody but with status and headers.
* WebFlux supports using a single value [reactive type](https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#webflux-reactive-libraries) (Flux or Mono) to produce the ResponseEntity asynchronously, and/or single and multi-value reactive types for the body.
* This allows a variety of async responses with ResponseEntity as follows:
* **ResponseEntity<Mono<T>> or ResponseEntity<Flux<T>>** make the response status and headers known immediately while the body is provided asynchronously at a later point. Use Mono if the body consists of 0..1 values or Flux if it can produce multiple values.
* **Mono<ResponseEntity<T>>** provides all three — response status, headers, and body, asynchronously at a later point. This allows the response status and headers to vary depending on the outcome of asynchronous request handling.
* **Mono<ResponseEntity<Mono<T>>> or Mono<ResponseEntity<Flux<T>>>** are yet another possible, albeit less common alternative. They provide the response status and headers asynchronously first and then the response body, also asynchronously, second.