
Typesafe

Reactive Streams and Akka Streams

Markus Jura (@markusjura)
& Lutz Huehnken (@lutzhuehnken)

Streams

Common Use of Streams

- Bulk Data Transfer
- Real Time Data Sources
- Batch Processing of large data sets
- Monitoring and Analytics

What is a Stream?

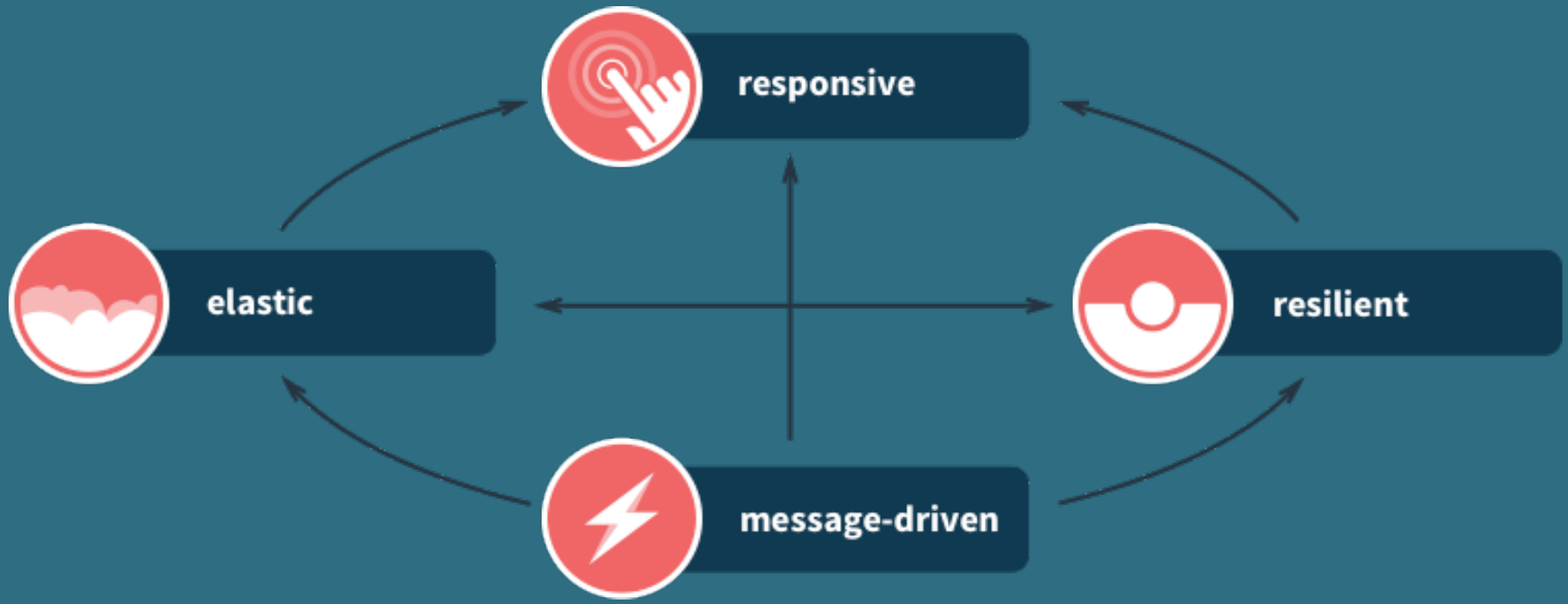
- Ephemeral flow of data
- Potentially unbounded in size
- Processed by describing transformation of data

Reactive Streams

Reactive Streams Projects / Companies

- Typesafe
  - Akka Streams
- Netflix
  - rxJava / rxScala
- Pivotal
  - Spring Reactor
- Redhat
  - Vert.x
- Oracle

Reactive

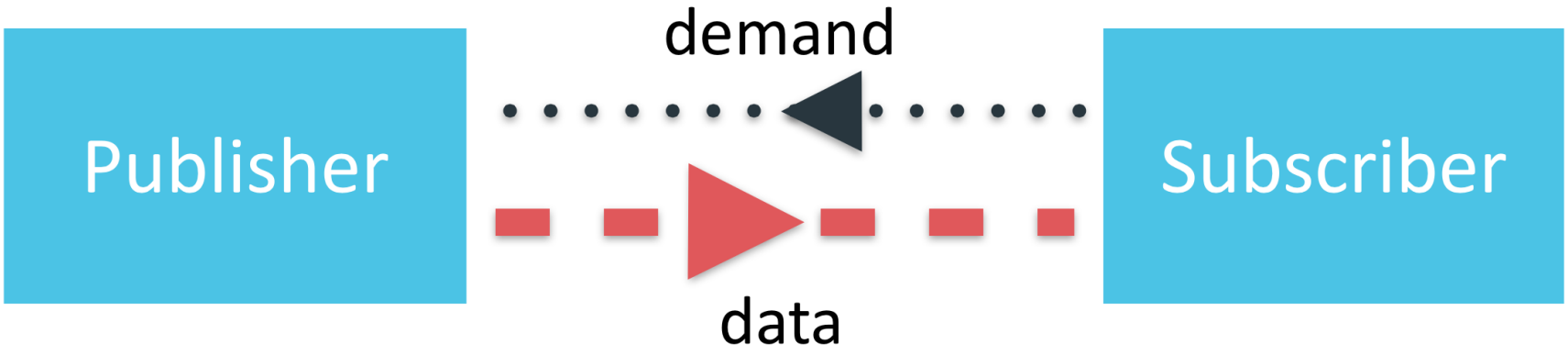


Reactive Streams Overview

- How do we:
  - Handle potentially infinite streams of data?
  - Handle data in a reactive manner?
  - Achieve asynchronous non-blocking data flow?
  - Avoid out of memory errors?

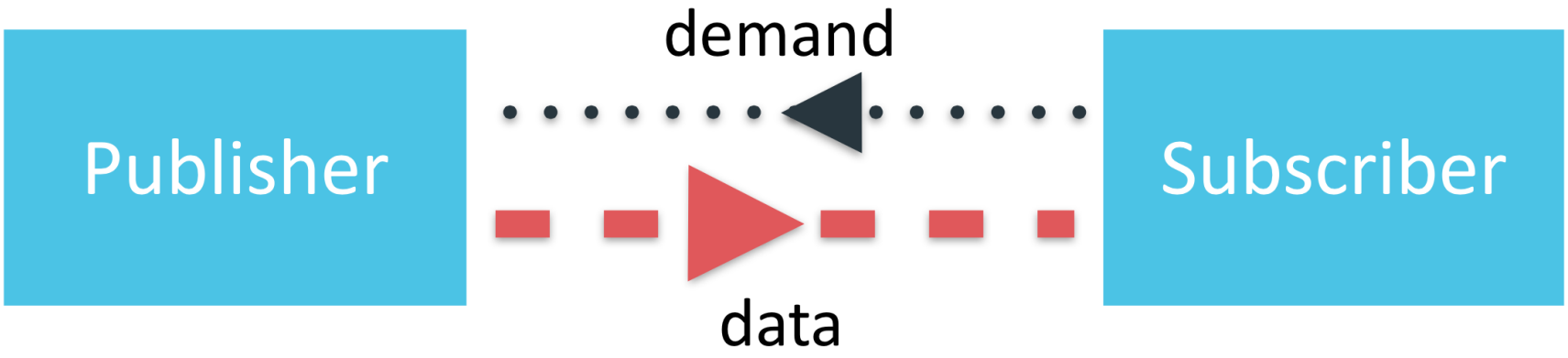
Supply and Demand

- Data Items Flow Downstream
- Demand Flows Upstream
- Data Items flow only when there is demand.



Dynamic Push-Pull

- “Push” behavior when consumer is faster
- “Pull” behavior when producer is faster
- Switches automatically between these
- Batching demand allows batching data



Reactive Streams Specification

- Interface Specification
  - Java interfaces for implementations
- TCK
  - Test Harness to validate implementations
- Specification Website
  - http://www.reactive-streams.org/

Publisher

```
package org.reactivestreams;
public interface Subscriber<T> {...}
public interface Publisher<T> {
  public void subscribe<T>(
    Subscriber<T> subscriber);
}
```

Subscriber

```
public interface Subscriber<T> {
  public void onSubscribe(
    Subscription subscription);
  public void onNext(T element);
  public void onComplete();
  public void onError(Throwable cause);
}
```

Subscription

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

Backpressure

- Downstream consumers pushing back on the producer to prevent flooding.
- In reactive-streams:
  - Consumers stop requesting more elements until they are ready to process more.
  - Producers only fire elements if there is demand.

Why Backpressure?

- Explicitly design for system overload
  - Demand is propagated throughout the WHOLE flow
  - Can decide WHERE to handle overload
- Limit the number of in-flight messages throughout the system
  - Bounded memory consumption
  - Bounded cpu contention
- Recipient is in control of incoming data rate
- Data in flight is bounded by signaled demand

Buffering

- We can prefetch stream elements by requesting more than we really need.
- We can use this technique to ensure no "sputter" in the stream.
- We can also use this technique to pull faster than downstream consumer.

Akka Streams & Actors

Basic Actor

```
case class Greeting(who: String)

class GreetingActor extends Actor with
ActorLogging {
  def receive = {
    case Greeting(who) => log.info("Hello " + who)
  }
}

val system = ActorSystem("MySystem")
val greeter: ActorRef =
  system.actorOf(Props[GreetingActor])

greeter ! Greeting("London Scala User Group")
```

Properties of Actors

- Message Based / Event Driven
- Isolated State
- Sane Concurrency Model
- Isolated Failure Handling (Supervision)

Akka Streams – A Bridge

- Converts Publisher/Subscriber API into Actor messages
- Simplify creating Publisher/Subscribers using Actors
- Attach Reactive streams into existing Akka applications

Creating an ActorSubscriber

```
import akka.stream._

class PrintlnActor extends Actor with ActorSubscriber {
  val requestStrategy = OneByOneRequestStrategy
  def receive: Receive = {
    case ActorSubscriberMessage.OnNext(element) =>
         println(element)
  }
}

val printlnActor:ActorRef =
  system.actorOf(Props[PrintlnActor], "println")

val subscriber = ActorSubscriber(printlnActor)
```

Creating an ActorPublisher

```
import akka.stream._

class IntActor extends Actor with ActorPublisher[Int] {
  def receive: Receive = {
    case ActorPublisherMessage.Request(elements) =>
      while (totalDemand > 0) { onNext(1) }
  }
}

val intActor: ActorRef =
  system.actorOf(Props[IntActor], "intActor")

val publisher =  ActorPublisher(intActor)
```

Why use Actors as Publishers?

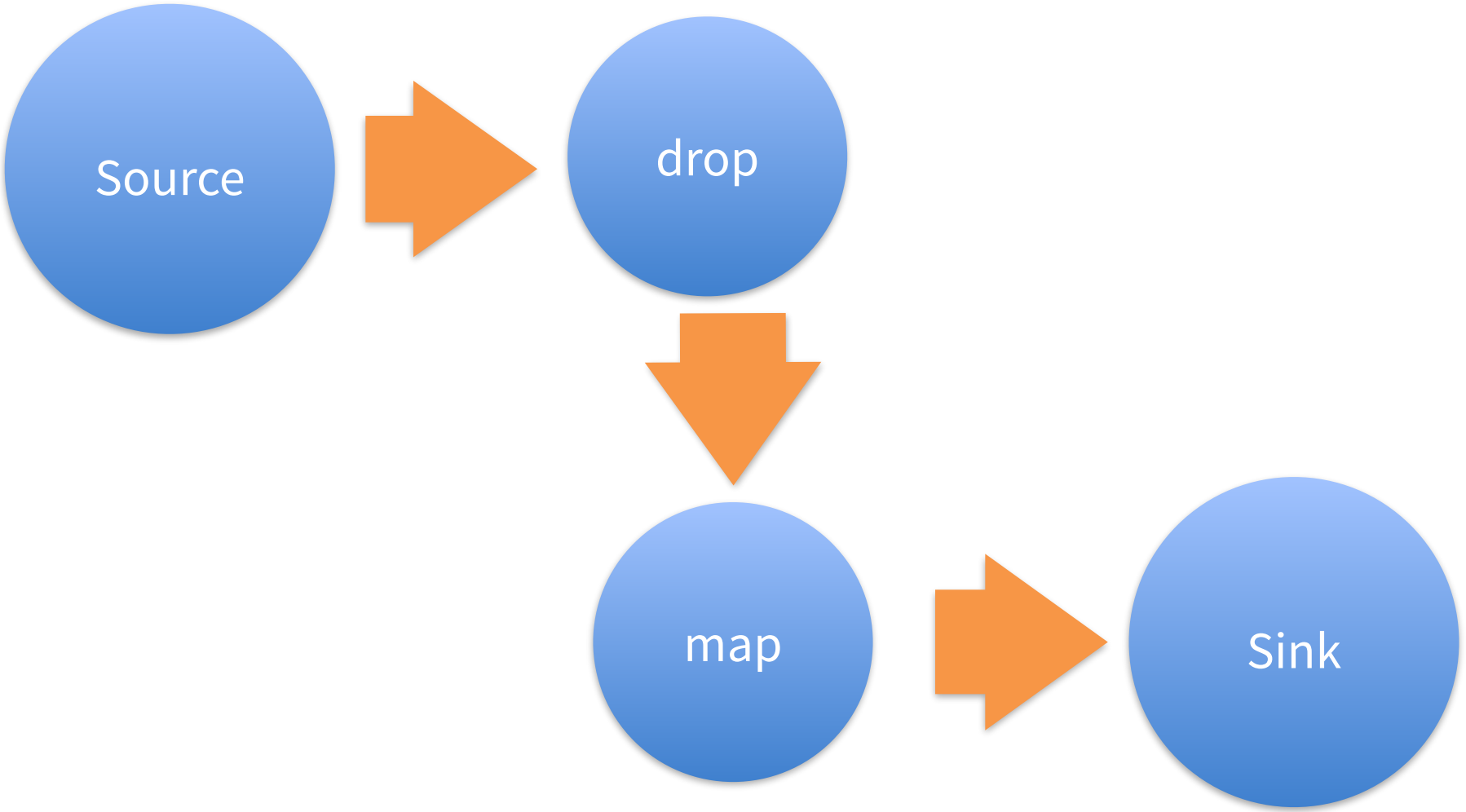
- Actors are smart
  - They can keep internal state to track demand and supply
  - They can buffer data to meet anticipated demand
- Actors are powerful
  - They can spin up child actors to meet demand
  - With Akka clustering, can spread load across multiple machines
- Actors are resilient
  - On exception, actor can be killed and restarted by supervisor
  - Actor interaction is thread-safe, and actor state is private

Actors Demo

Flows

Linear Transformations

Linear Transformations



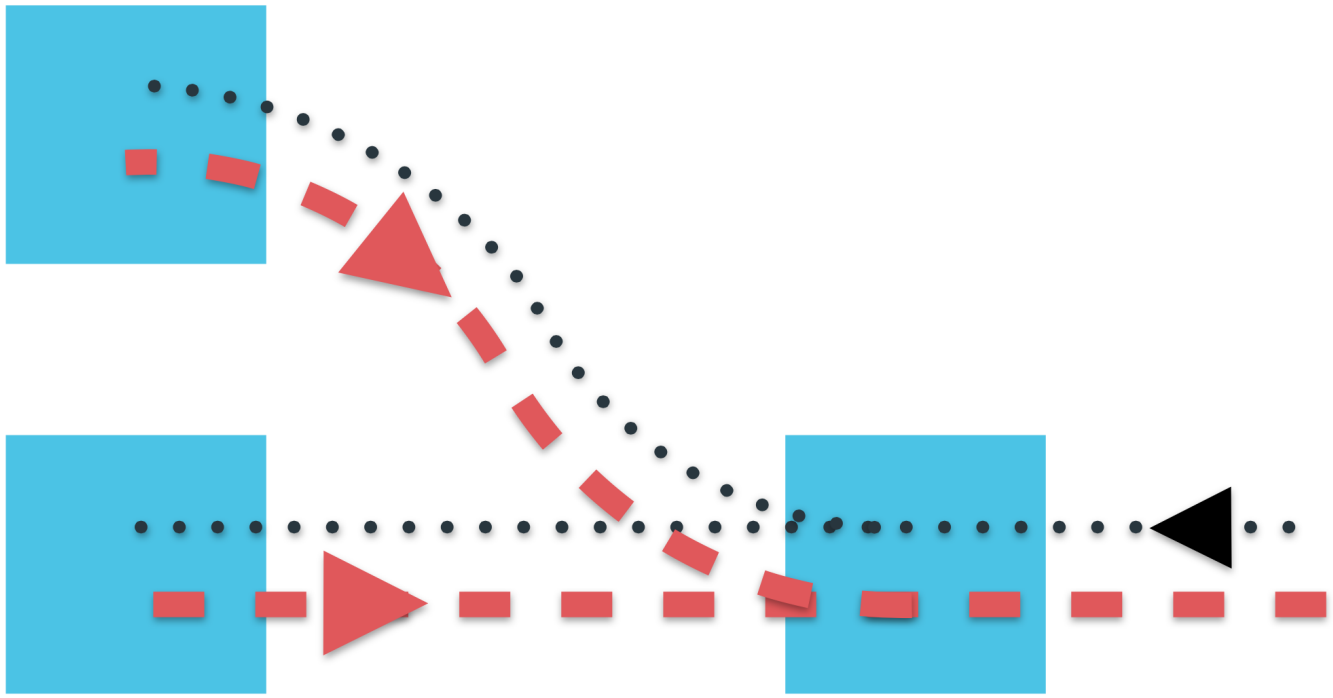
Linear Stream Transformations

- Deterministic (like for collections)
  - map, filter, collect, grouped, drop, take, groupBy, ...
- Time-Based
  - takeWithin, dropWithin, groupedWithin, ...
- Rate-Detached
  - expand, conflate, buffer, ...
- asynchronous
  - mapAsync, mapAsyncUnordered, ...

Non-linear Stream Transformations

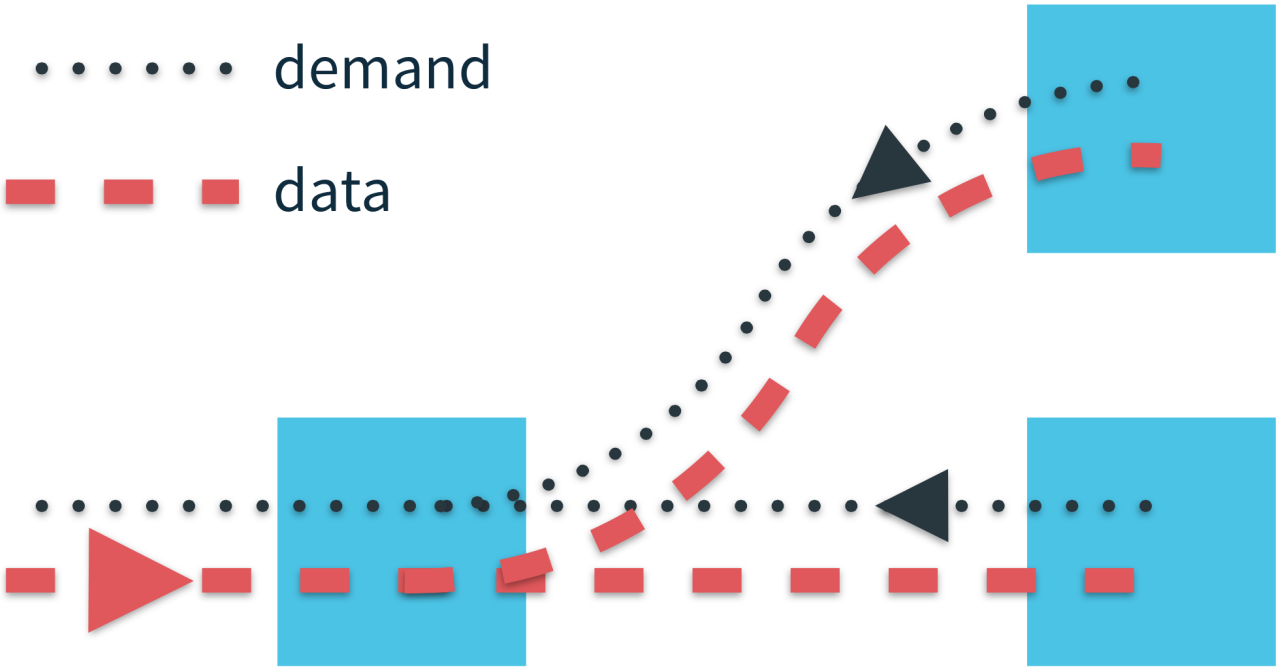
- Fan-In
  - merge, concat, zip
- Fan-Out
  - broadcast, route, unzip

Fan In



merging the data means splitting the demand

Fan Out



splitting the data means merging the demand

Materialization

- Akka Streams separate the ***what*** from the ***how***
  - declarative Source/Flow/Sink DSL to create blueprint
  - FlowMaterializer turns this into running Actors
- this allows alternative materialization strategies
  - optimization
  - verification / validation
  - cluster deployment
- only Akka Actors for now, but more to come!

FlowGraph Demo

Play

Streaming Data

- Play can stream data using Iteratees and Enumerators
- Streamed data through chunked encoding, using Ok.stream()
- But Iteratees and Enumerators are complicated.
- And Reactive Streams are simple.

Play 2.4 Experimental Features

- Reactive Streams Integration!
- Adapts Futures, Promises, Enumerators and Iteratees
- Note: potential for event loss, no performance tuning
- All access through `play.api.libs.streams.Streams`

Streaming video through Play

```
def stream = Action {
  val headers =
    Seq(CONTENT_TYPE -> "video/mp4",
        CACHE_CONTROL -> "no-cache")
  val framePublisher =
    video.FFMpeg.readFile(mp4, Akka.system)
  val frameEnumerator =
    Streams.publisherToEnumerator(framePublisher)
  val bytesEnumeratee = Enumeratee.map[Frame](encodeFrame)
  val chunkedVideoStream =
    frameEnumerator.through(bytesEnumeratee)
  Ok.stream(chunkedVideoStream).withHeaders(headers: _*)
}
```

Play Demo

Questions

Thanks

@markusjura
@lutzhuehnken