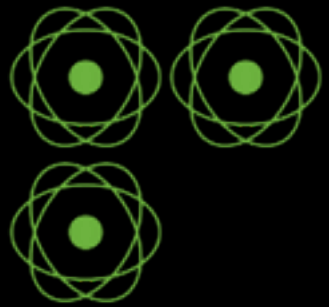
Reactor, Reactive Streams and the MicroService architecture

Stephane Maldini



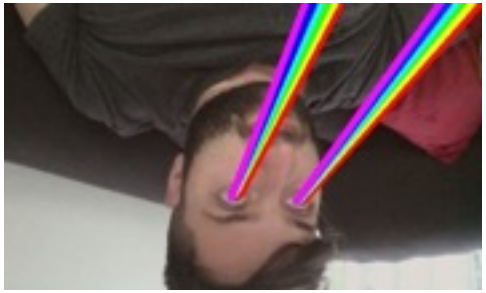
Pivotal

@smaldini - solve 9 issues, create 10 problems

Customer Success Organization @ Pivotal

Reactor Committer II

Try to contribute to Reactive-Streams




SAY MICROSERVICE
ONE MORE TIME!
memegenerator.net

NanoService, MicroService, NotTooBigService™…

Aperture Sciences Test 981:
Observe the following examples

NanoService, MicroService, NotTooBigService™…

cat file.csv

NanoService, MicroService, NotTooBigService™…

cat file.csv | grep ‘doge’

NanoService, MicroService, NotTooBigService™…

cat file.csv | grep ‘doge’ | sort

NanoService, MicroService, NotTooBigService™…

POST [json] http://dogecoin.money/send/id

NanoService, MicroService, NotTooBigService™…

POST [json] http://dogecoin.money/send/id

—> GET [json] http://dogeprofile.money/id

NanoService, MicroService, NotTooBigService™…

POST [json] http://dogecoin.money/send/id

—> GET [json] http://dogeprofile.money/id

—> POST [json] http://nsa.gov.us/cc/trace/id

NanoService, MicroService, NotTooBigService™…

```
userService.auth(username,password)
```

NanoService, MicroService, NotTooBigService™…

userService.auth(username,password)

—> userService.hashPassword(password)

NanoService, MicroService, NotTooBigService™…

userService.auth(username,password)

—> userService.hashPassword(password)

—> userService.findByNameAndHash(name)

NanoService, MicroService, NotTooBigService™…

- A SomethingService will **always need to interact**
  - With the user
  - With other services

- The **boundary** between services is the real deal

And this threat has a name

And this threat has a name

Latency

UberFact : *Humans don’t really enjoy waiting*

UberFact : *Humans don’t really enjoy waiting*

Neither do The Machines

What is latency doing to you ?

- **Loss of revenues**
  - because users **switched to another site/app**
  - because services are **compounding inefficiency**
  - because **aggressive scaling** will be needed

What is latency doing to you ?

- **Loss of revenues**
  - because users **switched to another site/app**
  - because services are **compounding inefficiency**
  - because **aggressive scaling** will be needed

‘2 bucks prediction’ : tech team turnover will increase and keep mourning about how crap is their design

Loading 

Loading 

All hail Reactive Programming

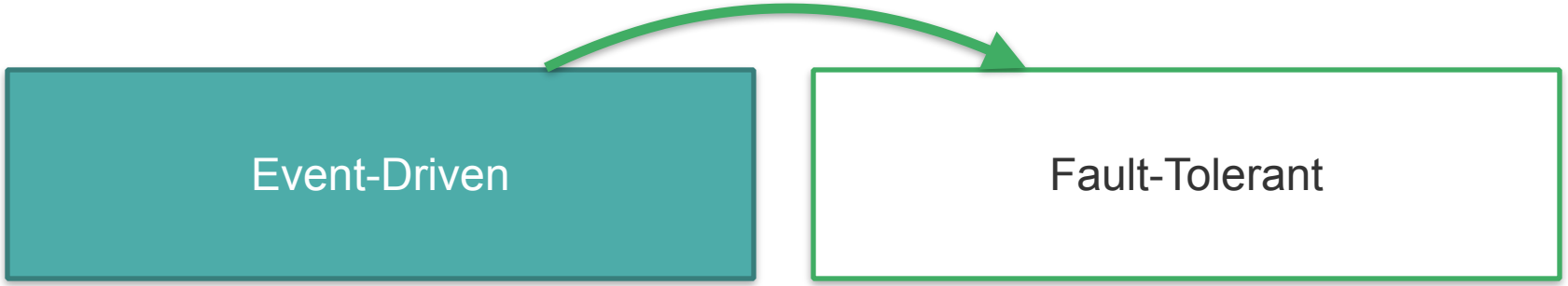
- A possible answer to this issue
- The very nature of **Reactor**, look at the name dude
- A fancy buzz-word that might work better than MDA or SOA
- A simple accumulation of years of engineering

No sh*t, what is Reactive Programming ?

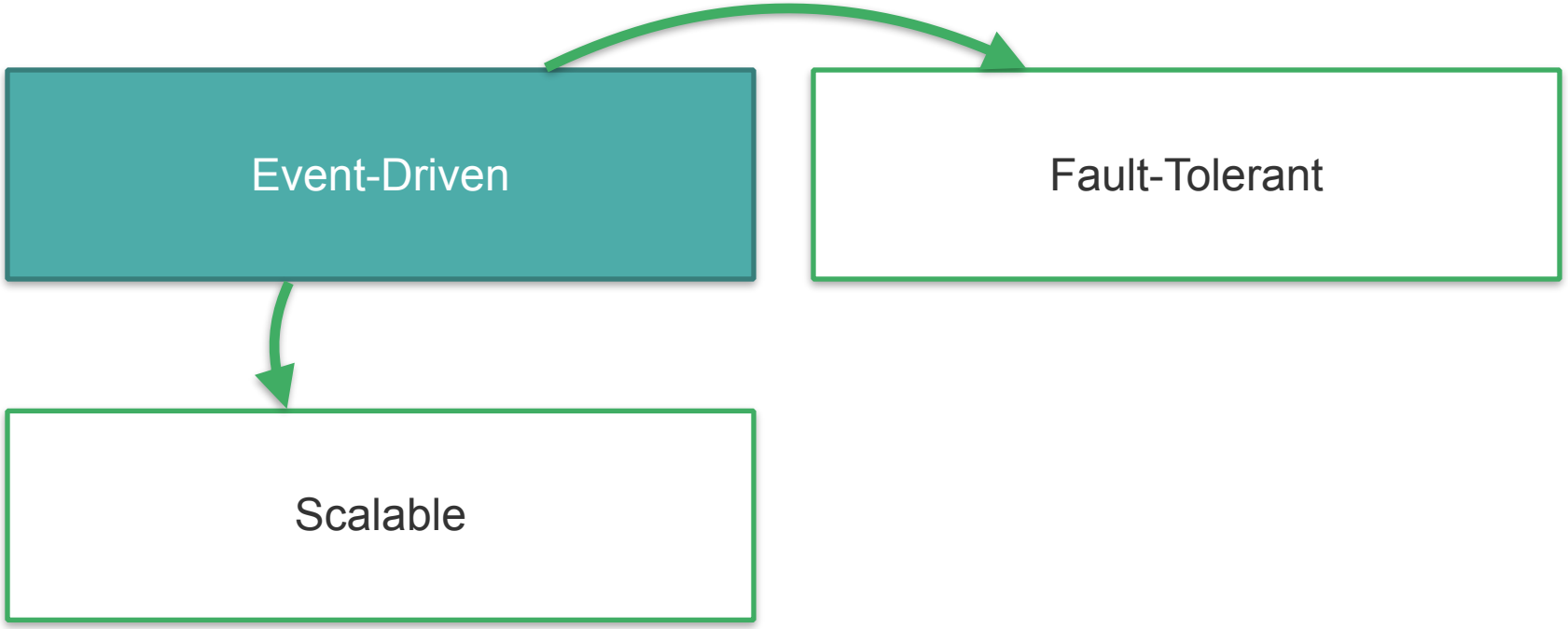
No sh*t, what is Reactive Programming ?

Event-Driven

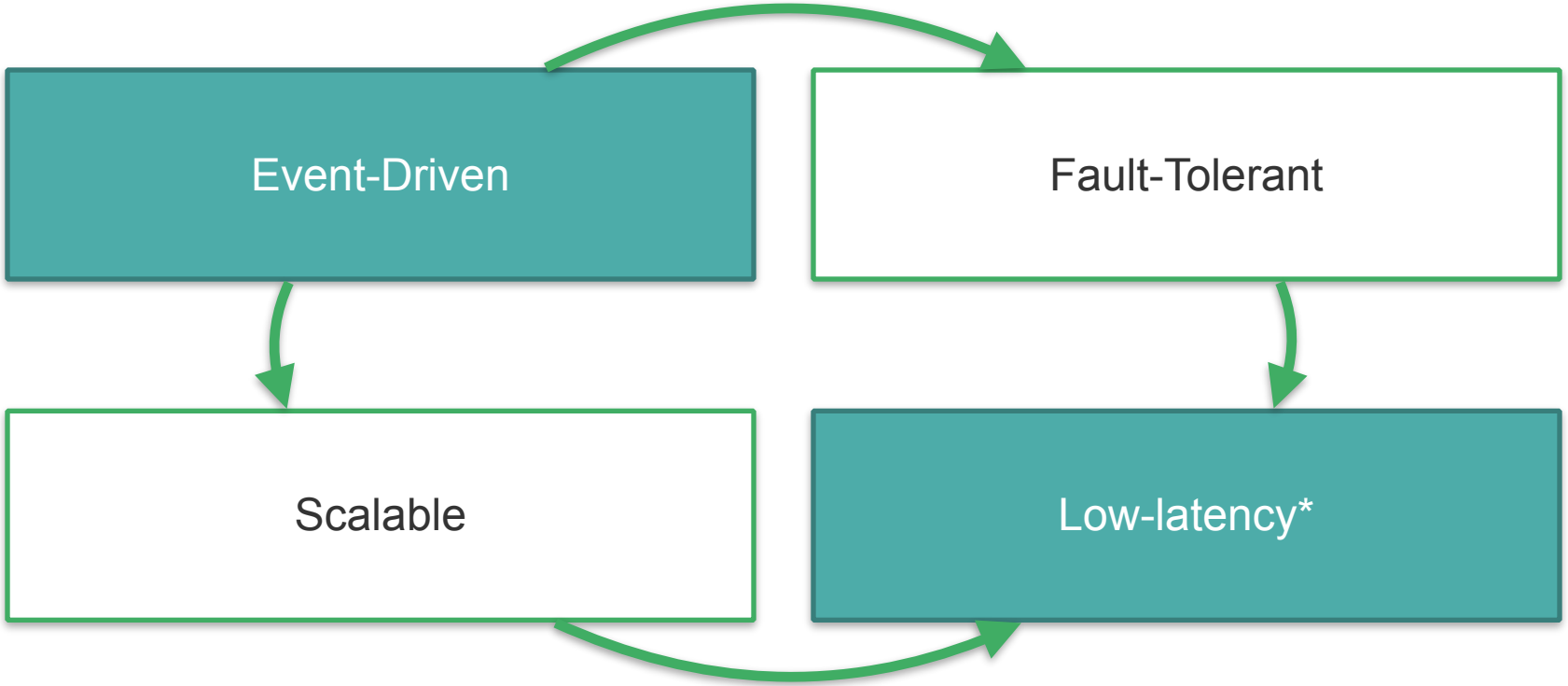
No sh*t, what is Reactive Programming ?



No sh*t, what is Reactive Programming ?



No sh*t, what is Reactive Programming ?



Reactive Architecture ?

- A **Reactive** system **MUST** be resilient
  - splitting concerns to achieve error bulk-heading and modularity
- A **Reactive** system **MUST** be scalable
  - scale-up : partition work across CPUs
  - scale-out : distribute over peer nodes

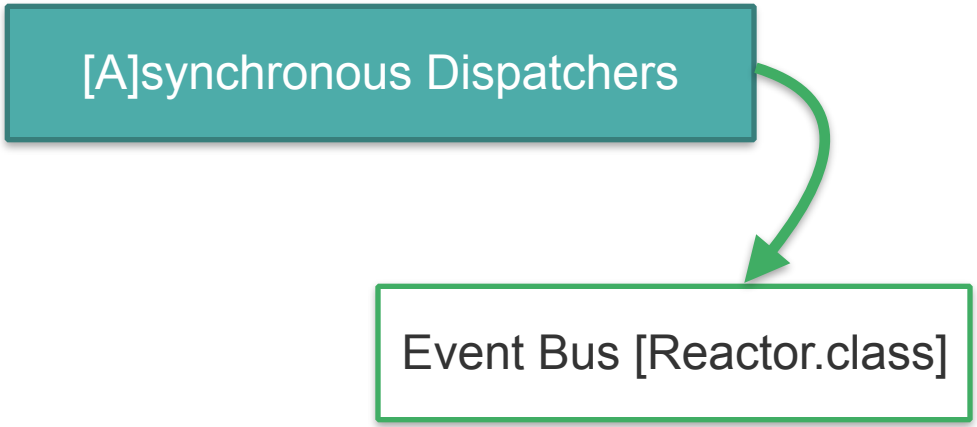
Reactor has 99 problems but Latency isn’t one

Reactor-Core features

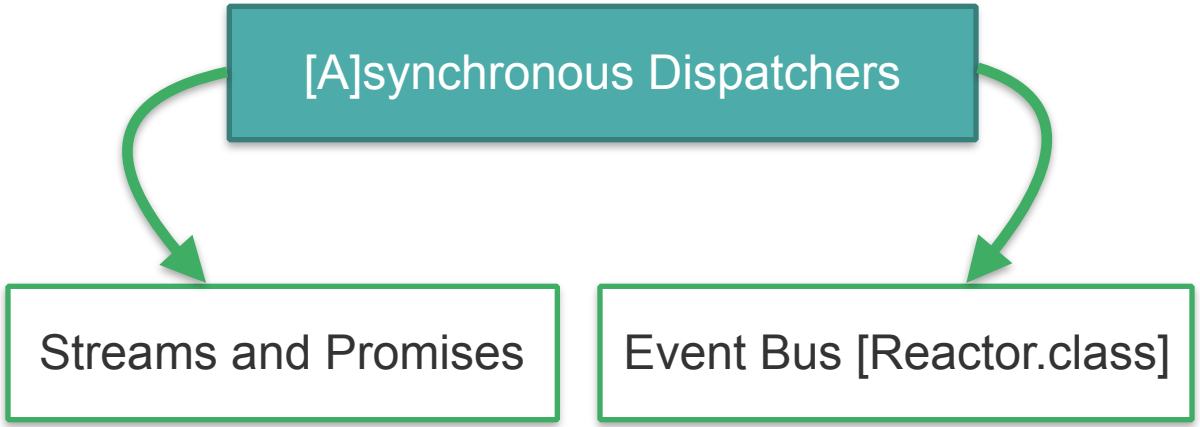
Reactor-Core features

[A]synchronous Dispatchers

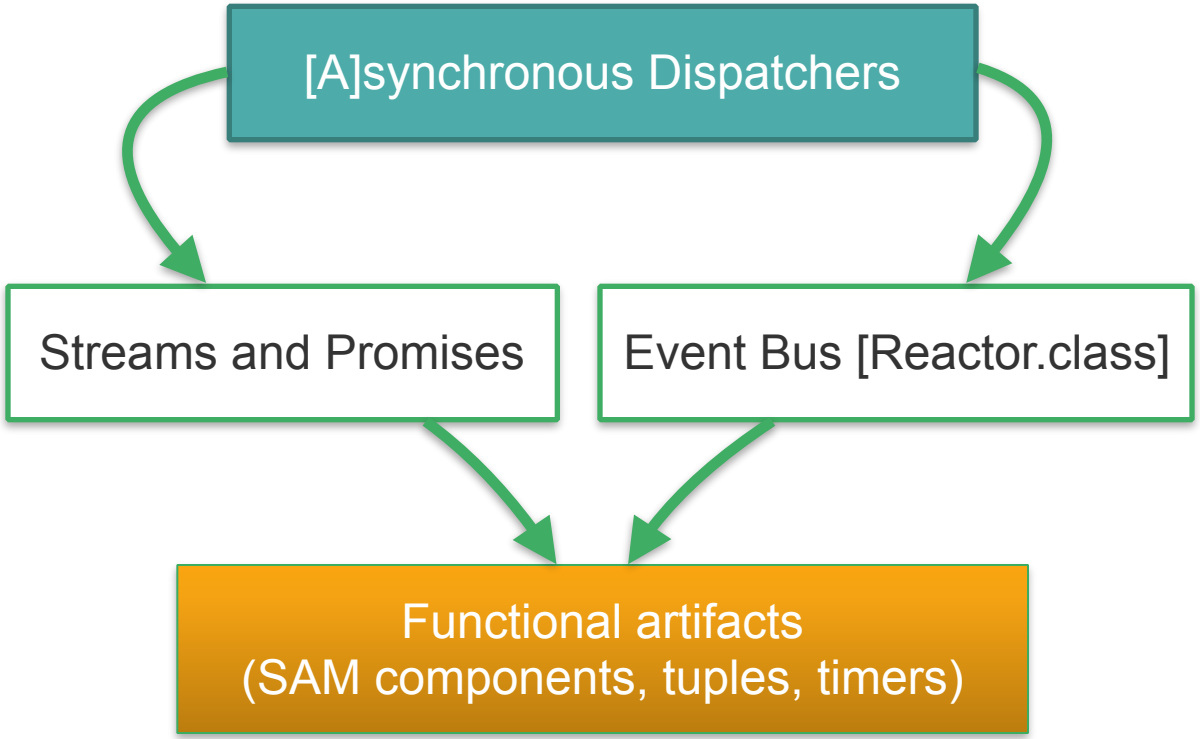
Reactor-Core features



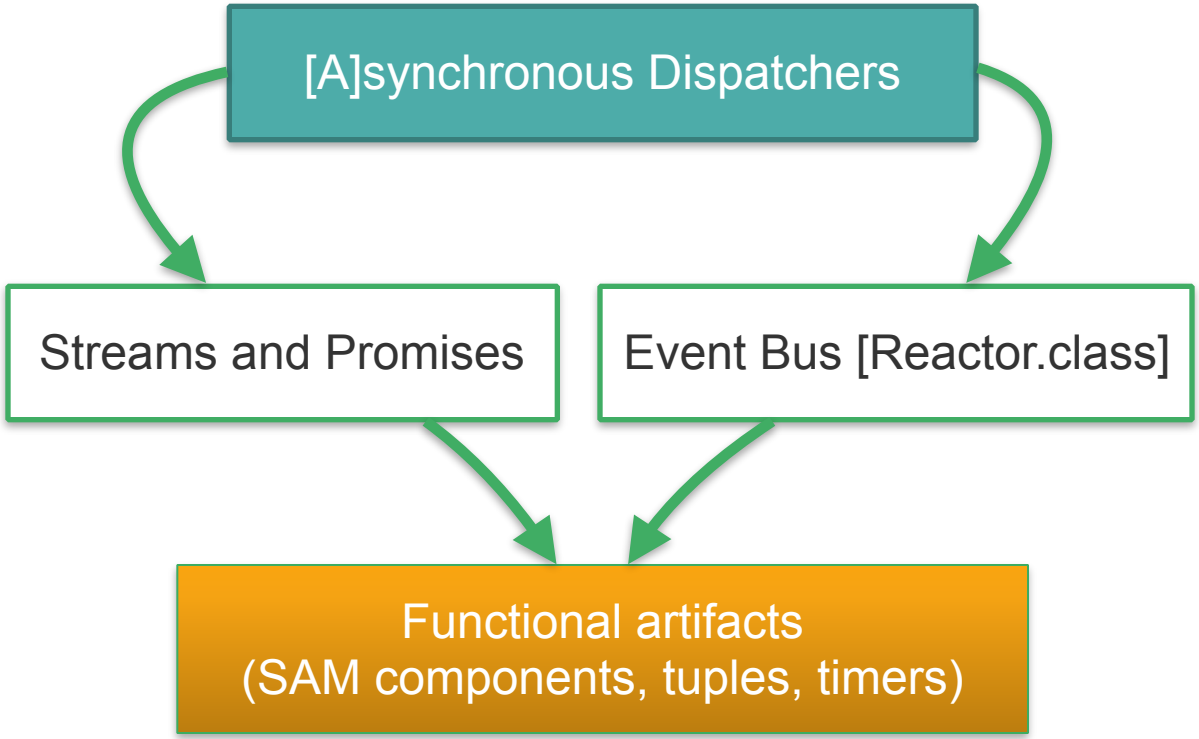
Reactor-Core features



Reactor-Core features

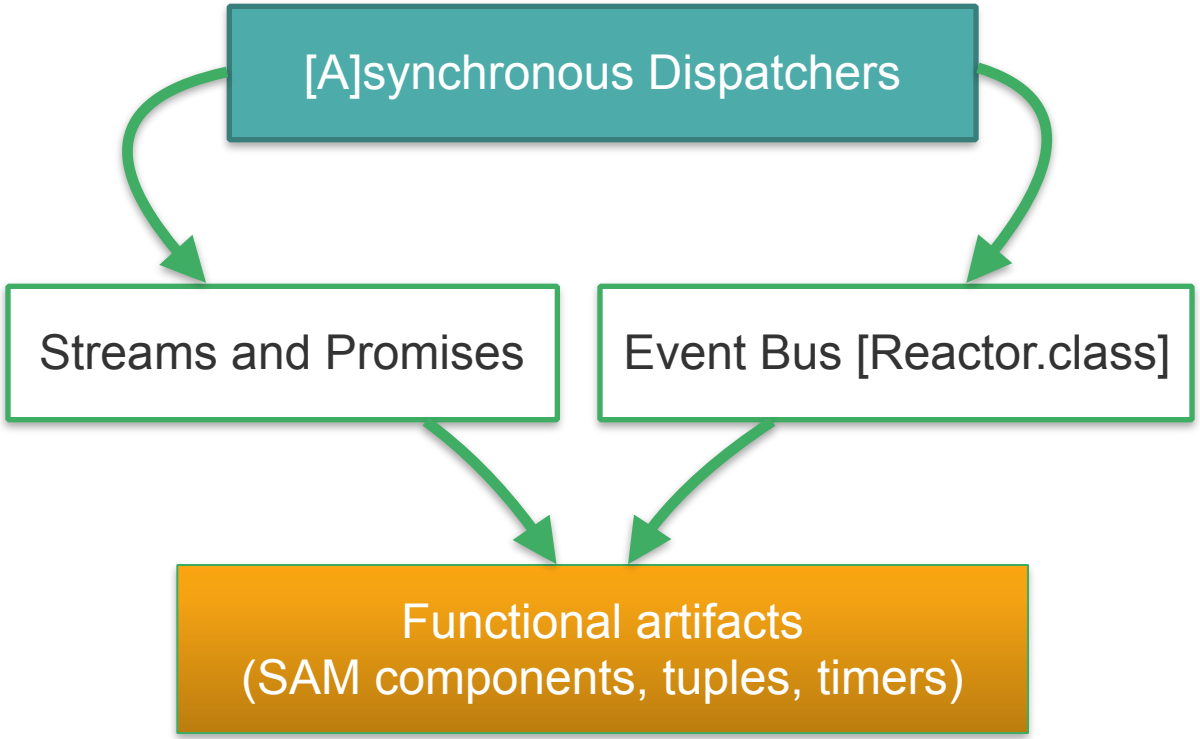


Reactor-Core features



Fast IO
[buffer, net,
persistent queues,
codec]

Reactor-Core features



Fast IO
[buffer, net, persistent queues, codec]

Fast Data
[allocators, batch-processors]

Stream?

Stream

Stream!

Using a Stream ?

```
import reactor.rx.spec.Streams

def stream = Streams.defer()

stream.map{ name ->
    Tuple.of(name, 'so wow')
}.map{ tuple ->
    Tuple.of(tuple.t1, "$tuple.t2, much sad")
}.consume{ tuple ->
    println "bye bye ! $tuple.t2... $tuple.t1"
}

stream.broadcastNext('Doge')
```

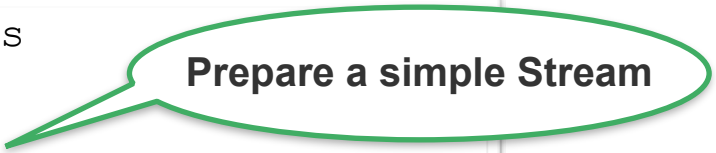
Using a Stream ?

```
import reactor.rx.spec.Streams

def stream = Streams.defer()

stream.map{ name ->
    Tuple.of(name, 'so wow')
}.map{ tuple ->
    Tuple.of(tuple.t1, "$tuple.t2, much sad")
}.consume{ tuple ->
    println "bye bye ! $tuple.t2... $tuple.t1"
}

stream.broadcastNext('Doge')
```



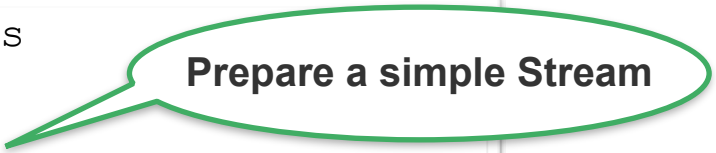
Using a Stream ?

```
import reactor.rx.spec.Streams

def stream = Streams.defer()

stream.map{ name ->
    Tuple.of(name, 'so wow')
}.map{ tuple ->
    Tuple.of(tuple.t1, "$tuple.t2, much sad")
}.consume{ tuple ->
    println "bye bye ! $tuple.t2... $tuple.t1"
}

stream.broadcastNext('Doge')
```



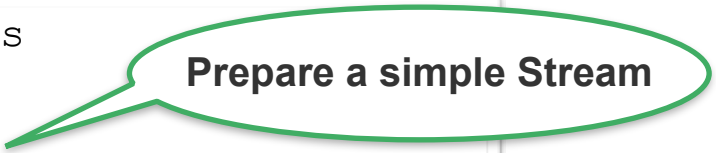
Using a Stream ?

```
import reactor.rx.spec.Streams

def stream = Streams.defer()

stream.map{ name ->
    Tuple.of(name, 'so wow')
}.map{ tuple ->
    Tuple.of(tuple.t1, "$tuple.t2, much sad")
}.consume{ tuple ->
    println "bye bye ! $tuple.t2... $tuple.t1"
}

stream.broadcastNext('Doge')
```



Using a Stream ?

```
import reactor.rx.spec.Streams

def stream = Streams.defer()

stream.map{ name ->
    Tuple.of(name, 'so wow')
}.map{ tuple ->
    Tuple.of(tuple.t1, "$tuple.t2, much sad")
}.consume{ tuple ->
    println "bye bye ! $tuple.t2... $tuple.t1"
}

stream.broadcastNext('Doge')
```

Prepare a simple Stream

1st step

2nd step

Terminal callback

Using a Stream ?

```
import reactor.rx.spec.Streams

def stream = Streams.defer()

stream.map{ name ->
    Tuple.of(name, 'so wow')
}.map{ tuple ->
    Tuple.of(tuple.t1, "$tuple.t2, much sad")
}.consume{ tuple ->
    println "bye bye ! $tuple.t2... $tuple.t1"
}

stream.broadcastNext('Doge')
```

Prepare a simple Stream

1st step

2nd step

Terminal callback

Send some data into the stream

Using a Stream ?

- Embedded data-processing
- Event Processing
- Metrics, Statistics
- Micro-Batching
- Composition
- Error Handling

Defining a Stream

- Represents a **sequence of data,** possibly **unbounded**
- Provide for processing API such as **filtering and enrichment**
- **Not** a ***Collection,*** **not** a ***Storage***

Stream VS Event Bus [Reactor]

- Works great combined (stream distribution)
- Type-checked flow
- Publisher/Subscriber tight control
- No Signal concurrency

Stream VS Event Bus [Reactor]

- Works great combined (stream distribution)
- Type-checked flow
- Publisher/Subscriber tight control
- No Signal concurrency

Rule of thumb:

if nested event composition > 2, switch to Stream

Hot Stream vs Cold Stream

- An Hot Stream multi-casts real-time signals
  - think **Trade, Tick, Mouse Click, Websocket**

- A Cold Stream uni-casts deferred signals
  - think **File, Array, Computation result (Future)**

Reactor : Iterable Cold Stream

```
Streams
    .just(1, 2, 3, 4, 5)
    .take(3)
    .subscribe(System.out::println);
```

Reactor : AMQP Hot Stream

```
LapinStreams
        .fromQueue(queueName)
        .dispatchOn(Environment.get())
        .timeout(30)
        .subscribe(System.out::println);
```

Reactor : AMQP Hot Stream

```
LapinStreams
        .fromQueue(queueName)
        .dispatchOn(Envi
        .timeout(30)
        .subscri                    .println);
```

COMING SOON IN A REPOSITORY NEAR YOU

Introducing Reactive Streams Specification !

What is defined by Reactive Streams ?

What is defined by Reactive Streams ?

Async non-blocking data sequence

What is defined by Reactive Streams ?

Async non-blocking data sequence

Minimal resources requirement

What is defined by Reactive Streams ?

Async non-blocking data sequence

Interoperable protocol
(Threads, Nodes…)

Minimal resources requirement

What is defined by Reactive Streams ?

Async non-blocking data sequence

Interoperable protocol
(Threads, Nodes…)

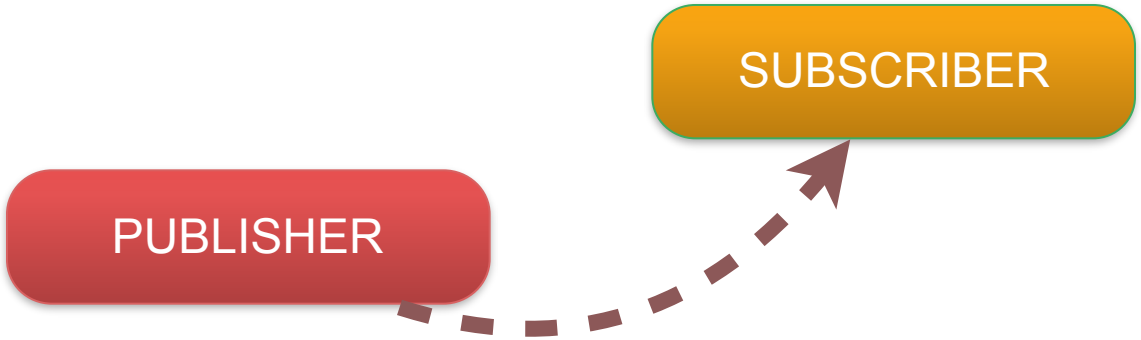
Async non-blocking flow-control

Minimal resources requirement

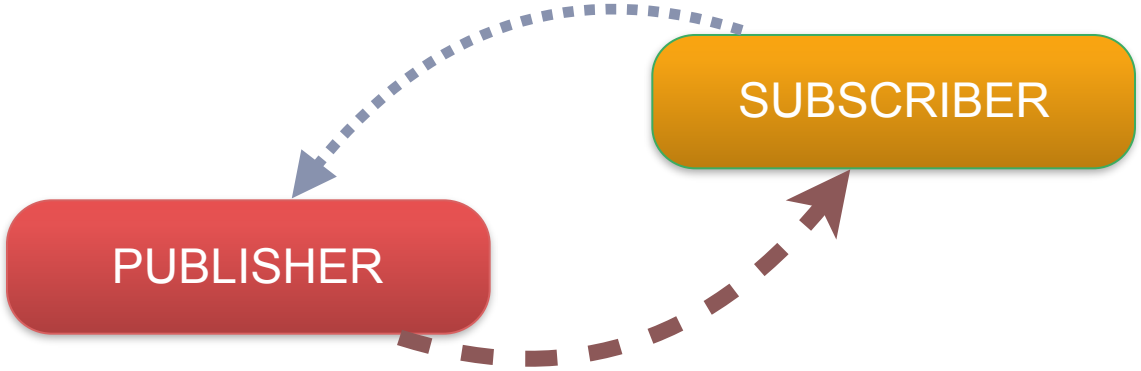
Reactive-Streams: Dynamic Message-Passing

Reactive-Streams: Dynamic Message-Passing

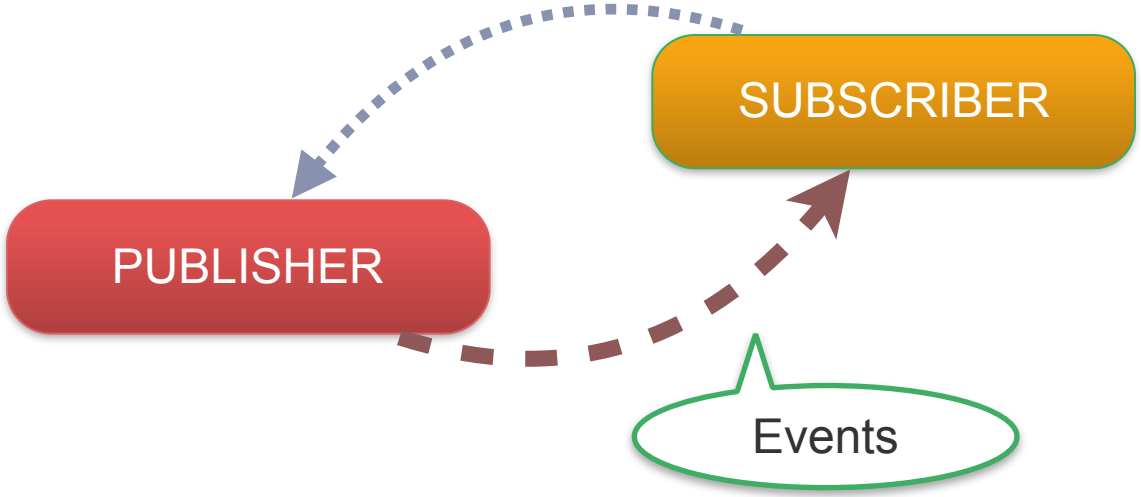
Reactive-Streams: Dynamic Message-Passing



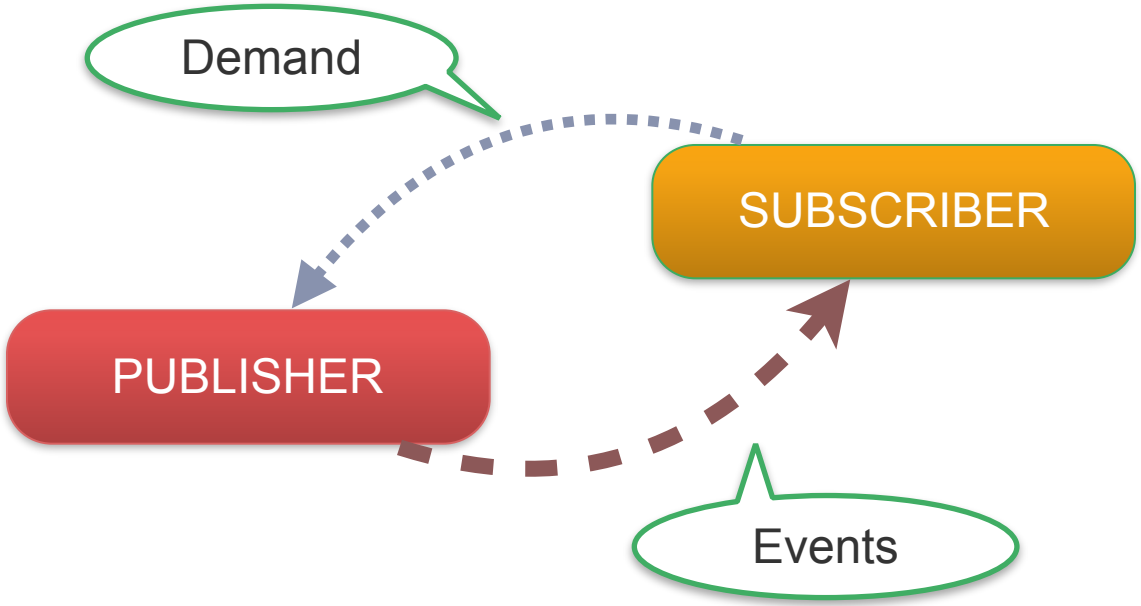
Reactive-Streams: Dynamic Message-Passing



Reactive-Streams: Dynamic Message-Passing



Reactive-Streams: Dynamic Message-Passing



Now You Know

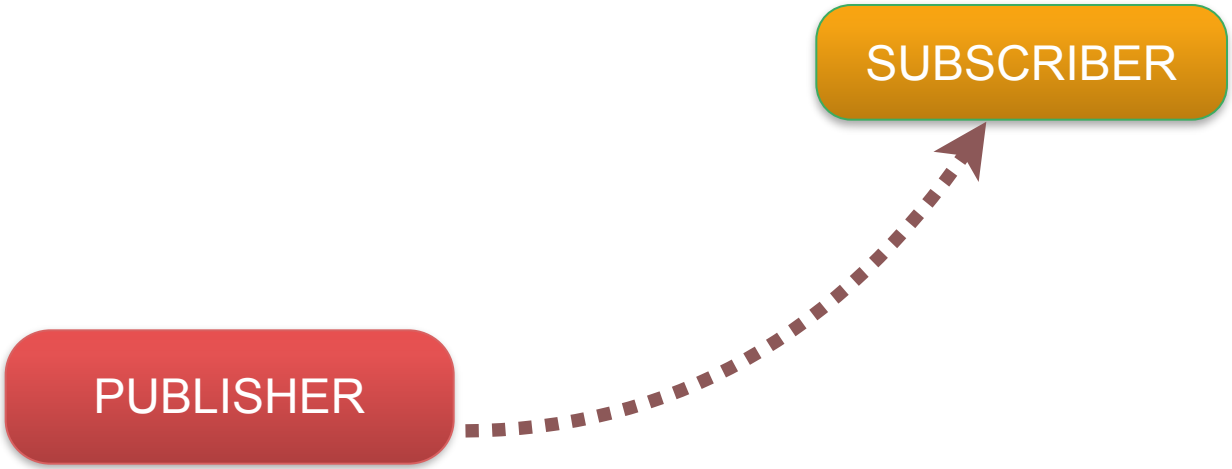
- It is not only queue-based pattern:
  - Signaling demand on a slower **Publisher** == no buffering
  - Signaling demand on a faster **Publisher** == buffering

- Data volume is bounded by a **Subscriber**
  - Scaling dynamically if required

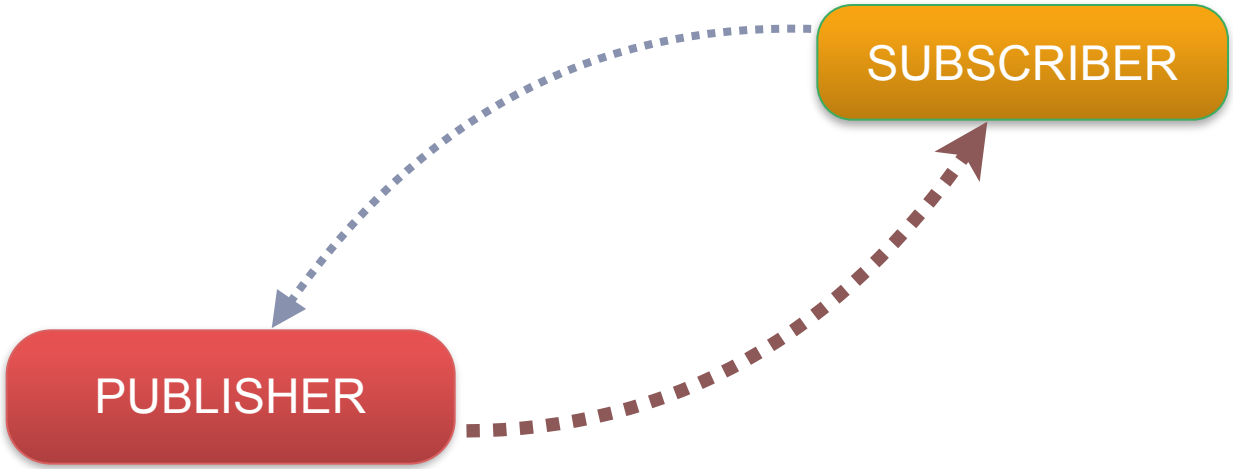
Out Of The Box : Flow Control

Out Of The Box : Flow Control

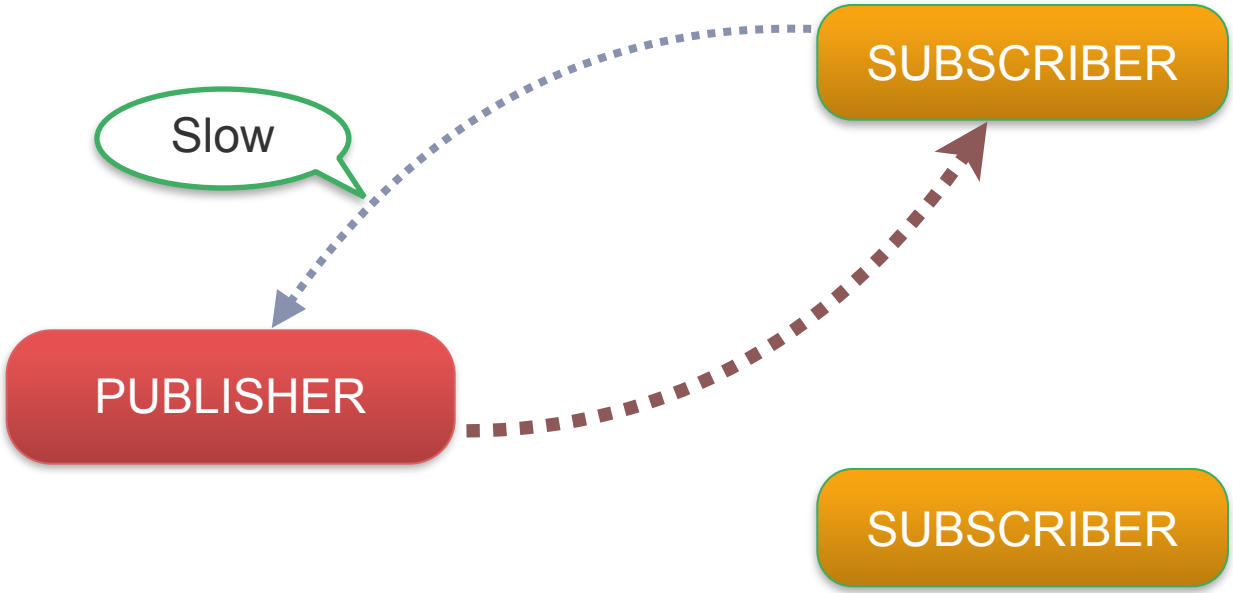
Out Of The Box : Flow Control



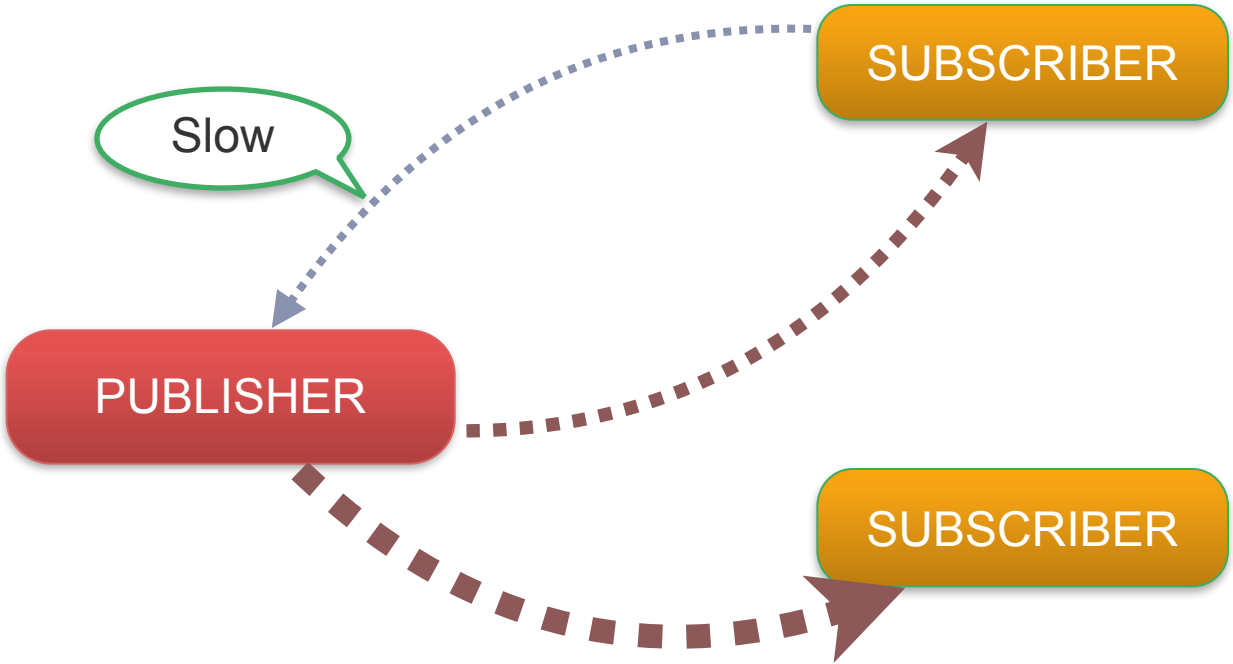
Out Of The Box : Flow Control



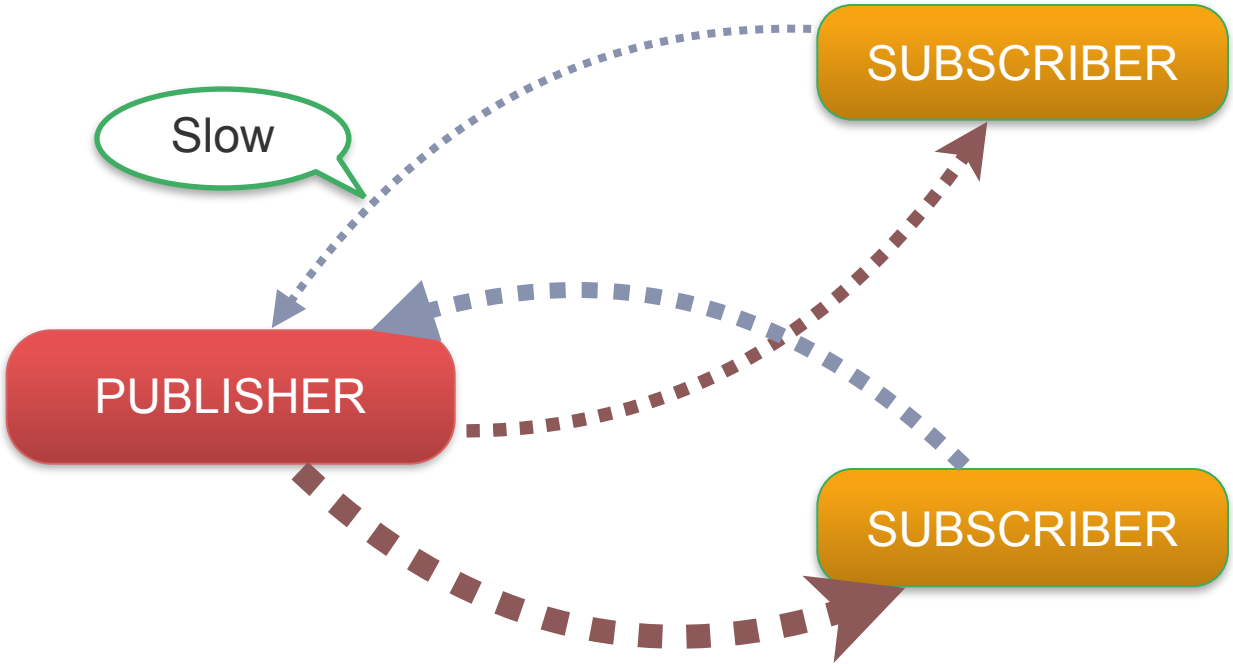
Out Of The Box : Flow Control



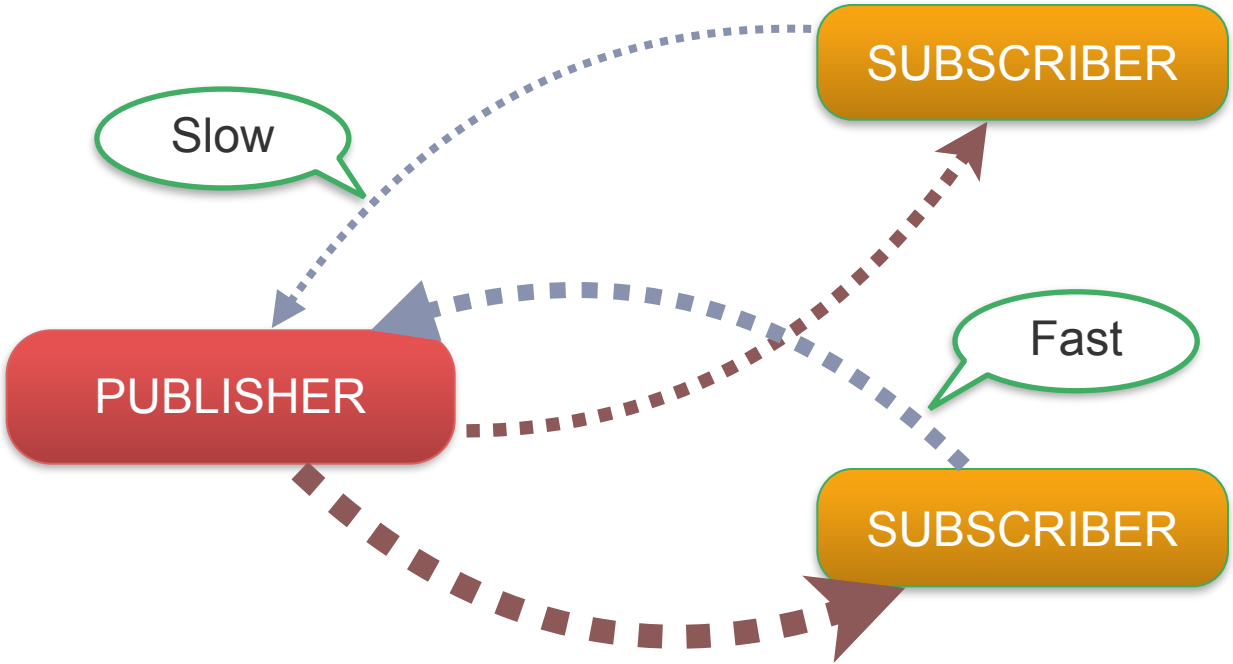
Out Of The Box : Flow Control



Out Of The Box : Flow Control



Out Of The Box : Flow Control



Reactive Streams: Signals

onError | (**onSubscribe** **onNext*** (**onError** | **onComplete**)?)

Reactive Streams: Joining forces

Doug Lea – SUNY Oswego

Reactive Streams: Joining forces

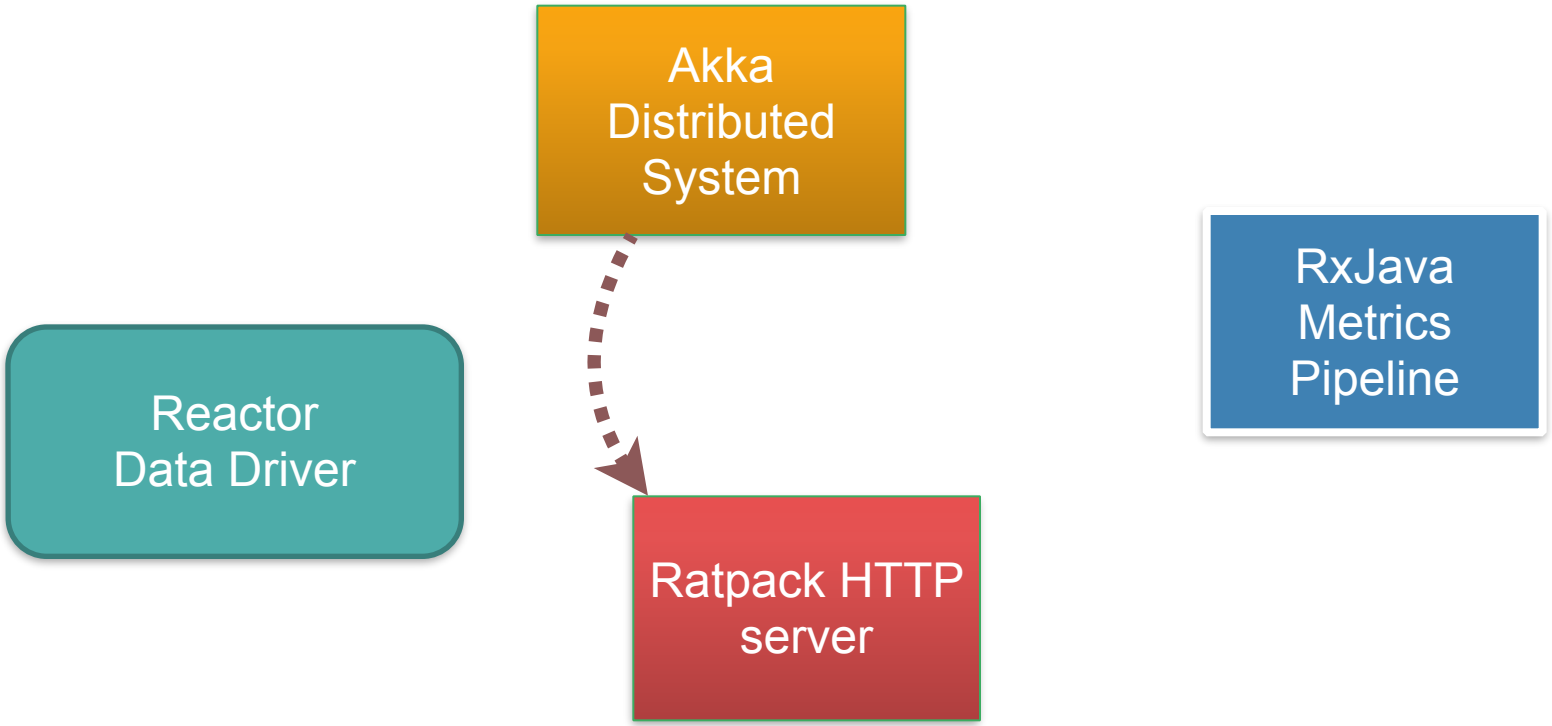
Akka
Distributed
System

RxJava
Metrics
Pipeline

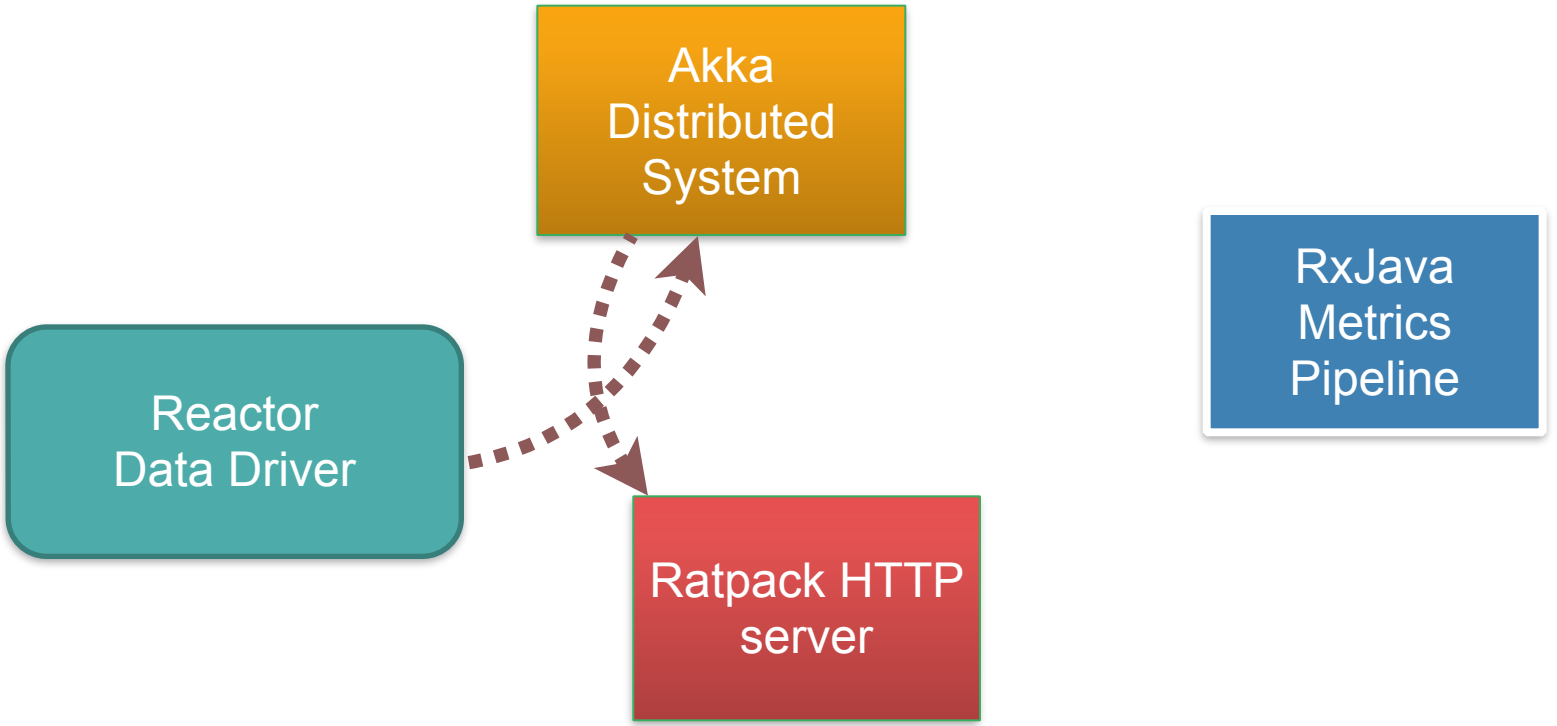
Reactor
Data Driver

Ratpack HTTP
server

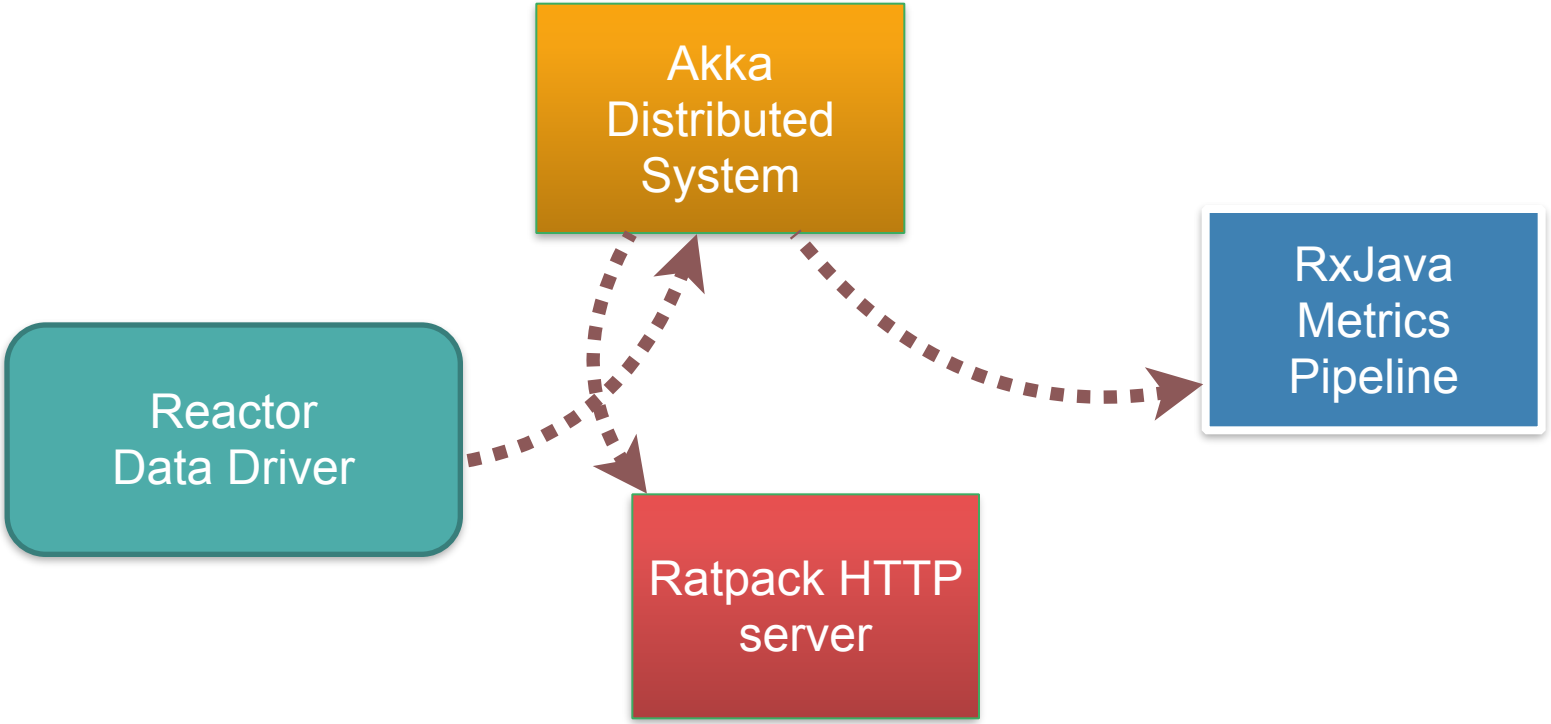
Reactive Streams: Joining forces



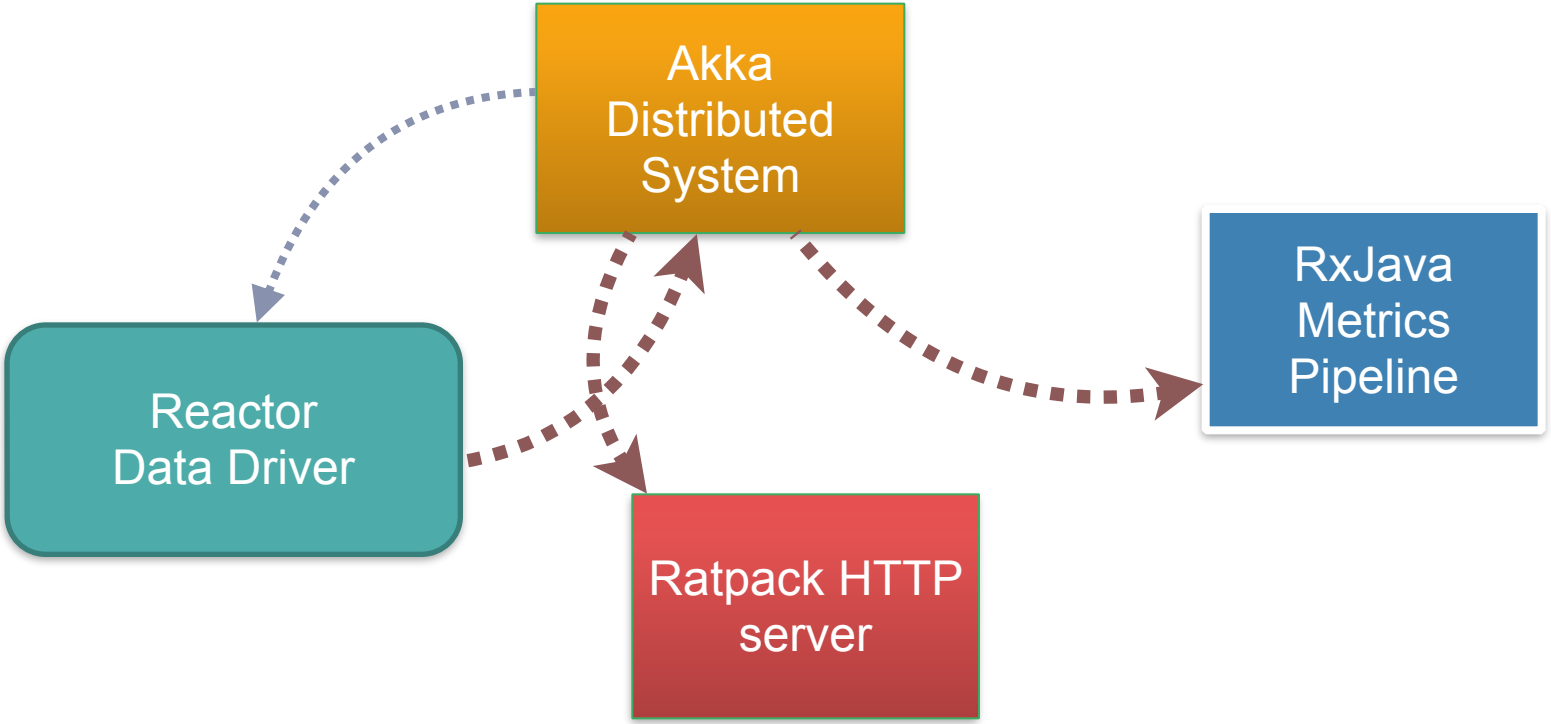
Reactive Streams: Joining forces



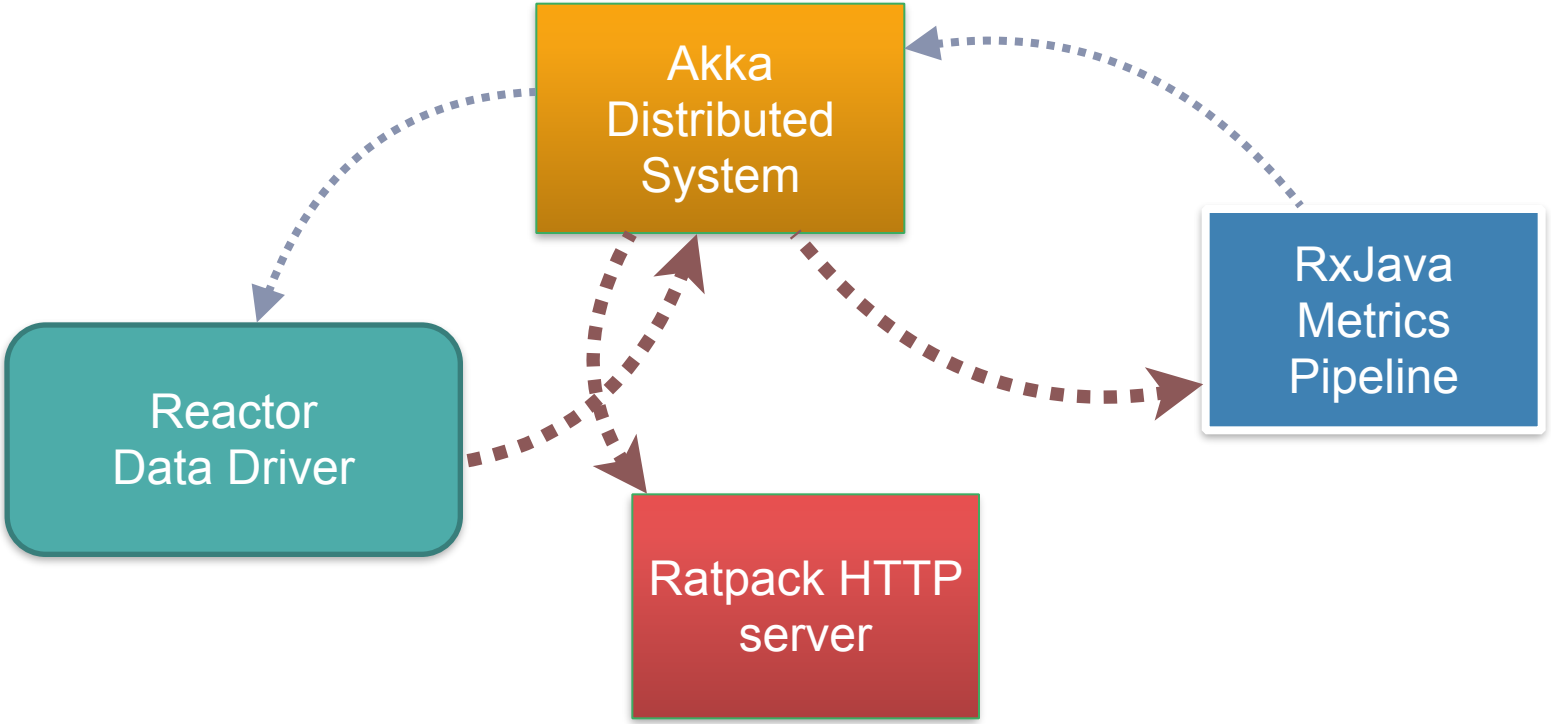
Reactive Streams: Joining forces



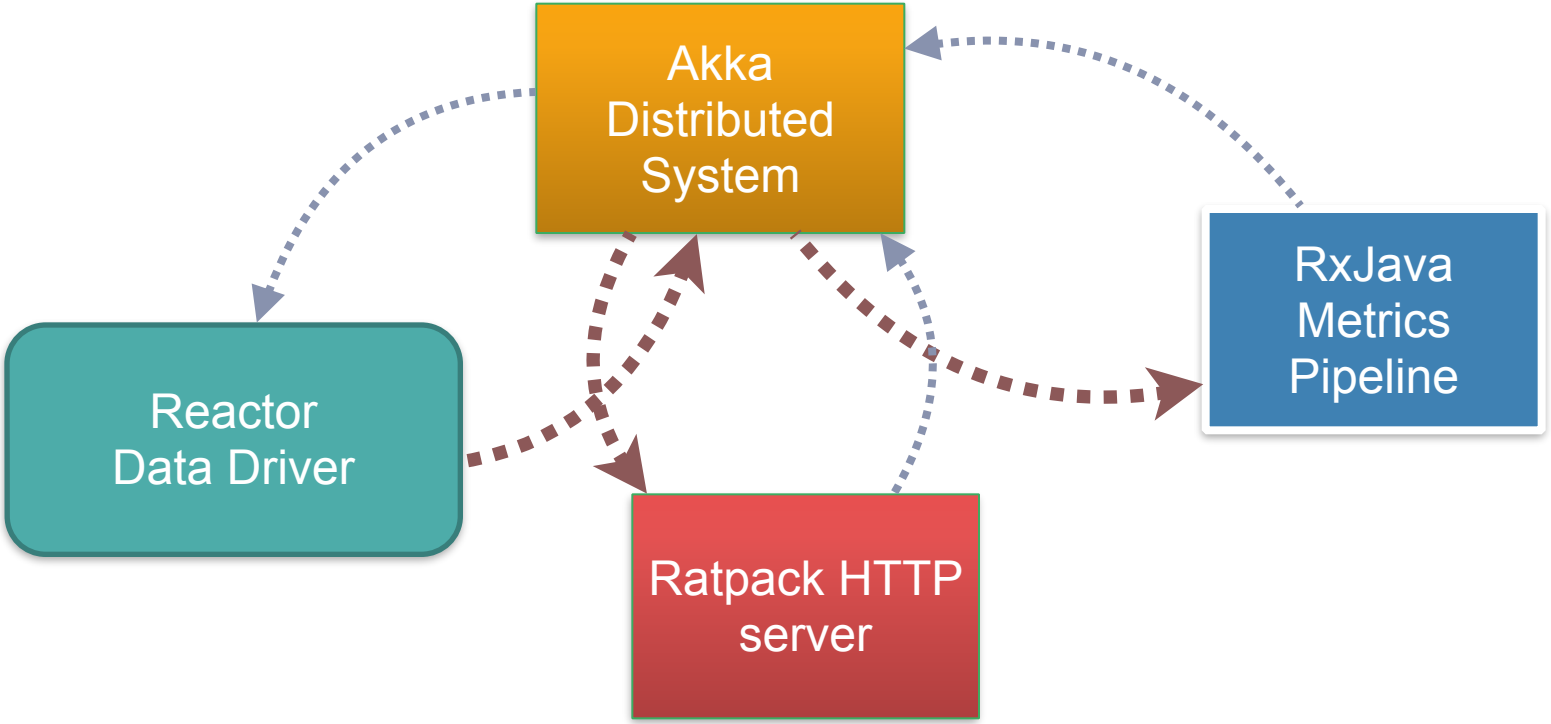
Reactive Streams: Joining forces



Reactive Streams: Joining forces



Reactive Streams: Joining forces



Reactive Streams: An industrial matured spec

- **Semantics**
  - **Single document** listing full rules
  - Open enough to allow for various patterns

- 4 **API** Interfaces
  - **Publisher**, **Subscriber**, **Subscription**, ***Processor***

- **TCK** to verify implementation behavior

Reactive Streams: org.reactivestreams

```
public interface Publisher<T> {
    public void subscribe(Subscriber<T> s);
}

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

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

Reactive Streams: org.reactivestreams

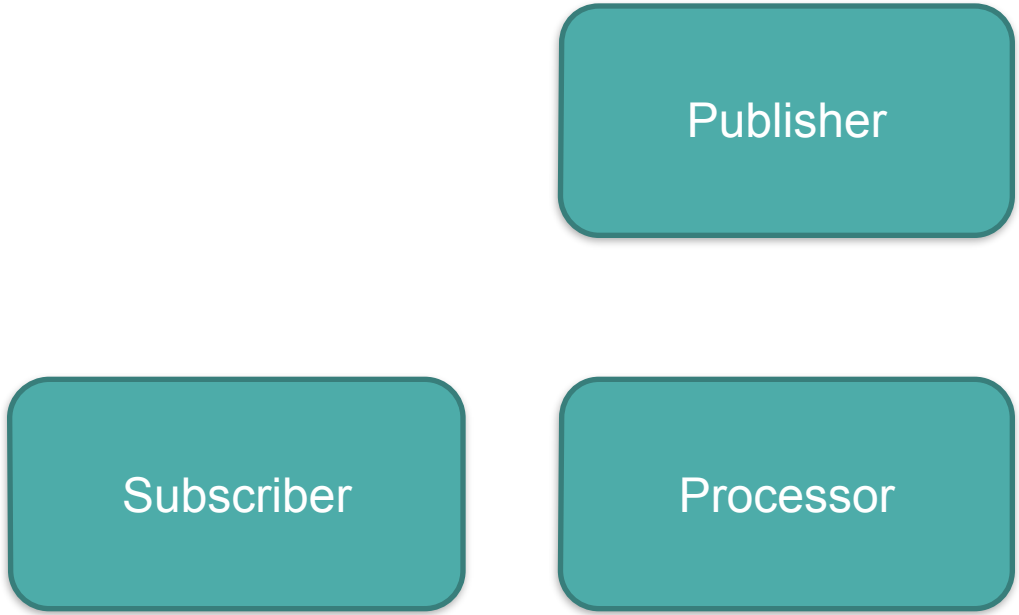
```
public interface Publisher<T> {
    public void subscribe(Subscriber<T> s);
}

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

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

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

Reactor <=> Reactive Streams

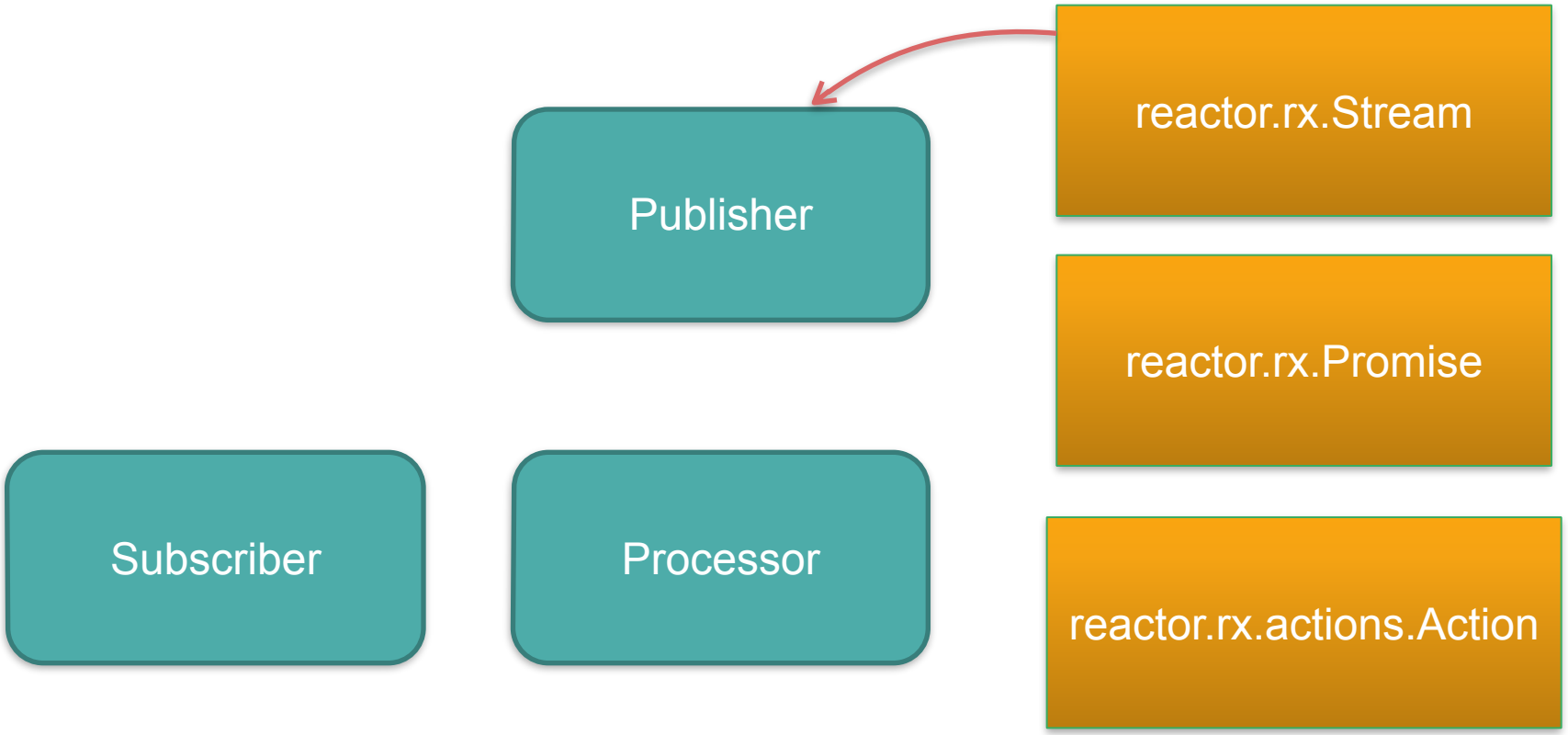


reactor.rx.Stream

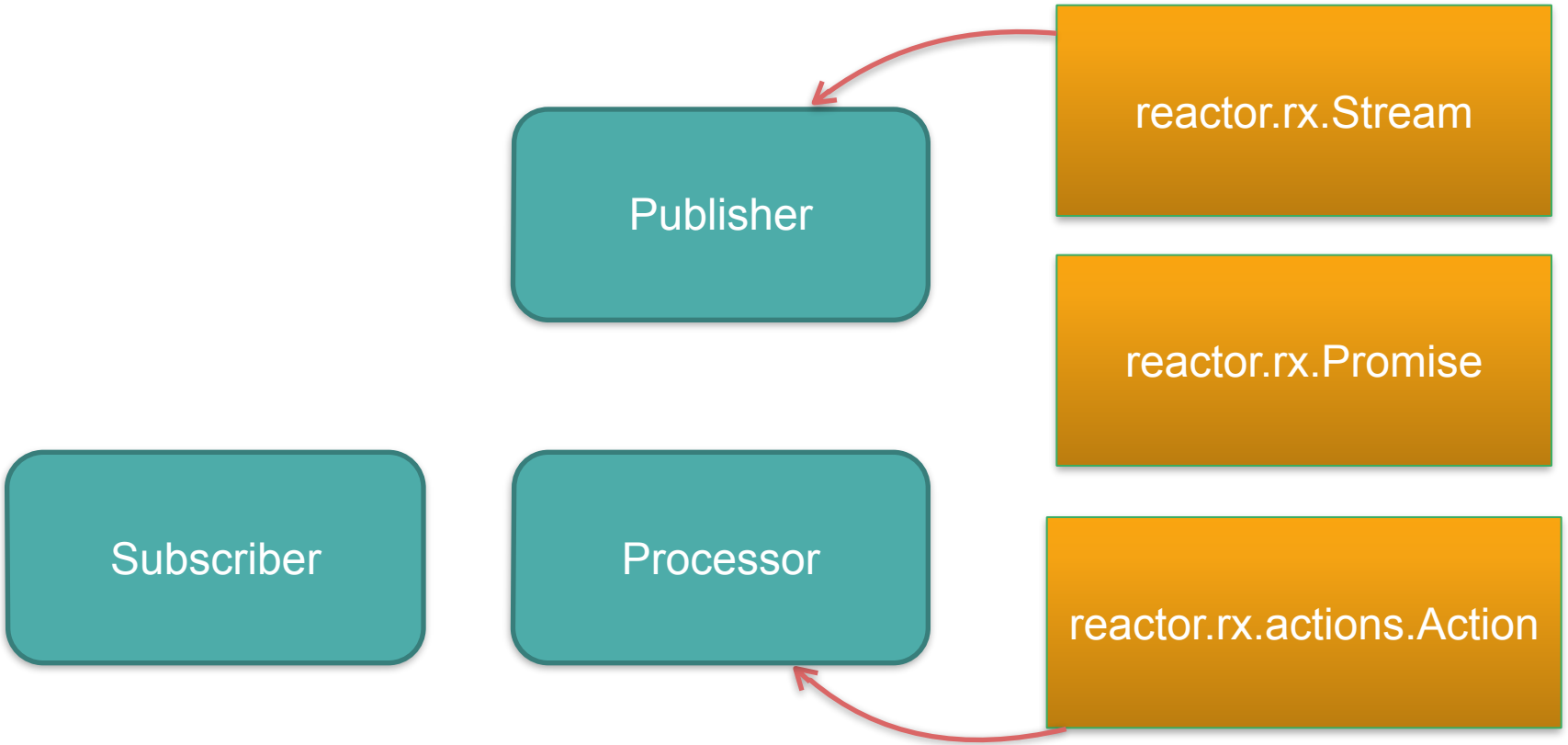
reactor.rx.Promise

reactor.rx.actions.Action

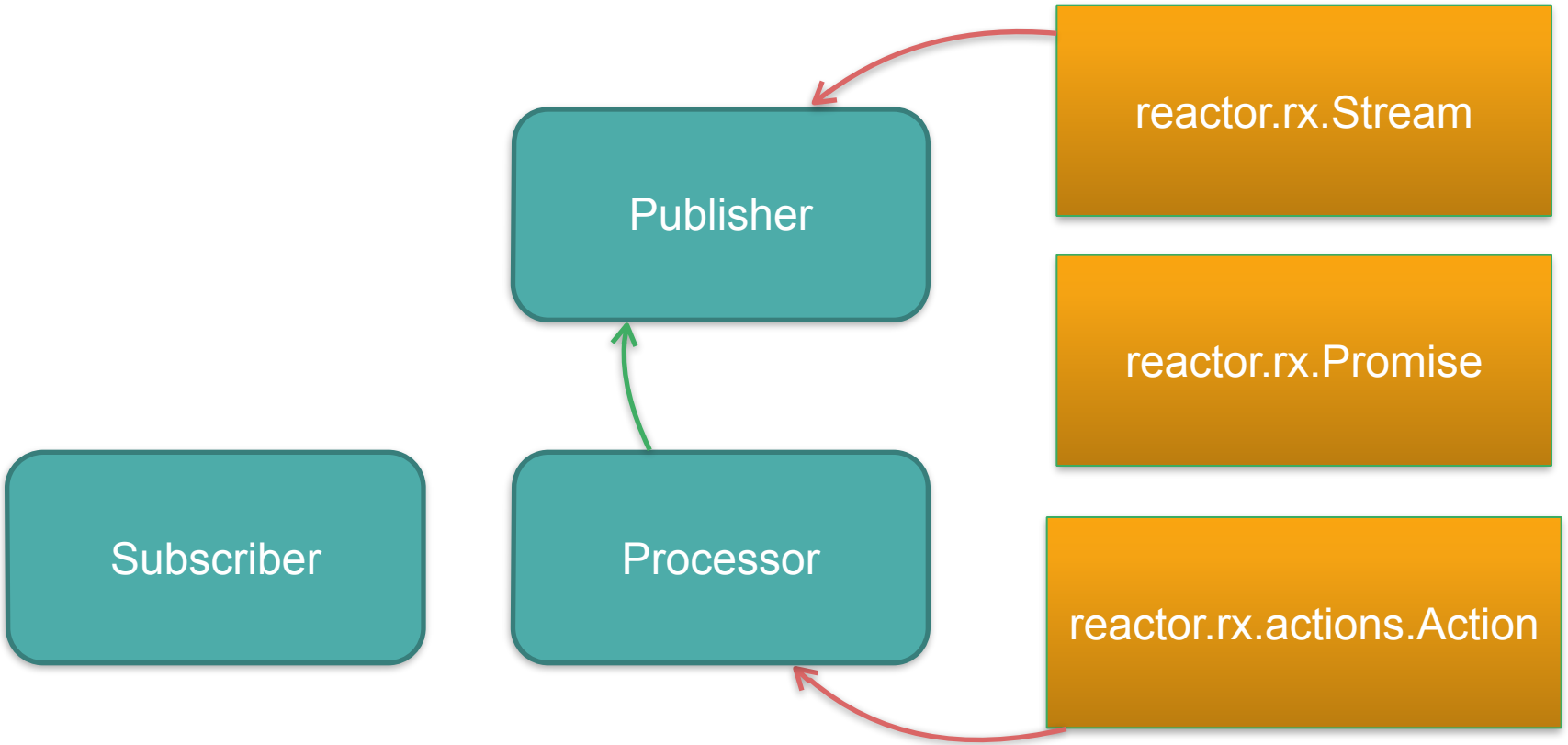
Reactor <=> Reactive Streams



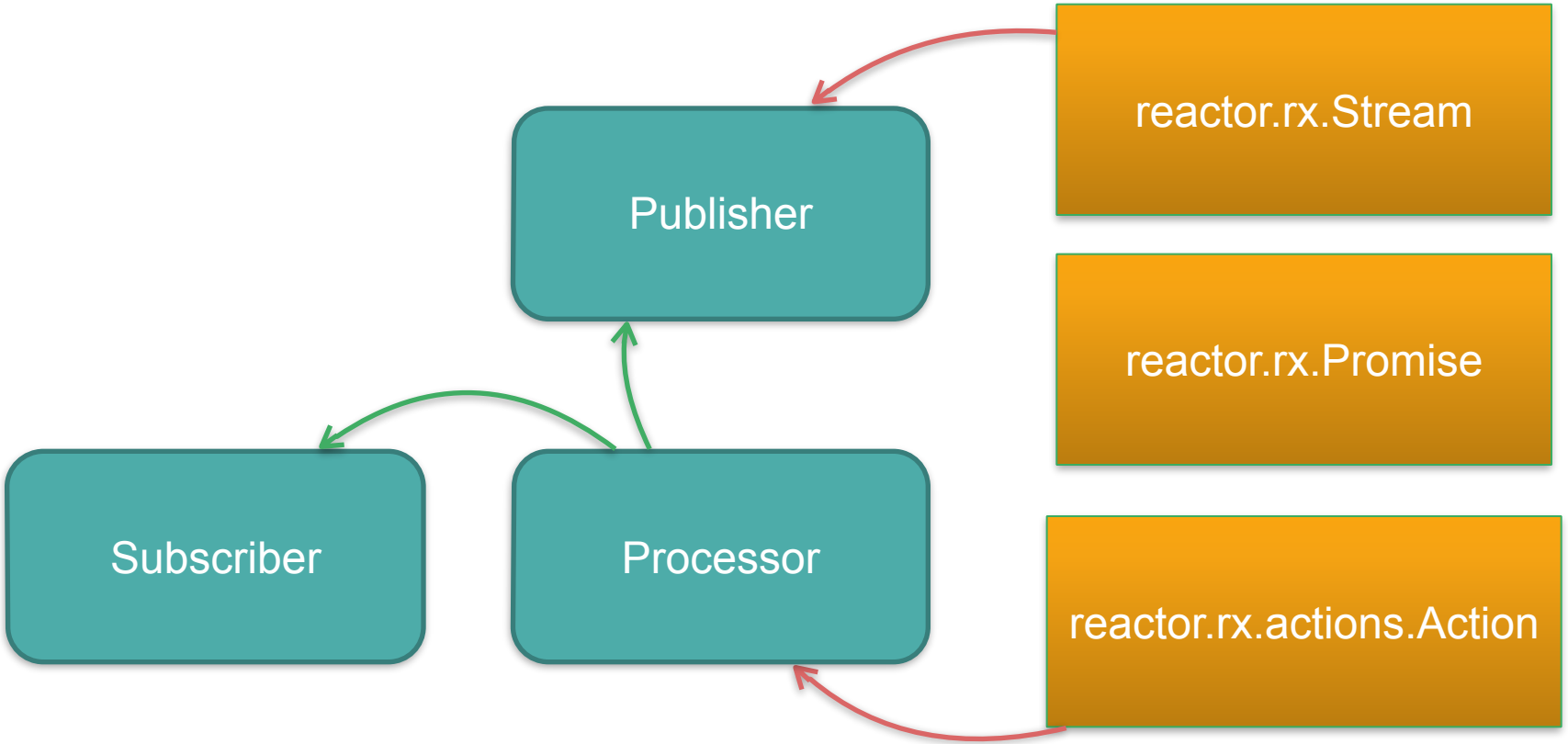
Reactor <=> Reactive Streams



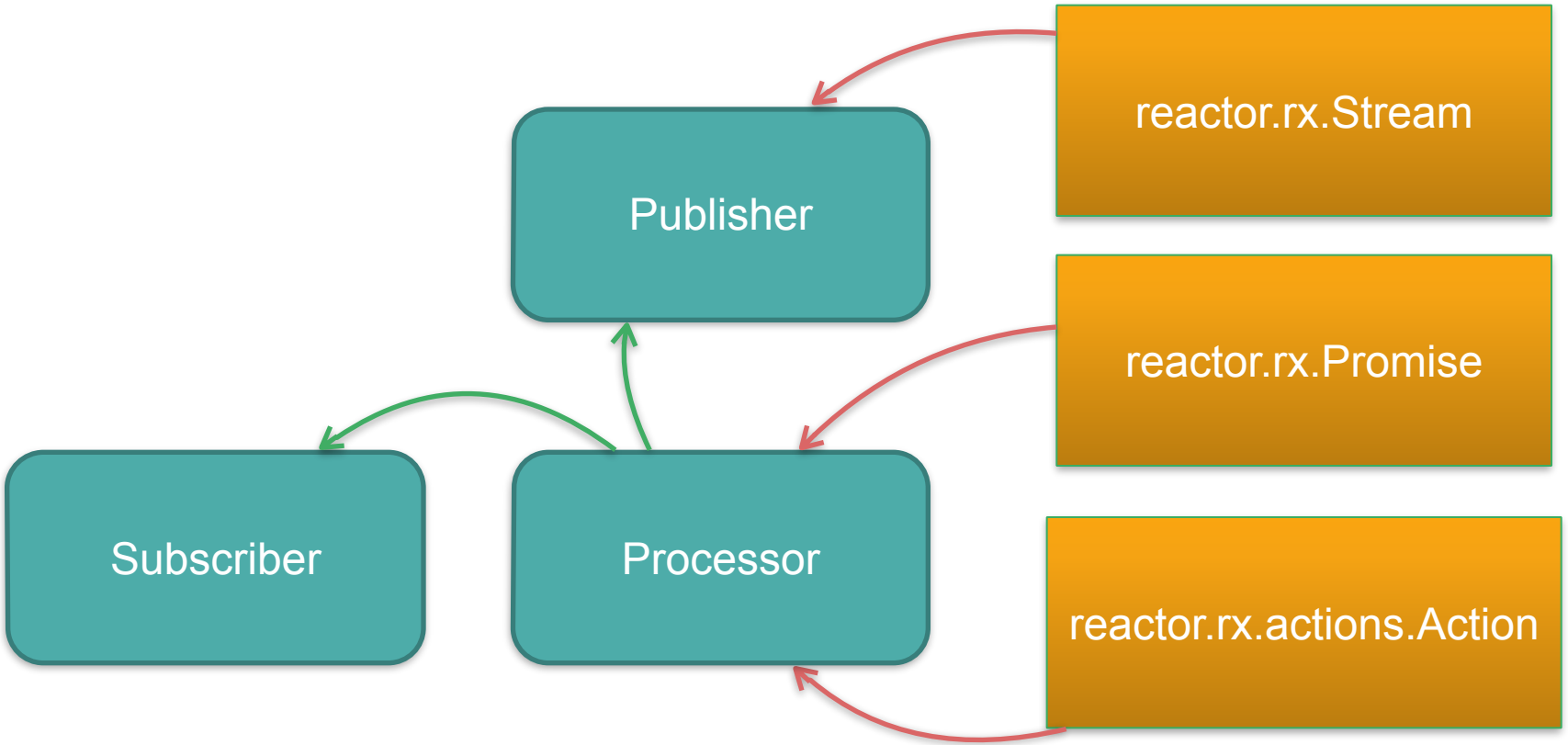
Reactor <=> Reactive Streams



Reactor <=> Reactive Streams



Reactor <=> Reactive Streams



What about **RxJava mate !** All those hipsters use it.

Reactor == RxJava

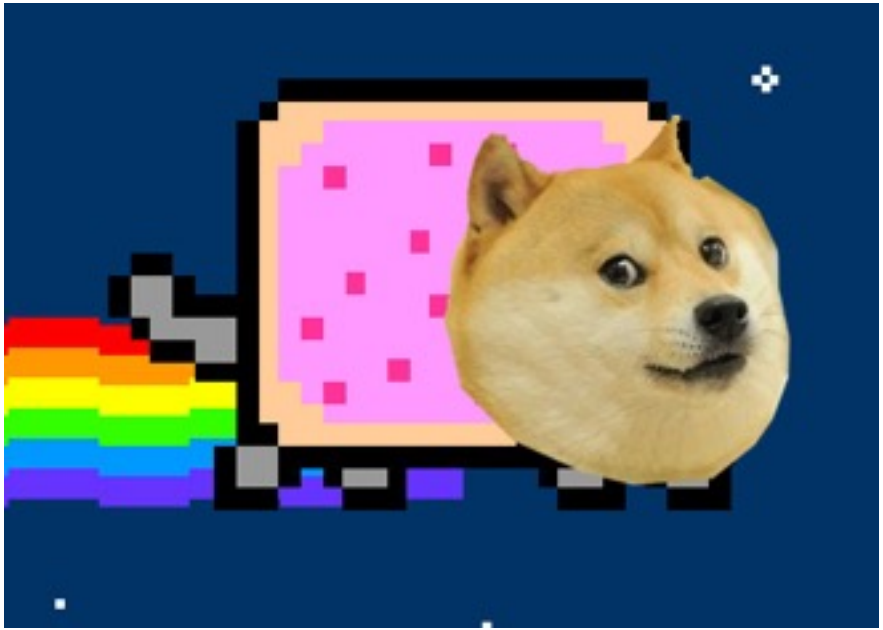
- Reactor Streams 2.0 are inspired by **Rx and RxJava**
  - http://msdn.microsoft.com/en-gb/data/gg577609.aspx
  - **Naming and behavior** is mostly aligned with **RxJava** (just, flatMap, retry…)
  - **Rx Patterns** should apply to Reactor Streams
  - **Lightweight**, embeddable

Reactor != RxJava

- Reactor Streams mission:
  - **Data throughput** over Functional facilities (Dispatchers, Subscription model, pre-allocation)
  - **Pivotal integration** (Spring.io, RabbitMQ, Redis, CloudFoundry, Gemfire…)
  - **Native Reactive Streams**, all Stream actions benefit from back pressure model and can talk to any implementation

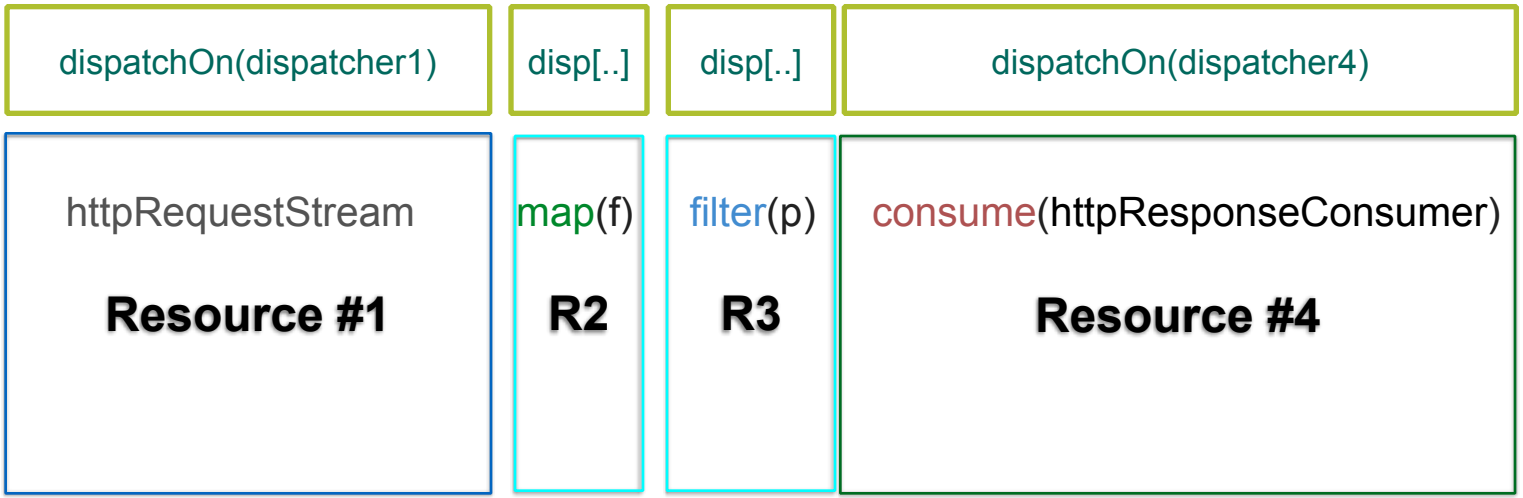
slidesSubscription.request(18);

//talkSubscription.cancel();

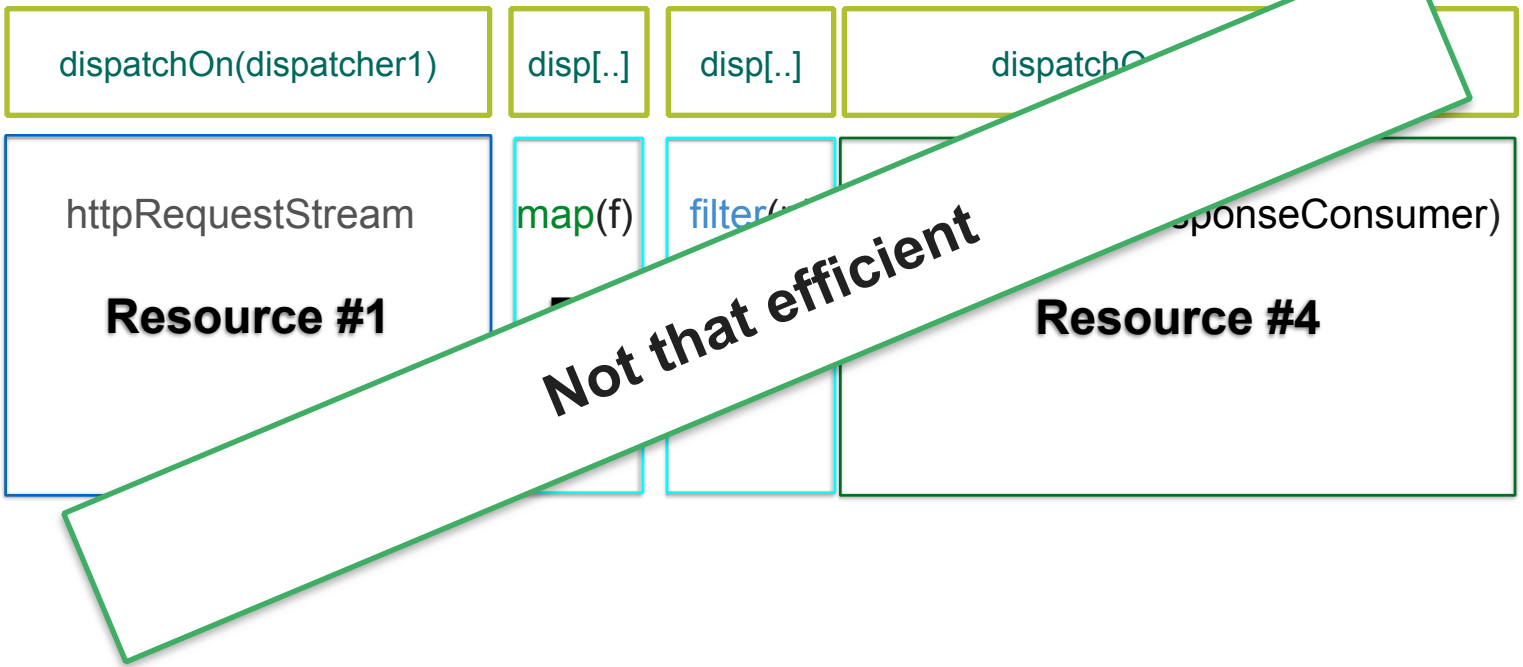


Let’s finish with practical patterns to empower your Reactive Service

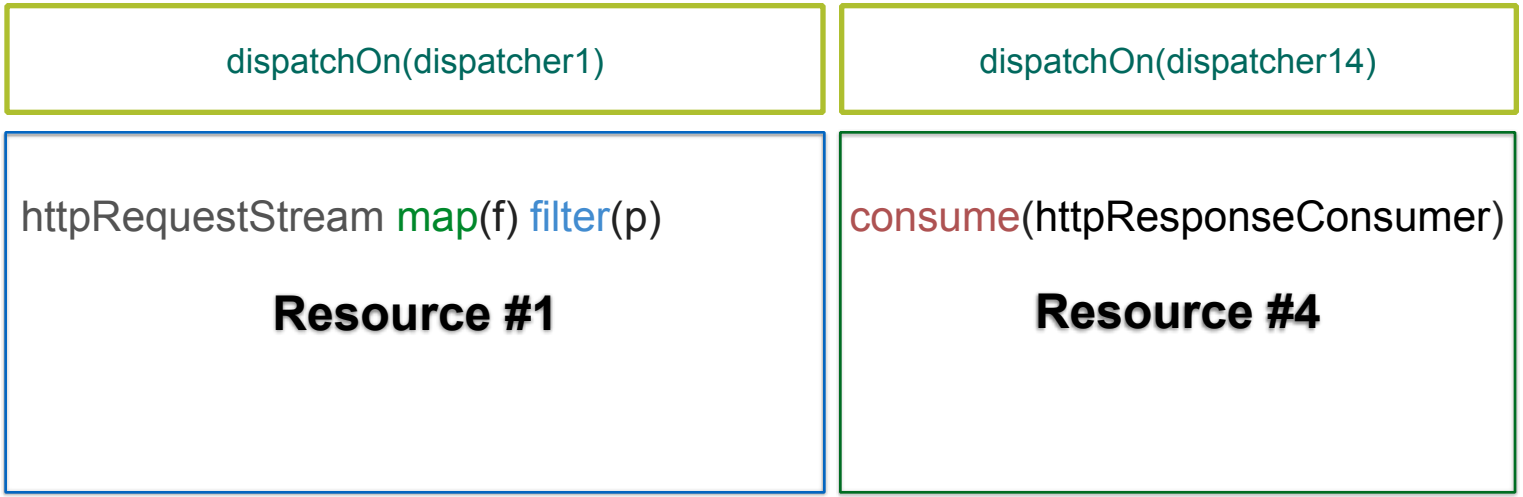
Reactive Streams: Async Boundaries



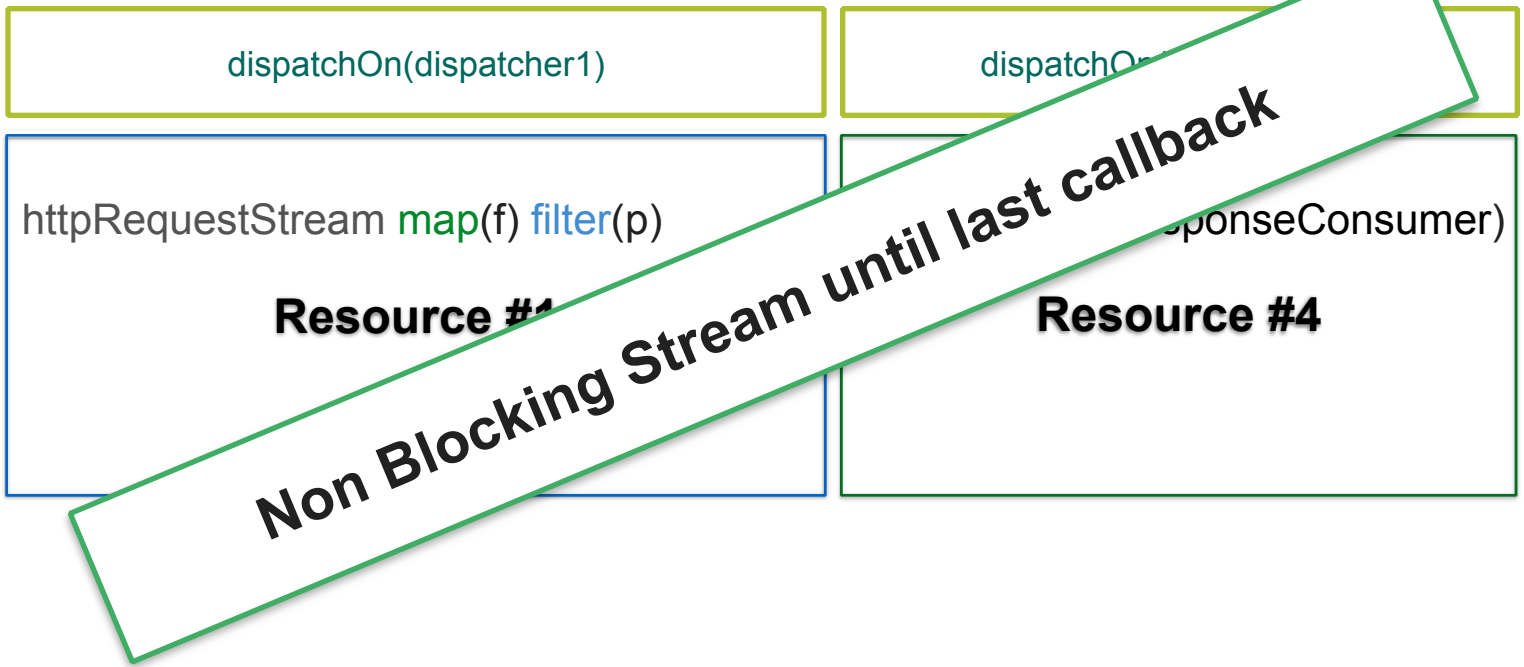
Reactive Streams: Async Boundaries



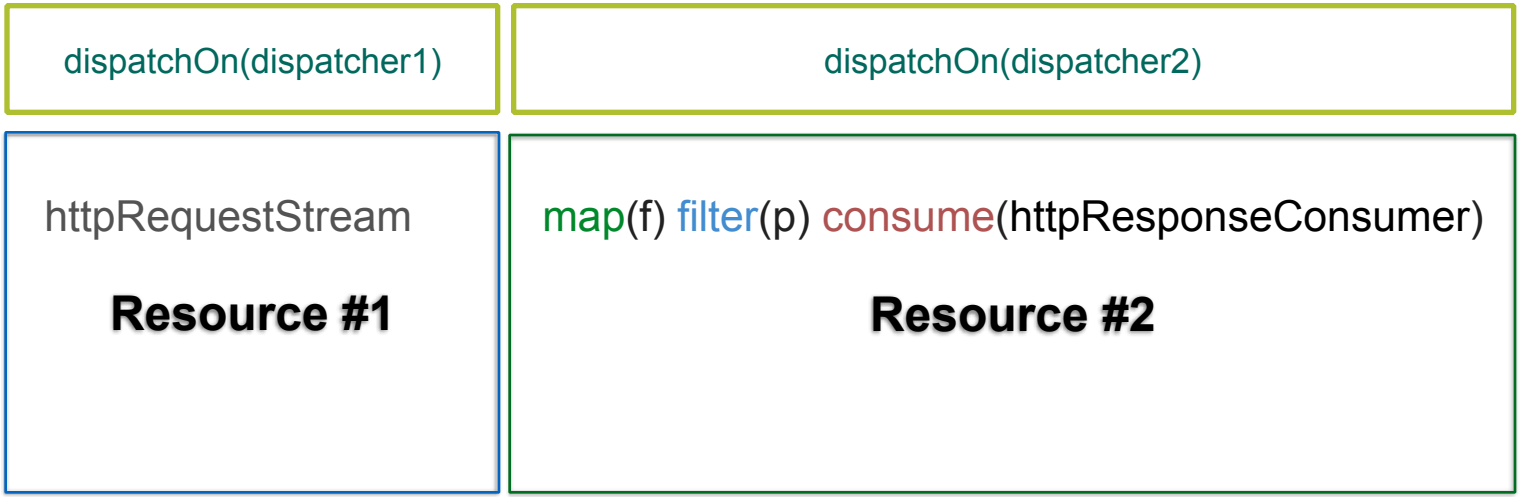
Reactive Streams: Async Boundaries



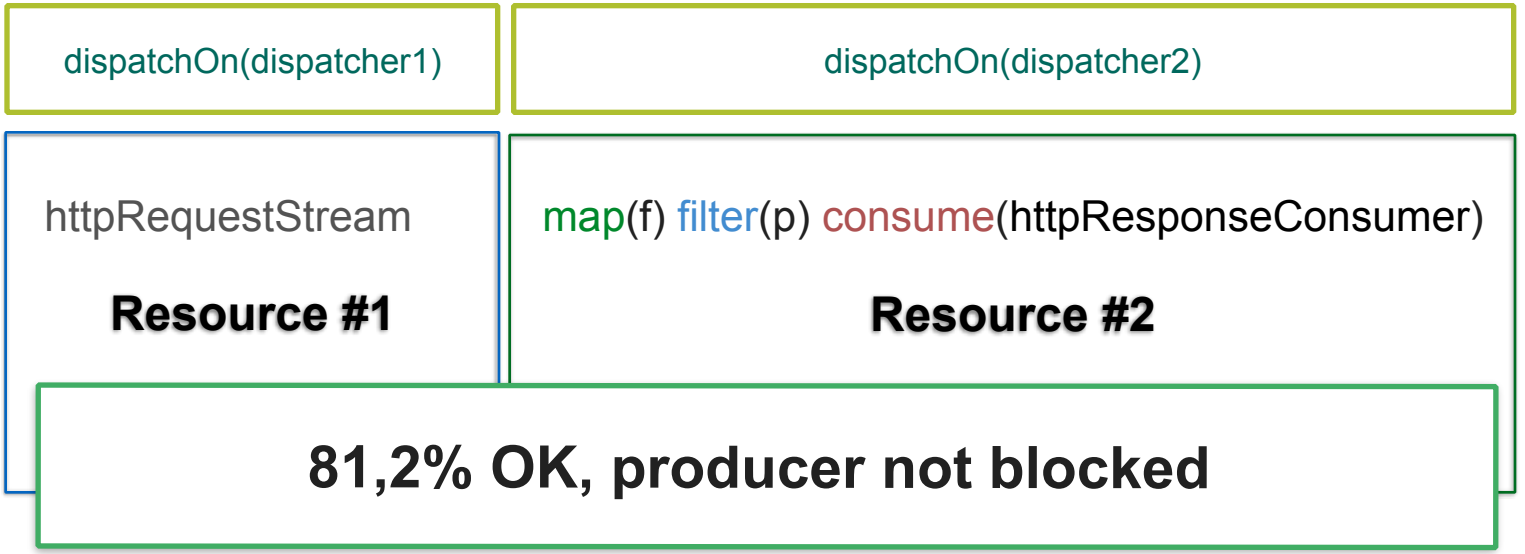
Reactive Streams: Async Boundaries



Reactive Streams: Async Boundaries



Reactive Streams: Async Boundaries



A full slide just to say something about FlatMap

A full slide just to say something about FlatMap

FlatMap Bucket Challenge ! Nominate 3 friends to explain *flatMap()*

Another one, FlatMap is that cool

Another one, FlatMap is that cool

flatMap() is nothing more than the functional alternative to RPC. Just a way to say “Ok bind this incoming data to this sub-flow and listen for the result, dude”.

Another one, FlatMap is that cool

Another one, FlatMap is that cool

Usually in Functional Reactive Programming, *flatMap* is often used for crossing an async boundary.

Another one, FlatMap is that cool

Another one, FlatMap is that cool

This applies to Reactor too, but ALL Reactor actions also have built-in backpressure support and dispatching facilities

Before you miss it, a FlatMap example

```
Streams.just('doge').flatMap{ name ->
    Streams.just(name)
           .observe{ println 'so wow' }
           .map{ 'much monad'}
}.consume{
    assert it == 'much monad'
}
```

Before you miss it, a FlatMap example

Feed a dynamic Sub-Stream with a name

```
Streams.just('doge').flatMap{ name ->
    Streams.just(name)
           .observe{ println 'so wow' }
           .map{ 'much monad'}
}.consume{
    assert it == 'much monad'
}
```

Before you miss it, a FlatMap example

Feed a dynamic Sub-Stream with a name

```
Streams.just('doge').flatMap{ name ->
    Streams.just(name)
           .observe{ println 'so wow' }
           .map{ 'much monad'}
}.consume{
    assert it == 'much monad'
}
```

Sub-Stream definition

Before you miss it, a FlatMap example

Feed a dynamic Sub-Stream with a name

```
Streams.just('doge').flatMap{ name ->
    Streams.just(name)
           .observe{ println 'so wow' }
           .map{ 'much monad'}
}.consume{
    assert it == 'much monad'
}
```

Sub-Stream definition

Sub-Stream result is **merged back to the top-level Steam**

Doing IO work (esp. O) : Cloning Pipeline

```
deferred = Streams.<String>defer(Environment.get());
deferred
    .parallel(8)
    .map(stream -> stream
        .map(i -> i)
        .reduce(2, service::reducePairAsMap)
        .consume(service::forwardToOutput)
).drain();
```

Doing IO work (esp. O) : Cloning Pipeline

Will create 8 clones competing for upstream data

```
deferred = Streams.<String>defer(Environment.get());
deferred
    .parallel(8)
    .map(stream -> stream
        .map(i -> i)
        .reduce(2, service::reducePairAsMap)
        .consume(service::forwardToOutput)
).drain();
```

Doing IO work (esp. O) : Cloning Pipeline

Will create 8 clones competing for upstream data

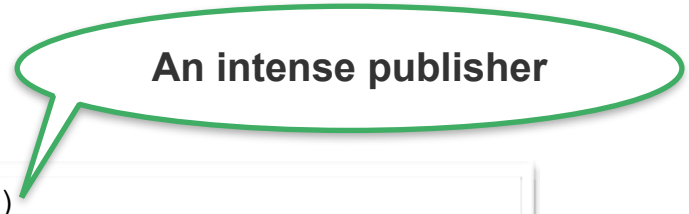
```
deferred = Streams.<String>defer(Environment.get());
deferred
    .parallel(8)
    .map(stream -> stream
        .map(i -> i)
        .reduce(2, service::reducePairAsMap)
        .consume(service::forwardToOutput)
).drain();
```

Start consuming the full Stream until complete

Dealing with a nervous Publisher

```
Streams.range(1, Integer.MAX_VALUE)
        .dispatchOn(Environment.masterDispatcher())
        .sample(2l, TimeUnit.SECONDS)
        .dispatchOn(Environment.cachedDispatcher())
        .consume{
            println it
        }
```

Dealing with a nervous Publisher




```
Streams.range(1, Integer.MAX_VALUE)
        .dispatchOn(Environment.masterDispatcher())
        .sample(2l, TimeUnit.SECONDS)
        .dispatchOn(Environment.cachedDispatcher())
        .consume{
            println it
        }
```

Dealing with a nervous Publisher

Assigned with a global dispatcher

An intense publisher

```
Streams.range(1, Integer.MAX_VALUE)
        .dispatchOn(Environment.masterDispatcher())
        .sample(2l, TimeUnit.SECONDS)
        .dispatchOn(Environment.cachedDispatcher())
        .consume{
            println it
        }
```

Dealing with a nervous Publisher

Assigned with a global dispatcher

An intense publisher

```
Streams.range(1, Integer.MAX_VALUE)
        .dispatchOn(Environment.masterDispatcher())
        .sample(2l, TimeUnit.SECONDS)
        .dispatchOn(Environment.cachedDispatcher())
        .consume{
            println it
        }
```

Retaining a single value every 2s

Dealing with a nervous Publisher

Assigned with a global dispatcher

An intense publisher

```
Streams.range(1, Integer.MAX_VALUE)
        .dispatchOn(Environment.masterDispatcher())
        .sample(2l, TimeUnit.SECONDS)
        .dispatchOn(Environment.cachedDispatcher())
        .consume{
            println it
        }
```

Retaining a single value every 2s

Dispatching the samples
with a different dispatcher to avoid publisher contention

Failproof: Implementing a Circuit Breaker

```
def closeCircuit = stream
def openCircuit = fallback ?: Streams.<T>fail(new Exception("service unavailable"))
def circuitSwitcher = Streams.switchOnNext()

stream
      .materialize()
      .window(maxSignals, maxTime)
      .flatMap{ s ->
            s.reduce(["failures":0, "success":0]){ tuple ->
                  if(tuple.t2.isOnNext()) tuple.t1.success++
                  else if(tuple.t2.isOnError()) tuple.t1.failures++
            }
      }
      .consume({ streamHealth ->
            if(streamHealth.failures/(streamHealth.failures + streamHealth.success) > threshold) {
                  Streams.timer(closeTimeout).consume{
                        circuitSwitcher.onNext(closeCircuit)
                  }
                  circuitSwitcher.onNext(openCircuit)
            } else {
                  circuitSwitcher.onNext(closeCircuit)
            }
      }, circuitSwitcher.&onError, circuitSwitcher.&onComplete)

//start with close circuit
circuitSwitcher.onNext(closeCircuit)
circuitSwitcher
```

Failproof: Implementing a Circuit Breaker

```
def closeCircuit = stream
def openCircuit = fallback ?: Streams.<T>fail(new Exception("service unavailable"))
def circuitSwitcher = Streams.switchOnNext()

stream
      .materialize()
      .window(maxSignals, maxTime)
      .flatMap{ s ->
            s.reduce(["failures":0, "success":0]){ tuple ->
                  if(tuple.t2.isOnNext()) tuple.t1.success++
                  else if(tuple.t2.isOnError()) tuple.t1.failures++
            }
      }
      .consume({ streamHealth ->
            if(streamHealth.failures/(streamHealth.failures + streamHealth.success) > threshold) {
                  Streams.timer(closeTimeout).consume{
                        circuitSwitcher.onNext(closeCircuit)
                  }
                  circuitSwitcher.                    ircuit)
            } else {
                  circuitS                            rcuit)
            }
      }, circuitSw                                witcher.&onComplete)

//start with close
circuitSwitcher.onNe                        rcuit)
circuitSwitcher
```

TL;DR

Failproof: Implementing a Circuit Breaker

```
def closeCircuit = stream
def openCircuit = fallback ?: Streams.<T>fail(new Exception("service unavailable"))
def circuitSwitcher = Streams.switchOnNext()

stream
      .materialize()
      .window(maxSignals, maxTime)
      .flatMap{ s ->
            s.reduce(["failures":0, "success":0]){ tuple ->
                  if(tuple.t2.isOnNext()) tuple.t1.success++
                  else if(tuple.t2.isOnError()) tuple.t1.failures++
            }
      }
      .consume({ streamHealth ->
            if(streamHealth.failures/(streamHealth.failures + streamHealth.success) > threshold) {
                  Streams.timer(closeTimeout).consume{
                        circuitSwitcher.onNext(closeCircuit)
                  }
                  circuitSwitcher.
            } else {
                  circuitS
            }
      }, circuitSw                                    witc

//start with close
circuitSwitcher.onNe                                  rcuit)
circuitSwitcher
```

TL;DR

materialize

Failproof: Implementing a Circuit Breaker

```
def closeCircuit = stream
def openCircuit = fallback ?: Streams.<T>fail(new Exception("service unavailable"))
def circuitSwitcher = Streams.switchOnNext()

stream
      .materialize()
      .window(maxSignals, maxTime)
      .flatMap{ s ->
            s.reduce(["failures":0, "success":0]){ tuple ->
                  if(tuple.t2.isOnNext()) tuple.t1.success++
                  else if(tuple.t2.isOnError()) tuple.t1.failures++
            }
      }
      .consume({ streamHealth ->
            if(streamHealth.failures/(streamHealth.failures + streamHealth.success) > threshold) {
                  Streams.timer(closeTimeout).consume{
                        circuitSwitcher.onNext(closeCircuit)
                  }
                  circuitSwitcher.
            } else {
                  circuitS
            }
      }, circuitSw                                witch

//start with close
circuitSwitcher.onNe                          rcuit)
circuitSwitcher
```

TL;DR

materialize

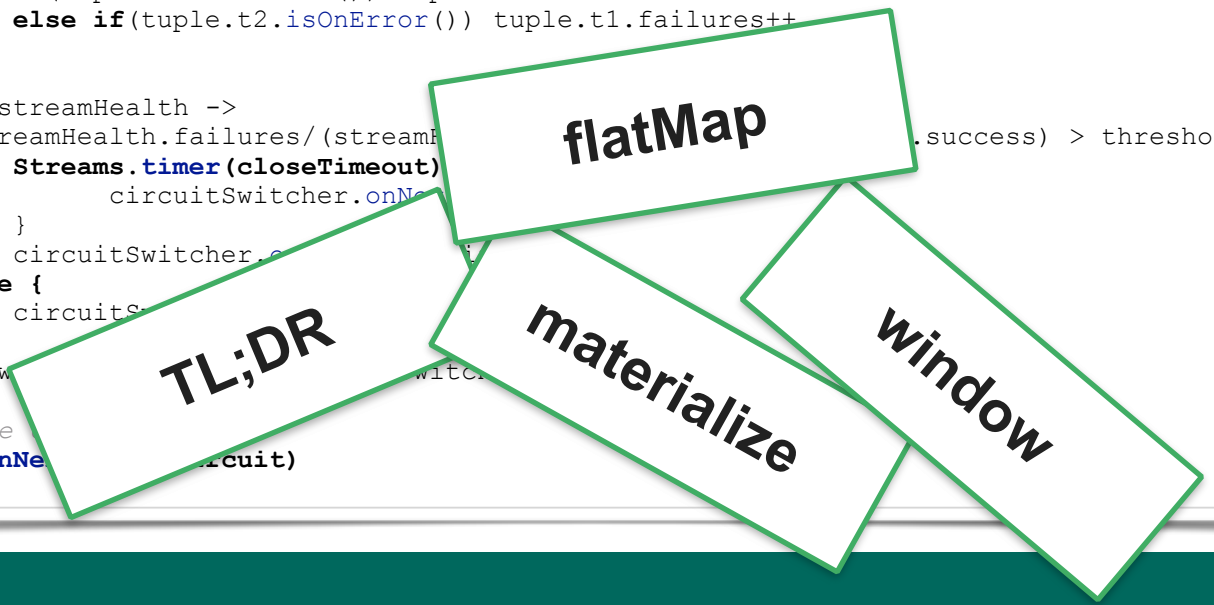
window

Failproof: Implementing a Circuit Breaker

```
def closeCircuit = stream
def openCircuit = fallback ?: Streams.<T>fail(new Exception("service unavailable"))
def circuitSwitcher = Streams.switchOnNext()

stream
      .materialize()
      .window(maxSignals, maxTime)
      .flatMap{ s ->
            s.reduce(["failures":0, "success":0]){ tuple ->
                  if(tuple.t2.isOnNext()) tuple.t1.success++
                  else if(tuple.t2.isOnError()) tuple.t1.failures++
            }
      }
      .consume({ streamHealth ->
            if(streamHealth.failures/(streamH                    .success) > threshold) {
                  Streams.timer(closeTimeout)
                        circuitSwitcher.onN
                  }
                  circuitSwitcher.
            } else {
                  circuitS
            }
      }, circuitSw                    witc

//start with close
circuitSwitcher.onNe                    rcuit)
circuitSwitcher
```

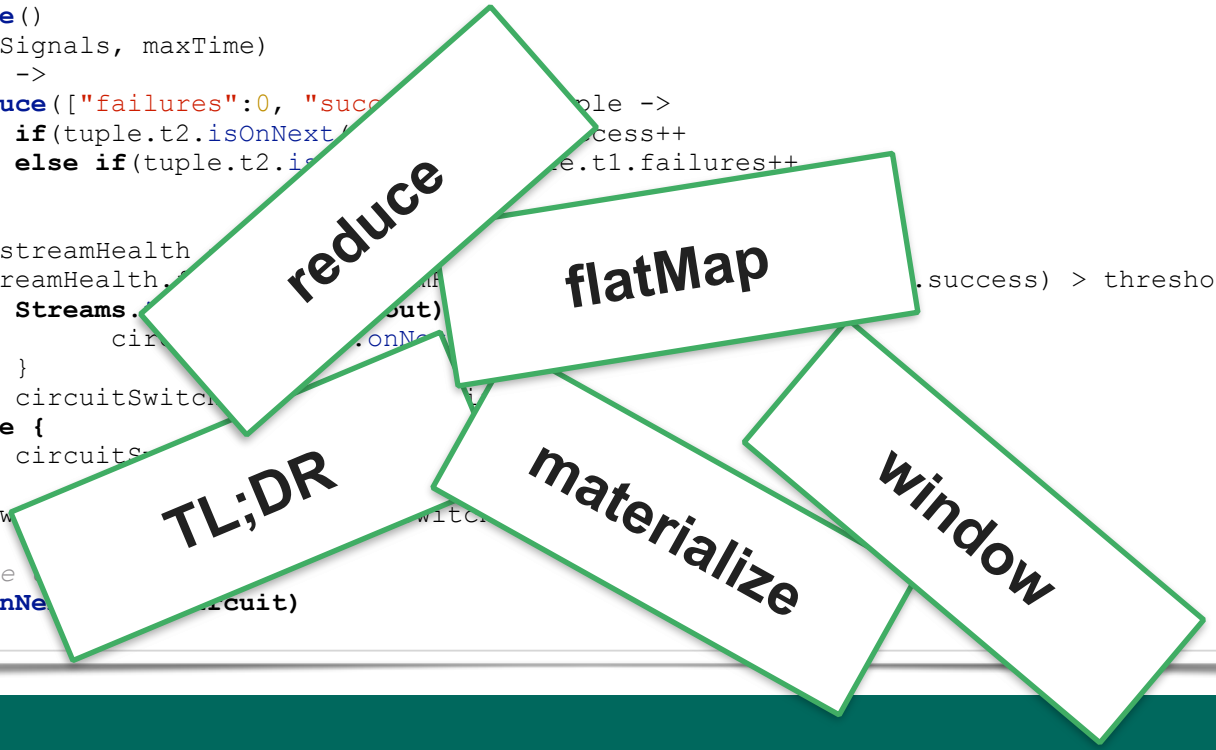


Failproof: Implementing a Circuit Breaker

```
def closeCircuit = stream
def openCircuit = fallback ?: Streams.<T>fail(new Exception("service unavailable"))
def circuitSwitcher = Streams.switchOnNext()

stream
      .materialize()
      .window(maxSignals, maxTime)
      .flatMap{ s ->
            s.reduce(["failures":0, "succ                    ple ->
                  if(tuple.t2.isOnNext(                    cess++
                  else if(tuple.t2.is                      e.t1.failures++
            }
      }
      .consume({ streamHealth
            if(streamHealth.                                 .success) > threshold) {
                  Streams.                                   but)
                        cir                                  .onN
                  }
                  circuitSwitc
            } else {
                  circuitS
            }
      }, circuitSw                                           witc

//start with close
circuitSwitcher.onNe                                         rcuit)
circuitSwitcher
```

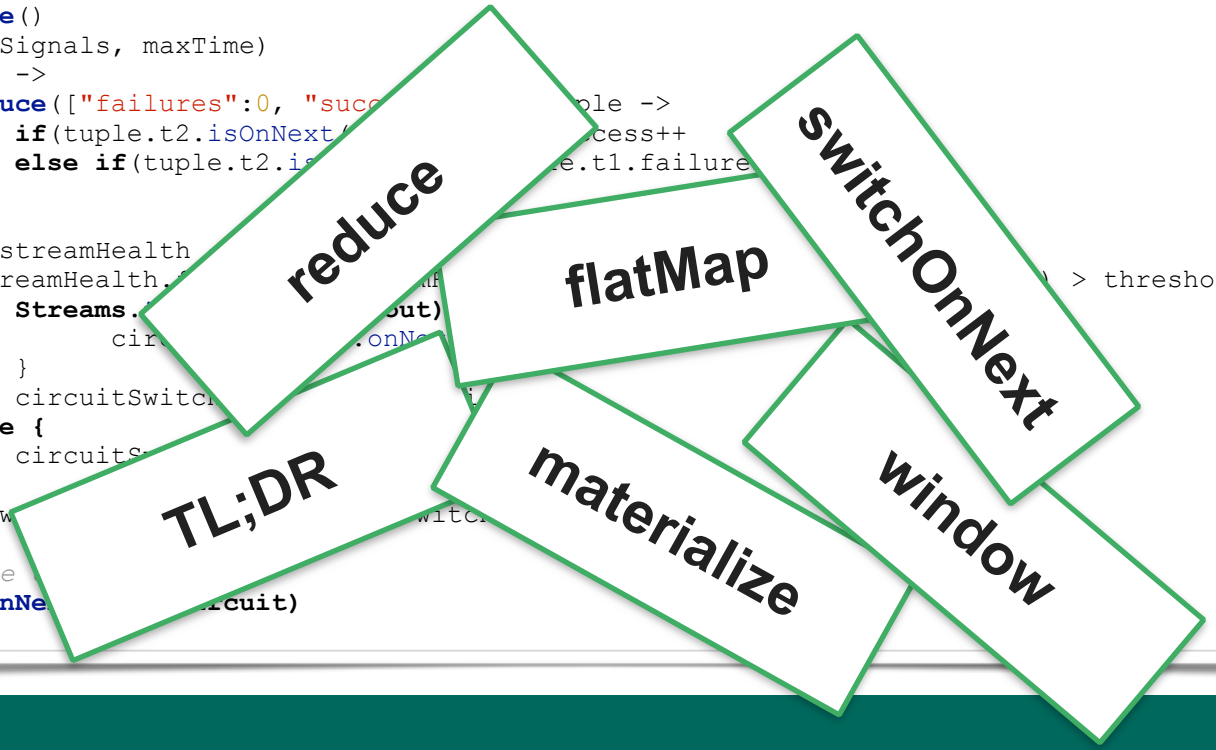


Failproof: Implementing a Circuit Breaker

```
def closeCircuit = stream
def openCircuit = fallback ?: Streams.<T>fail(new Exception("service unavailable"))
def circuitSwitcher = Streams.switchOnNext()

stream
      .materialize()
      .window(maxSignals, maxTime)
      .flatMap{ s ->
            s.reduce(["failures":0, "succ                    ple ->
                  if(tuple.t2.isOnNext(                    cess++
                  else if(tuple.t2.is                      e.t1.failure
            }
      }
      .consume({ streamHealth
            if(streamHealth.                                          ) > threshold) {
                  Streams.                                    out)
                        cir                                 .onNe
                  }
                  circuitSwitc
            } else {
                  circuitS
            }
      }, circuitSw                                    witc

//start with close
circuitSwitcher.onNe                                  rcuit)
circuitSwitcher
```

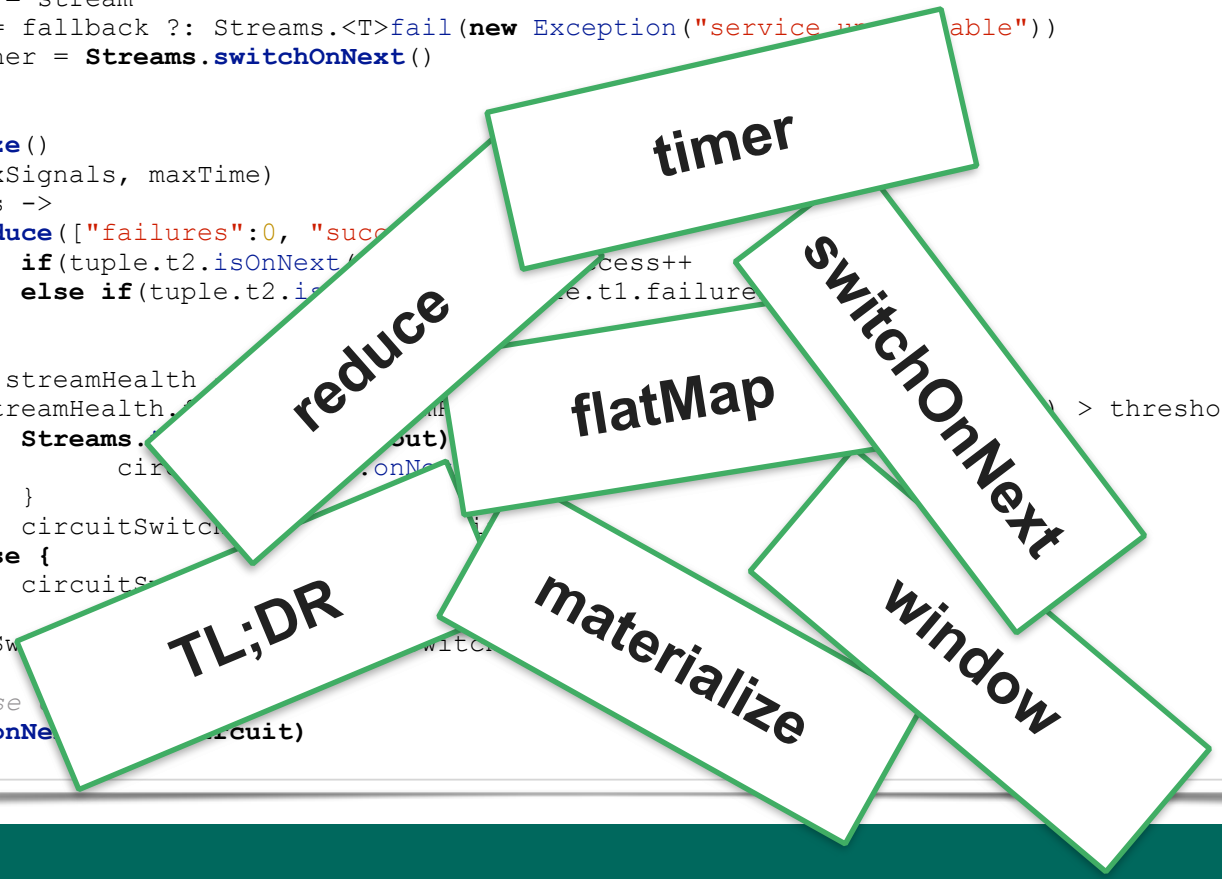


Failproof: Implementing a Circuit Breaker

```
def closeCircuit = stream
def openCircuit = fallback ?: Streams.<T>fail(new Exception("service u          able"))
def circuitSwitcher = Streams.switchOnNext()

stream
      .materialize()
      .window(maxSignals, maxTime)
      .flatMap{ s ->
            s.reduce(["failures":0, "succ
                  if(tuple.t2.isOnNext(
                  else if(tuple.t2.is
            }
      }
      .consume({ streamHealth
            if(streamHealth.
                  Streams.
                        cir
                  }
                  circuitSwitc
            } else {
                  circuitS
            }
      }, circuitSw

//start with close
circuitSwitcher.onNe          ircuit)
circuitSwitcher
```



Failproof: Implementing a Circuit Breaker

```
def fallback = Streams.<String>fail(new Exception("Fast fail")

//Open the circuit if there were more than 50% errors over 5 elements or within 3 sec.
//Will automatically close the circuit again after 2 sec.
Streams.circuitBreaker(stream, fallback, 5, 3, 0.5, 2)
     .retryWhen{ s -> s.zipWith(Streams.range(1,3)){ tuple.t2 }
                      .flatMap{ Streams.timer(it) }
                    }
     .consume (
          { println it },
          { println it.message },
          { println errors }
     )
```

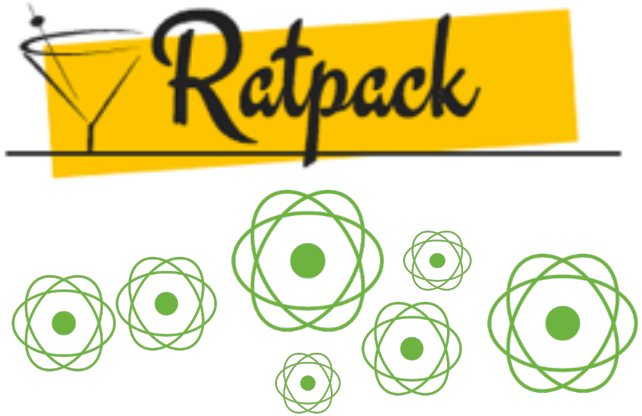
Failproof: Implementing a Circuit Breaker

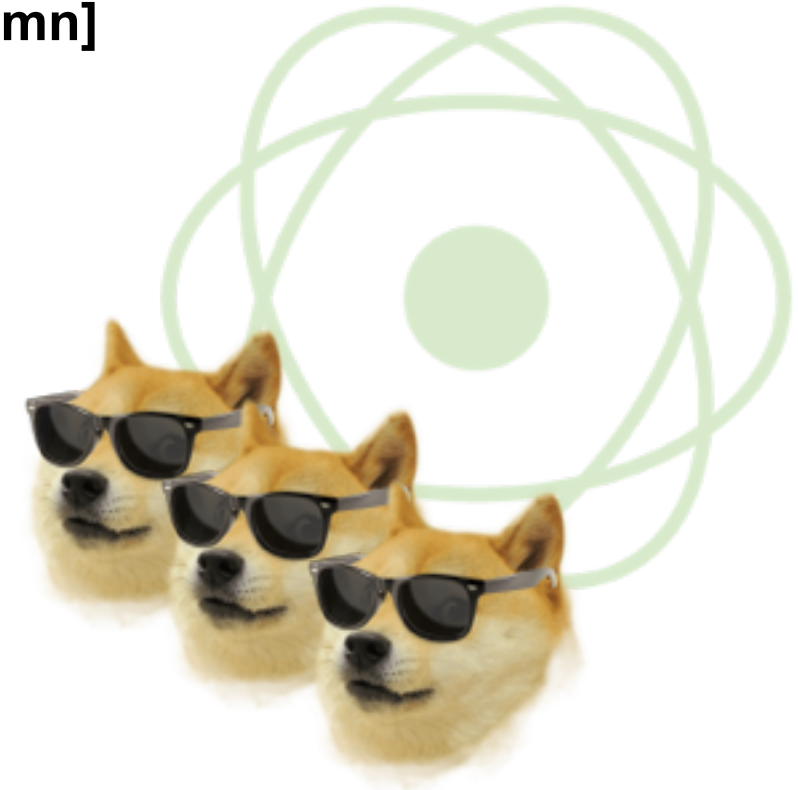
```
def fallback = Streams.<String>fail(new Exception("Fast fail")

//Open the circuit if there were more than 50% errors ov                    n 3 sec.
//Will automatically close the circuit again after 
Streams.circuitBreaker(stream, fallback, 5, 3, 
     .retryWhen{ s -> s.zipWith(Streams.ra
                     .flatMap{ Streams
                    }
     .consume (
          { println it }
          { println
          { pri
     )
```

Coming soon™ in M2

DEMO [if time left > 1.mn]





Featuring….

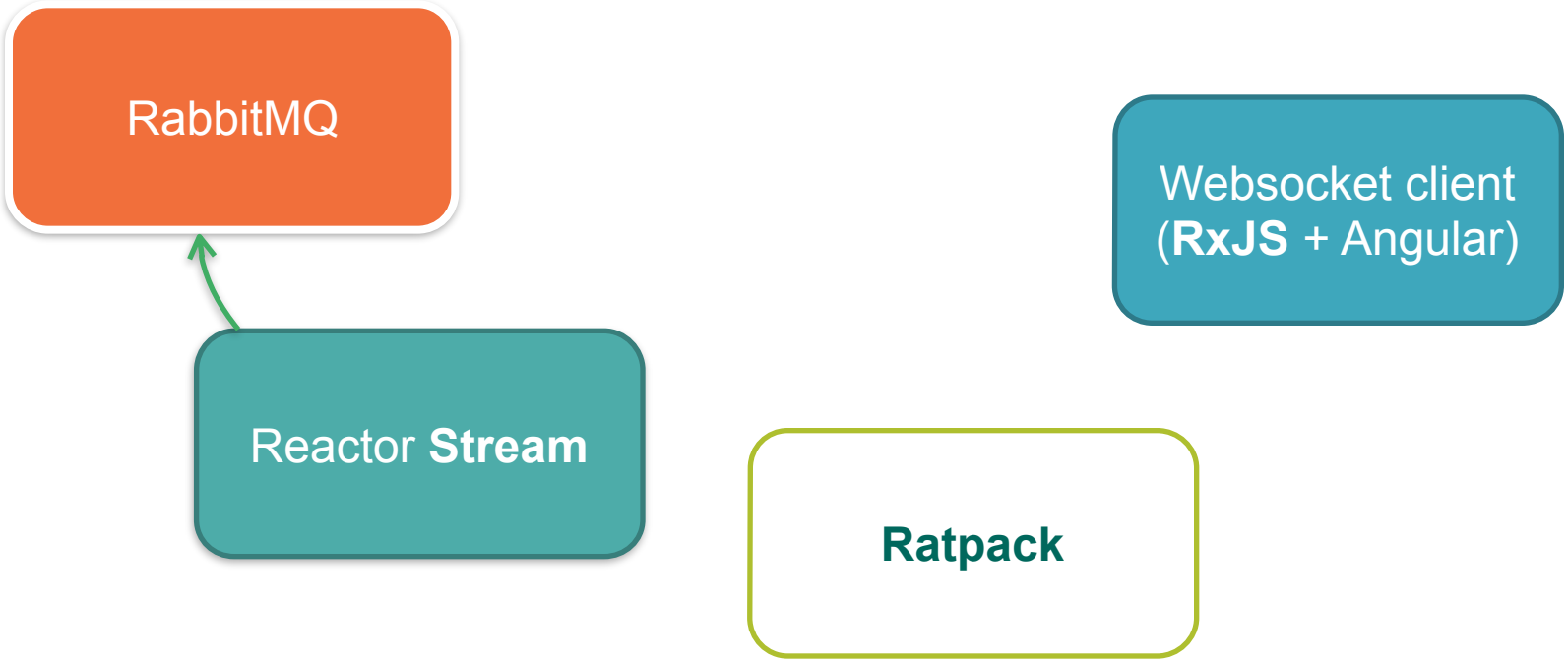
RabbitMQ

Websocket client (**RxJS** + Angular)

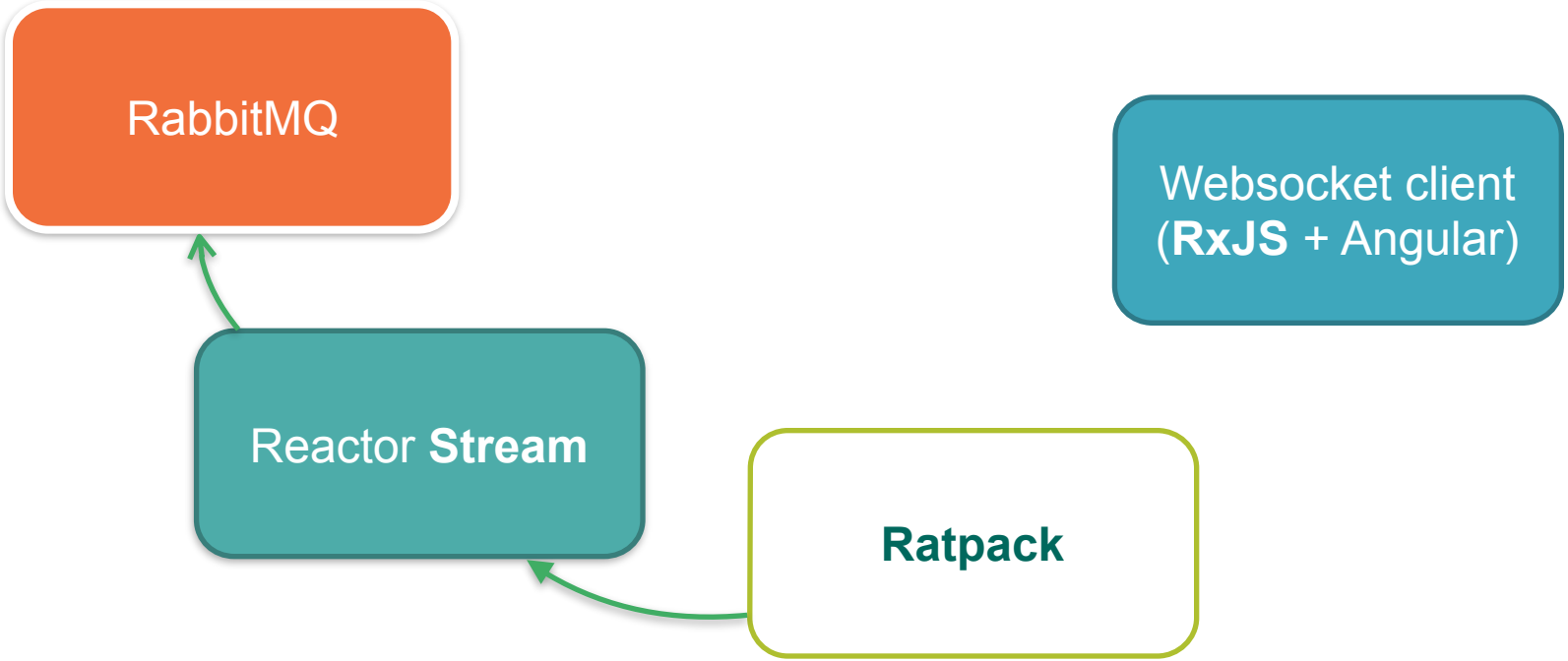
Reactor **Stream**

Ratpack

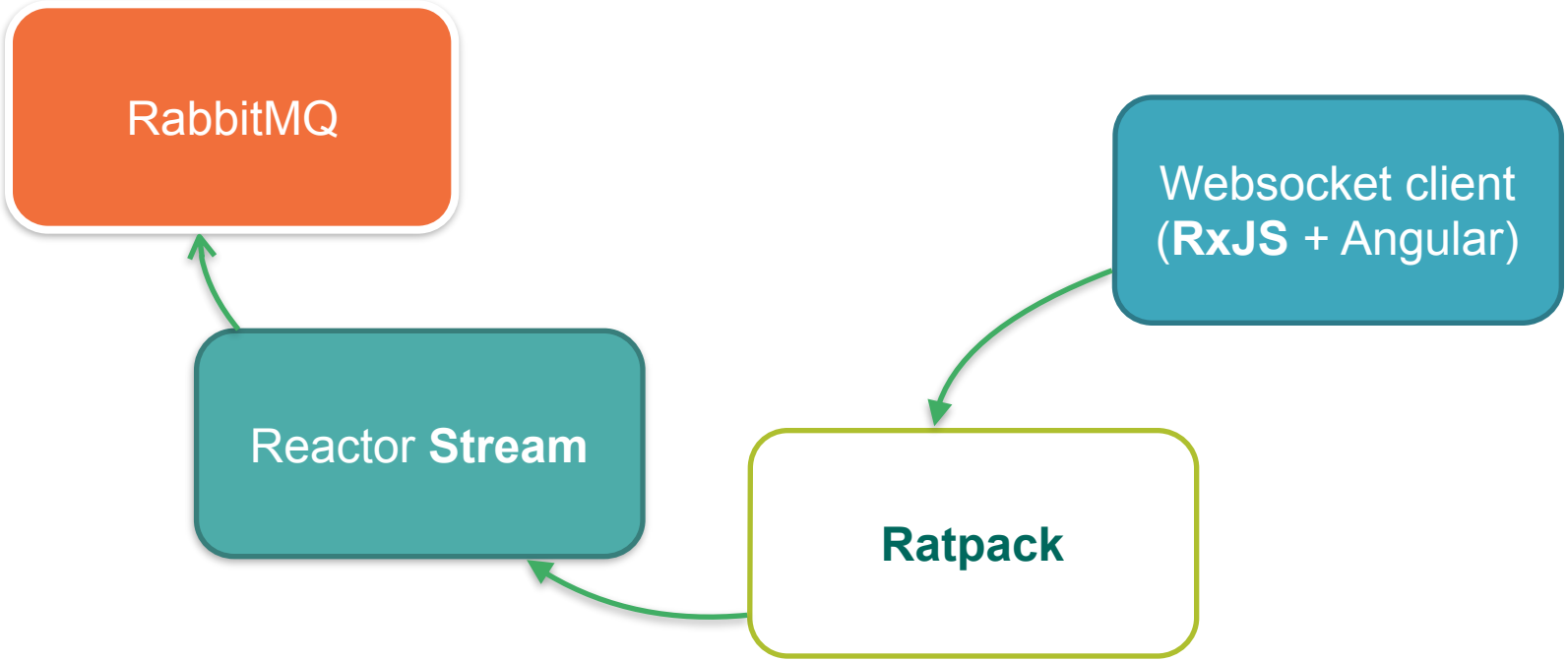
Featuring….


RabbitMQ
Reactor **Stream**
Websocket client
(**RxJS** + Angular)
Ratpack

Featuring….


RabbitMQ
Websocket client
(**RxJS** + Angular)
Reactor **Stream**
Ratpack

Featuring….


RabbitMQ
Websocket client
(**RxJS** + Angular)
Reactor **Stream**
Ratpack

Early adopters

- **Checkpoint**
  - **Reactor 2.0.0.M1** implements **0.4.0 - TCK OK**
  - **Akka Streams 0.10-M1** implements **0.4.0 - TCK OK**
  - **RxJava - ReactiveStreams 0.3.0** implements **0.4.0 - TCK EVALUATED**
  - **Ratpack 0.9.9** implements **0.4.0 - TCK OK**

- **Links**
  - **https://github.com/Netflix/RxJava**
  - **http://typesafe.com/blog/typesafe-announces-akka-streams**
  - **https://github.com/reactor/reactor**
  - **http://www.ratpack.io/manual/0.9.9/streams.html**

ReactiveStreams.onSubscribe(Resources)

- www.reactive-streams.org
- https://github.com/reactive-streams/reactive-streams

- **on maven central : 0.4.0**
  - **org.reactivestreams/reactive-streams**
  - **org.reactivestreams/reactive-streams-tck**

ReactiveStreams.onNext(Roadmap)

- Discussed for inclusion in JDK
- Close to release: 1.0.0.M1
  - Evaluating TCK before going 1.0 final
    - **Need 3 fully passing implementations before going 1.0.0.M1**

Reactor.onSubscribe(Resources)

- http://projectreactor.cfapps.io/
- https://github.com/reactor
- **Twitter: @projectReactor**
- **Blog Post: https://spring.io/blog/2014/10/21/reactor-2-0-0-m1-released-with-reactive-streams-integration**
- **on maven central : 2.0.0.M1, 2.0.0.BUILD-SNAPSHOT**
  - **org.projectreactor/reactor**

Q & A

session.onComplete()