REACTIVE STREAMS IN PRACTICE

Reactive Streams meetup Utrecht
29-03-2017

CODE.STAR

✉ joost@de-vries.name
joost-de-vries
@jouke

THREE ACTS

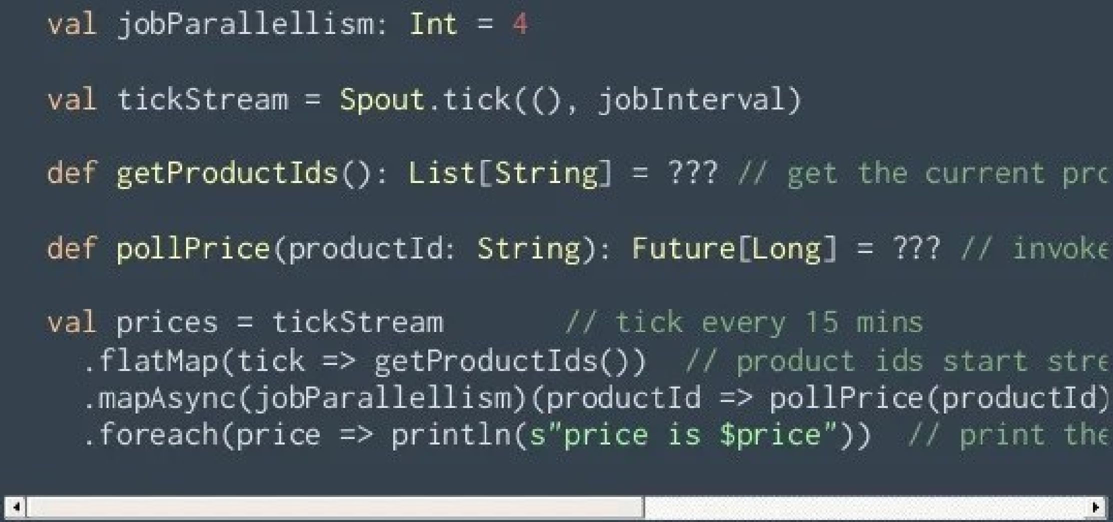
1. A little taste: some practical code
2. what is stream processing and what can we do with it
3. reactive streams

Let's talk about our experiences with streams

LET'S START WITH SOME CODE

We want to poll some market place every 15 minutes for the prices of our products.
And we don't want to be suspended, so we only do 4 requests at a time.'

LET'S START WITH SOME CODE



```
val jobParallellism: Int = 4

val tickStream = Spout.tick((), jobInterval)

def getProductIds(): List[String] = ??? // get the current pro

def pollPrice(productId: String): Future[Long] = ??? // invoke

val prices = tickStream       // tick every 15 mins
  .flatMap(tick => getProductIds())  // product ids start stre
  .mapAsync(jobParallellism)(productId => pollPrice(productId)
  .foreach(price => println(s"price is $price"))  // print the
```

This happens to be Scala

2

WHAT IS STREAM PROCESSING?

And what can we do with it?

LIKE A CROSS BETWEEN ITERATOR AND COMPLETABLEFUTURE

	synchronous		asynchronous
one	A getA()	⇒	CompletableFuture[A] getA()
many	Iterator[A]	⇒	Observable[A]

Using Java and RxJava

LIKE A CROSS BETWEEN ITERATOR AND COMPLETABLEFUTURE

	synchronous		asynchronous
one	A getA()	⇒	CompletableFuture[A] getA()
many	Iterator[A]	⇒	Observable[A]

We call for data vs we get called with data

pull vs push

polling vs reactive

HMM..

So how is a stream different from a collection?

A stream is potentially infinite
A stream is spread out in time instead of memory
A stream can be non repeatable

STREAMS

Asynchronous stream processing

Source

Out

Created by Piotrek Chuchla
from Noun Project

Flow

In, Out

Created by [illegible]
from Noun Project

Sink

In

..also Spouts, Pipes, Drains

Rx: Observable, Observer

CREATE A STREAM

```
const source = Rx.Observable.create((observer) => {
    observer.onNext(3)
    observer.onNext(2)
    observer.onNext(1) // 0 or more values
    observer.onCompleted() // potentially end or error
})
```

3, 2, 1 .

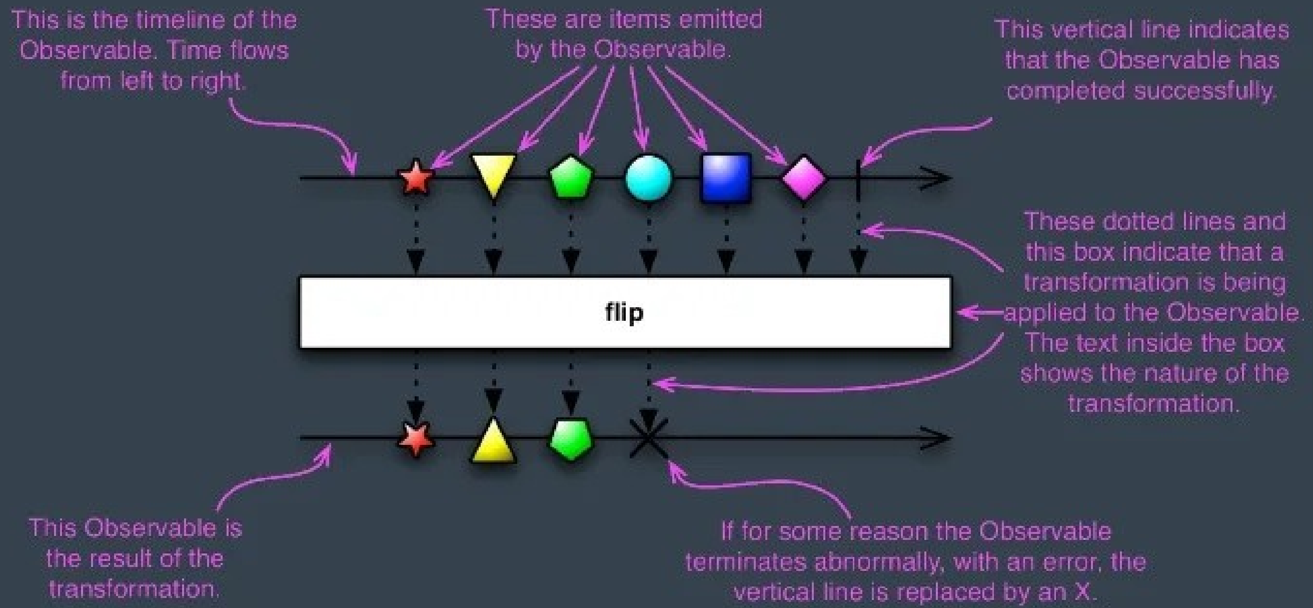
(element)* (complete|error)

A FAILING STREAM

```
const source = Rx.Observable.create((observer) => {
    observer.onNext(3)
    observer.onNext(2)
    observer.onError(new Error("boom"))
})
```

```
3, 2, 1 x
```

STREAMS IN DIAGRAMS



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

These are items emitted by the Observable.

This vertical line indicates that the Observable has completed successfully.

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

This Observable is the result of the transformation.

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

Marble diagrams

Streams are about time and change

See `reactivex.io`

③

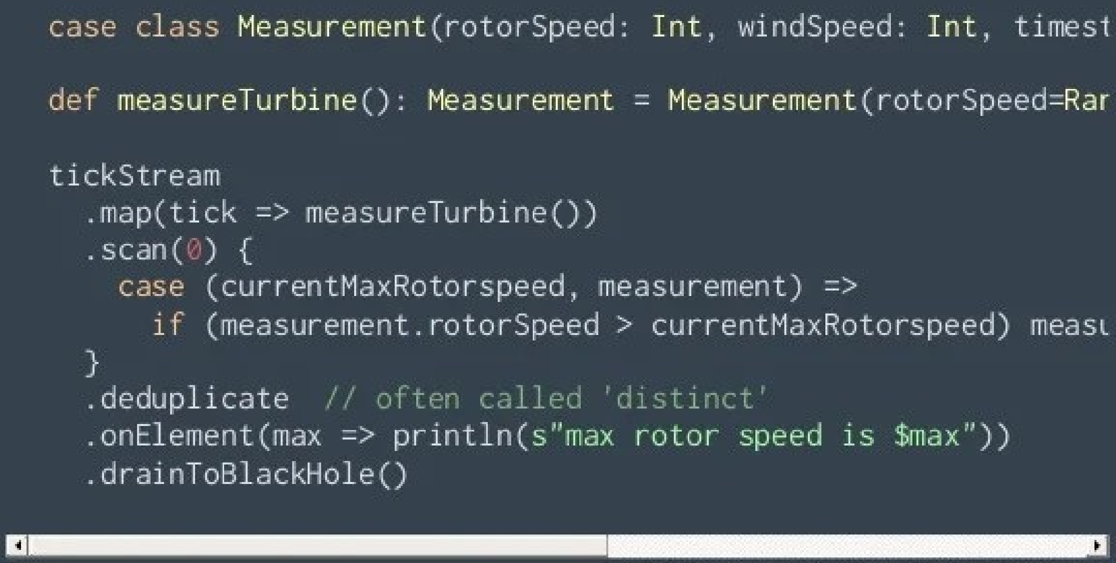
WHAT CAN WE DO WITH STREAMS?

WHAT CAN WE DO WITH STREAMS?

- IoT: dealing with sensor measurements
- System integration: pumping data between servers
- User interface code
- Server push

(Near) real time
Fast data
Immediate results as the inputs are coming in

DETERMINE HIGH WATER MARK FOR SENSOR MEASUREMENTS

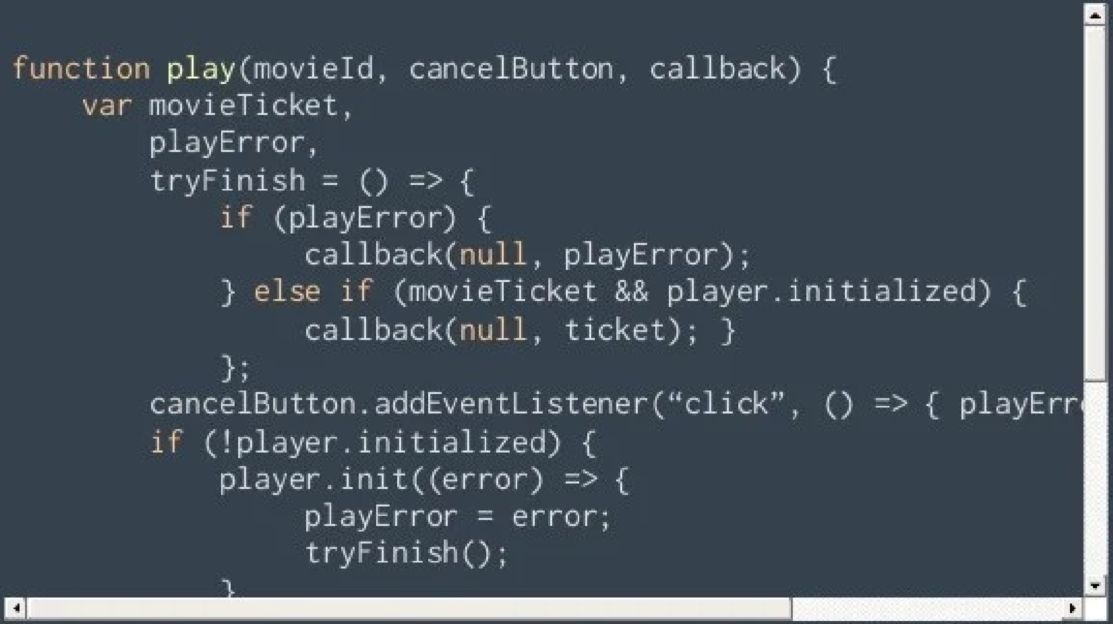


```
case class Measurement(rotorSpeed: Int, windSpeed: Int, timest

def measureTurbine(): Measurement = Measurement(rotorSpeed=Rar

tickStream
  .map(tick => measureTurbine())
  .scan(0) {
    case (currentMaxRotorspeed, measurement) =>
      if (measurement.rotorSpeed > currentMaxRotorspeed) measu
  }
  .deduplicate  // often called 'distinct'
  .onElement(max => println(s"max rotor speed is $max"))
  .drainToBlackHole()
```

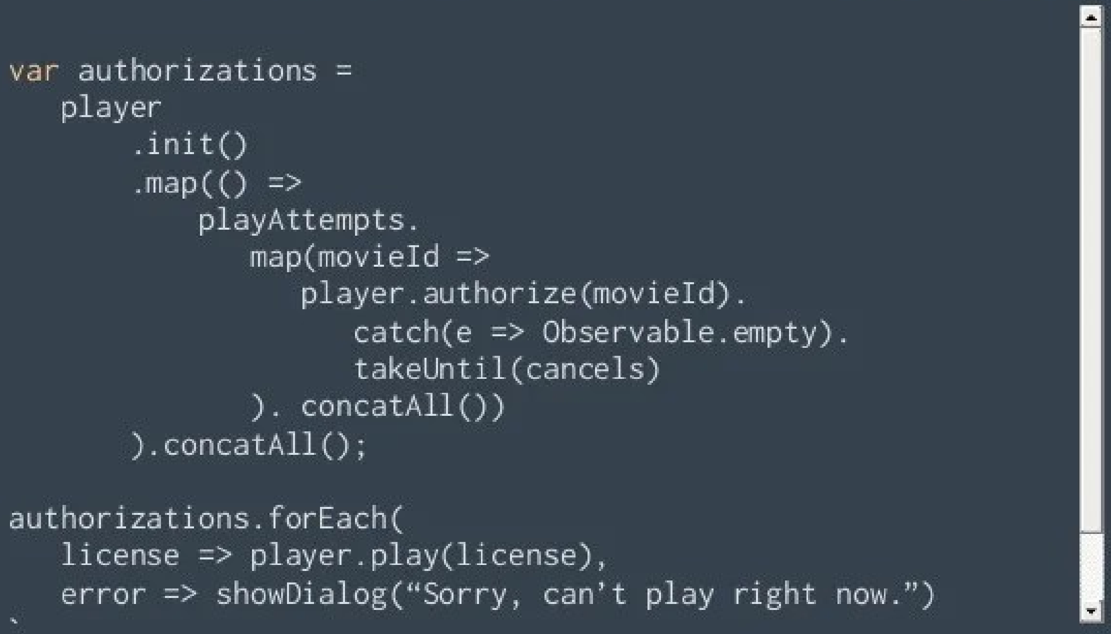
USER INTERFACE LOGIC: NETFLIX



```
function play(movieId, cancelButton, callback) {
    var movieTicket,
        playError,
        tryFinish = () => {
            if (playError) {
                 callback(null, playError);
            } else if (movieTicket && player.initialized) {
                 callback(null, ticket); }
            };
        cancelButton.addEventListener(“click”, () => { playErr
        if (!player.initialized) {
            player.init((error) => {
                 playError = error;
                 tryFinish();
            }
```

See Jafar Husains talk

USER INTERFACE LOGIC: NETFLIX

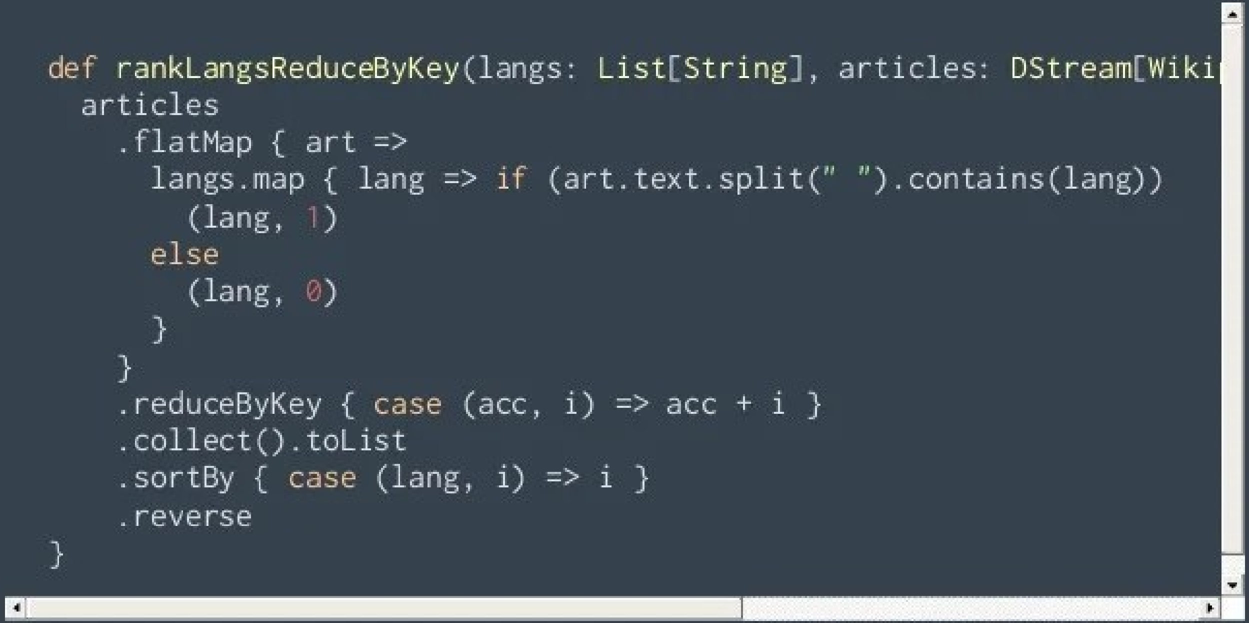


```
var authorizations =
   player
       .init()
       .map(() =>
           playAttempts.
              map(movieId =>
                 player.authorize(movieId).
                    catch(e => Observable.empty).
                    takeUntil(cancels)
              ). concatAll())
       ).concatAll();

authorizations.forEach(
   license => player.play(license),
   error => showDialog("Sorry, can't play right now.")
)
```

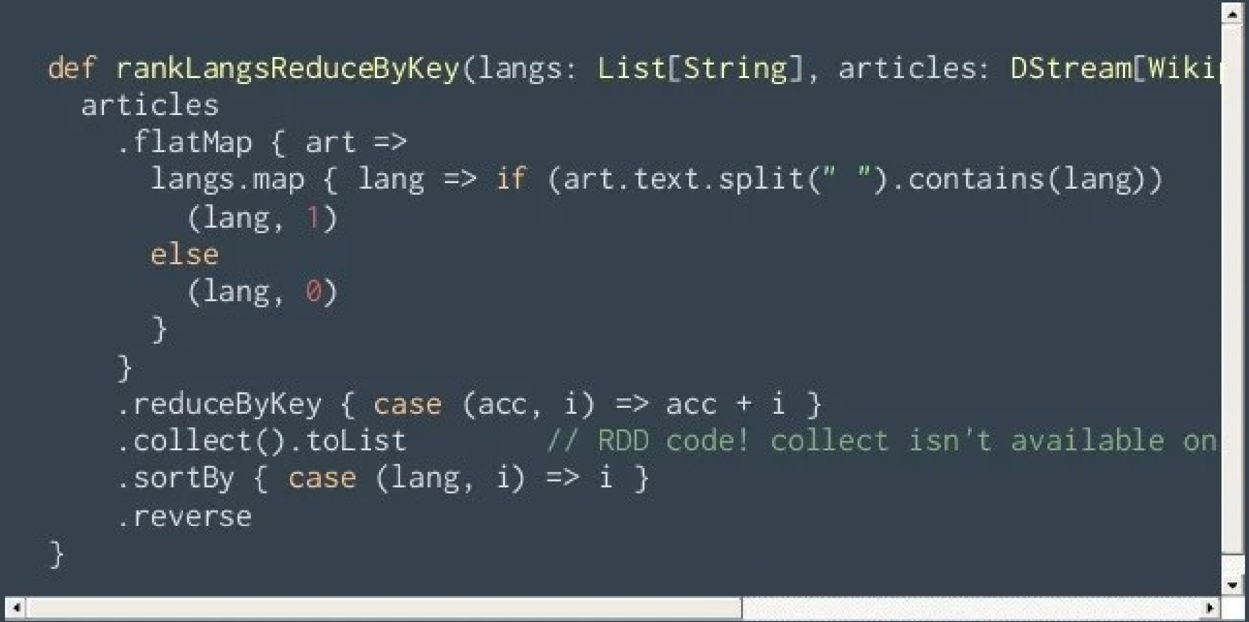
See Jafar Husains talk

CLUSTERED STREAMS

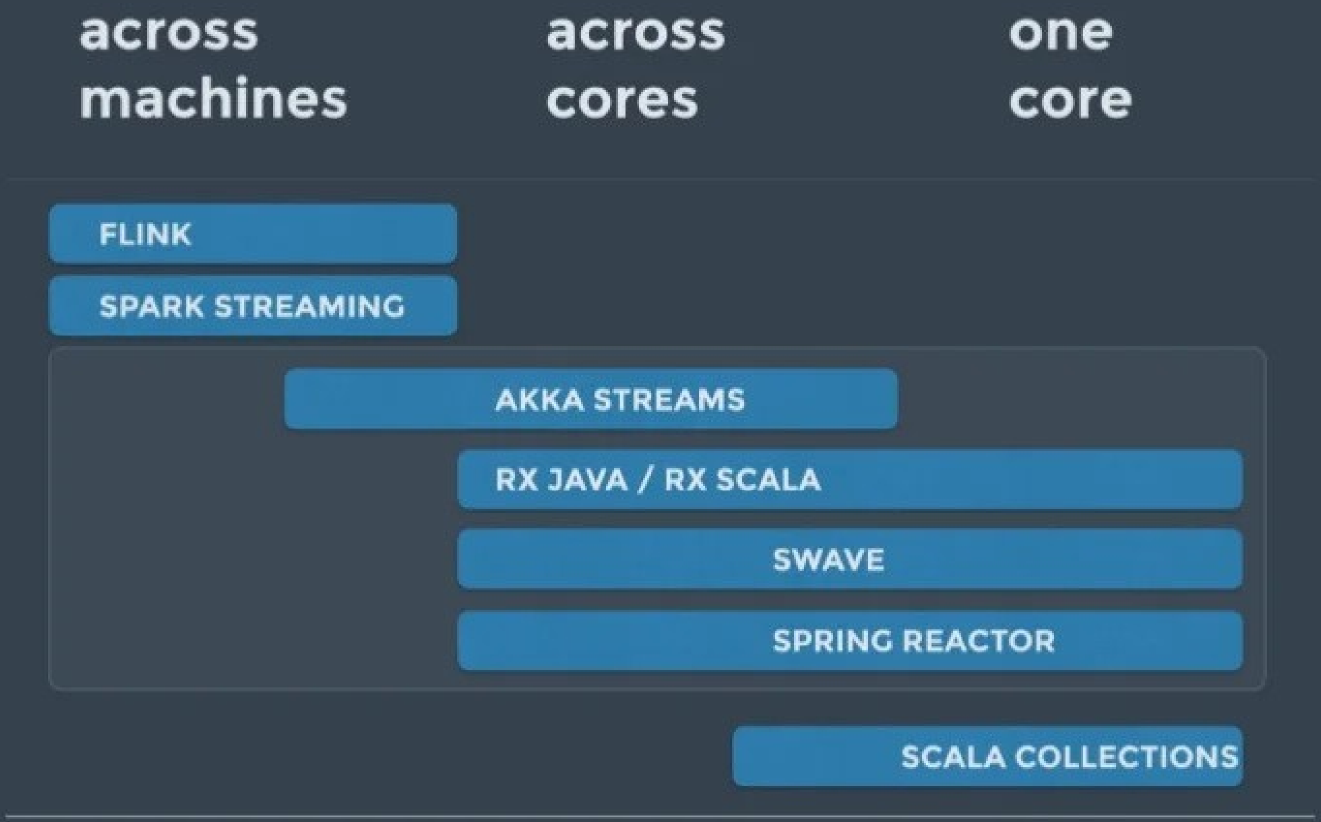


```
def rankLangsReduceByKey(langs: List[String], articles: DStream[Wiki
  articles
    .flatMap { art =>
      langs.map { lang => if (art.text.split(" ").contains(lang))
        (lang, 1)
      else
        (lang, 0)
      }
    }
    .reduceByKey { case (acc, i) => acc + i }
    .collect().toList
    .sortBy { case (lang, i) => i }
    .reverse
}
```

CLUSTERED STREAMS



```
def rankLangsReduceByKey(langs: List[String], articles: DStream[Wiki
  articles
    .flatMap { art =>
      langs.map { lang => if (art.text.split(" ").contains(lang))
        (lang, 1)
      else
        (lang, 0)
      }
    }
    .reduceByKey { case (acc, i) => acc + i }
    .collect().toList        // RDD code! collect isn't available on
    .sortBy { case (lang, i) => i }
    .reverse
}
```



across machines

across cores

one core

FLINK

SPARK STREAMING

AKKA STREAMS

RX JAVA / RX SCALA

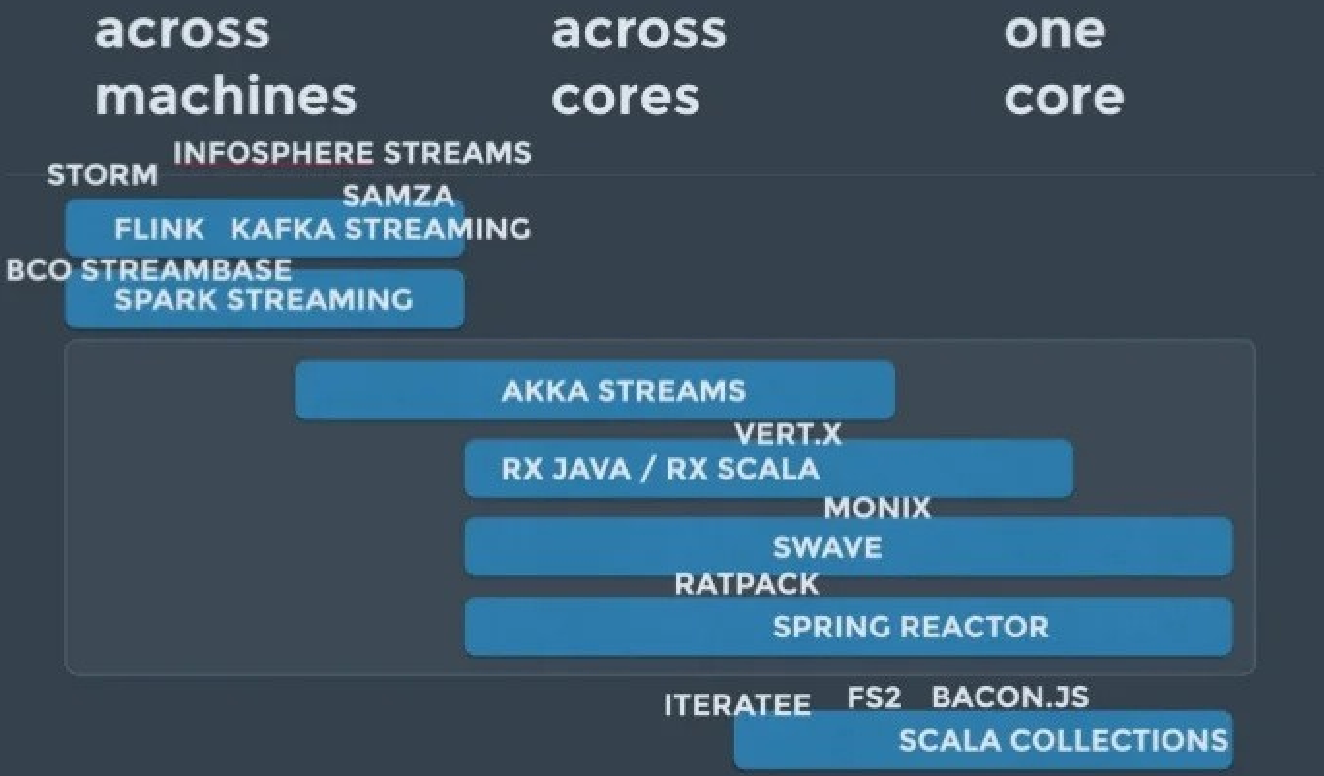
SWAVE

SPRING REACTOR

SCALA COLLECTIONS

Types of stream processing

Not only JVM: JS, .Net, Swift,..



across machines

across cores

one core

INFOSPHERE STREAMS

STORM

SAMZA

FLINK KAFKA STREAMING

BCO STREAMBASE

SPARK STREAMING

AKKA STREAMS

VERT.X

RX JAVA / RX SCALA

MONIX

SWAVE

RATPACK

SPRING REACTOR

ITERATEE FS2 BACON.JS

SCALA COLLECTIONS

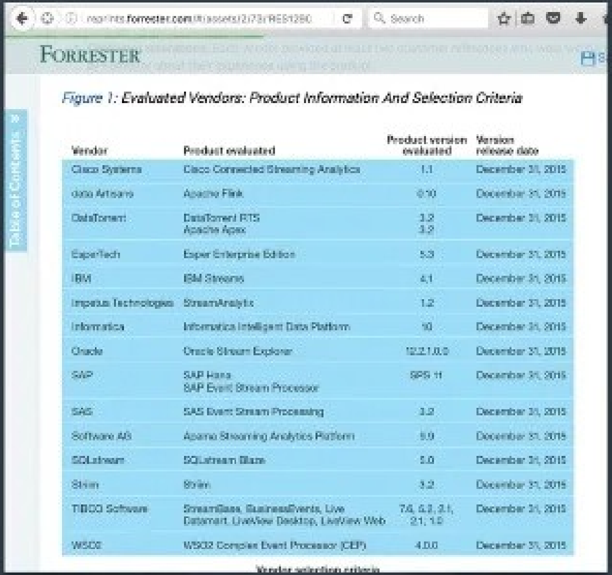


Figure 1: Evaluated Vendors: Product Information And Selection Criteria

Vendor	Product evaluated	Product version evaluated	Version release date
Cisco Systems	Cisco Connected Streaming Analytics	1.1	December 31, 2015
data Artisans	Apache Flink	0.10	December 31, 2015
DataTorrent	DataTorrent RTS Apache Apex	3.2 3.2	December 31, 2015
EsperTech	Esper Enterprise Edition	5.3	December 31, 2015
IBM	IBM Streams	4.1	December 31, 2015
Impetus Technologies	StreamAnalytix	1.2	December 31, 2015
Informatica	Informatica Intelligent Data Platform	10	December 31, 2015
Oracle	Oracle Stream Explorer	12.2.1.0.0	December 31, 2015
SAP	SAP Hana SAP Event Stream Processor	SPS 11	December 31, 2015
SAS	SAS Event Stream Processing	3.2	December 31, 2015
Software AG	Apama Streaming Analytics Platform	9.9	December 31, 2015
SQLstream	SQLstream Blaze	5.0	December 31, 2015
Striim	Striim	3.2	December 31, 2015
TIBCO Software	StreamBase, BusinessEvents, Live Datamart, LiveView Desktop, LiveView Web	7.6, 5.3, 2.1, 2.1, 1.0	December 31, 2015
WSO2	WSO2 Complex Event Processor (CEP)	4.0.0	December 31, 2015

Vendor selection criteria

④

WHY STREAMS?

START WORKING WHEN DATA AVAILABLE

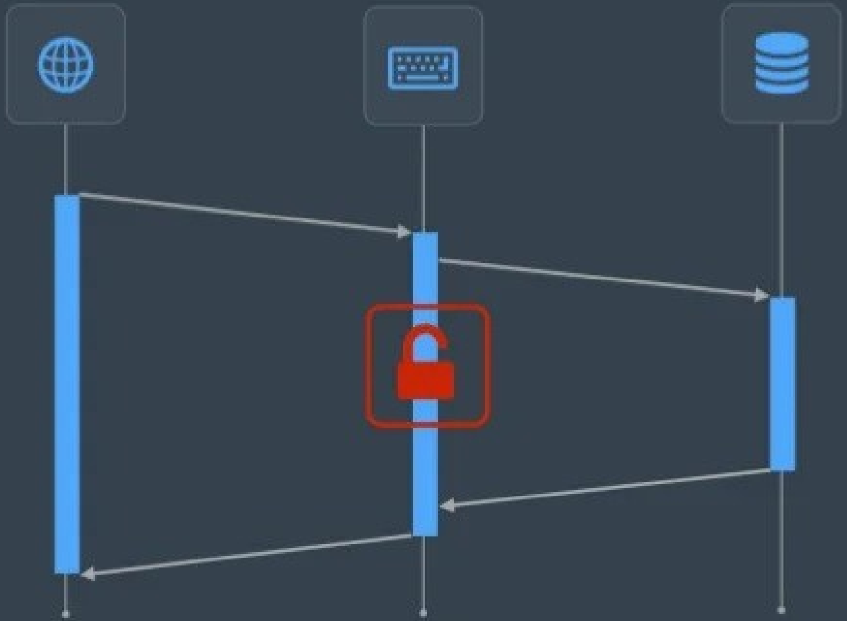
- Push instead of pull
- Least latency
- Process stuff without loading it all in memory

Reverse of query / polling

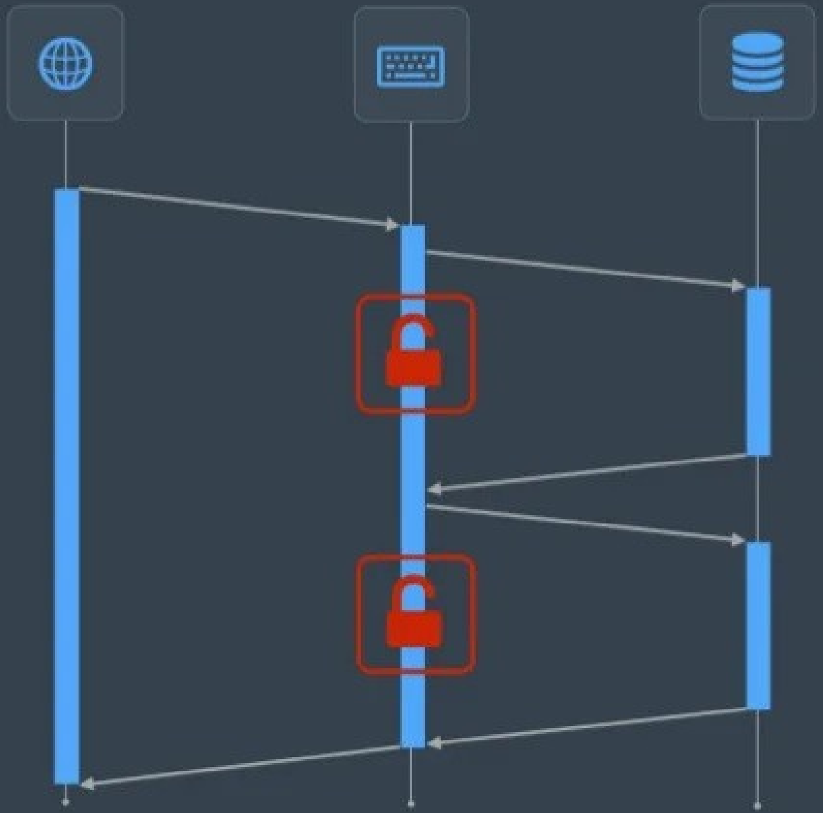
NON BLOCKING BOUNDED USE OF RESOURCES

- limited threads
- bounded memory usage (see backpressure)
- important for microservices / IoT-mobile long running reqs

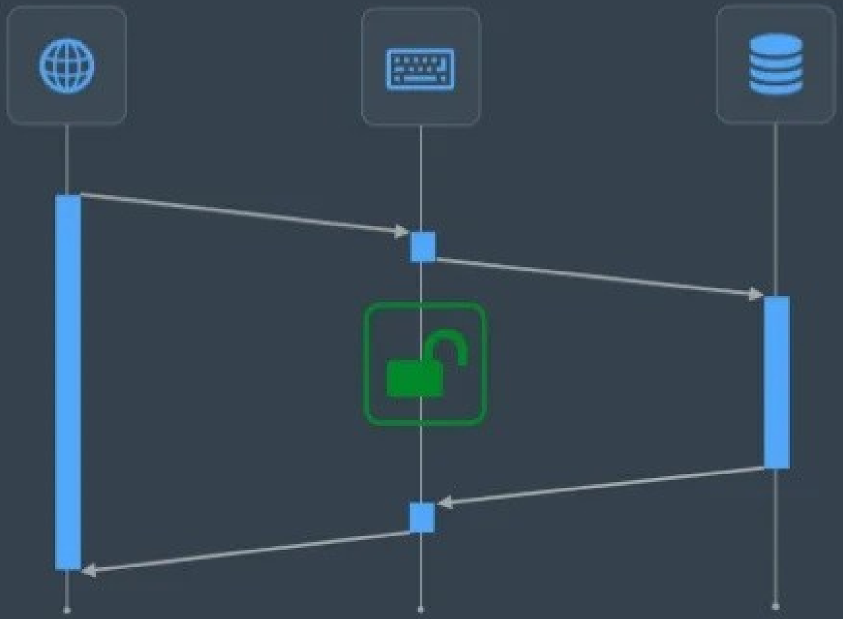
BLOCKING WAIT



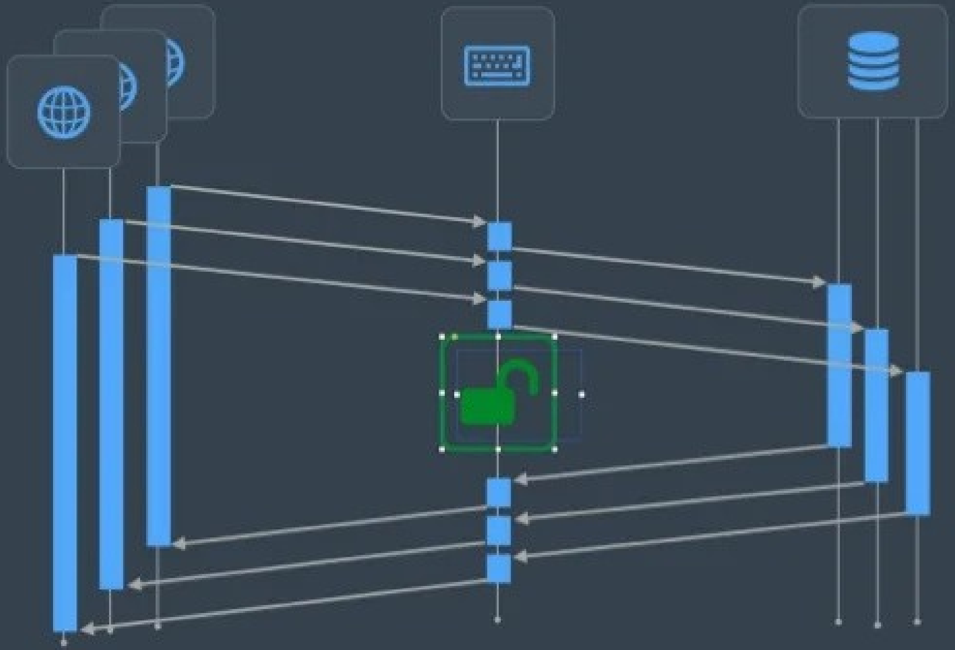
HIGHER LATENCY FOR USER



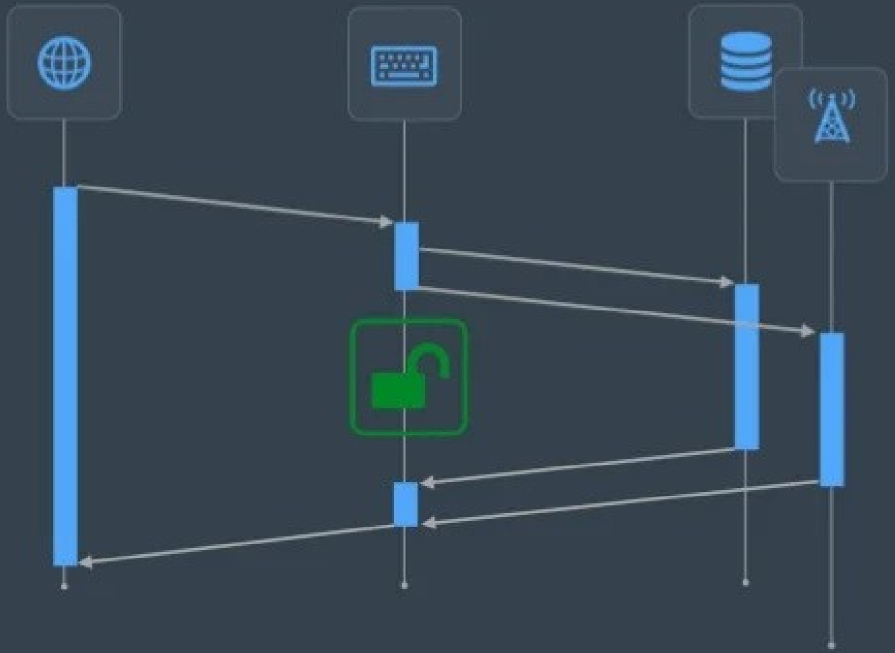
ASYNCHRONOUS PROCESSING



MULTIPLE CLIENTS



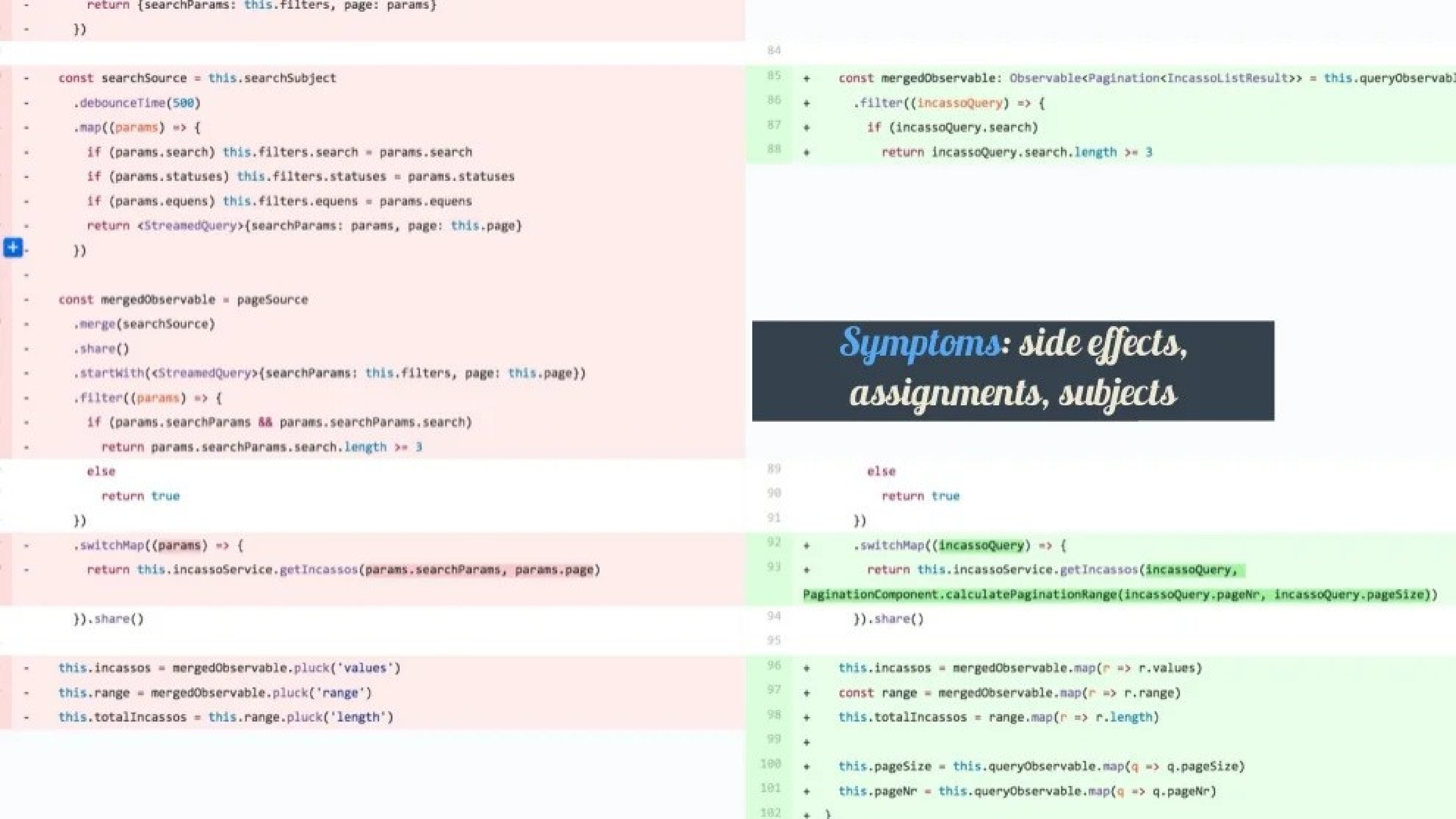
PARALLEL PROCESSING



CLEAN PROGRAMMING MODEL

- Concurrency model
- (flat)map: collection, parallel collection, clustered collection, stream

Functional programming



Symptoms: side effects, assignments, subjects

```
-       return {searchParams: this.filters, page: params}
-     })

-   const searchSource = this.searchSubject
-     .debounceTime(500)
-     .map((params) => {
-       if (params.search) this.filters.search = params.search
-       if (params.statuses) this.filters.statuses = params.statuses
-       if (params.equens) this.filters.equens = params.equens
-       return <StreamedQuery>{searchParams: params, page: this.page}
-     })
-
-   const mergedObservable = pageSource
-     .merge(searchSource)
-     .share()
-     .startWith(<StreamedQuery>{searchParams: this.filters, page: this.page})
-     .filter((params) => {
-       if (params.searchParams && params.searchParams.search)
-         return params.searchParams.search.length >= 3
        else
          return true
      })
-     .switchMap((params) => {
-       return this.incassoService.getIncassos(params.searchParams, params.page)

      }).share()

-   this.incassos = mergedObservable.pluck('values')
-   this.range = mergedObservable.pluck('range')
-   this.totalIncassos = this.range.pluck('length')
```

```

+   const mergedObservable: Observable<Pagination<IncassoListResult>> = this.queryObservabl
+     .filter((incassoQuery) => {
+       if (incassoQuery.search)
+         return incassoQuery.search.length >= 3
        else
          return true
      })
+     .switchMap((incassoQuery) => {
+       return this.incassoService.getIncassos(incassoQuery,
PaginationComponent.calculatePaginationRange(incassoQuery.pageNr, incassoQuery.pageSize))
      }).share()

+   this.incassos = mergedObservable.map(r => r.values)
+   const range = mergedObservable.map(r => r.range)
+   this.totalIncassos = range.map(r => r.length)
+
+   this.pageSize = this.queryObservable.map(q => q.pageSize)
+   this.pageNr = this.queryObservable.map(q => q.pageNr)
+ }
```

UI LOGIC IN TYPESCRIPT RXJS

```
const incassos: Observable<Pagination<IncassoListResult>> = this.queryOb
  .filter((incassoQuery) => {
    if (incassoQuery.search)
      return incassoQuery.search.length >= 3
    else
      return true
  })
  .switchMap((incassoQuery) => {
    return this.incassoService.getIncassos(incassoQuery, PaginationCompo
  }).share()
```

Move assignments and subjects

to the outside of your code

Observable in, Observable out

Use pure transformations

⑤

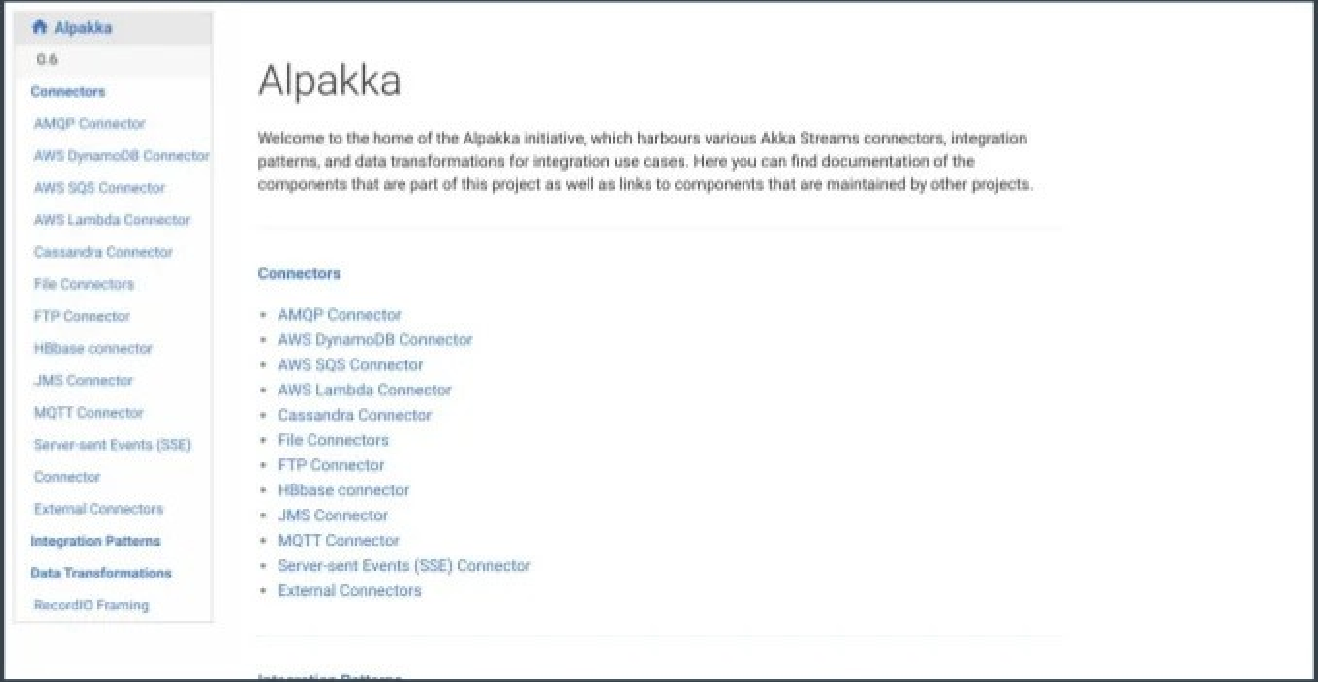
REACTIVE STREAMS

What makes them different?

REACTIVE STREAMS

- Interoperability standard
  - Oracle
  - Lightbend
  - Pivotal
  - Netflix
  - Redhat
- Supports backpressure

STREAM ANYTHING



Alpakka
0.6

Connectors

- AMQP Connector
- AWS DynamoDB Connector
- AWS SQS Connector
- AWS Lambda Connector
- Cassandra Connector
- File Connectors
- FTP Connector
- HBbase connector
- JMS Connector
- MQTT Connector
- Server-sent Events (SSE) Connector
- External Connectors

Integration Patterns

Data Transformations

- RecordIO Framing

Alpakka

Welcome to the home of the Alpakka initiative, which harbours various Akka Streams connectors, integration patterns, and data transformations for integration use cases. Here you can find documentation of the components that are part of this project as well as links to components that are maintained by other projects.

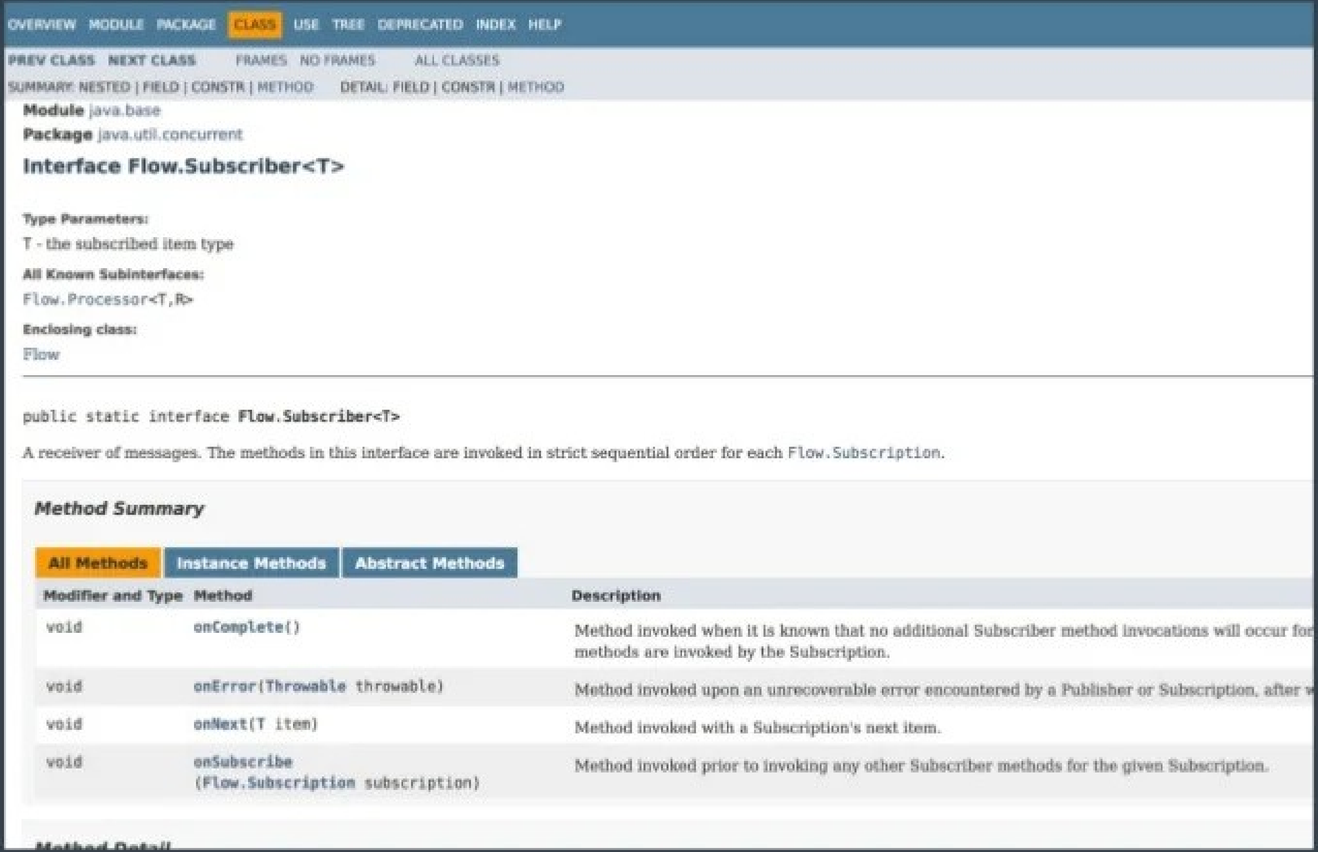
Connectors

- AMQP Connector
- AWS DynamoDB Connector
- AWS SQS Connector
- AWS Lambda Connector
- Cassandra Connector
- File Connectors
- FTP Connector
- HBbase connector
- JMS Connector
- MQTT Connector
- Server-sent Events (SSE) Connector
- External Connectors

Integration Patterns

Example from Akka
Spring Cloud will support Reactor (?)

RS INTERFACES IN JAVA 9


OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Module java.base

Package java.util.concurrent

Interface Flow.Subscriber<T>

Type Parameters:

T - the subscribed item type

All Known Subinterfaces:

`Flow.Processor<T,R>`

Enclosing class:

Flow

```
public static interface Flow.Subscriber<T>
```

A receiver of messages. The methods in this interface are invoked in strict sequential order for each `Flow.Subscription`.

Method Summary

All Methods
Instance Methods
Abstract Methods

Modifier and Type	Method	Description
`void`	`onComplete()`	Method invoked when it is known that no additional Subscriber method invocations will occur for methods are invoked by the Subscription.
`void`	`onError(Throwable throwable)`	Method invoked upon an unrecoverable error encountered by a Publisher or Subscription, after w
`void`	`onNext(T item)`	Method invoked with a Subscription's next item.
`void`	`onSubscribe(Flow.Subscription subscription)`	Method invoked prior to invoking any other Subscriber methods for the given Subscription.

Method Detail

BACKPRESSURE

What if up stream is *faster*?

Buffer

Drop elements

Slow down

BACKPRESSURE

Doesn't make sense for sensor data, mouse clicks, ...

But if you can't lose events...

Down stream communicates demand

⑥

RESOURCES

- Async JavaScript at Netflix," Jafar Husain https://www.infoq.com/presentations/netflix-rx-extensions
- Erik Meijer on observables
- http://reactivex.io/
- Talks by Stephane Maldini
- etc

SOURCES

- Overview of types of streams: Mathias Doenitz talk on Swave
- Erik Meijer on observables

Fin