Reactor 3

the Reactive foundation for the JVM

About me

&

how to get in touch

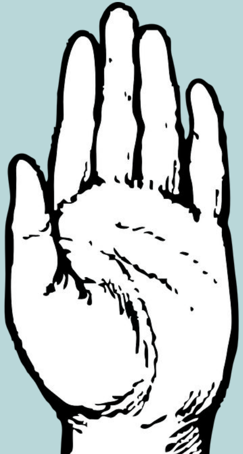
Pivotal

@SimonBasle

First

a

Survey



Who Here Uses...

Who Here Uses…

Java 8

Who Here Uses...

Java 8

RxJava

Who Here Uses...

Java 8

RxJava

Reactive Streams

Who Here Thinks...

Who Here Thinks...

Reactor is a fork of Atom editor

Who Here Thinks…

Reactor is *a* fork *of* Atom *editor*

Reactor is *a new* Spring *project*

Who Here Thinks...

Reactor is *a* fork *of* Atom *editor*

Reactor is *a new* Spring *project*

Reactor is *some* Asynchronous *stuff*

Who Here Thinks…

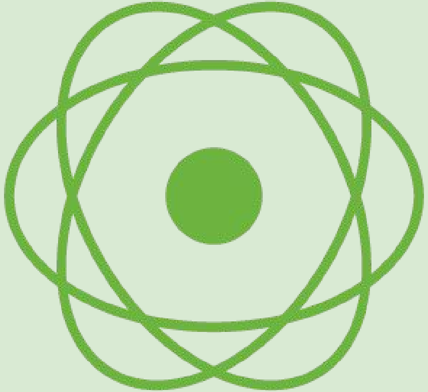
Reactor is *a* fork *of* Atom *editor*

Reactor is *a new* Spring *project*

Reactor is *some* Asynchronous *stuff*

end-of-day conf slot is best for naps?

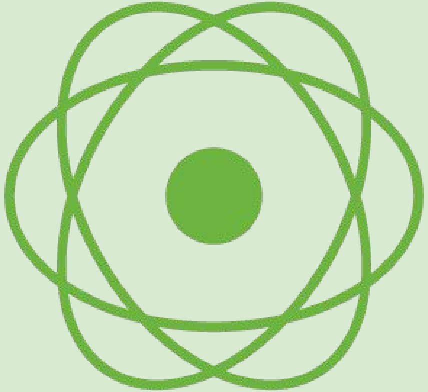
the Agenda



Reactive

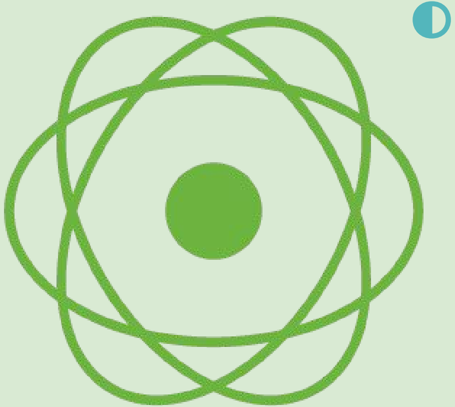
Programming

101



Reactive
Programming
101

Reactor 3
types & operators



Reactive
Programming
101

Reactor 3
types &
operators

backpressure
and
other beasts

Reactive
Programming
101

Reactor 3
types &
operators

backpressure
and
other beasts

testing
and
debugging

Reactive Programming **101**

Reactor 3 types & operators

Reactor and **Spring**

backpressure and **other beasts**

testing and **debugging**

Reactive Programming 101

Reactor 3 types & operators

reactor-netty

reactor-kafka...

Reactor and Spring

backpressure and other beasts

testing and debugging

Reactive Programming 101

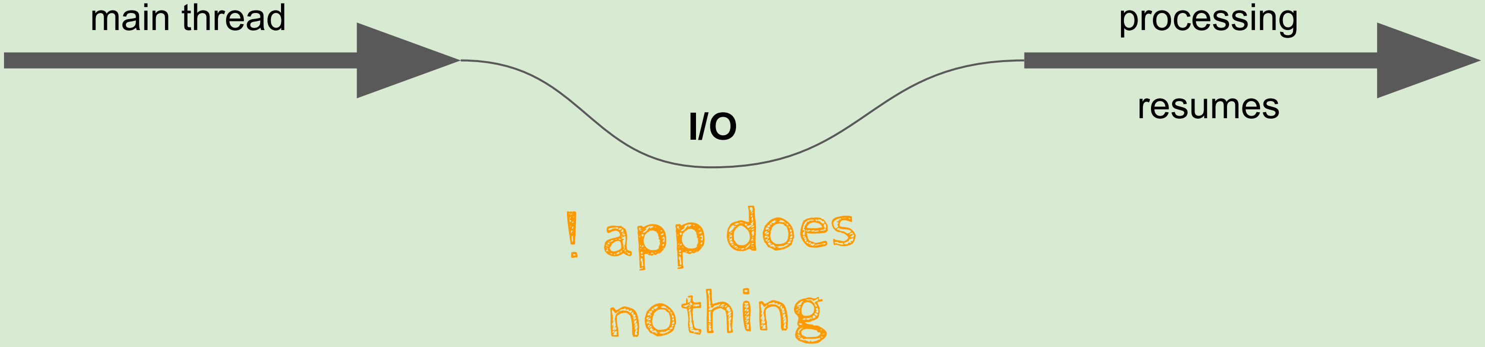
what does it bring to the table?

WHY?

WHY?

because blocking is evil

sync/blocking



main thread

I/O

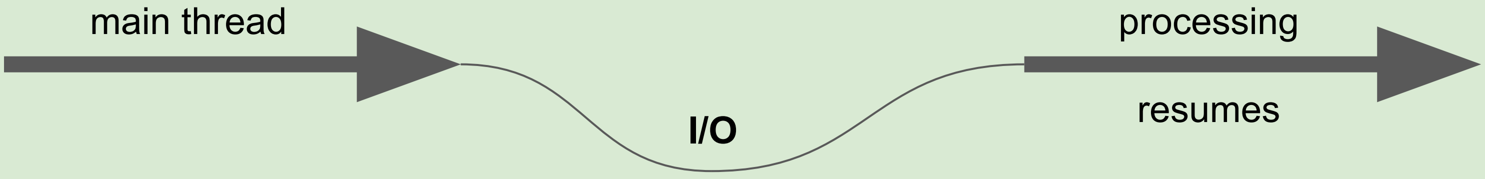
processing

resumes

! app does nothing

sync/blocking

BAD



main thread

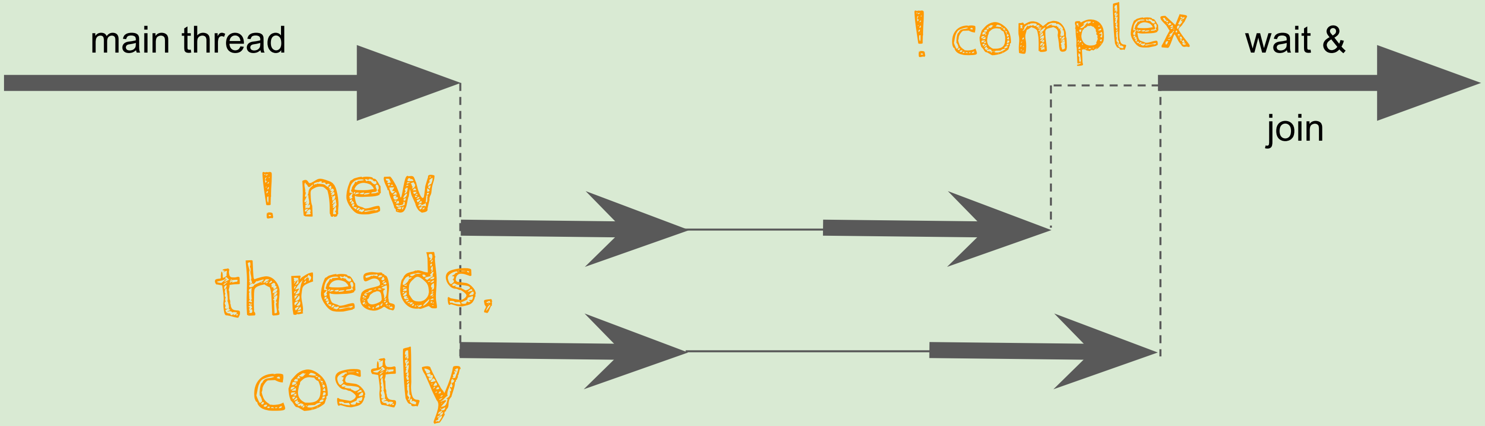
I/O

processing

resumes

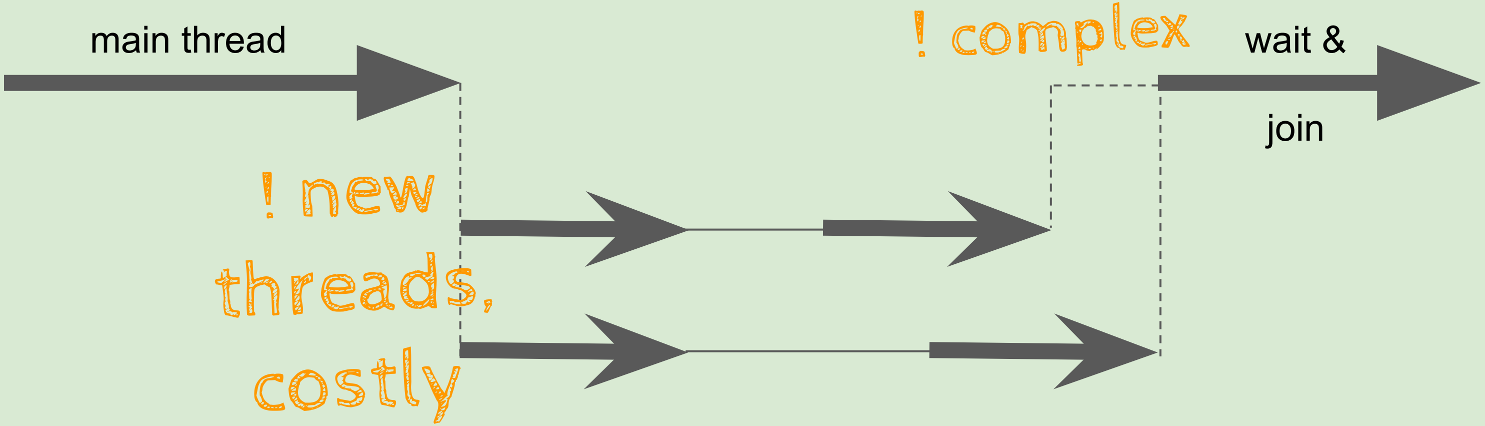
! app does nothing

async & blocking

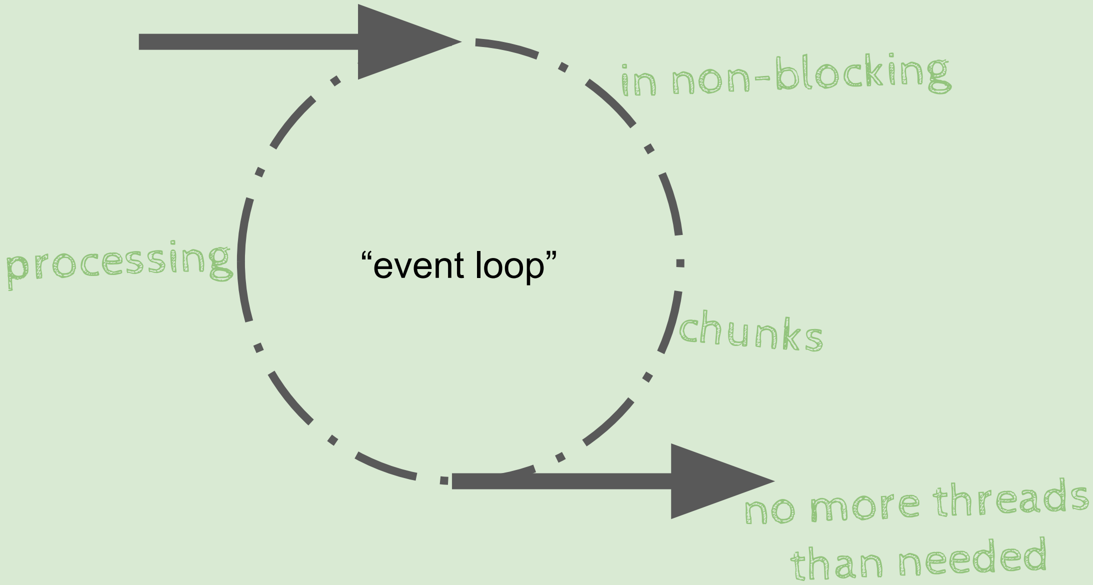


async & blocking

BAD



async & nonblocking



how do you achieve that

without losing your mind ?

Reactive Programming

> “Composing **asynchronous**
> & **event-based** sequences,
> using **non-blocking** operators”

without sacrifice

Callbacks ?

callback hell !

not readable

without sacrifice

Futures ?

easy to block

hard to compose

Pull?

~~Pull?~~ Push!

Pull? Push!

(or actually a little bit of Both)

Iterable

-

Iterator

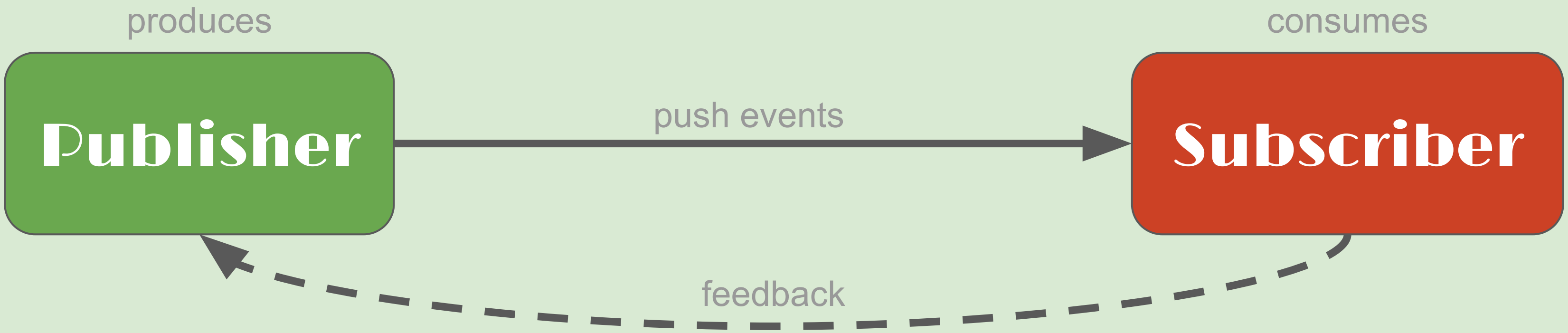
vs

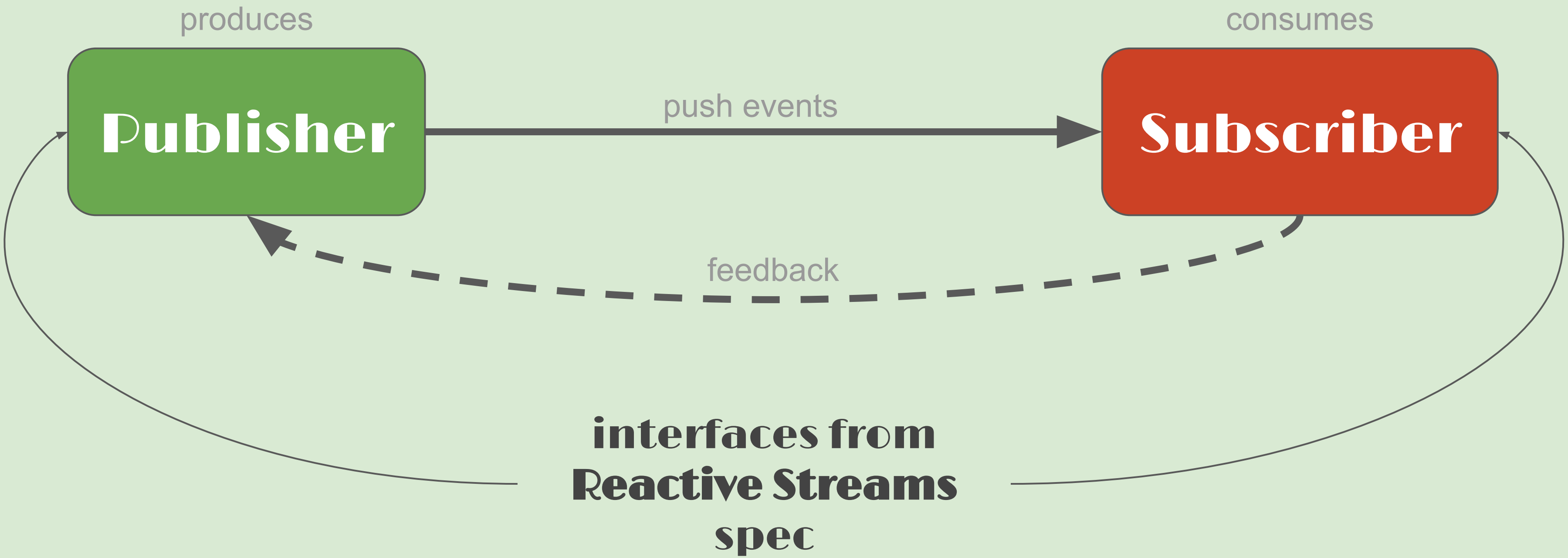
Publisher

-

Subscriber

Data in *Flux*


produces
Publisher
push events
Subscriber
consumes
feedback



produces

Publisher

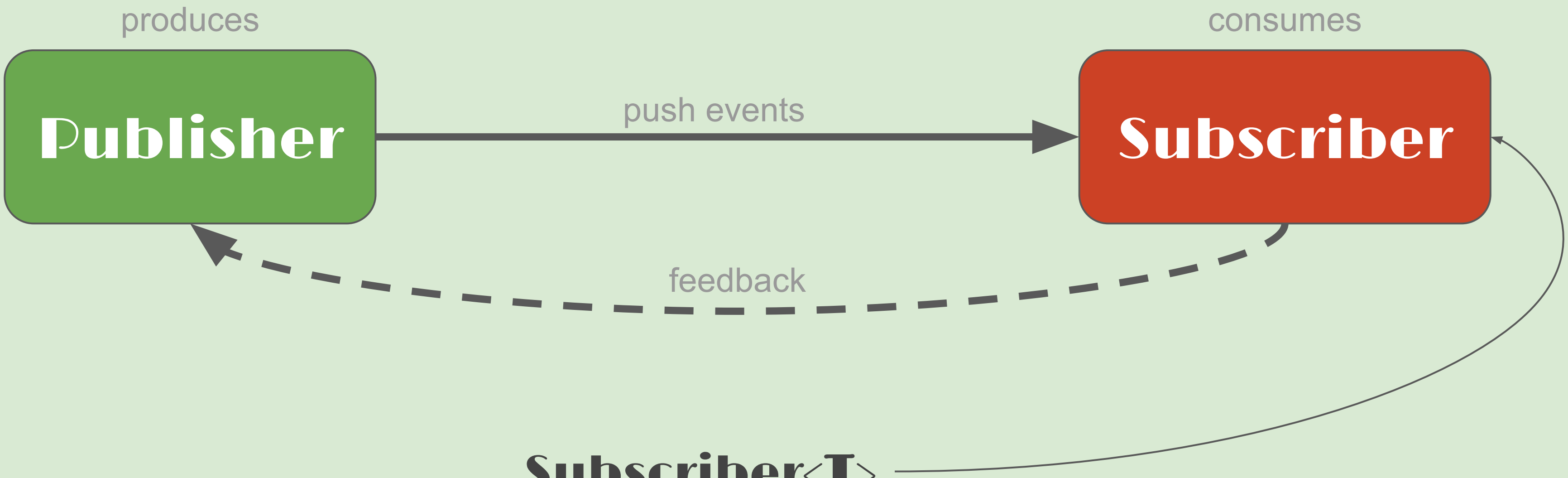
push events

consumes

Subscriber

feedback

interfaces from
Reactive Streams
spec



produces

Publisher

push events

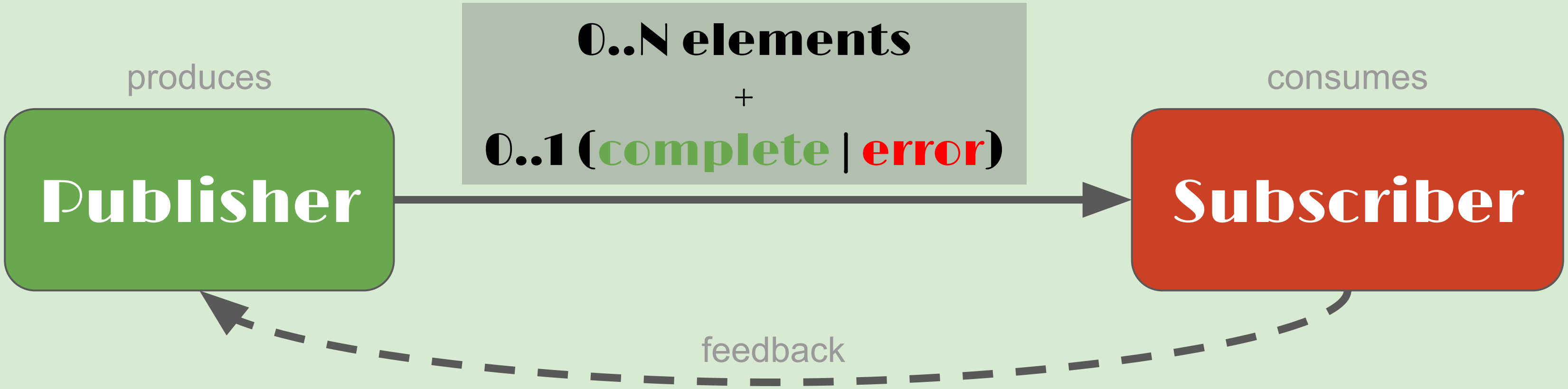
consumes

Subscriber

feedback

Subscriber<T>

```
onNext(T)
onComplete();
onError(Throwable);
```



0..N elements

+

0..1 (complete | error)

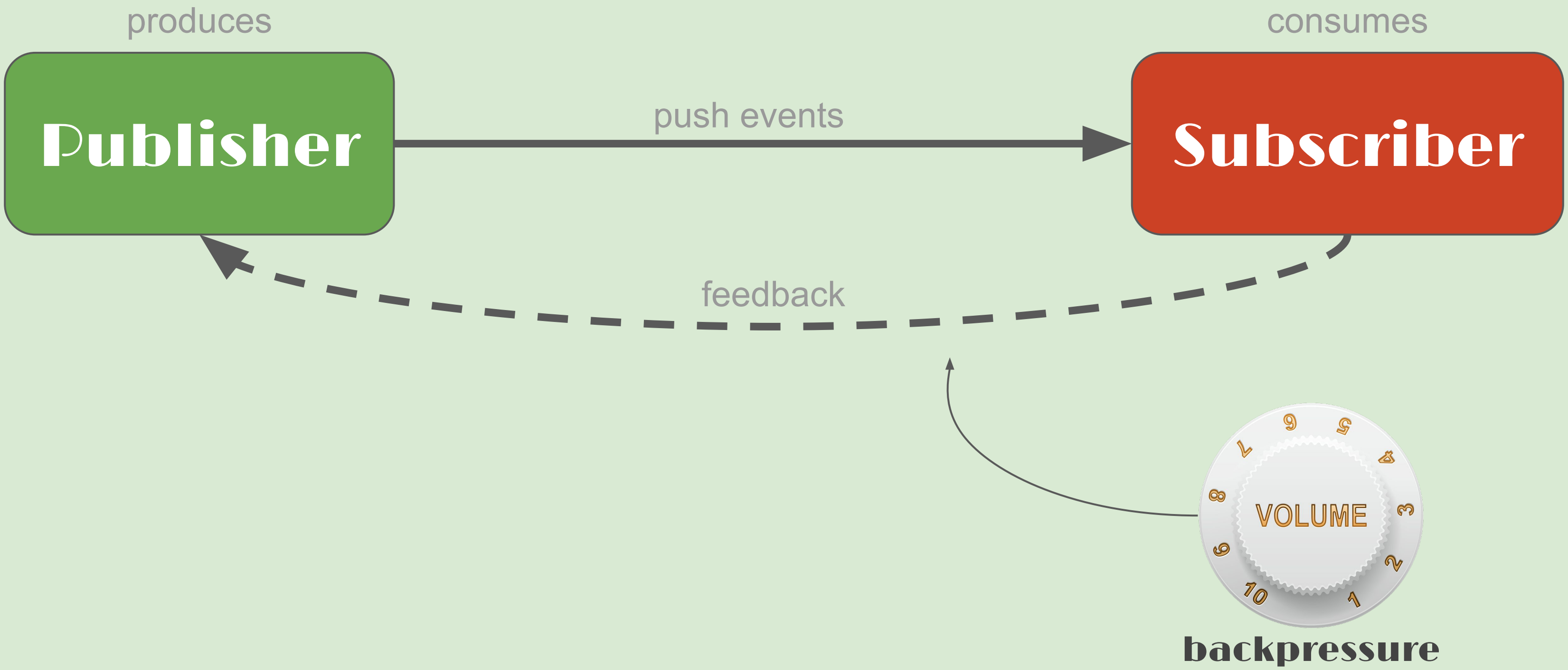
produces

Publisher

consumes

Subscriber

feedback



produces

Publisher

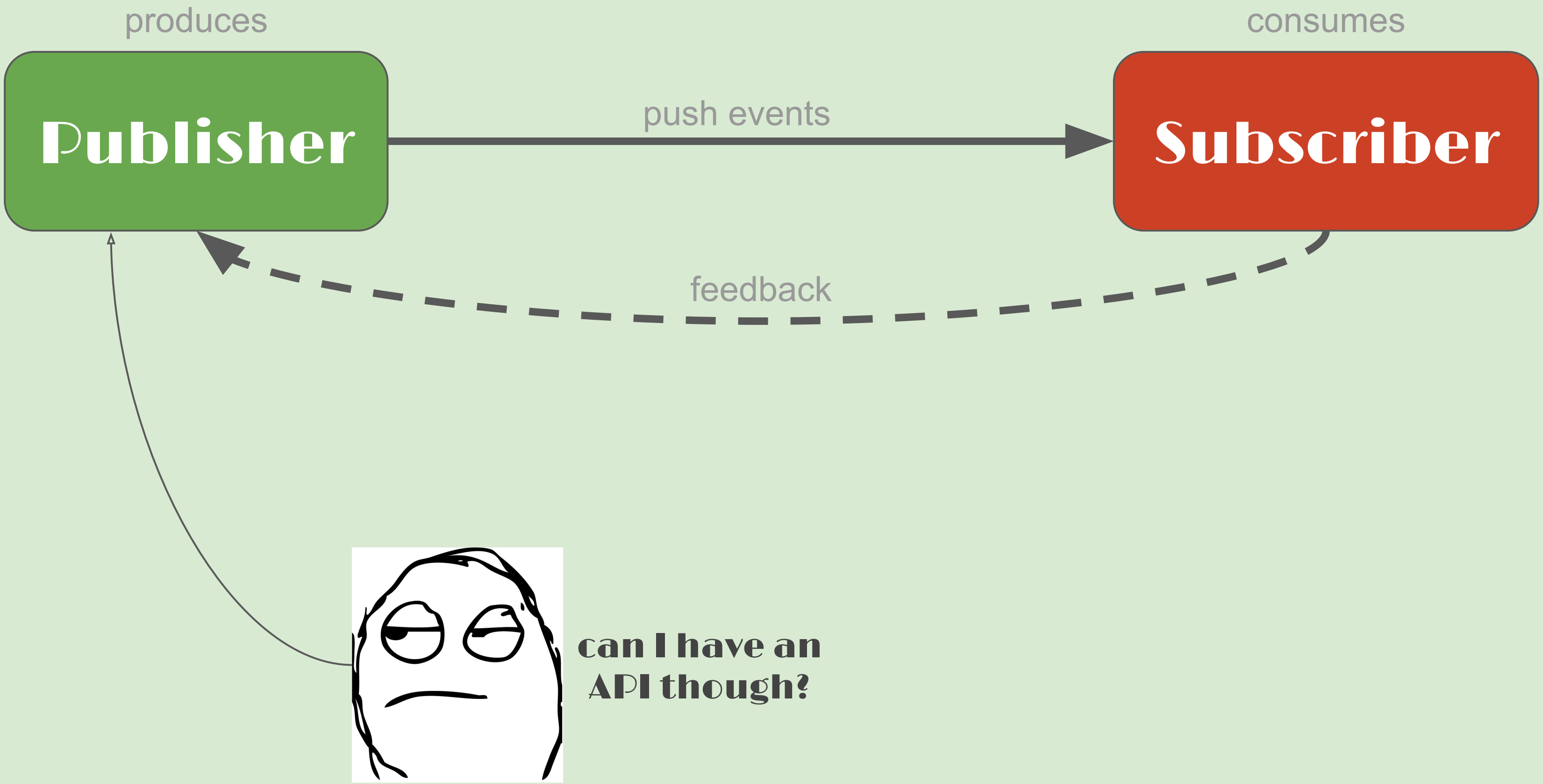
push events

consumes

Subscriber

feedback

backpressure



produces

consumes

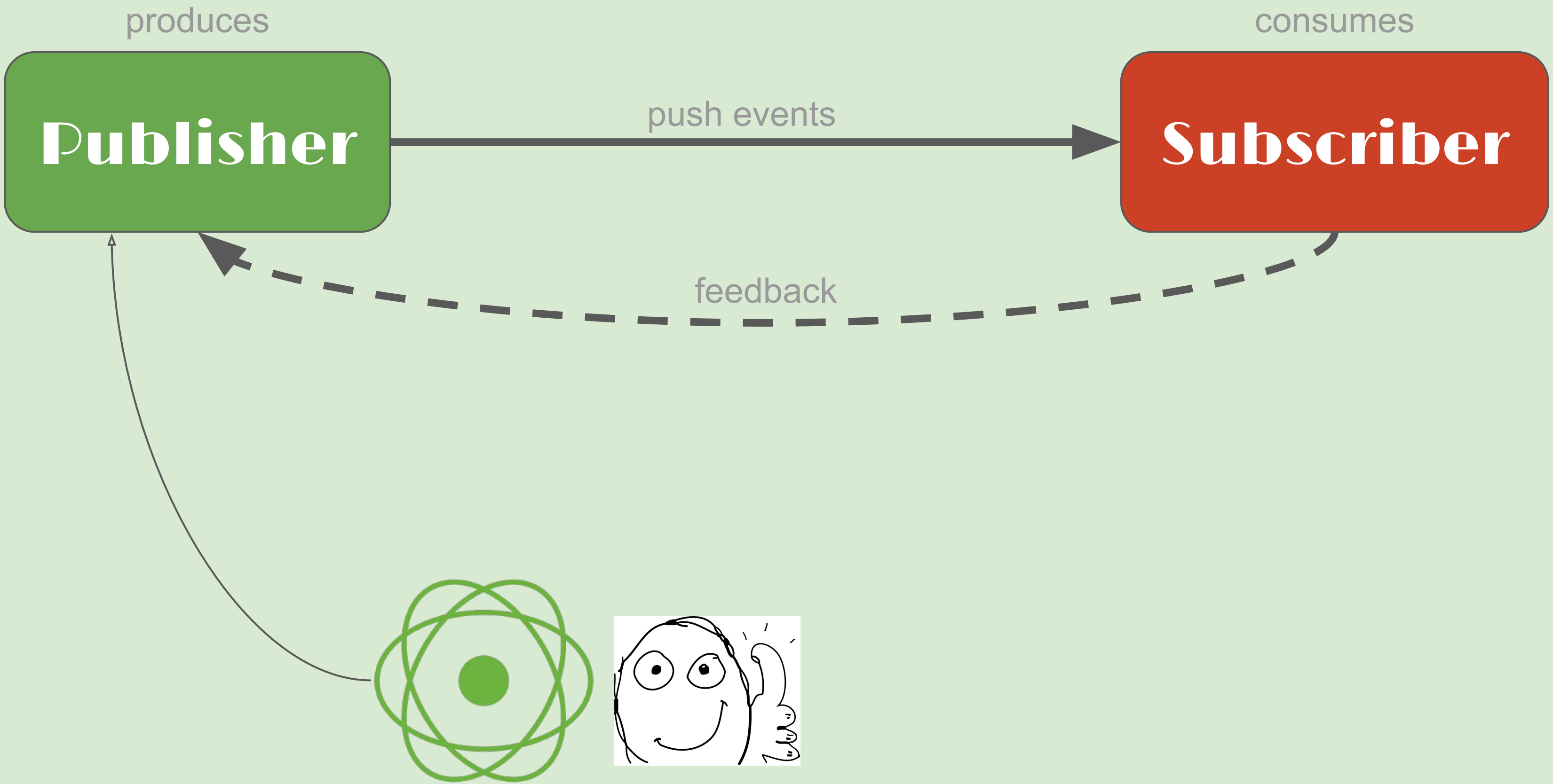
Publisher

push events

Subscriber

feedback

can I have an API though?



produces

Publisher

push events

consumes

Subscriber

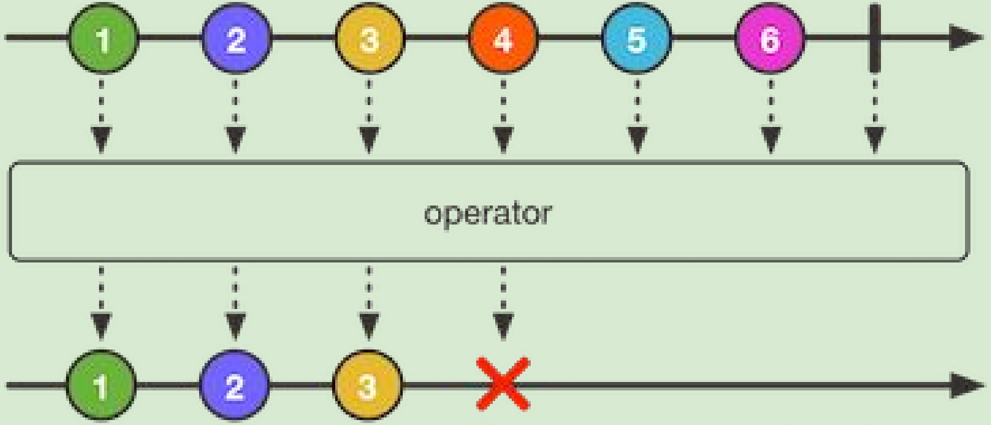
feedback

Reactor 3

types and operators

Flux<T>

for 0..N elements



These are items emitted by the Flux.

This vertical line indicates that the Flux has completed successfully.

This is the timeline of the Flux. Time flows from left to right.

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.

This Flux is the result of the transformation.

If for some reason the Flux terminates abnormally, with an error, the vertical line is replaced by an X.

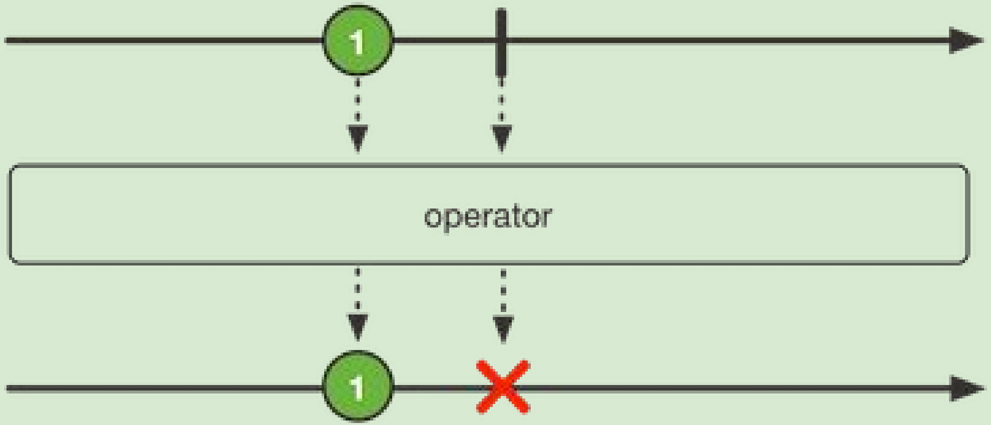
Mono<T>

*for **at most 1** element*

This is the eventual item emitted by the Mono.

This vertical line indicates that the Mono has completed successfully.

This is the timeline of the Mono. Time flows from left to right.



These dotted lines and this box indicate that a transformation is being applied to the Mono. The text inside the box shows the nature of the transformation.

This Mono is the result of the transformation.

If for some reason the Mono terminates abnormally, with an error, the vertical line is replaced by an X.

Reactive Streams

all the way

focus on Java 8

focus on Java 8

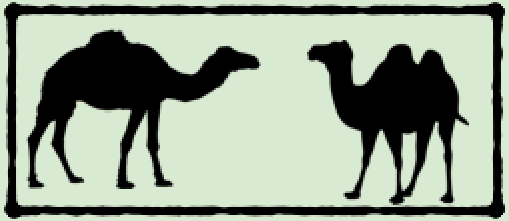
Duration, CompletableFuture, Streams

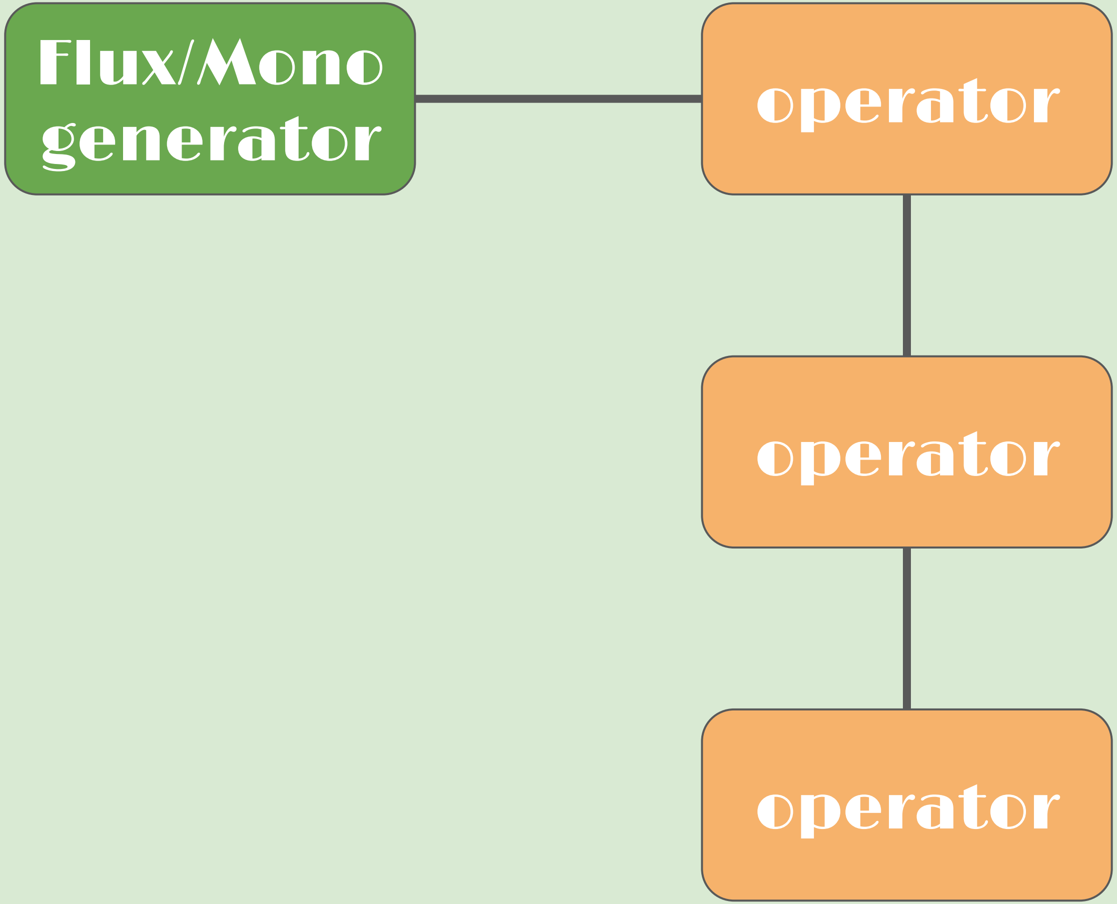
an Rx-inspired API

*with a vocabulary of **operators** similar to **RxJava**...*

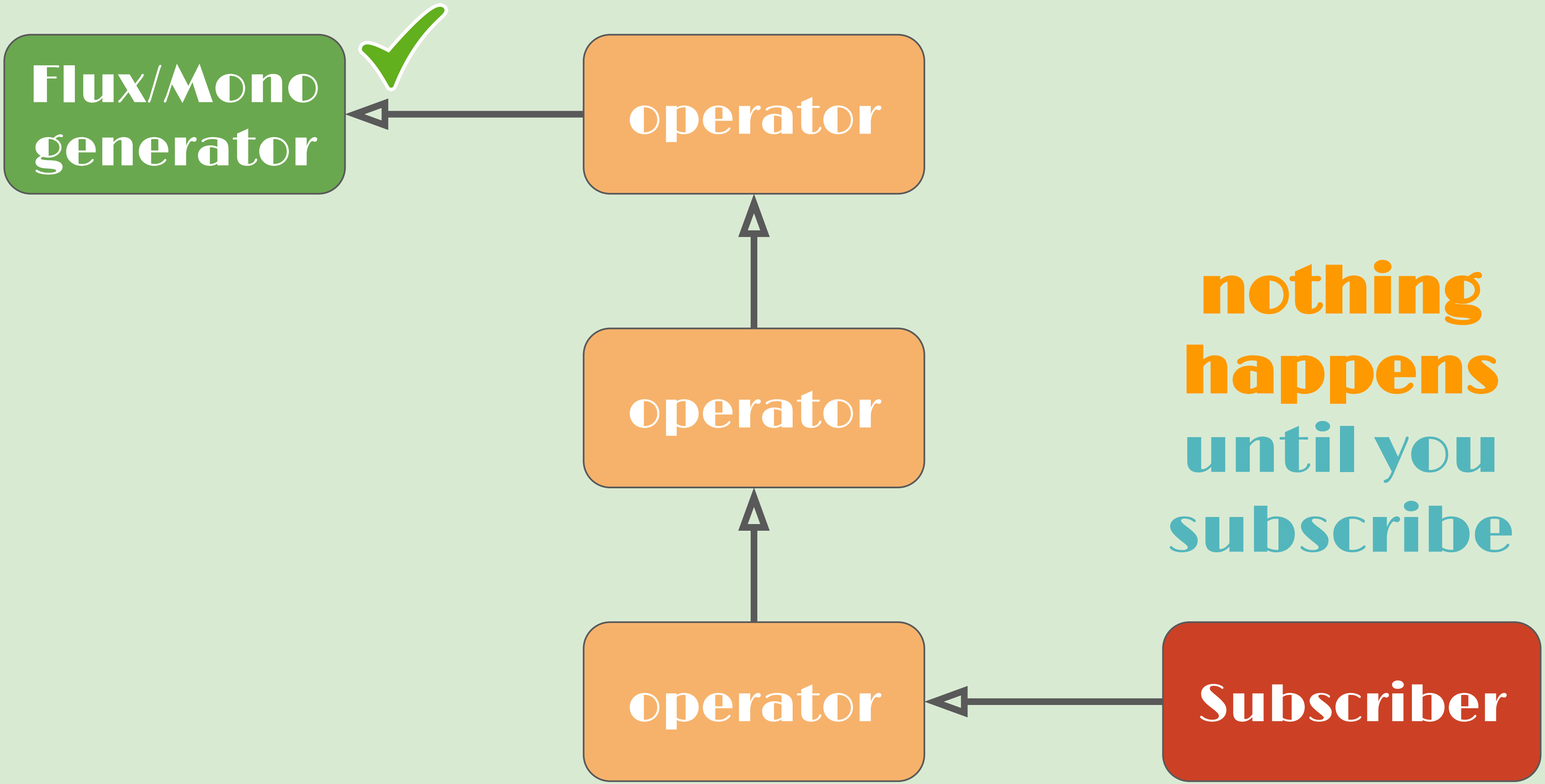
an Rx-inspired API

...but not exactly the same





nothing happens until you subscribe



Flux/Mono generator

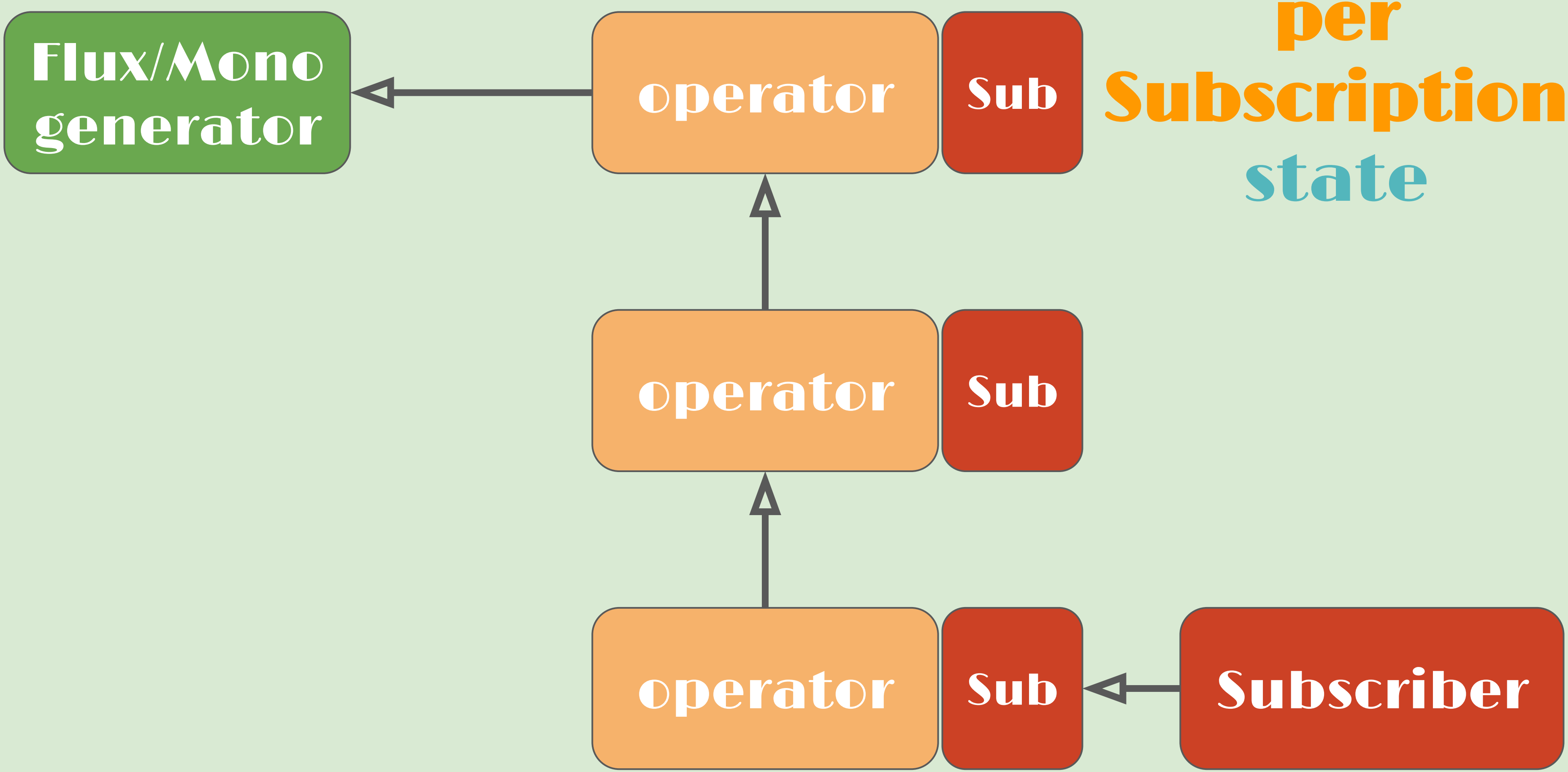
operator

operator

operator

Subscriber

nothing happens until you subscribe



per Subscription state

Flux/Mono generator

operator

Sub

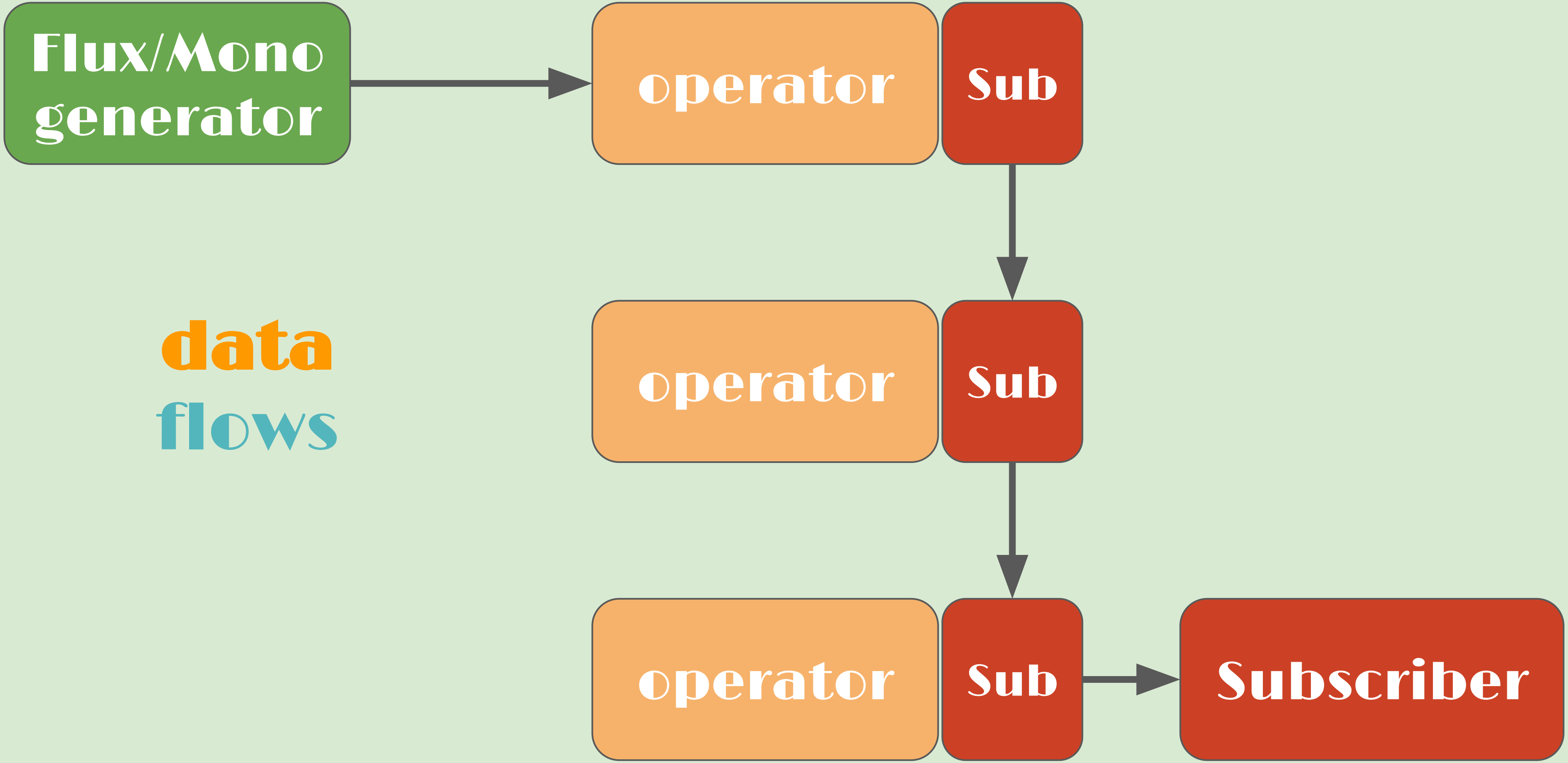
operator

Sub

operator

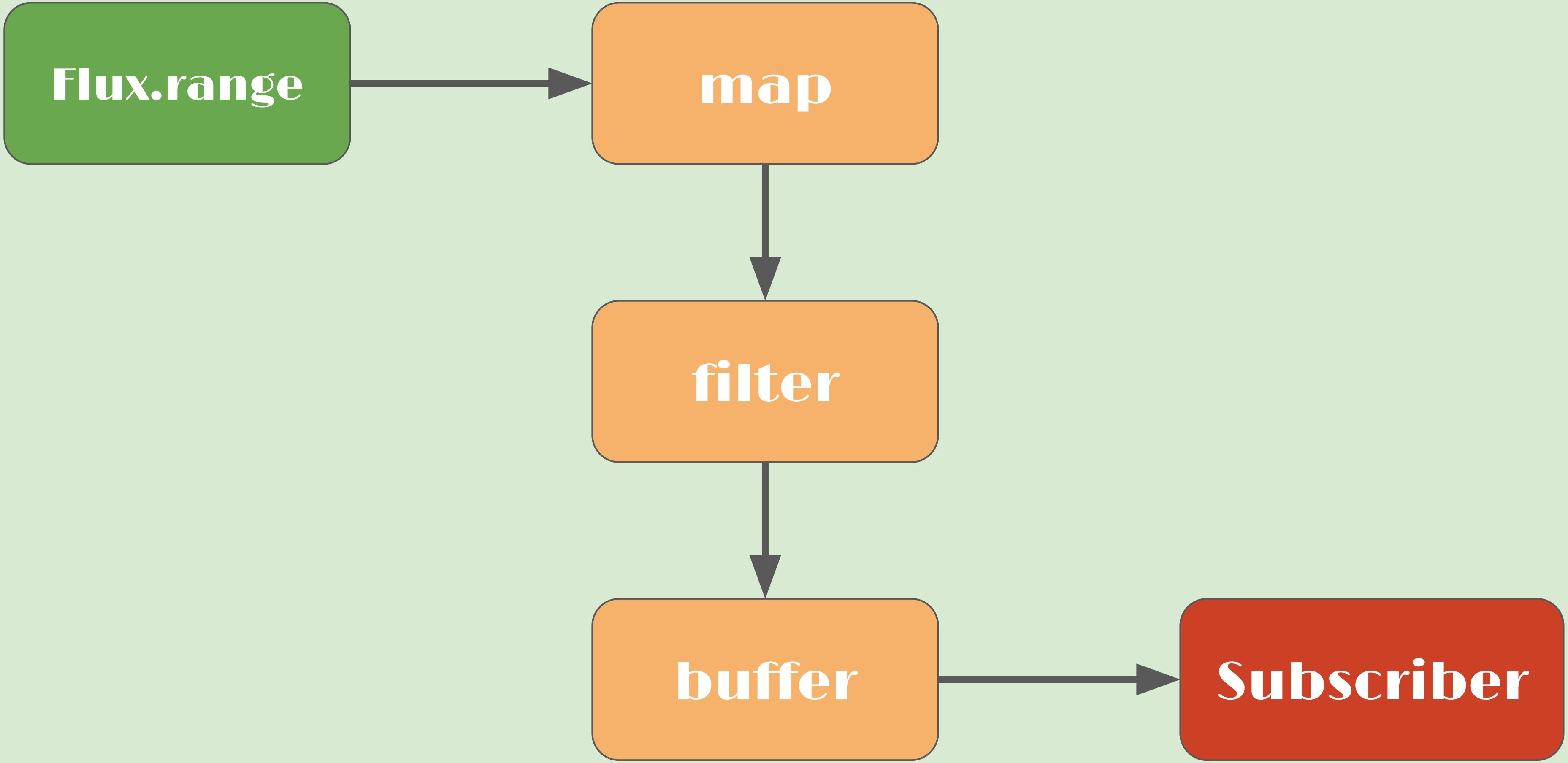
Sub

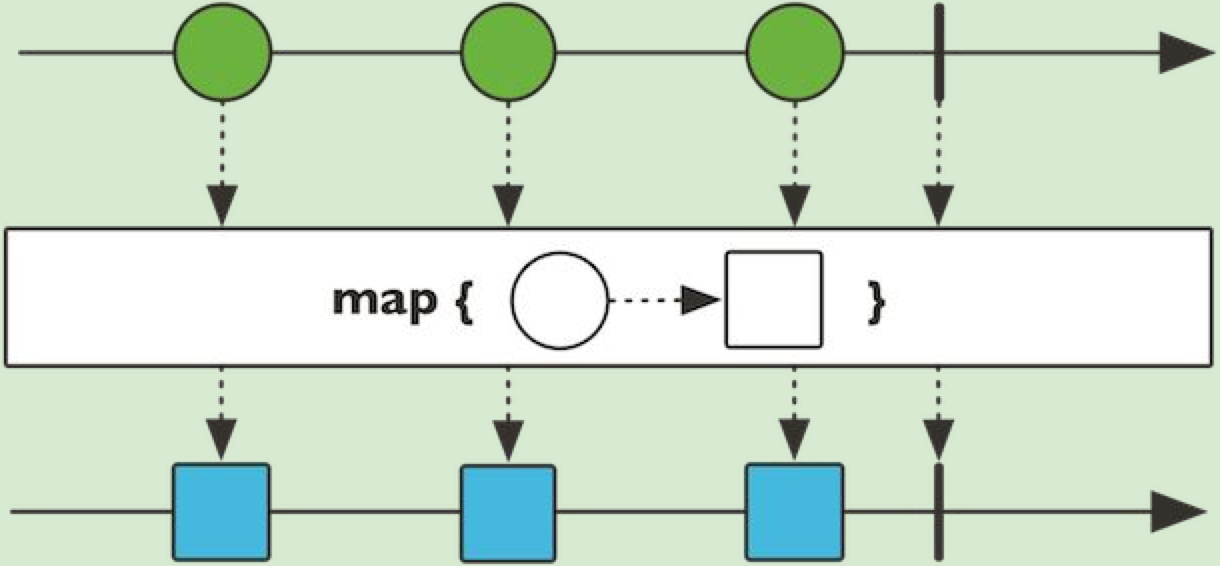
Subscriber

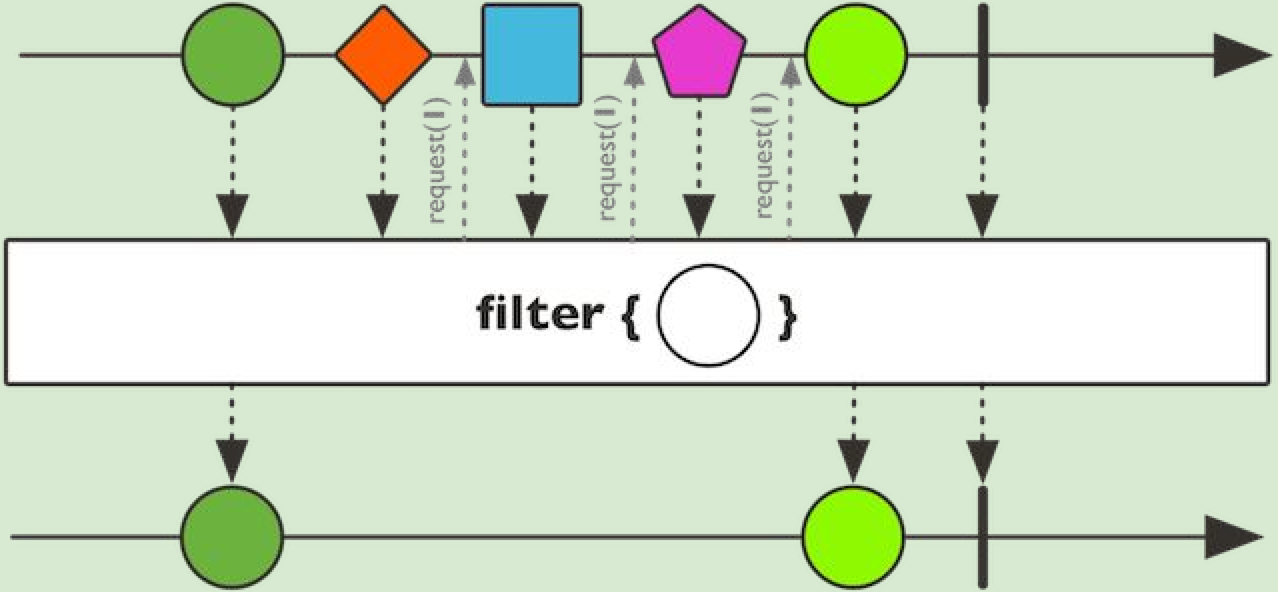

Flux/Mono generator
operator
Sub
operator
Sub
operator
Sub
Subscriber

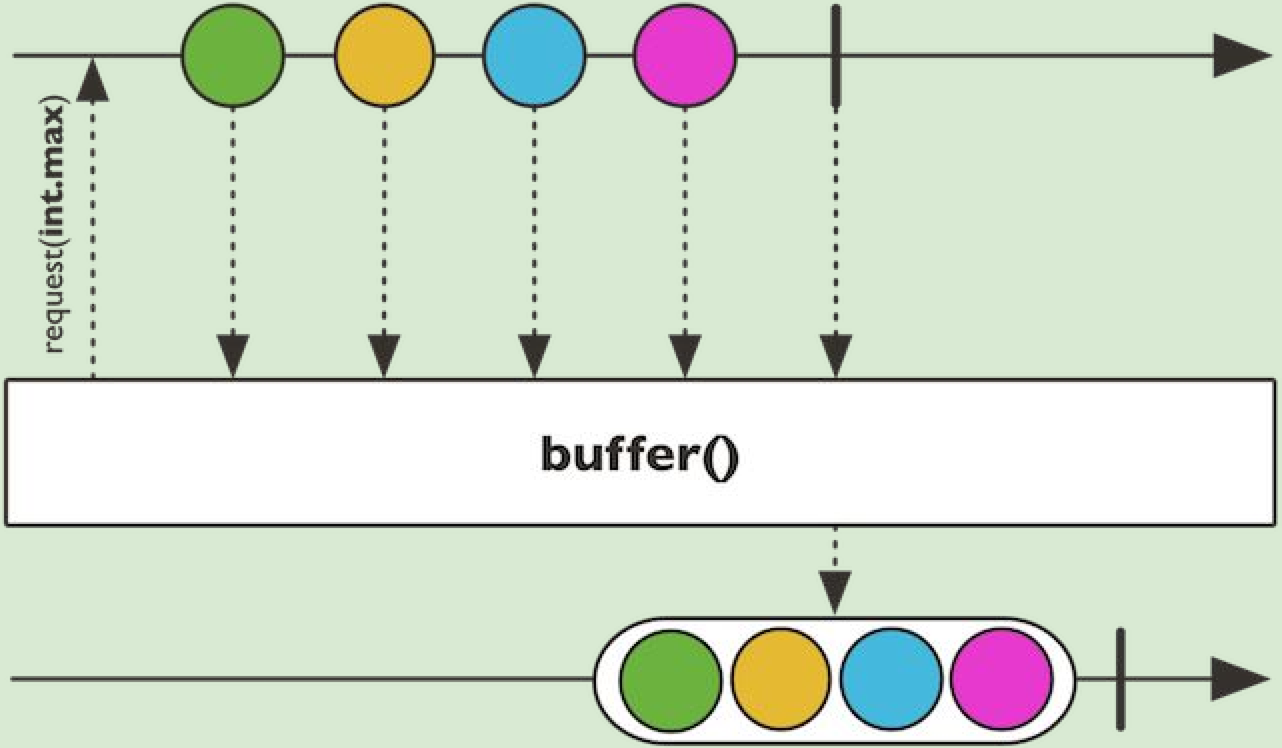
data flows

examples

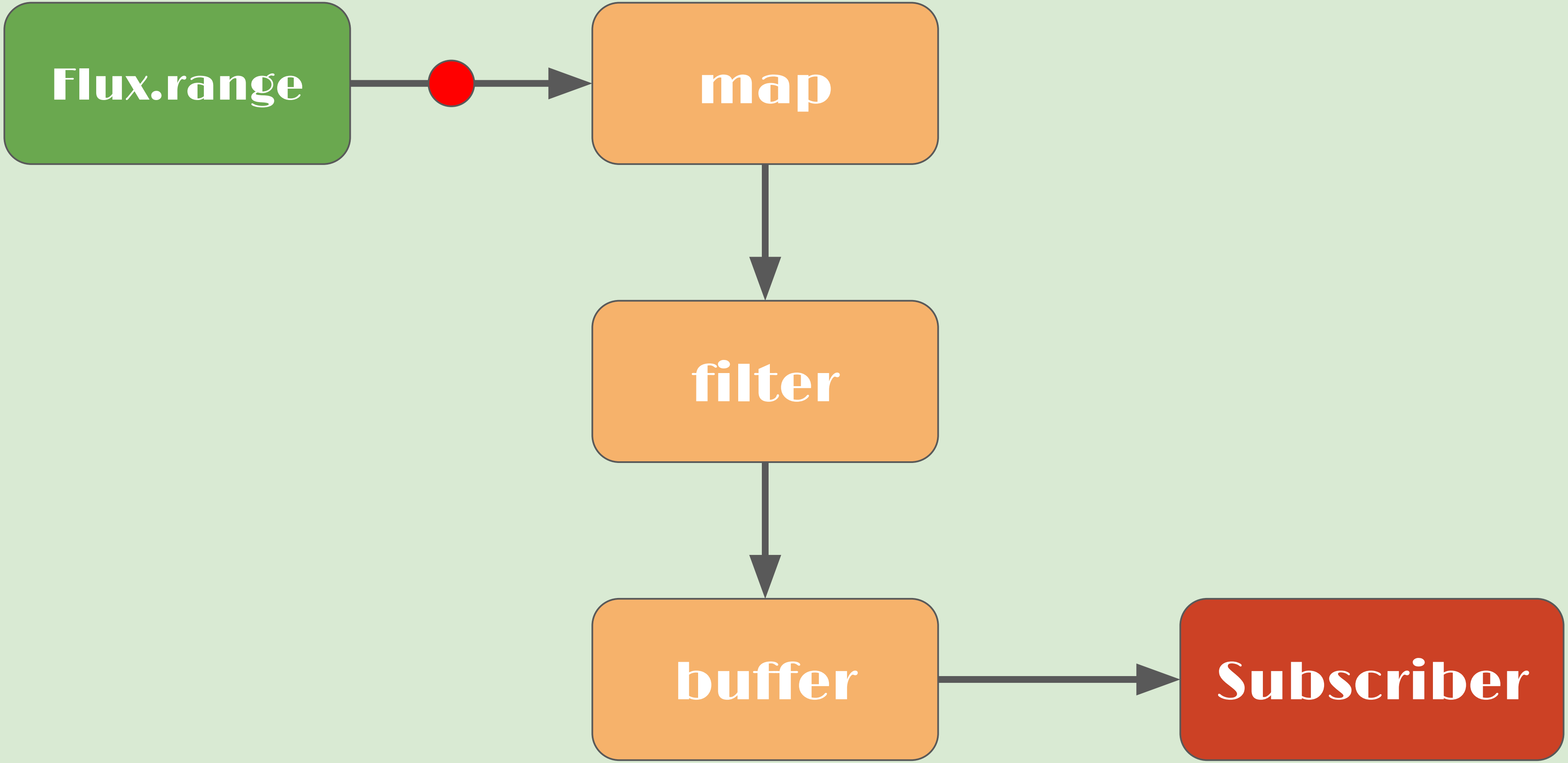

Flux.range
map
filter
buffer
Subscriber

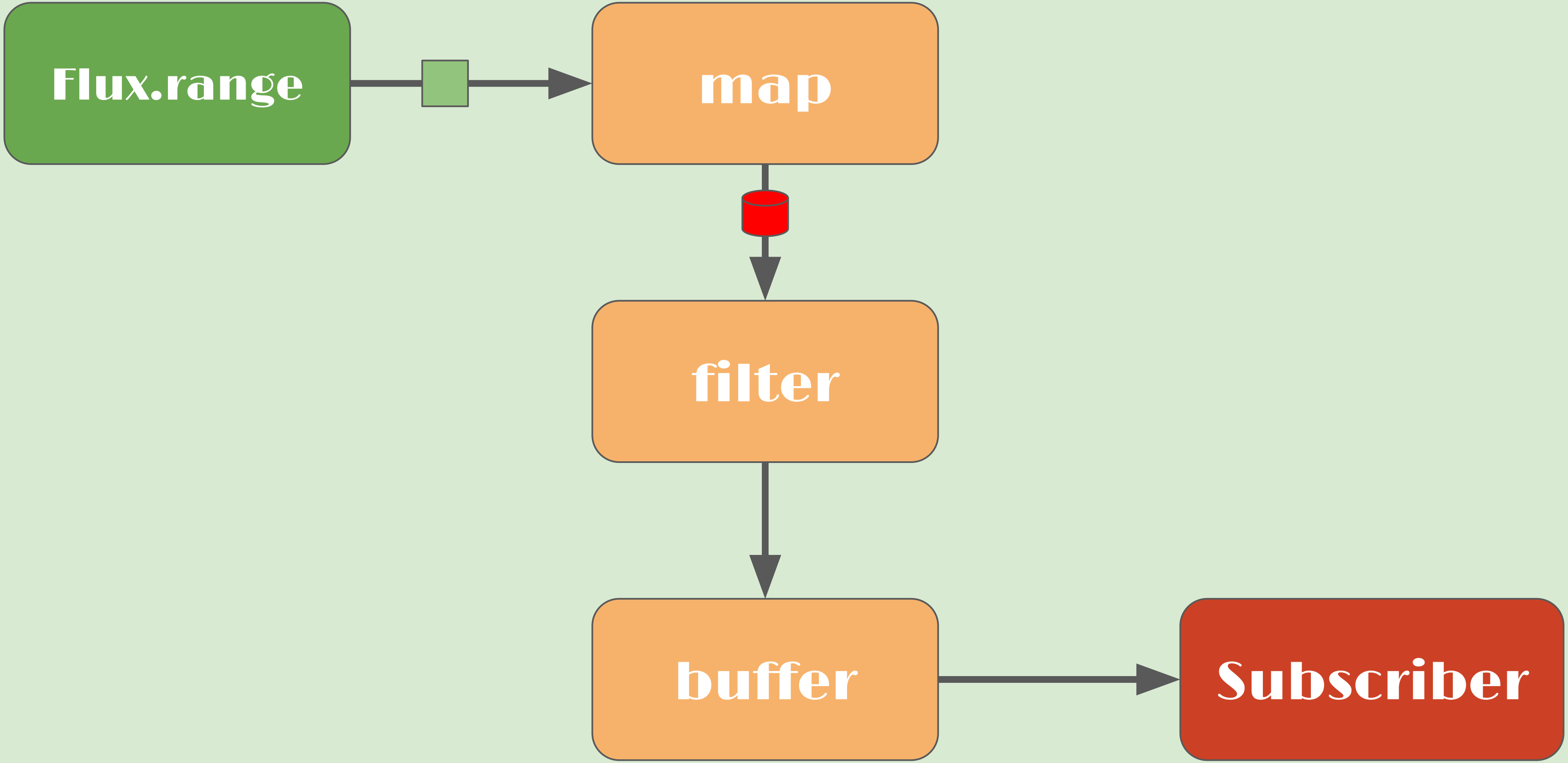

map { }

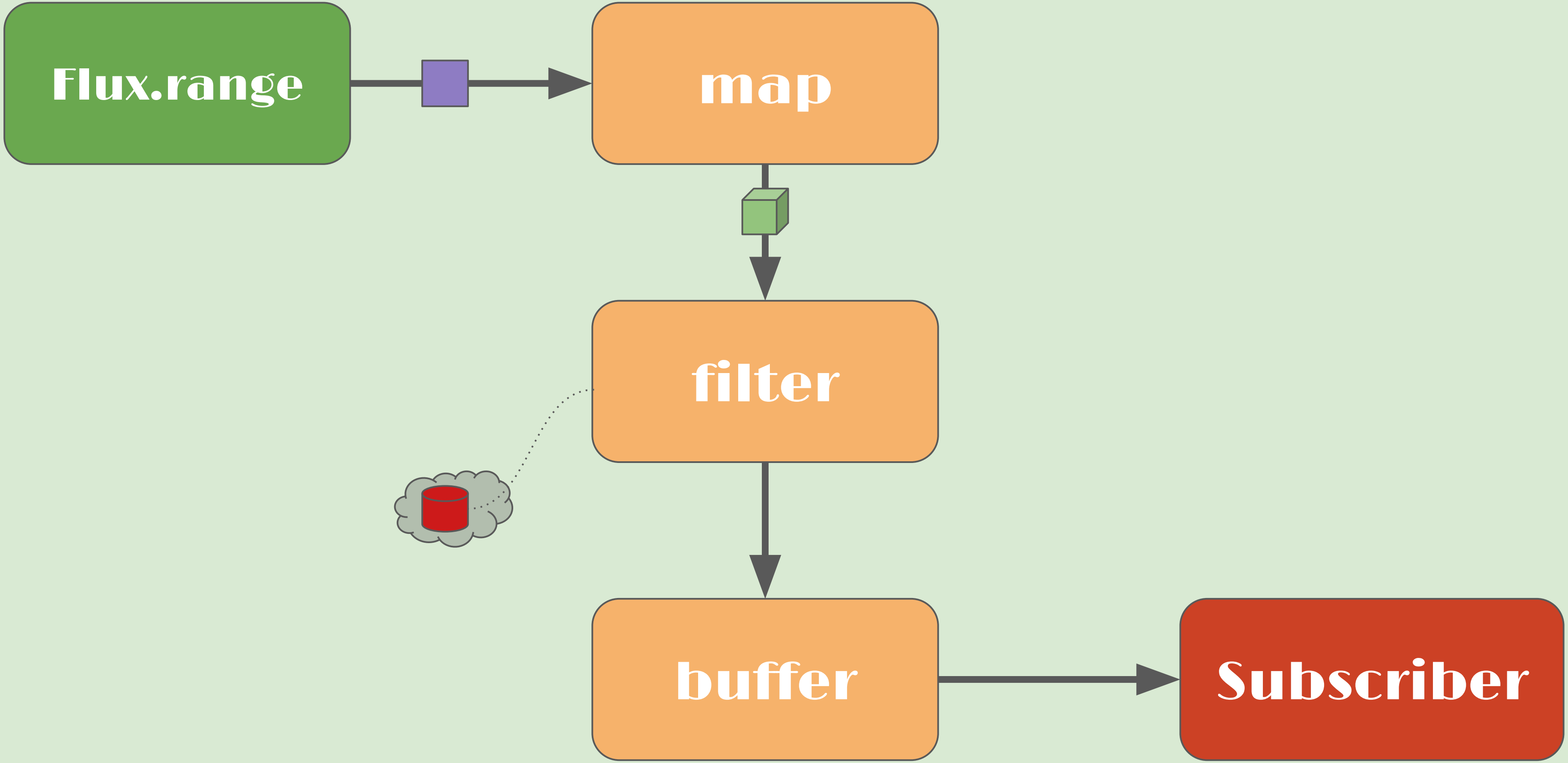

request(1)
request(1)
request(1)
filter { ○ }

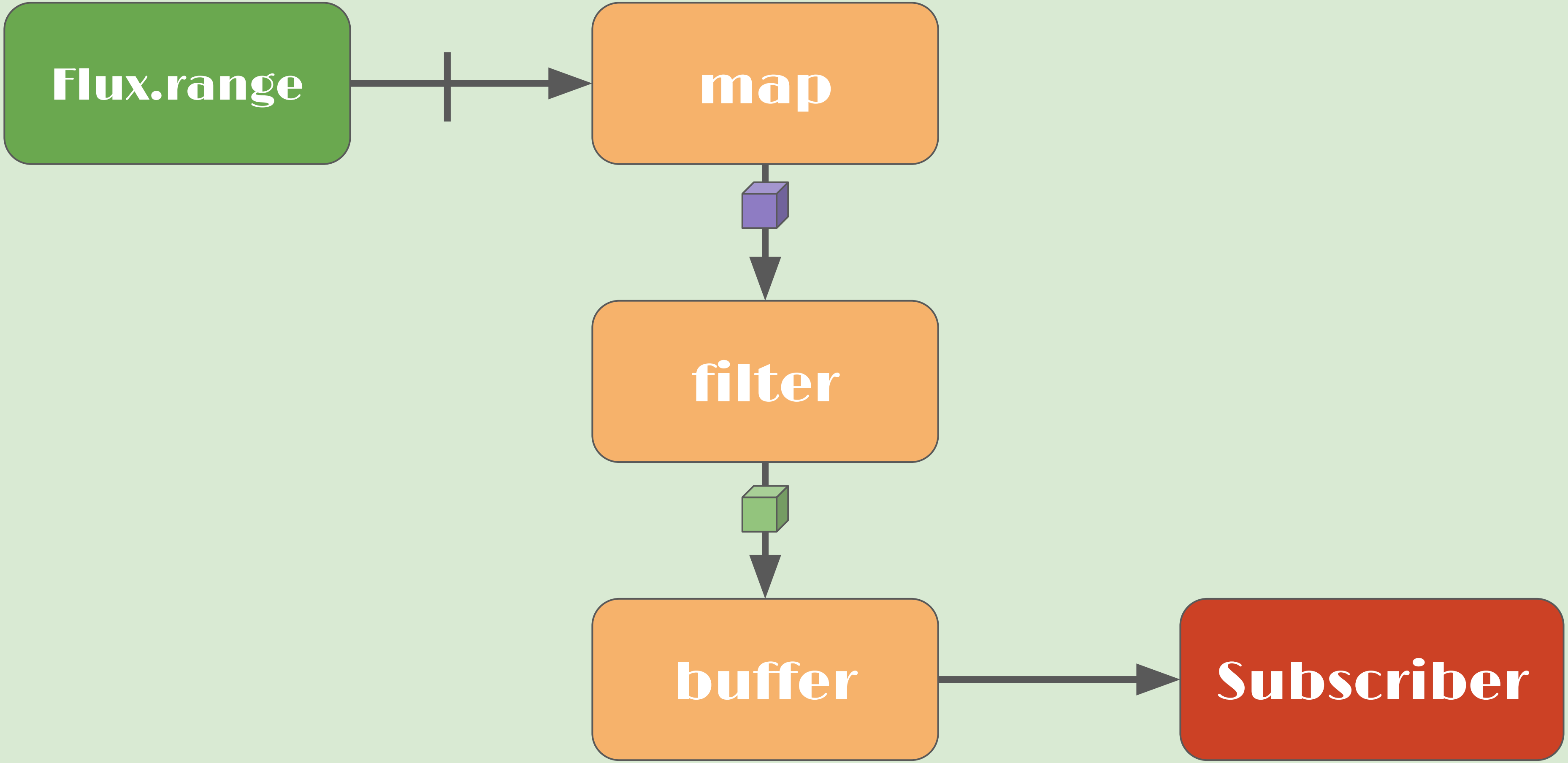

request(int.max)
buffer()

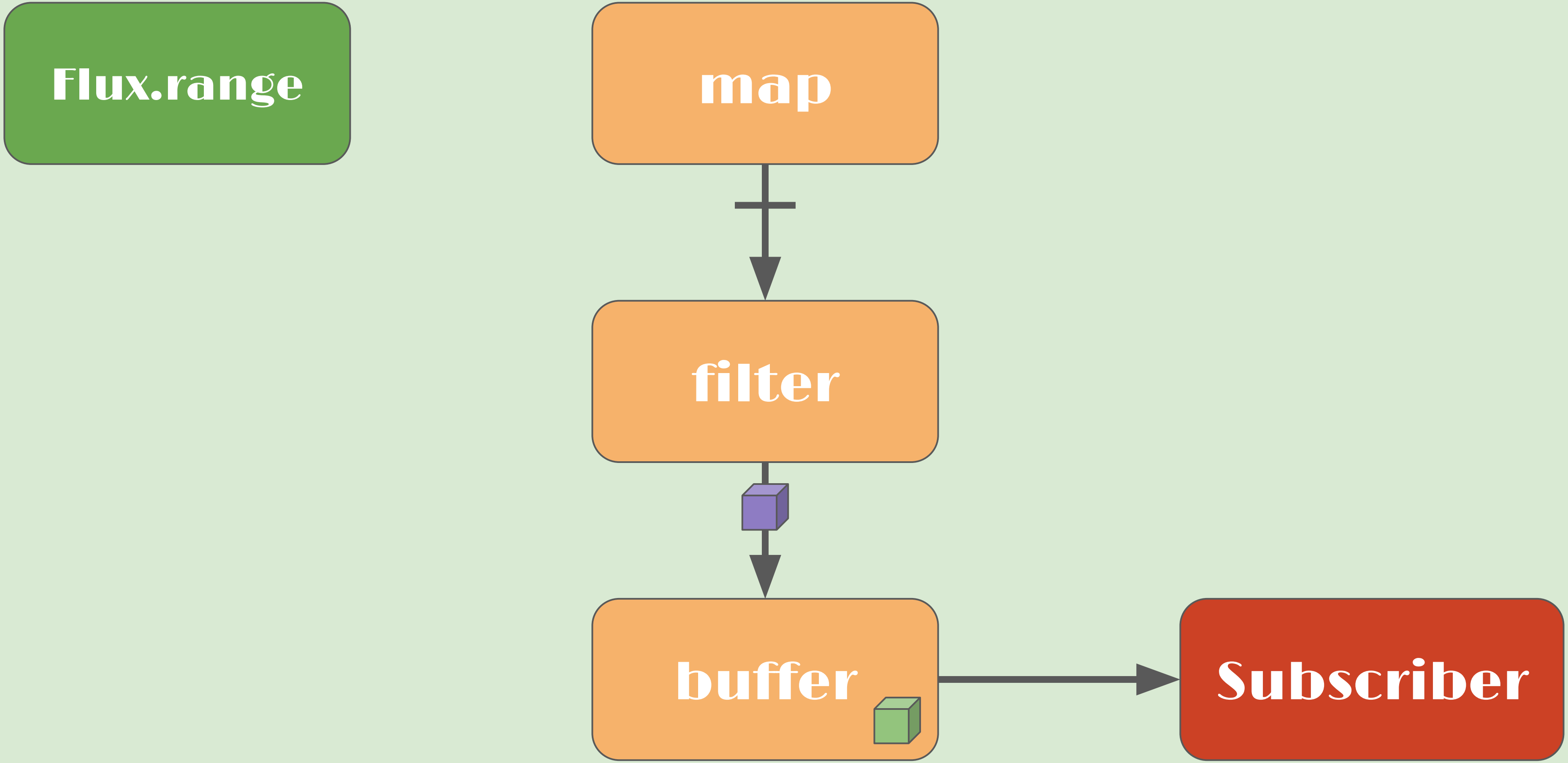
```
Flux.range(5, 3)
    .map(i -> i + 3)
    .filter(i -> i % 2 == 0)
    .buffer(3)
```


Flux.range
map
filter
buffer
Subscriber


Flux.range
map
filter
buffer
Subscriber


Flux.range
map
filter
buffer
Subscriber


Flux.range
map
filter
buffer
Subscriber



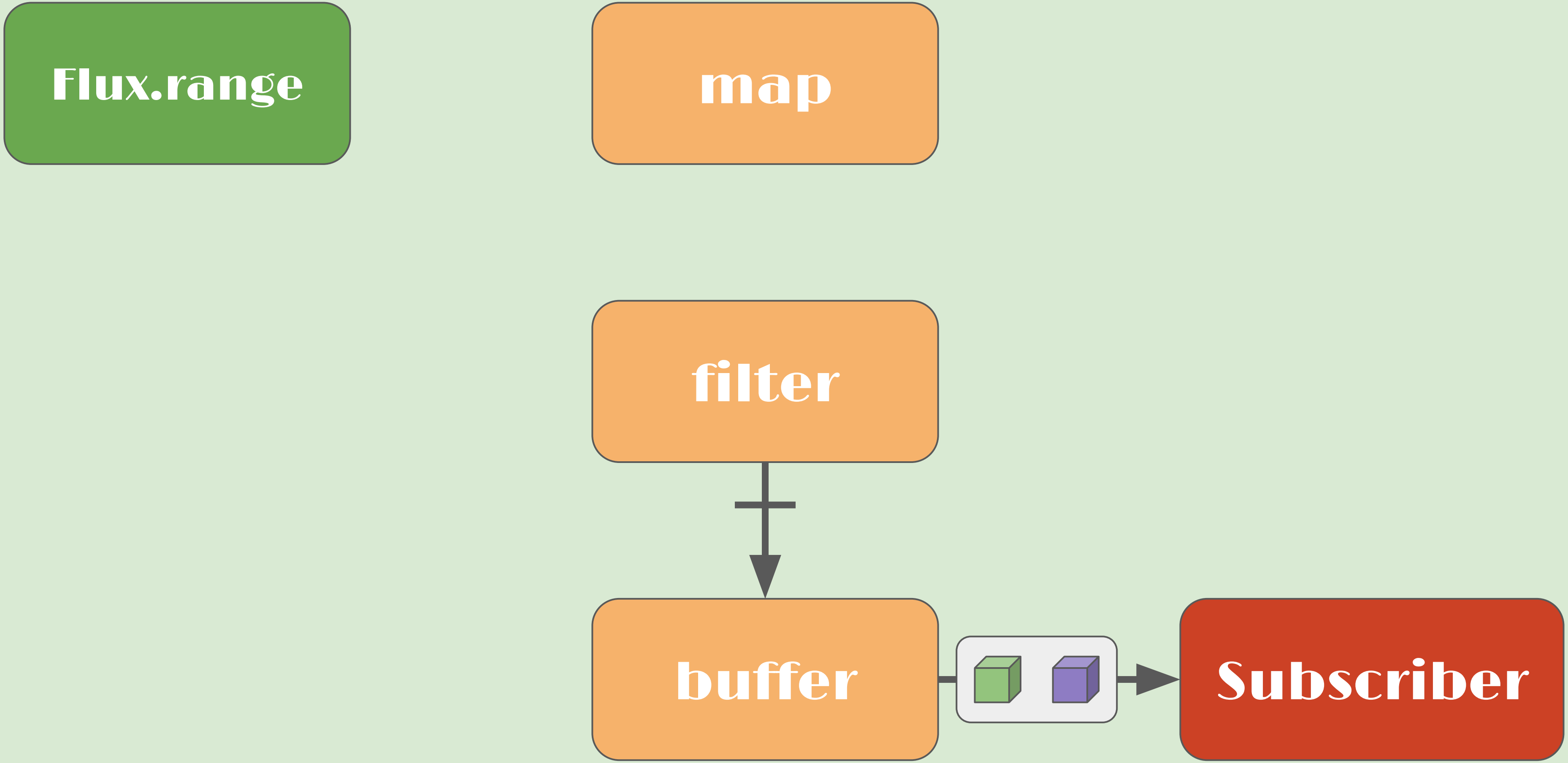
Flux.range

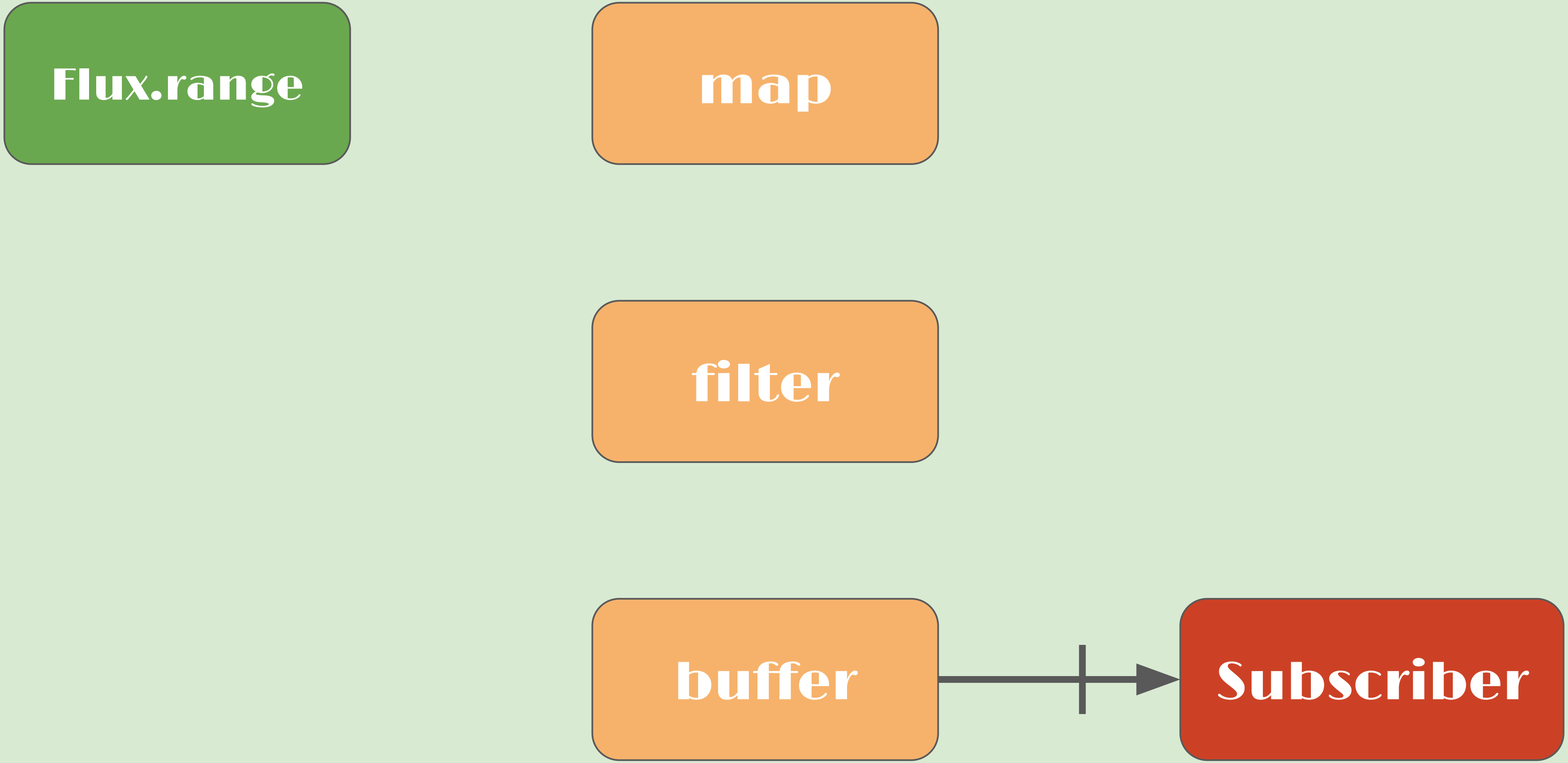
map

filter

buffer

Subscriber


Flux.range
map
filter
buffer
Subscriber



Flux.range

map

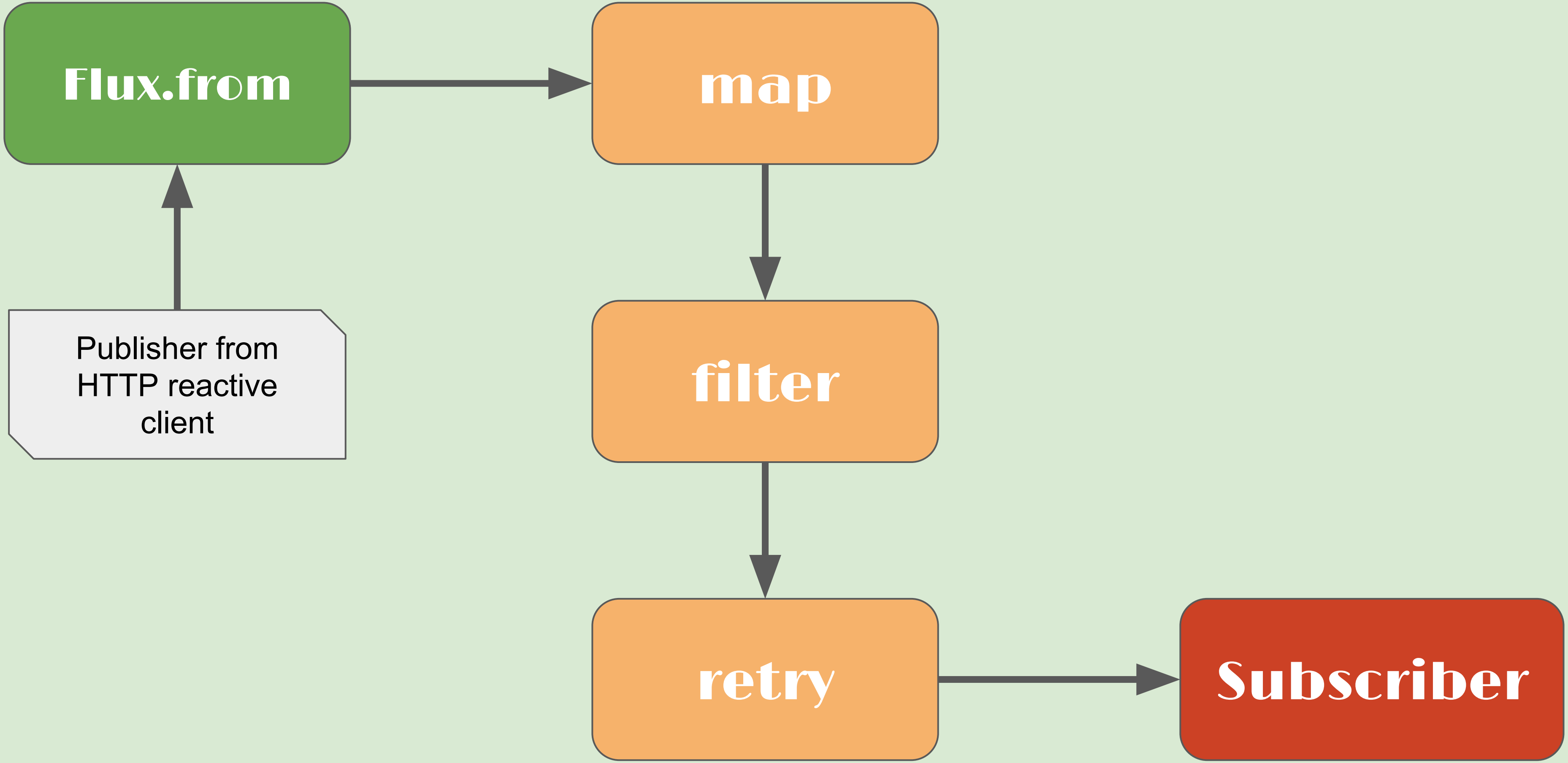
filter

buffer

Subscriber

```
Flux.range(5, 3)
    .map(i -> i + 3)
    .filter(i -> i % 2 == 0)
    .buffer(3)
```

```
5, 6, 7  |
8, 9, 10 |
8,    10 |
    [8,10]|
```



Flux.from

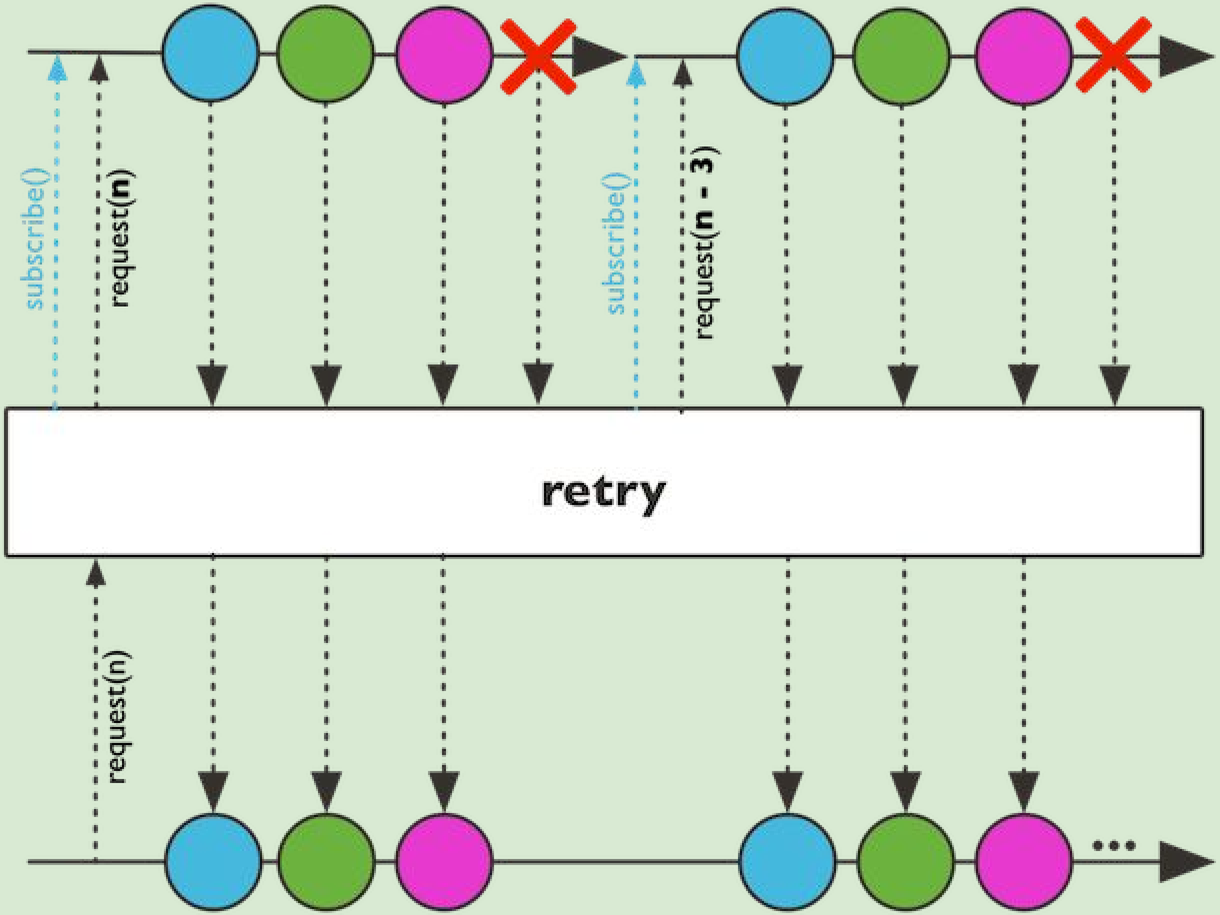
map

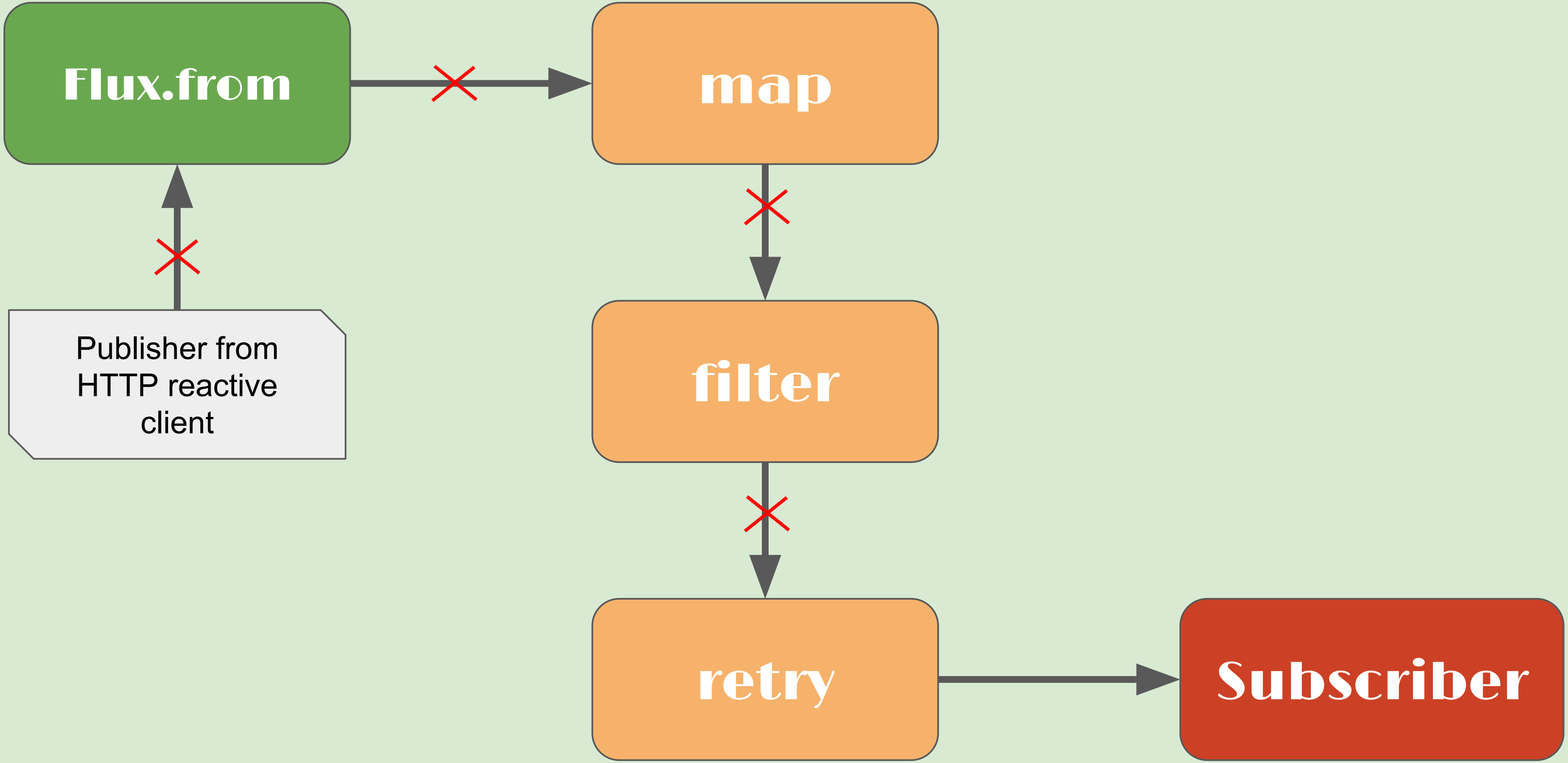
filter

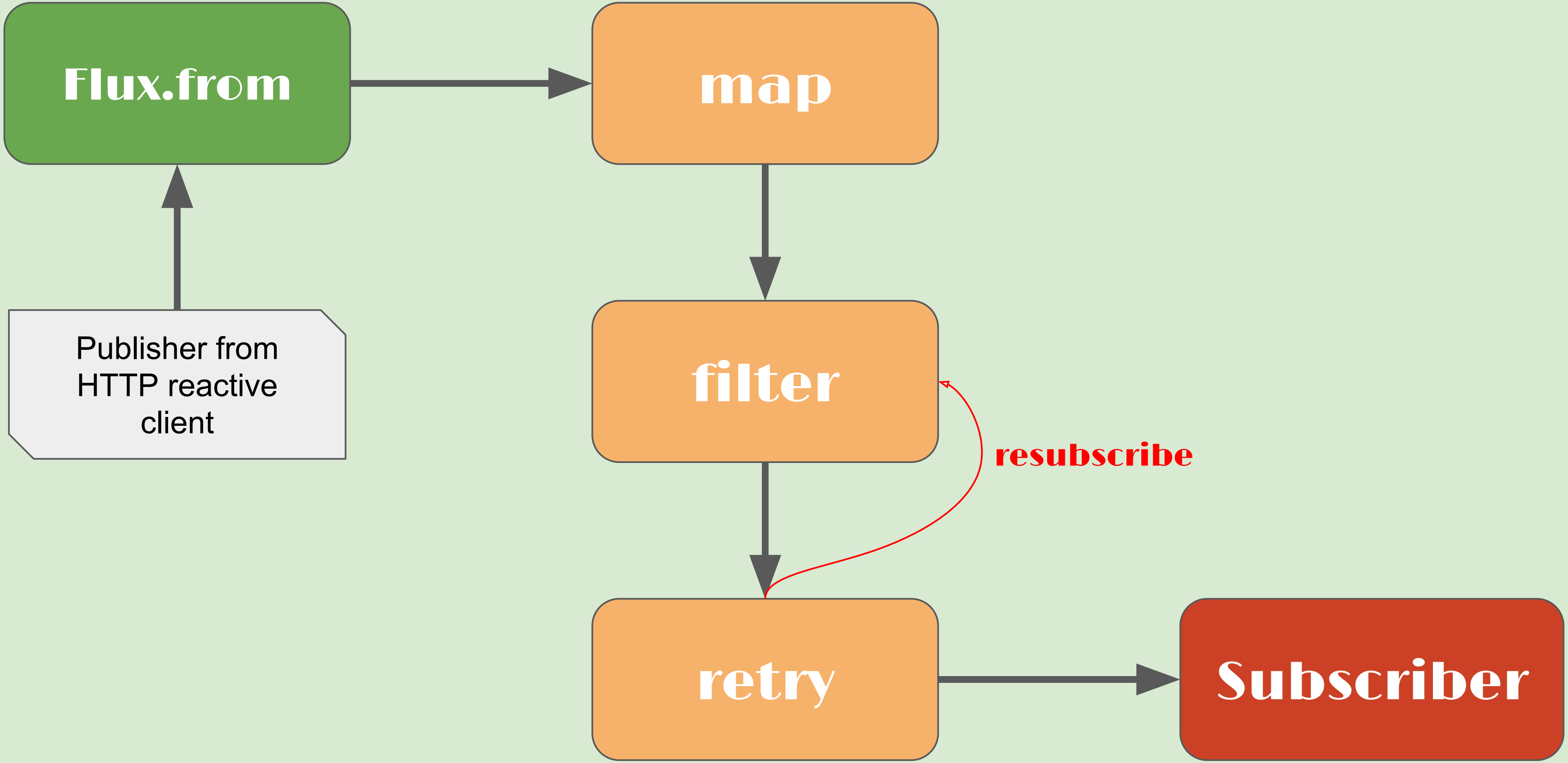
retry

Subscriber

Publisher from HTTP reactive client


subscribe()
request(n)
subscribe()
request(n - 3)
retry
request(n)
...


Flux.from
map
filter
retry
Subscriber
Publisher from HTTP reactive client



Flux.from

map

filter

retry

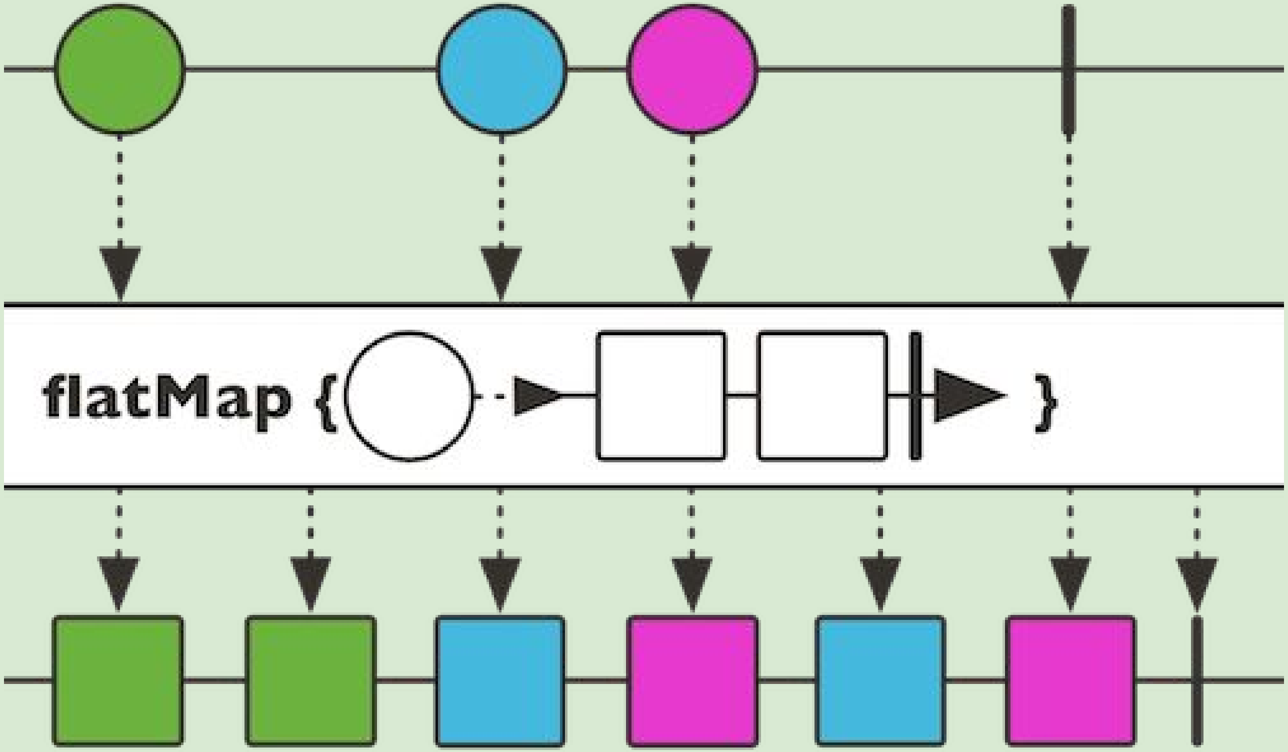
Subscriber

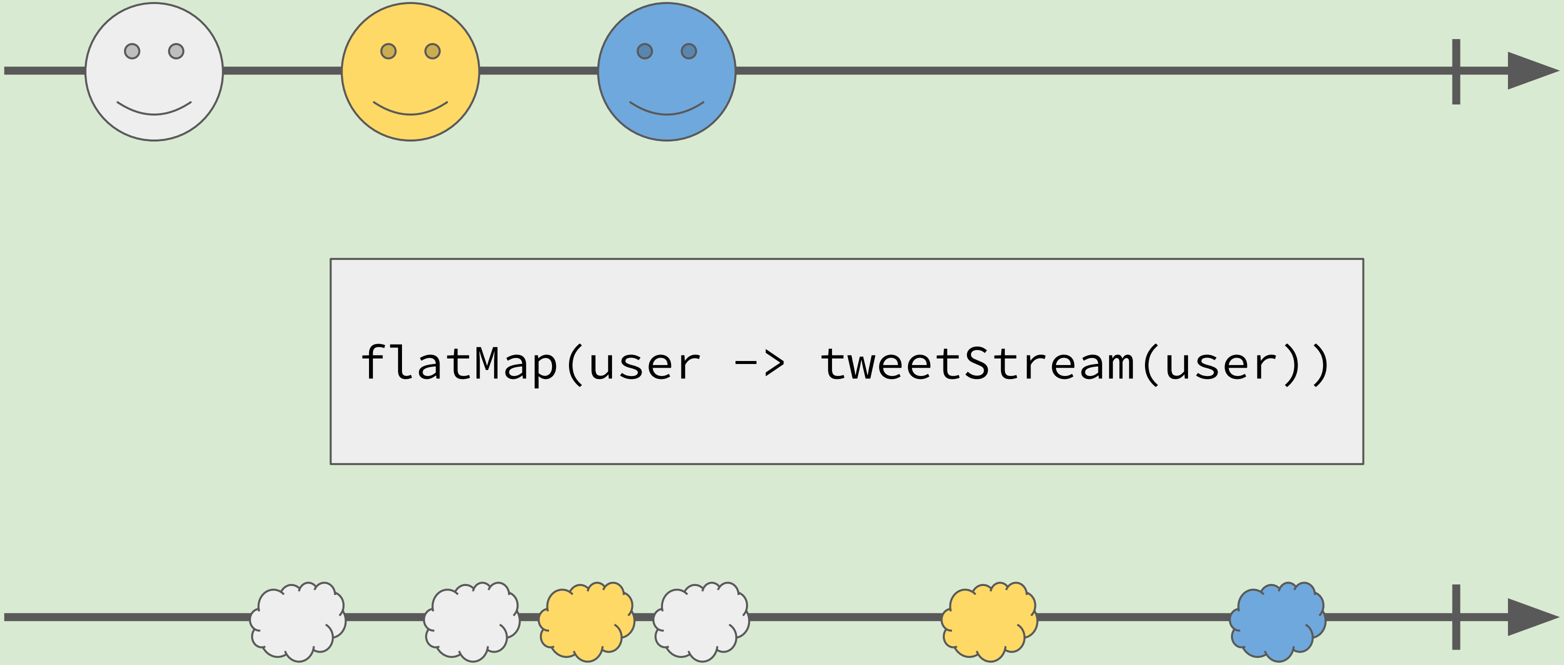
Publisher from HTTP reactive client

resubscribe

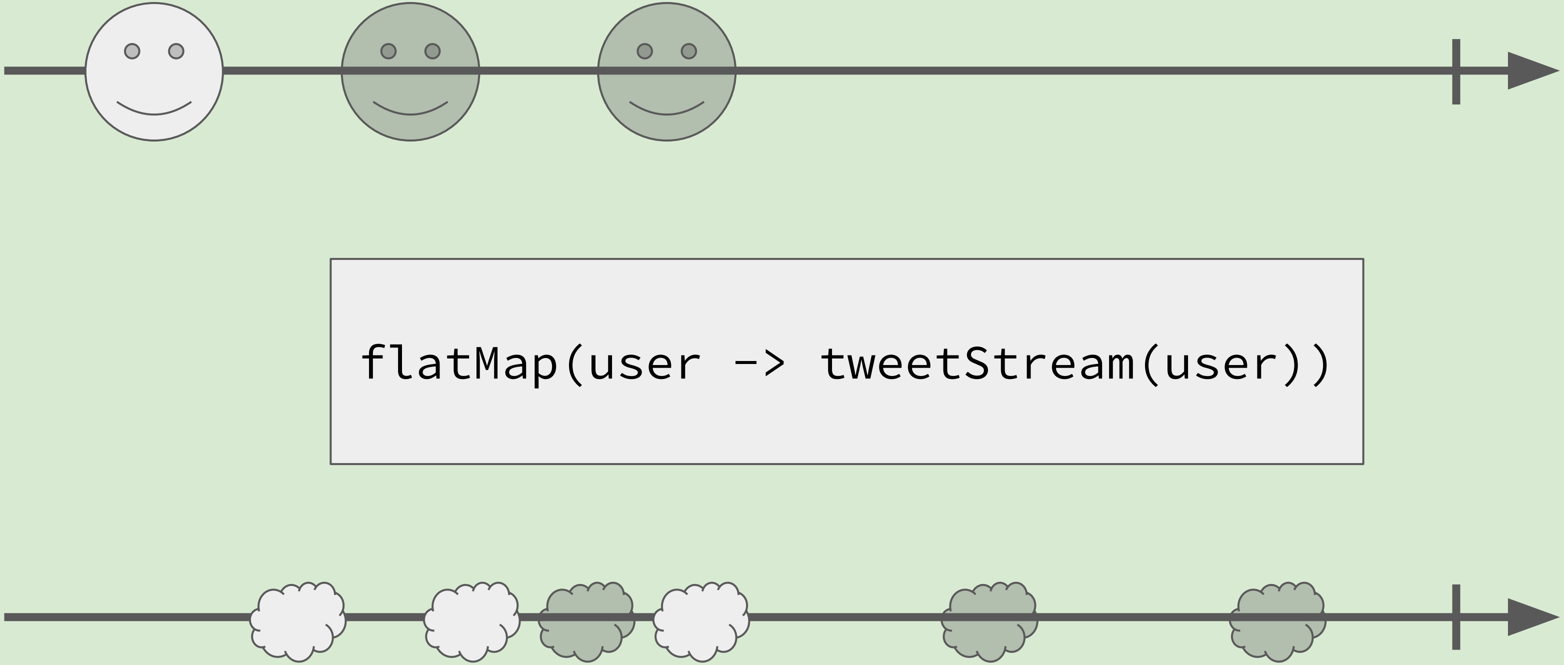
go DEEPER!

async sub-processes with flatMap

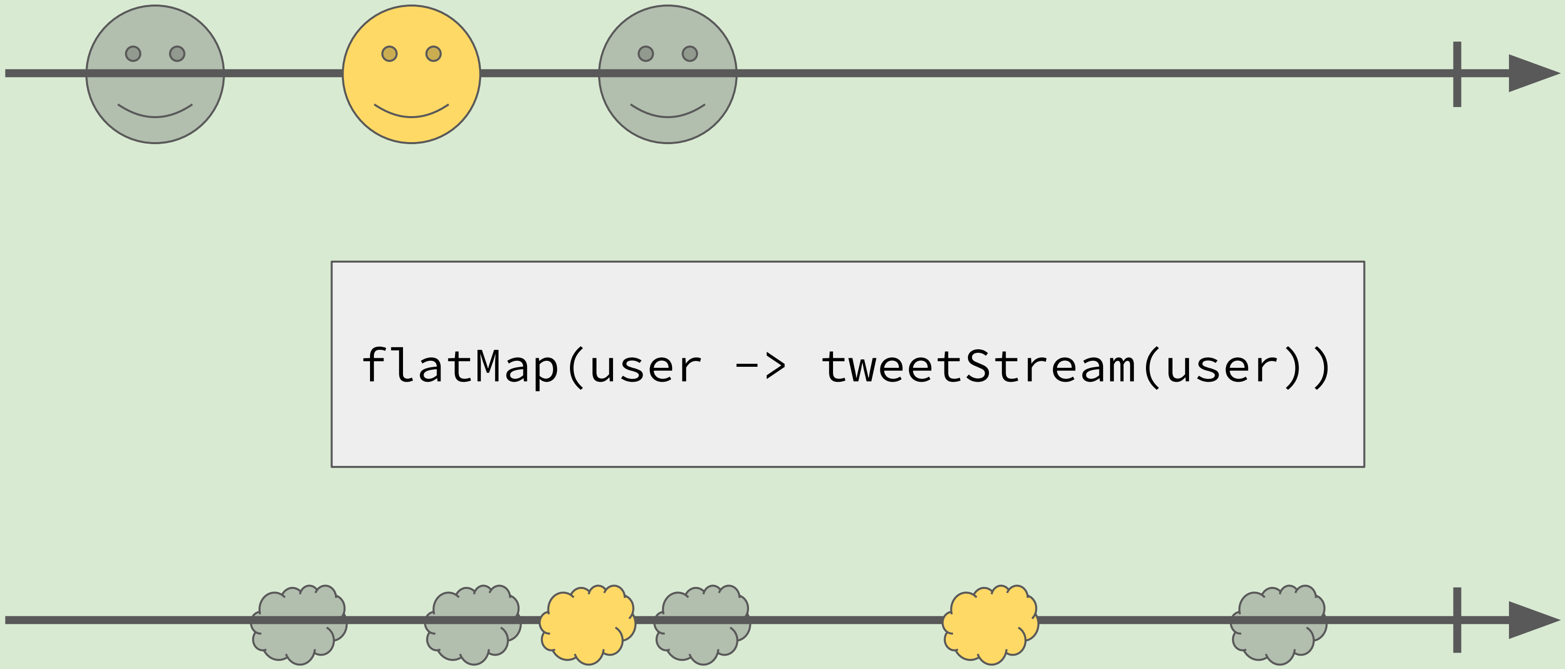

flatMap { }



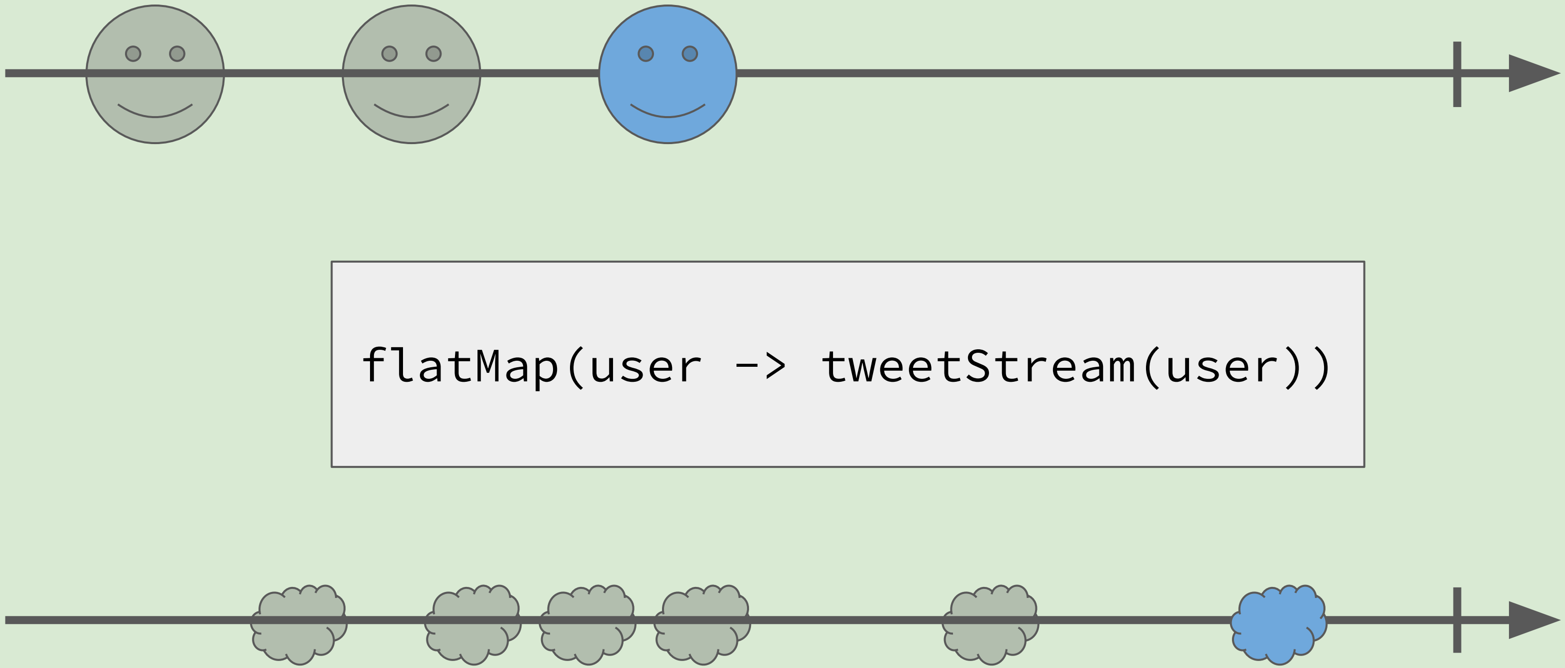
```
flatMap(user -> tweetStream(user))
```



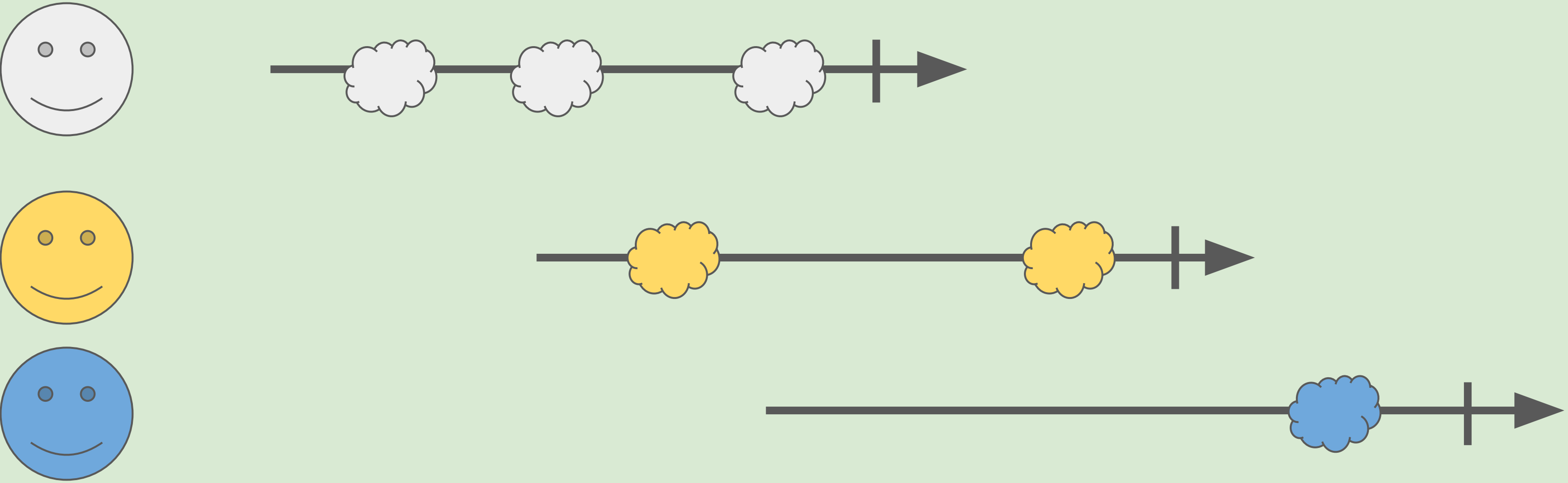
```
flatMap(user -> tweetStream(user))
```

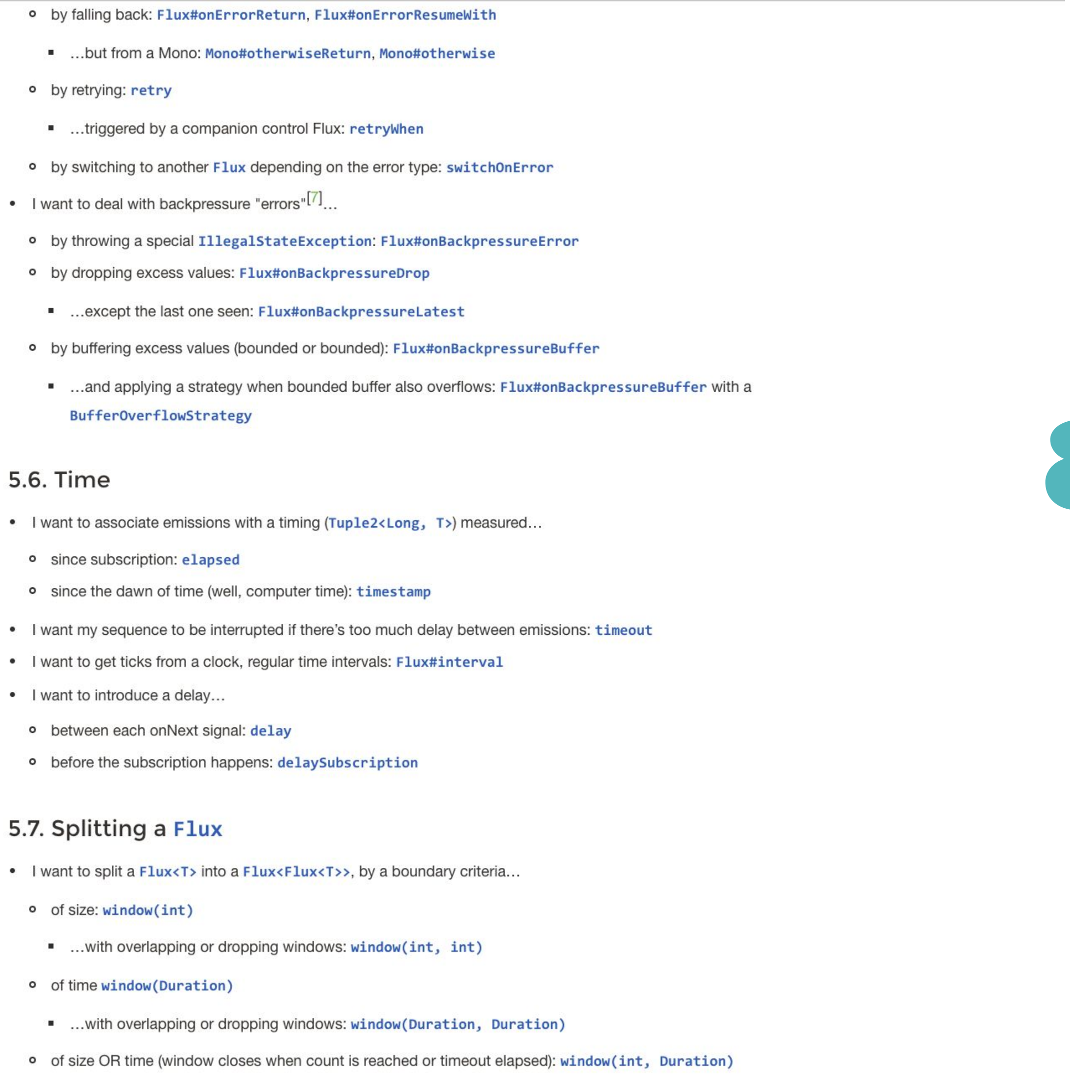


```
flatMap(user -> tweetStream(user))
```



```
flatMap(user -> tweetStream(user))
```





  - by falling back: Flux#onErrorReturn, Flux#onErrorResumeWith
    - …but from a Mono: Mono#otherwiseReturn, Mono#otherwise
  - by retrying: retry
    - …triggered by a companion control Flux: retryWhen
  - by switching to another Flux depending on the error type: switchOnError
- I want to deal with backpressure "errors"[7]…
  - by throwing a special IllegalStateException: Flux#onBackpressureError
  - by dropping excess values: Flux#onBackpressureDrop
    - …except the last one seen: Flux#onBackpressureLatest
  - by buffering excess values (bounded or bounded): Flux#onBackpressureBuffer
    - …and applying a strategy when bounded buffer also overflows: Flux#onBackpressureBuffer with a BufferOverflowStrategy

5.6. Time

- I want to associate emissions with a timing (Tuple2<Long, T>) measured…
  - since subscription: elapsed
  - since the dawn of time (well, computer time): timestamp
- I want my sequence to be interrupted if there's too much delay between emissions: timeout
- I want to get ticks from a clock, regular time intervals: Flux#interval
- I want to introduce a delay…
  - between each onNext signal: delay
  - before the subscription happens: delaySubscription

5.7. Splitting a Flux

- I want to split a Flux<T> into a Flux<Flux<T>>, by a boundary criteria…
  - of size: window(int)
    - …with overlapping or dropping windows: window(int, int)
  - of time window(Duration)
    - …with overlapping or dropping windows: window(Duration, Duration)
  - of size OR time (window closes when count is reached or timeout elapsed): window(int, Duration)

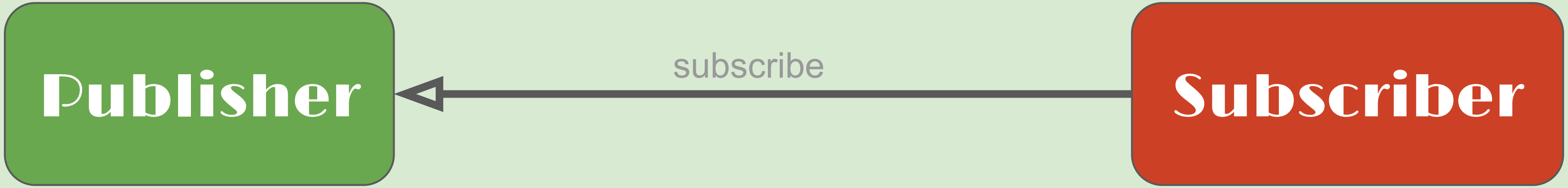
& much

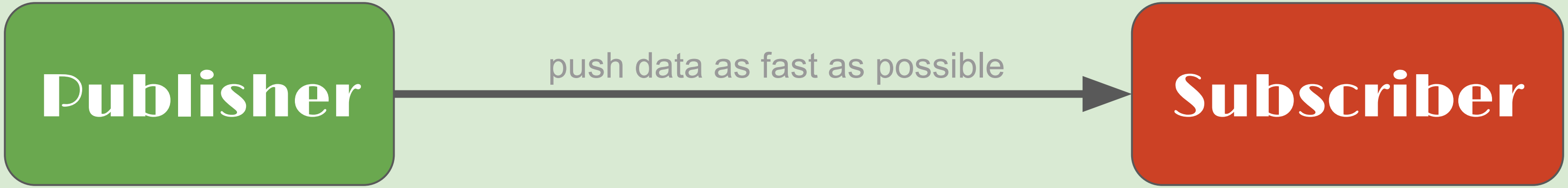
more…

“elements of functional programming”

BACKPRESSURE

and other beasts

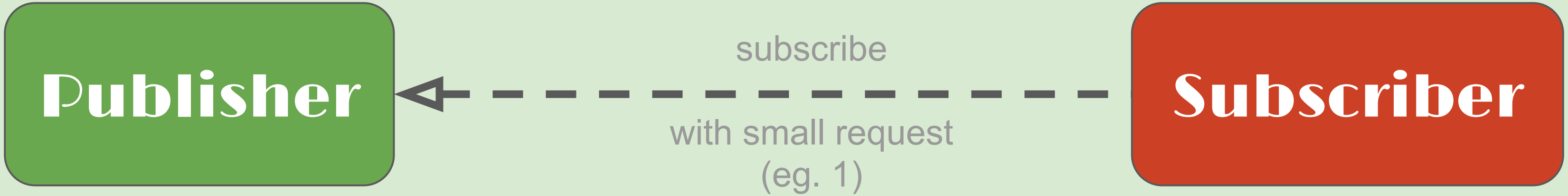

Publisher
subscribe
Subscriber



Publisher

push data as fast as possible

Subscriber

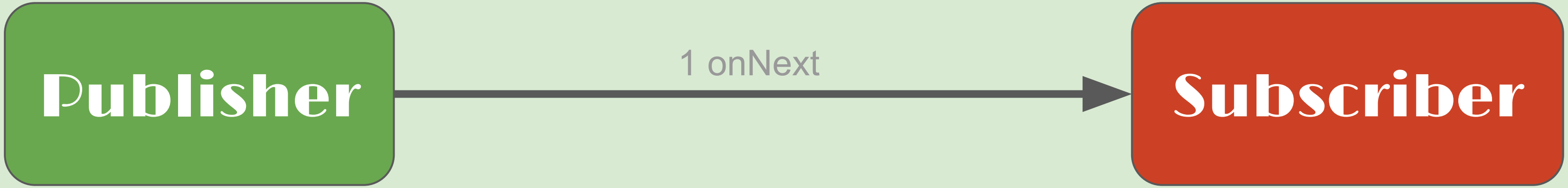


Publisher

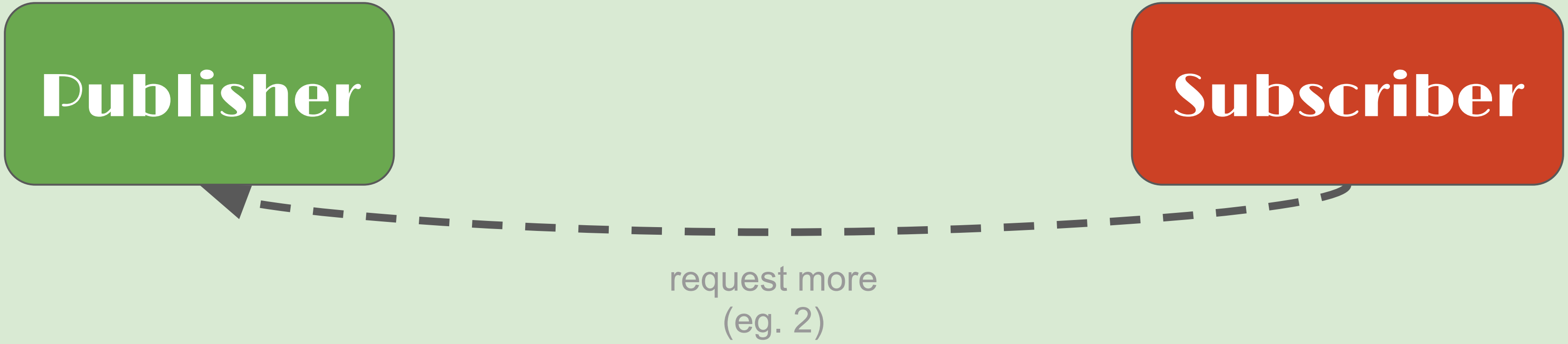
subscribe

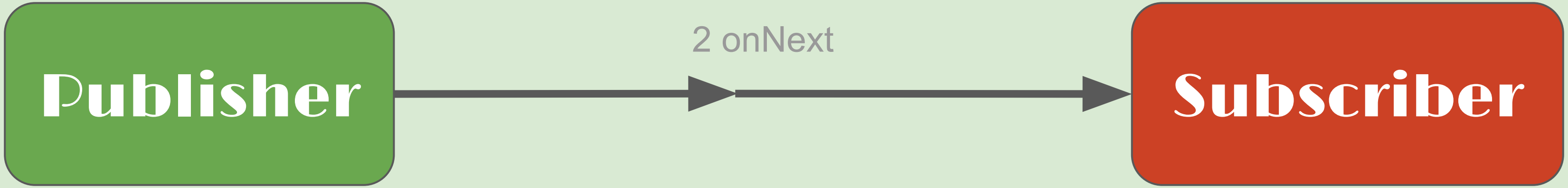
with small request
(eg. 1)

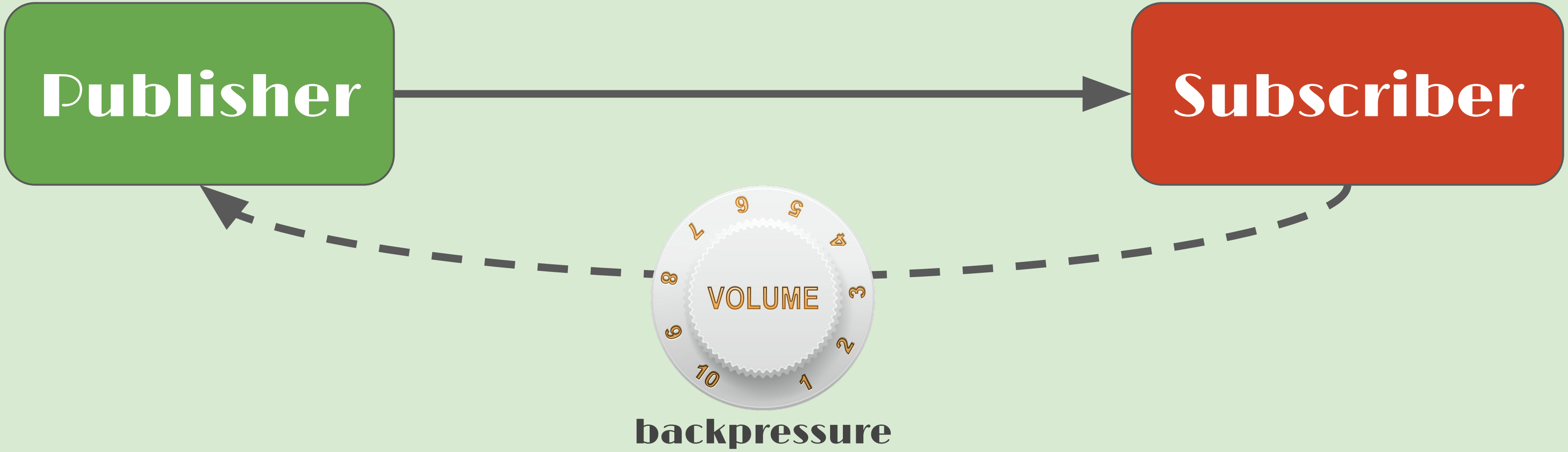
Subscriber



Publisher
1 onNext
Subscriber


Publisher
Subscriber
request more
(eg. 2)


Publisher
2 onNext
Subscriber



Publisher

Subscriber

backpressure

other ways of dealing with backpressure

eg. drop, buffer...

internal

optimisations

macro FUSION

avoids unnecessary request back-and-forth

micro FUSION

share internal structures for less allocation

threading

contexts

Reactor
is
agnostic

however it

facilitates switching

Schedulers

Schedulers

elastic, parallel, single, timer…

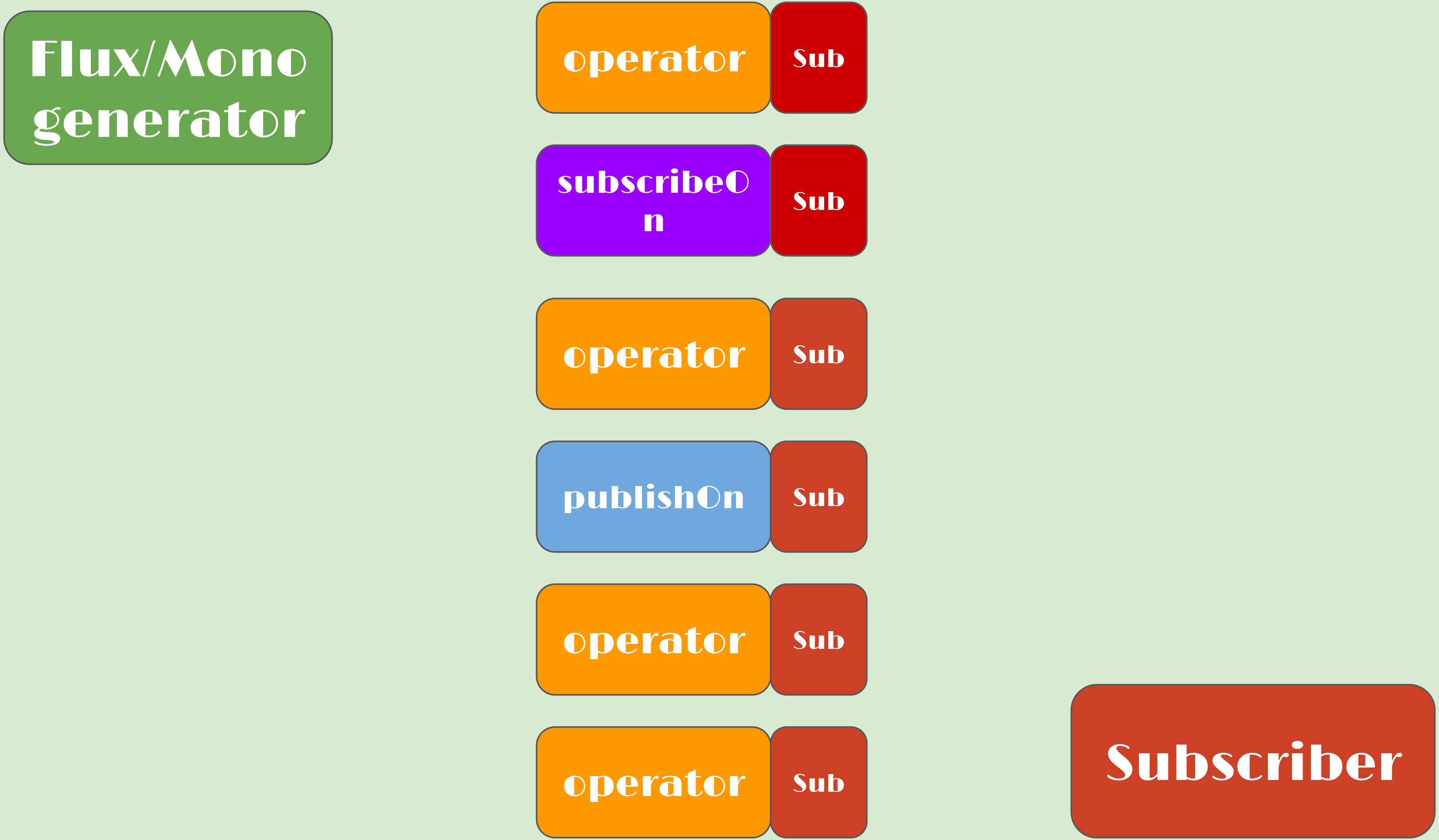
publishOn

switch rest of the flux on a thread

subscribeOn

make the subscription and request happen

on a particular thread



Flux/Mono generator

operator | Sub

subscribeOn | Sub

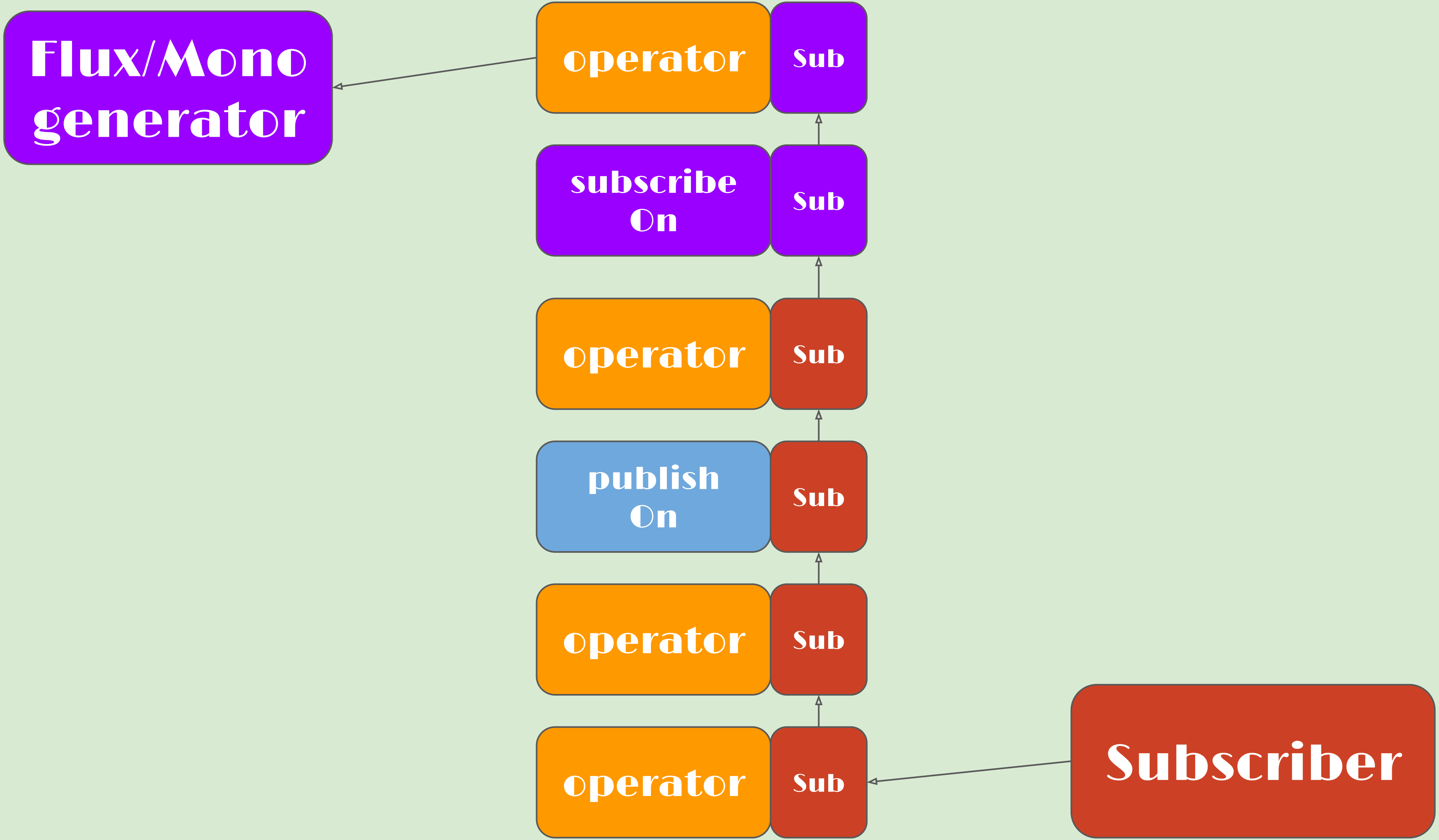
operator | Sub

publishOn | Sub

operator | Sub

operator | Sub

Subscriber



Flux/Mono generator

operator

Sub

subscribe On

Sub

operator

Sub

publish On

Sub

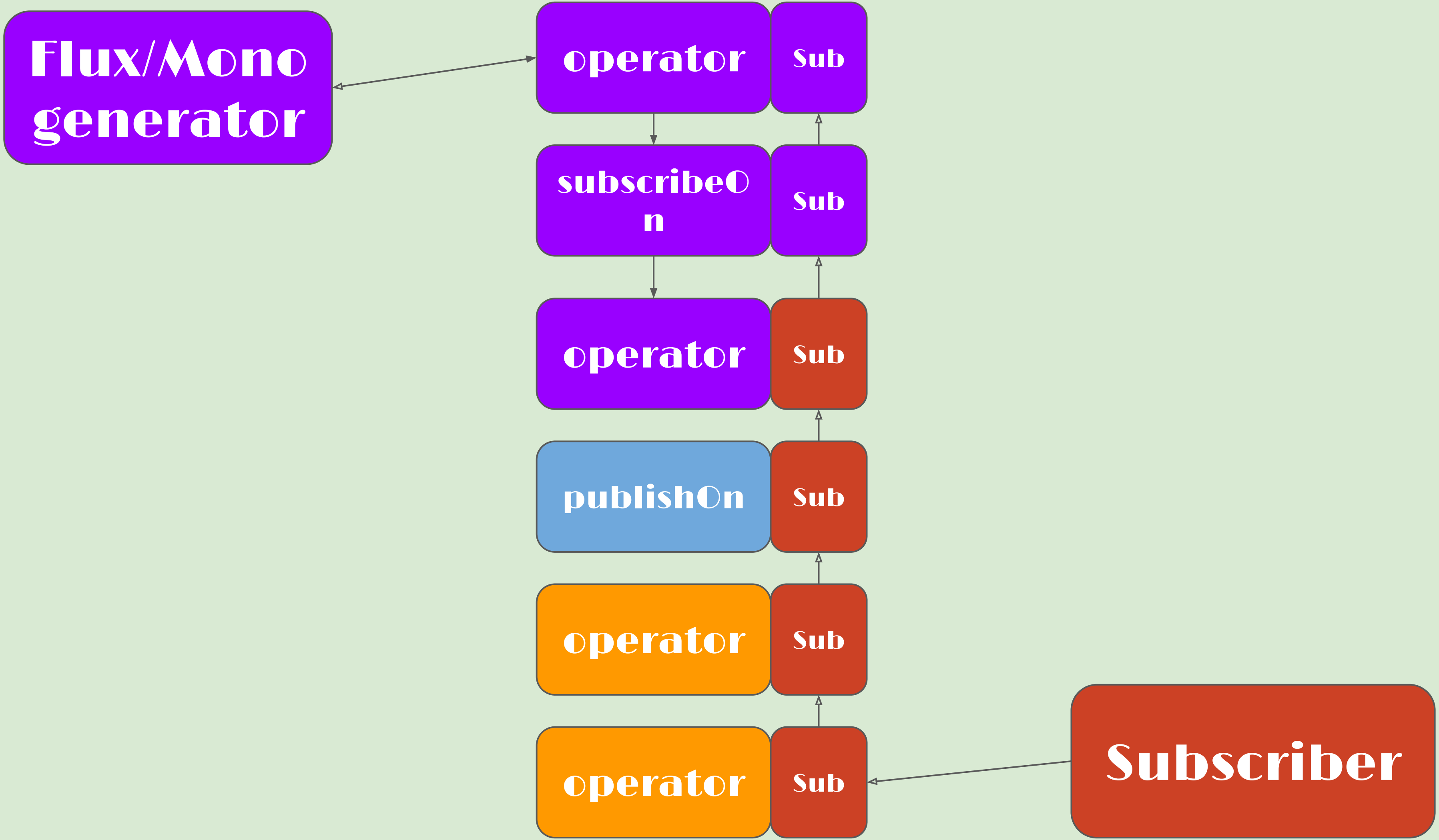
operator

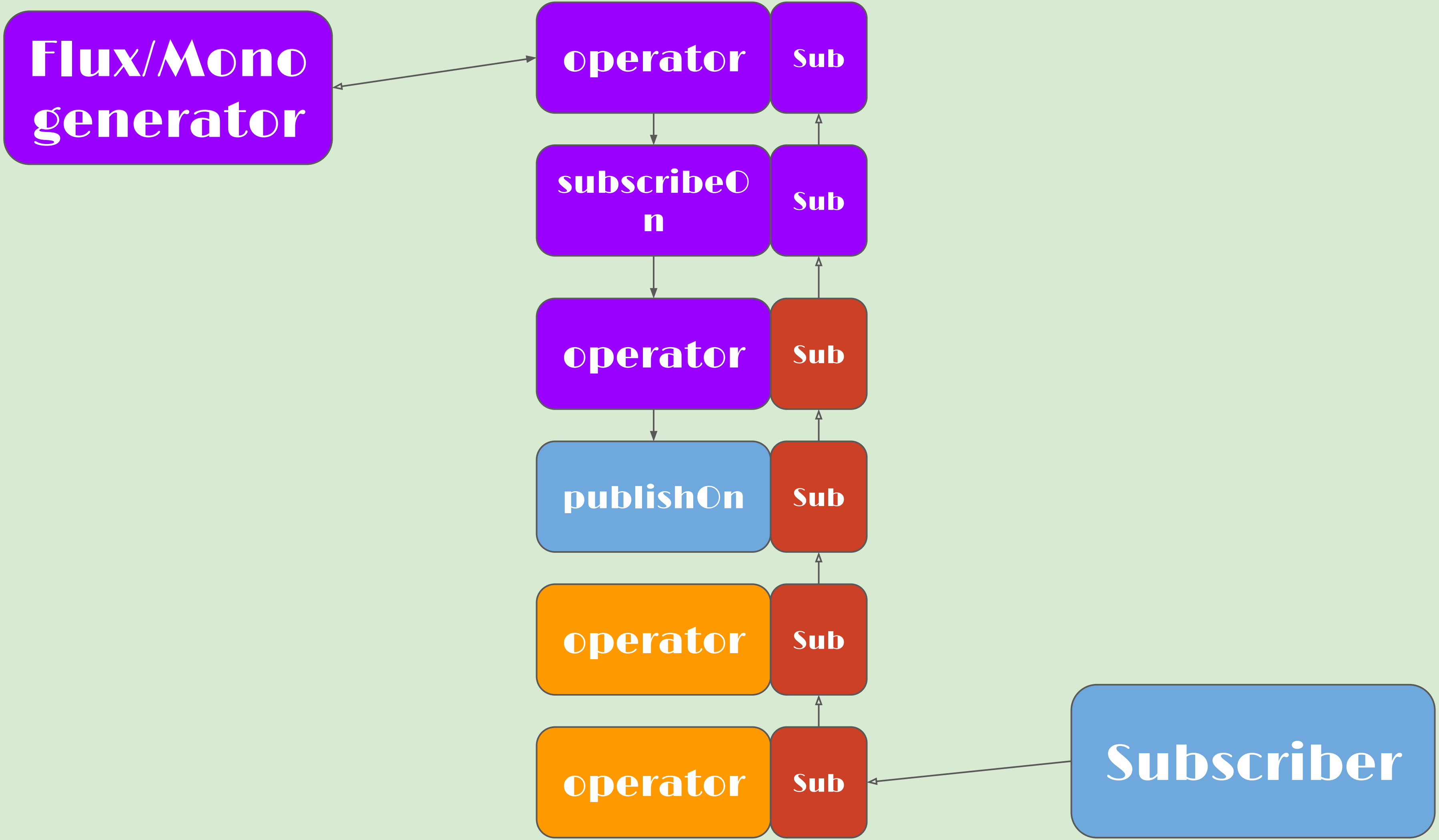
Sub

operator

Sub

Subscriber





Flux/Mono generator

operator

Sub

subscribeOn

Sub

operator

Sub

publishOn

Sub

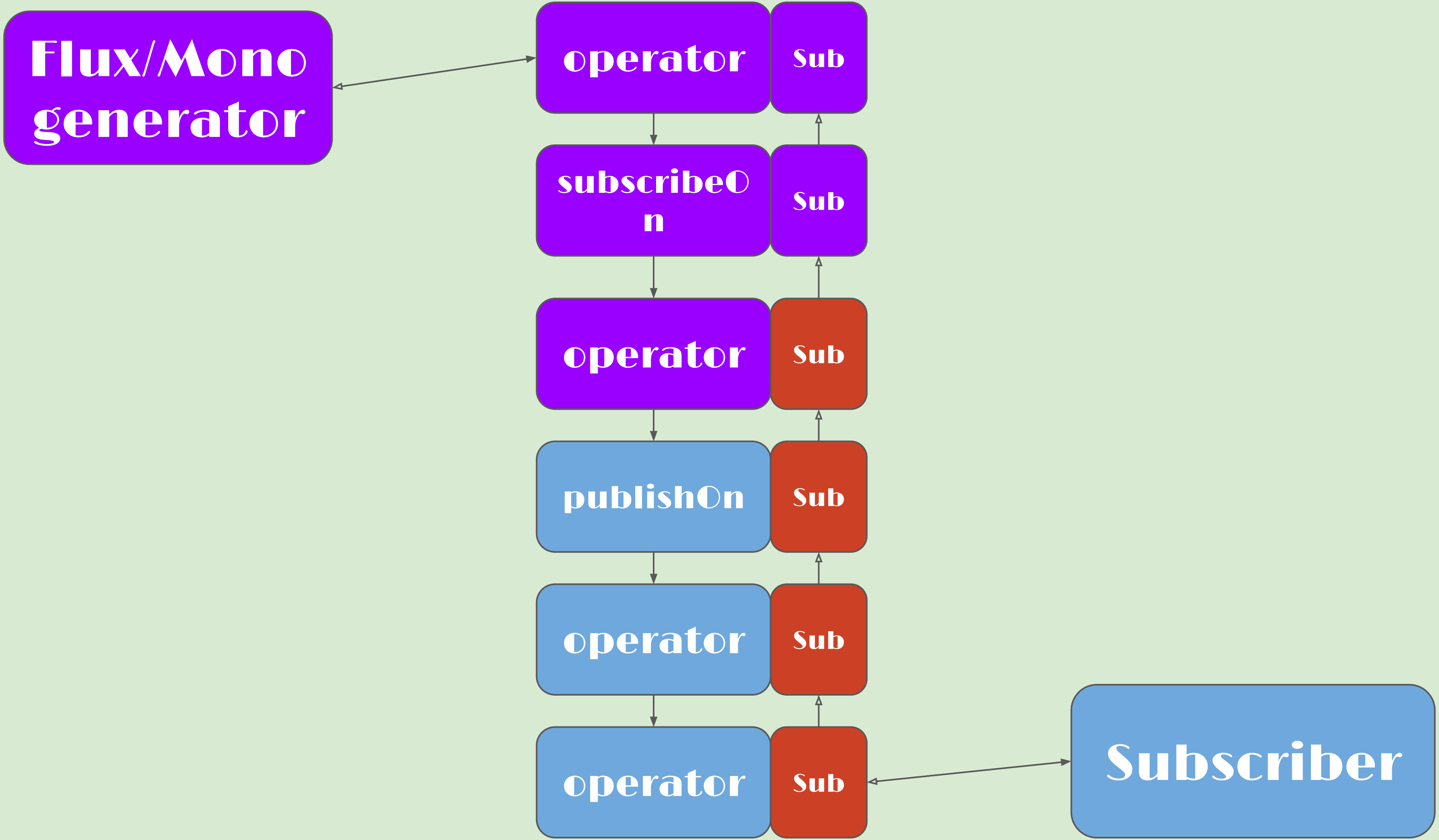
operator

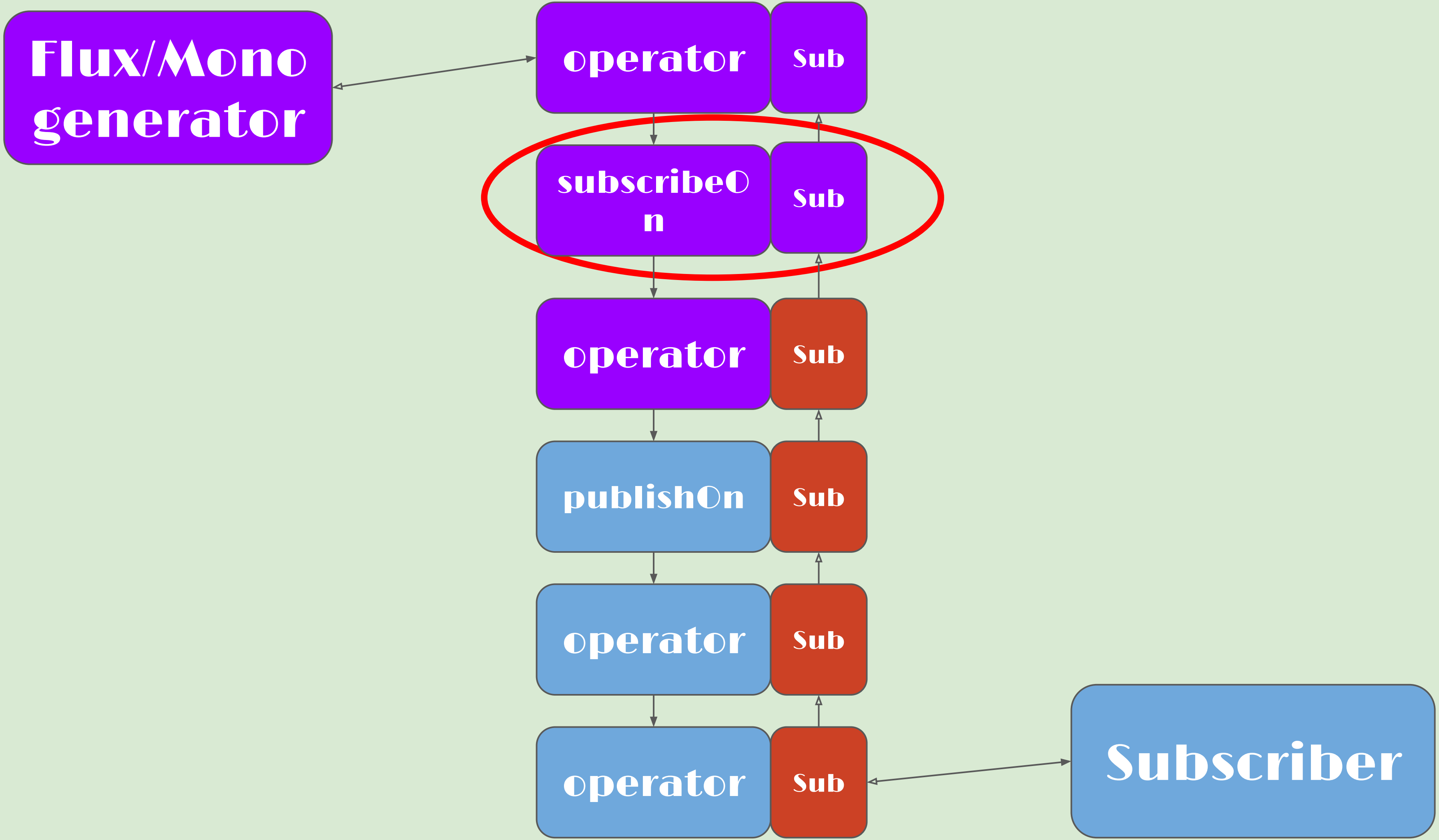
Sub

operator

Sub

Subscriber


Flux/Mono generator
operator
Sub
subscribeOn
Sub
operator
Sub
publishOn
Sub
operator
Sub
operator
Sub
Subscriber


Flux/Mono generator
operator
Sub
subscribeOn
Sub
operator
Sub
publishOn
Sub
operator
Sub
operator
Sub
Subscriber

lock free operators

lock free operators

and Work Stealing

Testing & Debugging

in an asynchronous world

Testing a Publisher

StepVerifier

Testing a Publisher

with Virtual Time *support*

Simulate a source

TestPublisher

Debugging Issues

stacktraces get hard to decipher

usually just show

where Subscription happens

```
java.lang.IndexOutOfBoundsException: Source emitted more than one item
	at reactor.core.publisher.MonoSingle$SingleSubscriber.onNext(MonoSingle.java:120)
	at
reactor.core.publisher.FluxOnAssembly$OnAssemblySubscriber.onNext(FluxOnAssembly.java:314)
...

...
	at reactor.core.publisher.Mono.subscribeWith(Mono.java:2668)
	at reactor.core.publisher.Mono.subscribe(Mono.java:2629)
	at reactor.core.publisher.Mono.subscribe(Mono.java:2604)
	at reactor.core.publisher.Mono.subscribe(Mono.java:2582)
	at reactor.guide.GuideTests.debuggingActivated(GuideTests.java:727)
```

Find where the Flux

was instantiated (assembly)

Checkpoint()

or full assembly tracing

Checkpoint()

or full assembly tracing

costly!

```
Assembly trace from producer [reactor.core.publisher.MonoSingle] :
    reactor.core.publisher.Flux.single(Flux.java:5335)
    reactor.guide.GuideTests.scatterAndGather(GuideTests.java:689)
    reactor.guide.GuideTests.populateDebug(GuideTests.java:702)
```

Reactor and Spring

Reactor and Spring

and do I need Spring to use Reactor ?

NO *philosoraptor* you don't

Reactor 3

is a dependency of

Spring 5

not the other way around

spring 5

Java 8

baseline

reactive

focus

new WEB stack

WebFlux

```
@RestController(“/user”)
public class UserController {

 @GetMapping(“/{id}”)
 Mono<User> getUser(String id) {...}

}
```

functional option

for Routing

Spring Data

reactive repositories

```
@GetMapping(“/{id}”)
Mono<User> getUser(String id) {
  return reactiveRepo.findOne(id);
}
```

Reactor and the Network

reactor-netty

reactor-netty

builds on Netty to provide

reactive I/O

Client / Server

TCP

or udp

Http

and WebSockets

```
HttpServer.create(0)
  .newHandler((in, out) -> out
    .sendWebsocket((i, o) ->
      o.options(opt -> opt.flushOnEach())
       .sendString(Flux.just("test")
                   .delayElementsMillis(100)
                   .repeat())
     )
  )
.block();
```

still a bit **low level**

still a bit **low level**

reactor-kafka

topics as

Flux<T>

reactive API

over Kafka Producer | Consumer

send(Flux)

into Kafka

Flux receive()

from Kafka

(*currently in* **MILESTONE 1**)

Questions?

Thanks!

Credits

- **Springfield Plant:** copyright FOX
- **Raised Hand:** CC0 (via Pixabay)
- **Checklist:** CC-By Crispy (via Flickr)
- **Robot Devil:** copyright FOX
- **Volume Knob:** CC0 (via Pixabay)
- **Camel Shape:** CC0 (via Pixabay)
- **Dromedary Shape:** CC-By-SA USPN,Whidou (via Wikimedia)
- **Dam:** CC-By-SA Matthew Hatton (via geograph.org.uk)
- **Cogs:** CC0 (via publicdomainpictures.net)
- **Thread Balls:** CC0 (via Pixabay)
- **The Fortune Teller:** Georges de la Tour (public domain)
- **Microphone:** CC0 (via Pexels)
- **End Sands:** CC0 (via Pixabay)
- **logos:** Pivotal, Spring, Twitter and Github logo copyright their respective companies.