What's Covered ?

- Introduction to Reactive Programming
- Advantages of Reactive Programming over traditional programming models
- Writing Reactive Programming code using **Project Reactor**
- Introduction **Spring WebFlux**
- Reactive Services using Spring WebFlux
  - **Three Reactive Rest Services** using Spring WebFlux
- JUnit test cases using **JUnit5**

Prerequisites:

- Knowledge of Spring Boot.
- Experience building RESTFUL APIs
- Knowledge of Java8 Lambdas, Streams.
- At least JDK 8 is needed.
- IntelliJ or any IDE (Eclipse, NetBeans etc.,)

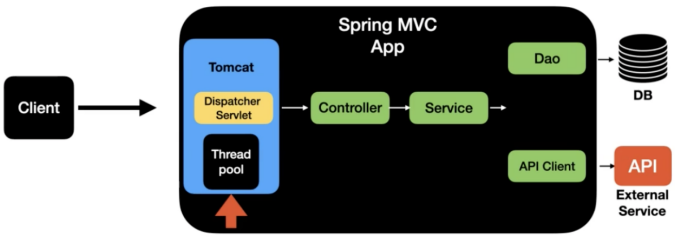
https://github.com/dilipsundarraj1/reactive-spring-webflux/tree/final

Why Reactive Programming ?

Expectations of the Application

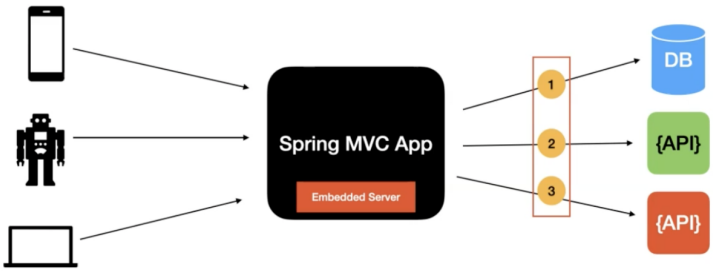
- Response times are expected in milliseconds
- No Downtime is expected
- Scale up automatically based on the load

Restful API using Spring Boot/MVC



- Concurrency is **Thread Per Request** model
- This style of building APIs are called **Blocking APIs**
- Wont scale for today's application needs

Restful API using Spring Boot/MVC



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

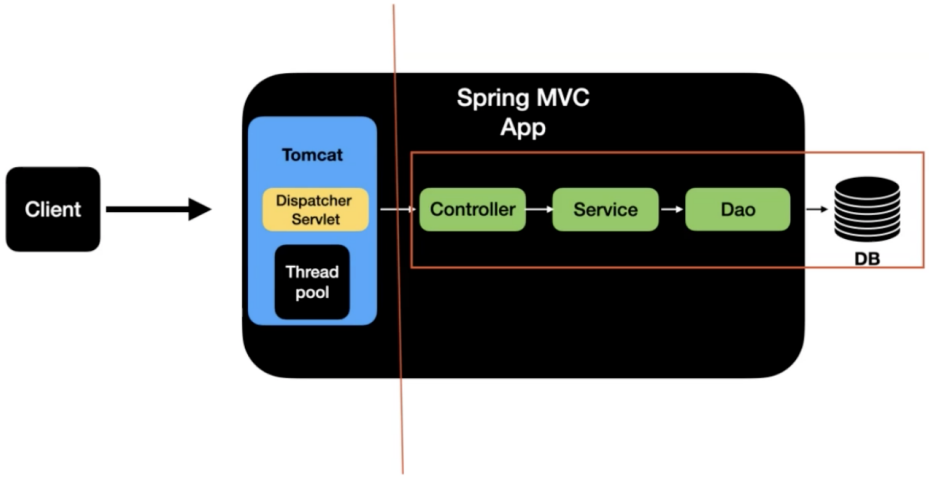
Spring MVC Limitations

- Thread pool size of Embedded tomcat in Spring MVC's is **200**
- Can we increase the thread pool size based on the need ?
  - Yes, only to a certain limit.
- Let's say you have a use case to support **10000** concurrent users.
  - Can we create a thread pool of size **10000** Threads ?
    - No

Thread and its Limitations

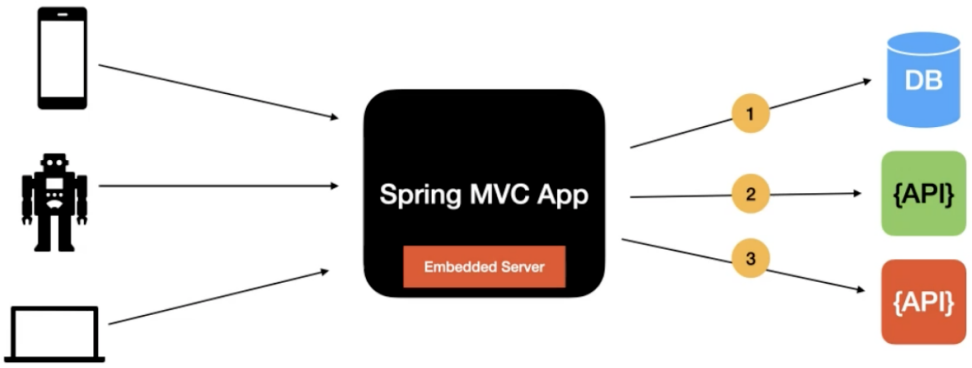
- Thread is an expensive resource
- It can easily take up to 1MB of heap space
- More threads means more memory consumption by the thread itself
- Less heap space for actually processing the request

Restful API using Spring Boot/MVC



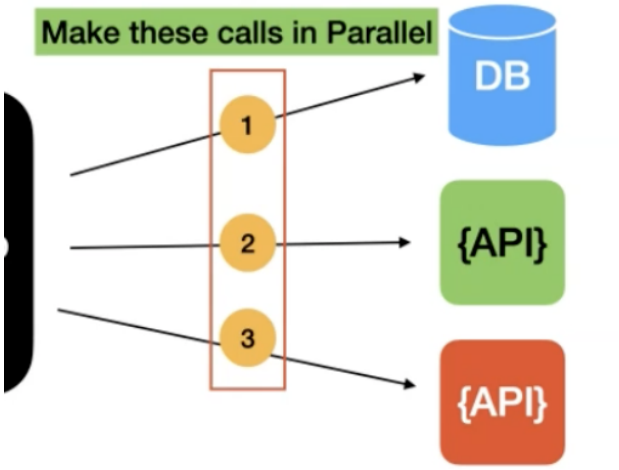
From the above pic mvc is Blocking Nature.

Restful API using Spring Boot/MVC



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

To overcome this summation of db + api + api



Lets explore the asynchrony options in Java

- Callbacks
- Futures

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**

Concurrency APIs in Java

Future	CompletableFuture

• 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
	• 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

Drawbacks of Spring MVC

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

Is there a better option available?

Reactive programming to the rescue

Summary

- Does this mean we should stop using Spring MVC?
  - No
  - This still works very well for many use cases

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
- Calling thread is released to do useful work

request (n) : Another call sent from behind the scene requesting for the data. Signal for DB from app that app is ready to consume the data. This is not a redundant call, there is a reason for it.

Whatever data we receive is in the form of a stream. on Next function will be called for this purpose. Once data completed onComplete() will be executed.

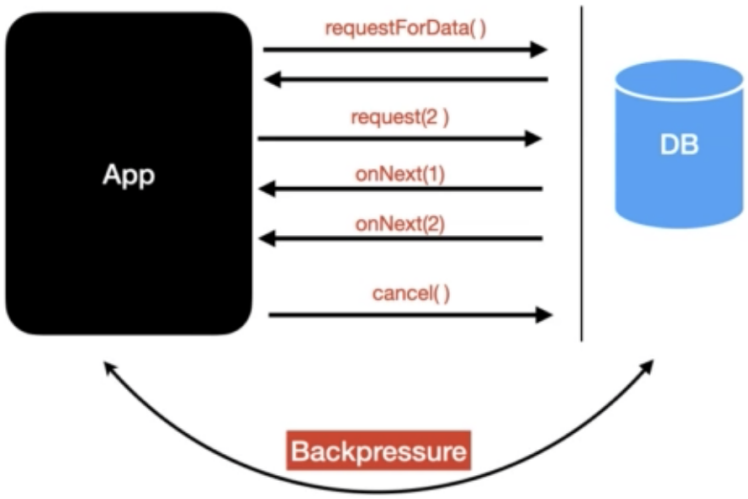
Push Based data streams model

These streams are different from java streams. These streams are reactive streams.

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



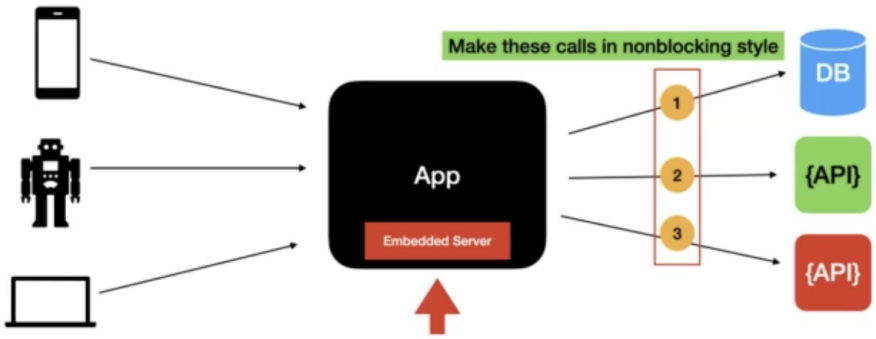
During cancel it may request for cancel or request for additional data
So It is not only a push-based data flow model. So reactive programming is a combination of push-pull based data flow model.

When to use Reactive Programming ?

Use Reactive Programming when there is need to build and support high load with the available resources

App that needs to support 400 TPS

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 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

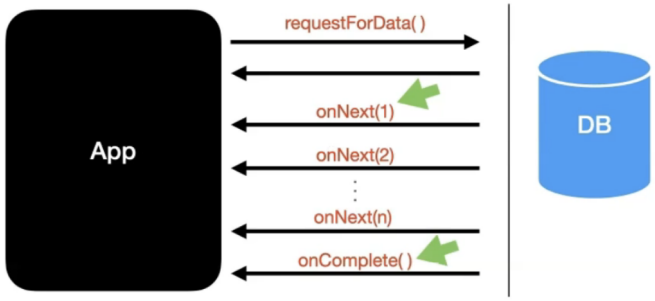
```
public interface Publisher<T> {

    public void subscribe(Subscriber<? super T> s); 1
}
```

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

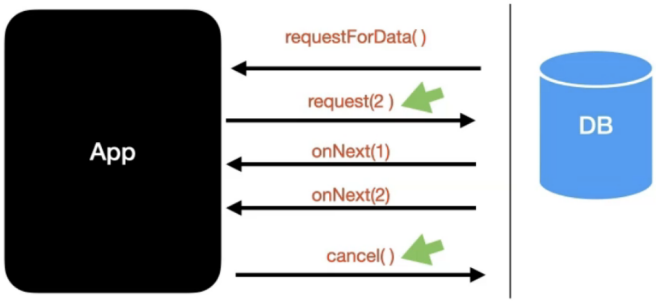
Subscriber

```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```



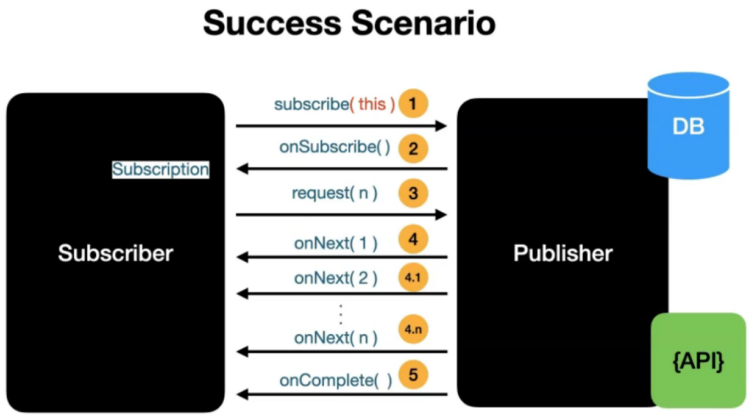
Subscription

```
public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

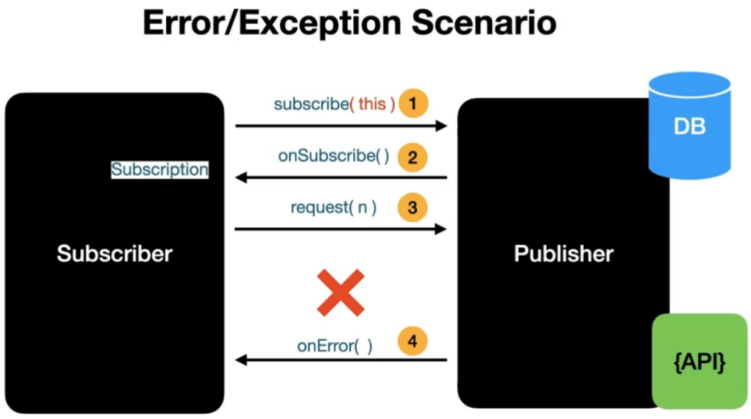


Subscription is the one which connects the app and datasource

Reactive Streams - How it works together ?



Reactive Streams - How it works together ?



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

Exceptions are treated as events not as exceptions.

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

Flow API

- Release as part of Java 9
- This holds the contract for reactive streams but no implementation is available as part of JRE

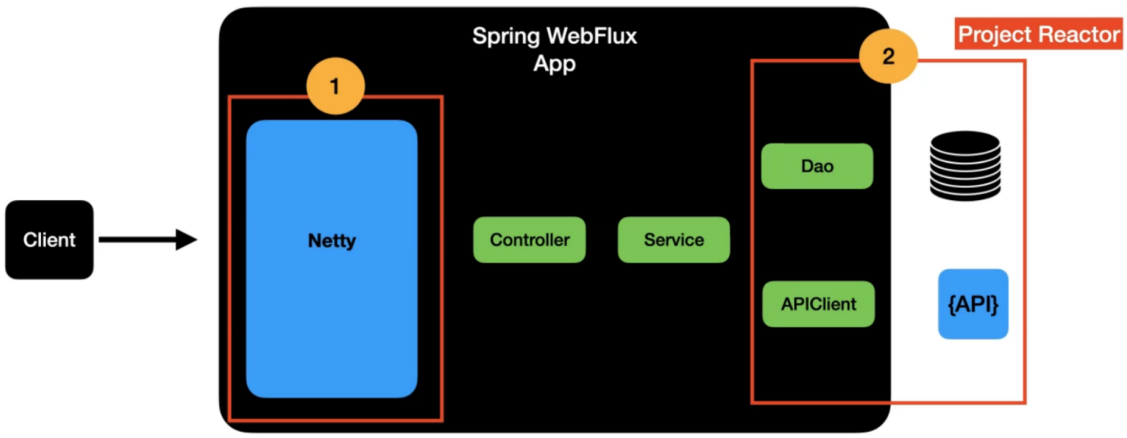
Implementation is still in the hands of the reactive library.

What is a Nonblocking or Reactive RestFul API ?

NonBlocking or Reactive RestFul 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



Project Reactor

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

Google Project Reactor - and click the site. https://projectreactor.io/

Click on documentation : it will tell us what are different modules we have

https://projectreactor.io/docs

There are different modules our focus is on Reactor Core / Reactor Test

Google Project Reactor reference guide

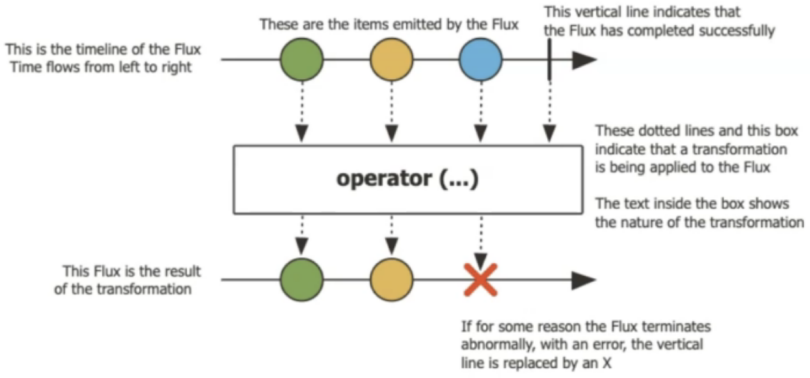
https://projectreactor.io/docs/core/release/reference/

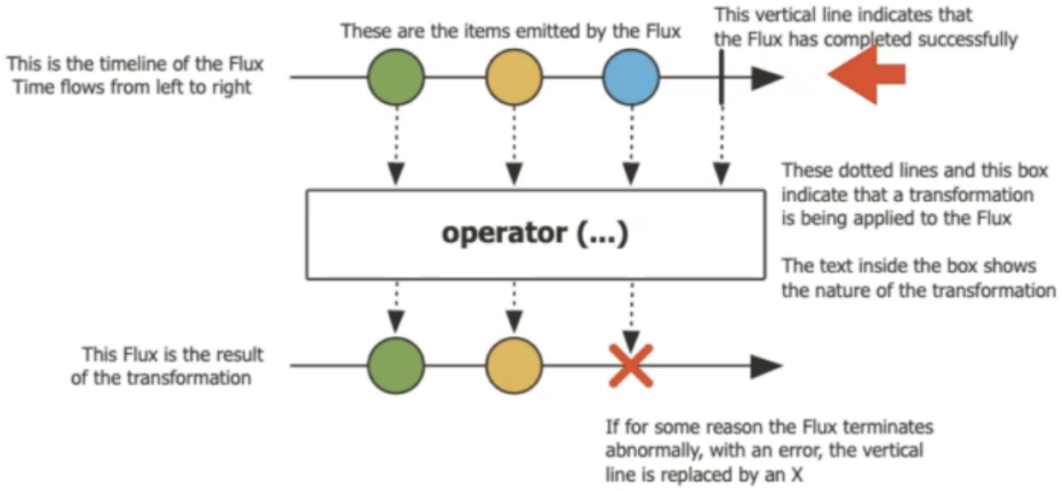
It is guide for on top of our work

Flux & Mono

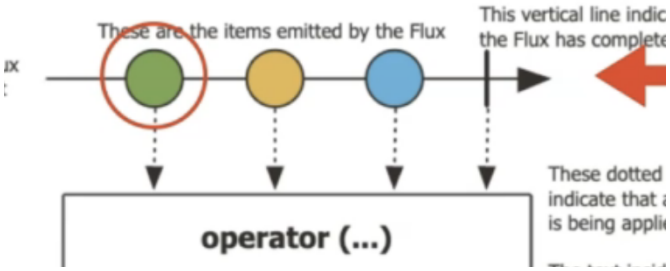
- Flux and Mono is a reactive type that implements the Reactive Streams Specification
- Flux and Mono is part of the **reactor-core** module
- Flux is a reactive type to represent 0 to N elements
- Mono is a reactive type to represent 0 to 1 element

Flux - 0 to N elements

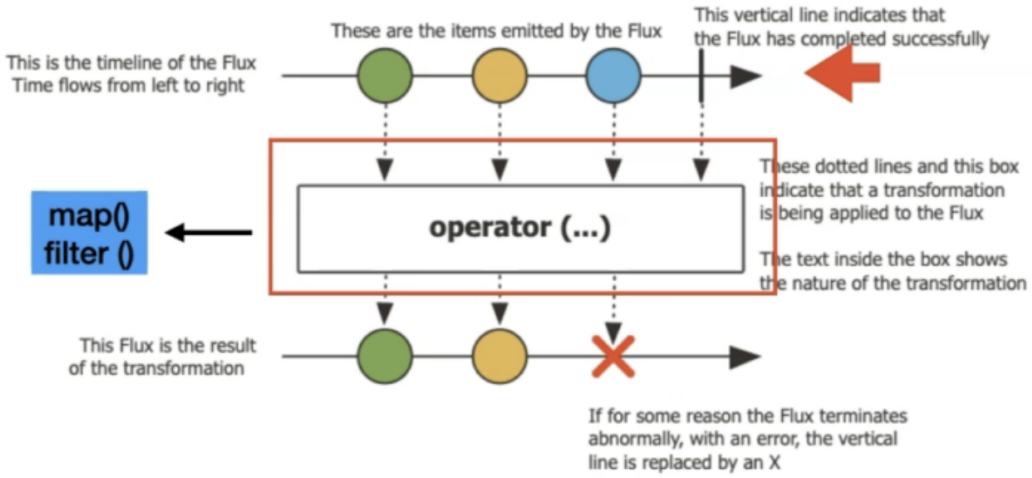




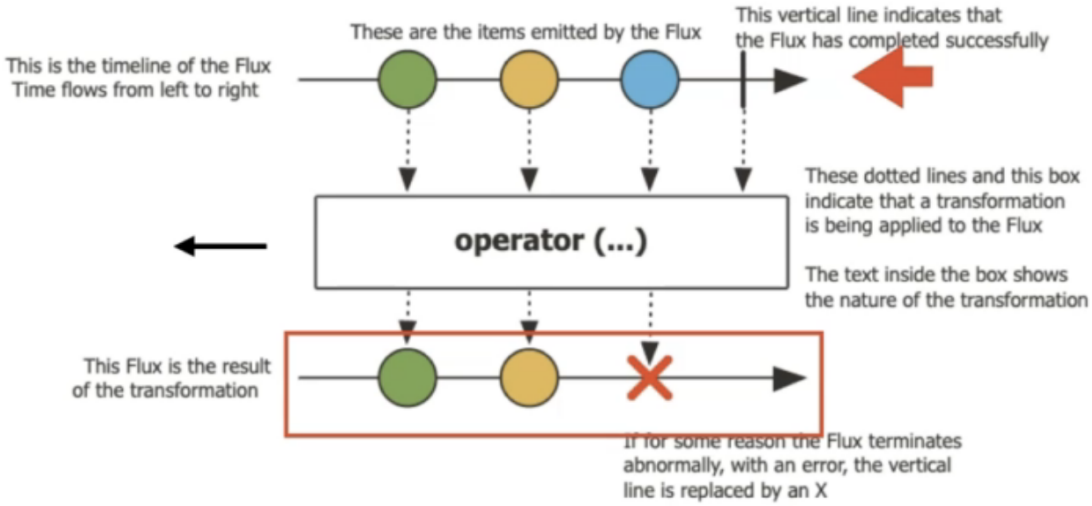
We have multiple marbles on the top represent datasource



Individual marbles represent data that will be sent using onNext operator in the subscriber

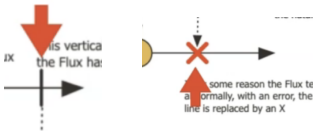


Operator layer used for data transformation on data we received. There are lot more for data transformation in addition to map and filter.



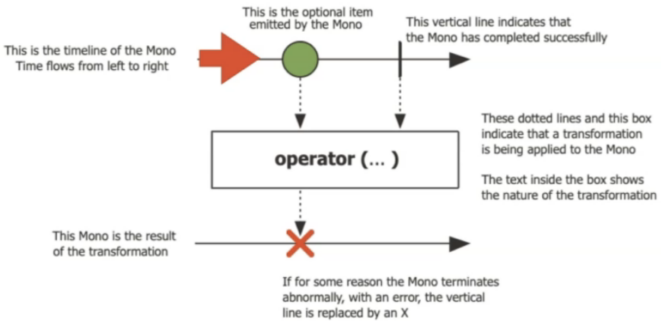
Result of transformation. Final result handover to subscriber.

Vertical Bar present no more data and on complete signal to subscriber



The cross represents some exception or error. onError signal sent to subscriber.

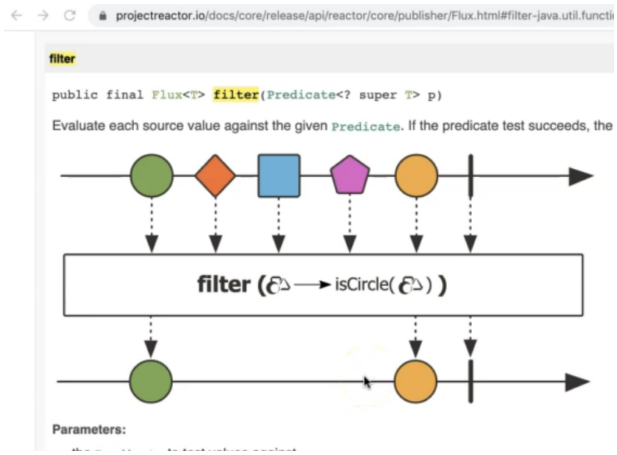
Mono - 0 to 1 Element



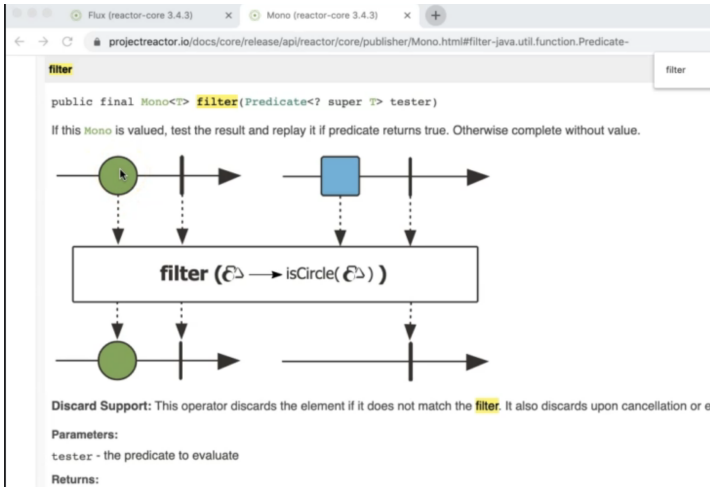
In Mono we have only single marble (data)

Google Flux Reactor :
https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html

In the above page somewhere we see operators by clicking the link we find a detailed explanation like below with a marble diagram. This one is for operator filter.

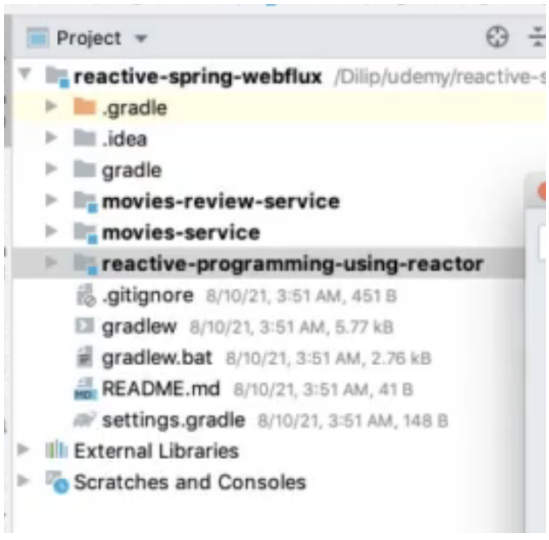


Search for mono Reactor:
https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html

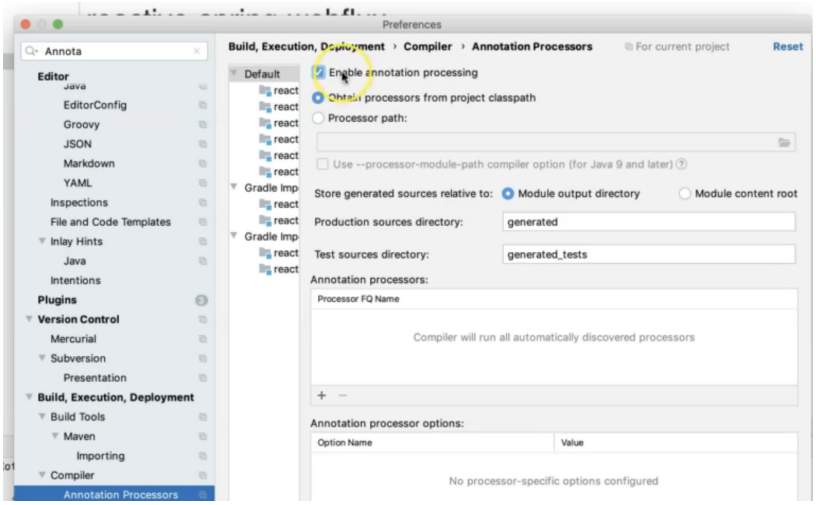


Setting up the Project For this Course

https://github.com/dilipsundarraj1/reactive-spring-webflux
Main branch is for beginning of the code
Final branch is for end of the code



Enable annotation processor in preference window



And click apply

plugins : Spring Assistance