Functional Stream Processing

with Scala

Adil Akhter

The next 45 minutes

... is about **Scalaz-Stream**:

- Objectives.
- Design Philosophy.
- Core Building Blocks & their Semantics.

Scalaz-Stream

scalaz.concurrent.Task

Task[A]

- Purely functional & compositional.
- Task[A] is a wrapper around **Future[Throwable \/ A]**.
  - *Asynchronous*: uses **`scalaz.concurrent.Future[A]`**.
  - *Handles exceptions*: with **`scalaz.\/`**, aka **`Either`**.

Task[A]

- Task separates *what to do* with the *monadic context* from *when to do*.

```
scala> Task{ println("Hello World!") }
res0: scalaz.concurrent.Task[Unit] = scalaz.concurrent.Task@2435b6e8

scala> res0.run
Hello World!
```

Task[A]

- Task separates *what to do* with *monadic context* from *when to do*.

```
scala> Task{ println("Hello World!") }
res0: scalaz.concurrent.Task[Unit] = scalaz.concurrent.Task@2435b6e8
scala> res0.run
Hello World!
scala> res0.run
Hello World!
```

Task[A]

- **Example**: Lifting computation into a Task:

```
import twitter4j._
val twitterClient = new TwitterFactory(twitterConfig).getInstance()

def statuses(query: Query): Task[List[Status]] =
  Task {
    twitterClient.search(query).getTweets.toList
  }
```

Task Constructors

- `def now[A](a: A): Task[A]`
- `def fail(e: Throwable): Task[Nothing]`
- `def fork[A](a: => Task[A]): Task[A]`
- …

Scalaz-Stream

Objectives

- Incremental IO & Stream Processing.
- Modularity & Composability.
- Resource-safety.
- Performance.

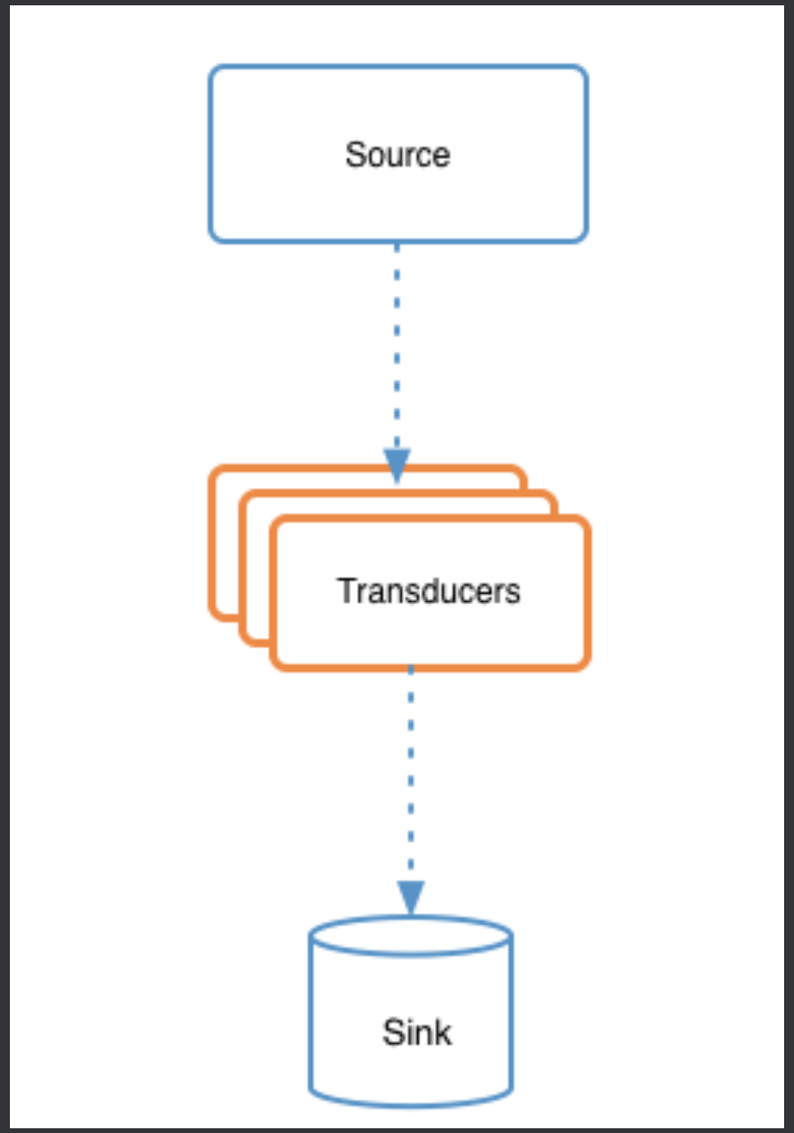
Canonical Example

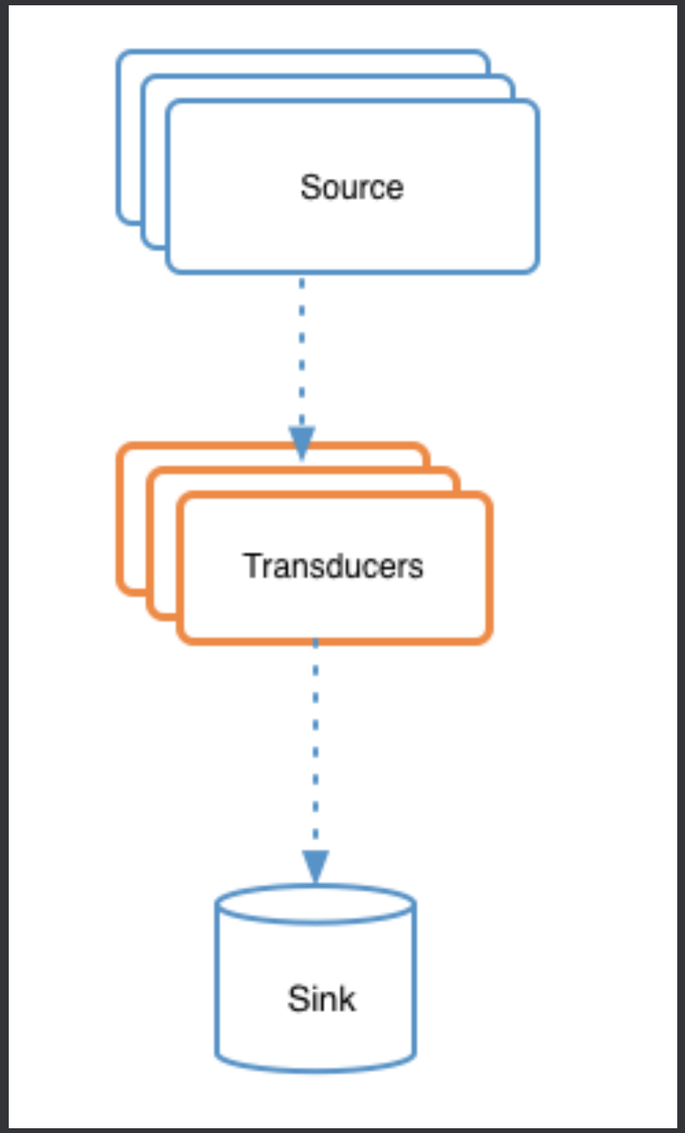
- File IO using Scalaz-Stream[1] :

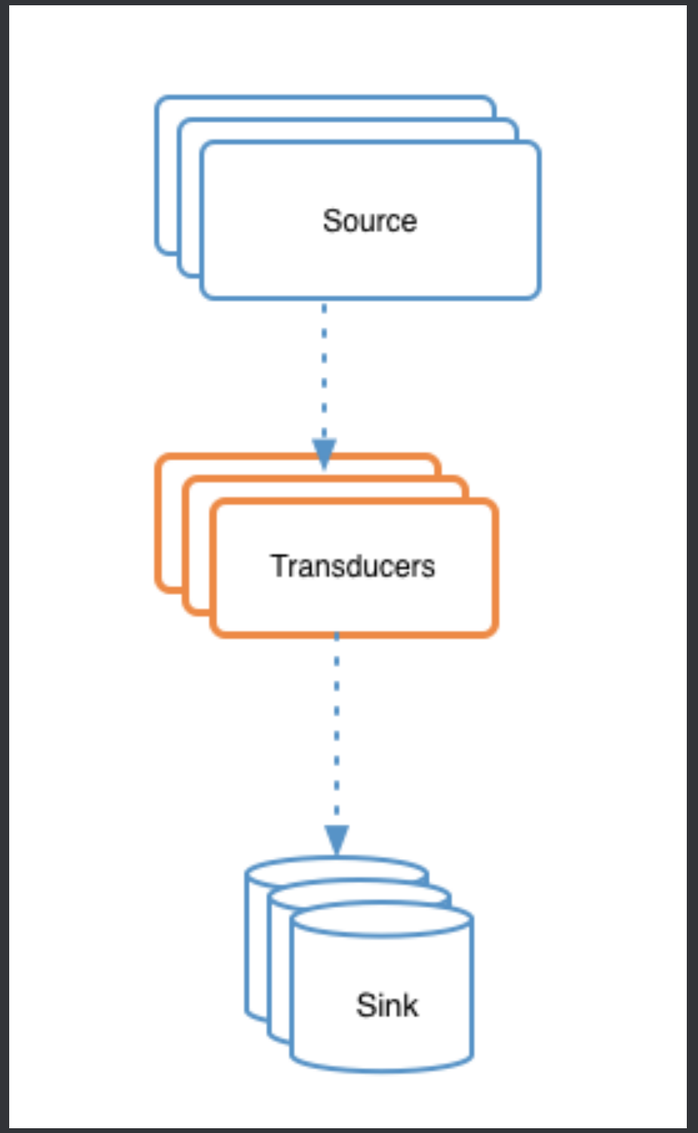
```
val streamConverter: Task[Unit] =
  io.linesR("testdata/fahrenheit.txt")
    .filter(s => !s.trim.isEmpty && !s.startsWith("//"))
    .map(line => fahrenheitToCelsius(line.toDouble).toString)
    .intersperse("\n")
    .pipe(text.utf8Encode)
    .to(io.fileChunkW("testdata/celsius.txt"))
    .run

streamConverter.run
```

[1] Source: https://github.com/functional-streams-for-scala/fs2/blob/topic/redesign/README.md


Source
Transducers
Sink


Source
Transducers
Sink


Source
Transducers
Sink

Scalaz-Stream

... provides an abstraction to declaratively specify on how to obtain stream of data.

Scalaz-Stream

... centers around one core type: **`Process`**.

Process[F, O]

... represents a **stream** of O values
which can interleave external requests
to evaluate expressions of form **F[_]**.

Process[F[_], O]

- F[_]: a context/container (e.g., **`scalaz.concurrent.Task`**)
  - effectful or not,
  - in which a computation runs.

Process[F[_], O]

- F[_]: a context/container (e.g., **`scalaz.concurrent.Task`**)
  - effectful or not,
  - in which a computation runs.
- Streams of **O**s is obtained from the context: **F**.

Process[F[_], O]

- F[_]: a context/container (e.g., **`scalaz.concurrent.Task`**)
  - effectful or not,
  - in which a computation runs.
- Streams of **O**s is obtained from the context: **F**.
- Process can accept different input types (e.g., F[_]).

Execution Model

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

import scalaz.stream._
import scalaz.concurrent.Task
import scala.concurrent.duration._

// Exiting paste mode, now interpreting.
```

Process

Process = Halt | Emit | Await

```
trait Process[F[_], O]
```

Halt

```
case class Halt(cause: Cause)
  extends Process[Nothing, Nothing]
```

```
scala> Process.halt
res19: scalaz.stream.Process0[Nothing] = Halt(End)
```

Emit[F[_],O]

```
case class Emit[F[_],O](
  head: Seq[O],
  tail: Process[F, O]) extends Process[F, O]
```

Emit[F[_],O]

```
scala> import Process._
import scalaz.stream.Process._

scala> emit(1)
res1: scalaz.stream.Process0[Int] = Emit(Vector(1))

scala> emit(1).toSource
res2: scalaz.stream.Process[scalaz.concurrent.Task,Int] = Emit(Vector(1))
```

Emit[F[_],O]

```
scala> emit(1)
res1: scalaz.stream.Process0[Int] = Emit(Vector(1))

scala> emit(1).toSource
res2: scalaz.stream.Process[scalaz.concurrent.Task,Int] = Emit(Vector(1))

scala> val p: Process[Task, Int] = emitAll(Seq(1,5,10,20))
p: scalaz.stream.Process[scalaz.concurrent.Task,Int] = Emit(List(1, 5, 10, 20))
```

Await[F[_], I, O]

```
case class Await[F[_], I, O](
  req: F[I],
  recv: I ⇒ Process[F, O],
  fallback: Process[F, O] = halt,
  cleanUp: Process[F,O] = halt) extends Process[F, O]
```

Await[F[_], I, O]

- Example: Build a Twitter Search Process to get tweets with **#spark** Tag.
- Defined earlier:

```
val twitterClient: Twitter = ???
def statuses(query: Query): Task[List[Status]] = ???
```

Await[F[_], I, O]

```
// Builds a Twitter Search Process for a given Query
def buildSearchProcess(query: Query): Process[Task, Status] = {
  val queryTask: Task[List[Status]] = statuses(query)
  await(queryTask){ statusList ⇒
    Process.emitAll(statusList)
  }
}
//In essence, it builds: Process: Await ⇒ (_ ⇒ Emit ⇒ Halt)
val akkaSearchProcess: Process[Task, Status] =
  buildSearchProcess(new Query("#spark"))
```

Await[F[_], I, O]

- Example: Build a *stream* of Positive Integers.

```
scala> import Process._
import Process._

scala> def integerStream: Process[Task, Int] = {
     |   def next (i: Int): Process[Task,Int] = await(Task(i)){ i ⇒
     |     emit(i) ++ next(i + 1)
     |   }
     |   next(1)
     | }
integerStream: scalaz.stream.Process[scalaz.concurrent.Task,Int]
```

Notable Approaches to Construct Process

- `Process.eval`
- `Process.repeatEval`
- `scalaz.stream.io` and `scalaz.stream.nio`
- `scalaz.stream.tcp`
- `scalaz.stream.async`
- …

Running a Process: run

- **`def run: F[Unit]`**: constructs the machinery that will execute the Process.
- Reduces the sequence of operations to a single operation, in the context of Task. Given:

```
scala> val intsP = integerStream.take(10)
intsP: scalaz.stream.Process[scalaz.concurrent.Task,Int] = ...

scala> val task = intsP.run
task: scalaz.concurrent.Task[Unit] = scalaz.concurrent.Task@fbce6f9

scala> task.run
// outputs nothing?
```

Running a Process: runLog

- `def runLog:F[Vector[O]]`: Gets all the intermediate results.

```
scala> intsP.runLog.run
res4: Vector[Int] = Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Running a Process: runLog

- What does happen when we execute **integerStream.runLog.run**?

```
def integerStream: Process[Task, Int] = {
  def next (i: Int): Process[Task,Int] = await(Task(i)){ i ⇒
    emit(i) ++ next(i + 1)
  }
  next(1)
}
```

Other Approaches

- `def runLast: F[Option[A]]`
- `def runFoldMap(f: A => B): F[B]`

Transformations

Transformations

```
// map
scala> integerStream.map(_ * 100).take(5).runLog.run
res5: Vector[Int] = Vector(100, 200, 300, 400, 500)
//flatMap
scala> integerStream.flatMap(i ⇒ emit(-i) ++ emit(-i-1)).take(5).runLog.run
res6: Vector[Int] = Vector(-1, -2, -2, -3, -3)
// Zip
scala> val zippedP: Process[Task, (Int, String)]
     | = integerStream.take(2) zip emitAll(Seq ("A", "B"))
scala> zippedP.runLog.run
res2: Vector[(Int, String)] = Vector((1,A), (2,B))
```

Composability

Process1[-I,+O]

- aka. Pipe
- Pure Transformations
- `Process[F, A].pipe(Process1[A, B]) => Process[F, B]`

```
scala> import process1._
scala> integerStream |> filter(i => i > 5) |> exists(_ == 10)
//res8: scalaz.stream.Process[Task,Boolean] = ...
```

Process1[-I,+O]

- `type Process1[-I,+O] = Process[Env[I,Any]#Is, O]`.
- **`process1`** object facilitates **`filter`**, **`drop`**, **`take`** and so on.

Channel[+F[_],-I,O]

- `type Channel[+F[_],-I,O] = Process[F, I => F[O]]`.
- `Process[F, A].through(Channel[Task, A, B]) => Process[Task, B]`.

Channel[+F[_],-I,O]

```
scala> val multiply10Channel: Channel[Task, Int, Int] =
     |   Process.constant { (i: Int) =>
     |     Task.now(i*10)
     |   }

scala> integerStream.take(5) through multiply10Channel
scala> res40.runLog.run
res41: Vector[Int] = Vector(10, 20, 30, 40, 50)
```

Sink[+F[_],-O]

- `Process[F, A].to(Sink[Task, A]) => Process[Task, Unit]`
- `type Sink[+F[_],-O] = Process[F, O => F[Unit]]`

Sink[+F[_],-O]

```
scala> val printInts: Sink[Task, Int] =
     |   Process.constant { (i: Int) =>
     |     Task.delay { println(i) }
     |   }

scala> val intPwithSink: Process[Task, Unit]
      | = integerStream to printInts
scala> intPwithSink.run.run
1
2
3
4
5
```

Example

Sentiment Analysis of Tweets

Source

- Building a source that queries Twitter in every 5 seconds.

```
def buildTwitterQuery(query: Query): Process1[Any, Query]
  = process1 lift { _ ⇒ query }
val source = awakeEvery(5 seconds) |> buildTwitterQuery(query)
```

Channel

- Building the *Twitter Query* Channel.

```
// defined earlier
def statuses(query: Query): Task[List[Status]] = ???

// Lift the Task to construct a channel
val queryChannel: Channel[Task, Query, List[Status]]
  = channel lift statuses
```

Channel

- Building the *Sentiment Analysis* Channel.

```
import SentimentAnalyzer._
def analyze(t: Tweet): Task[EnrichedTweet] =
  Task {
    EnrichedTweet(t.author, t.body, t.retweetCount, sentiment(t.body))
  }

val analysisChannel: Channel[Task, Tweet, EnrichedTweet] =
  channel lift analyze
```

Source |> Channel

```
def tweetsP(query: Query): Process[Task, Status] =
  source through queryChannel flatMap {
      Process emitAll _
    }
  // source = awakeEvery(5 seconds) |> buildTwitterQuery(query)
```

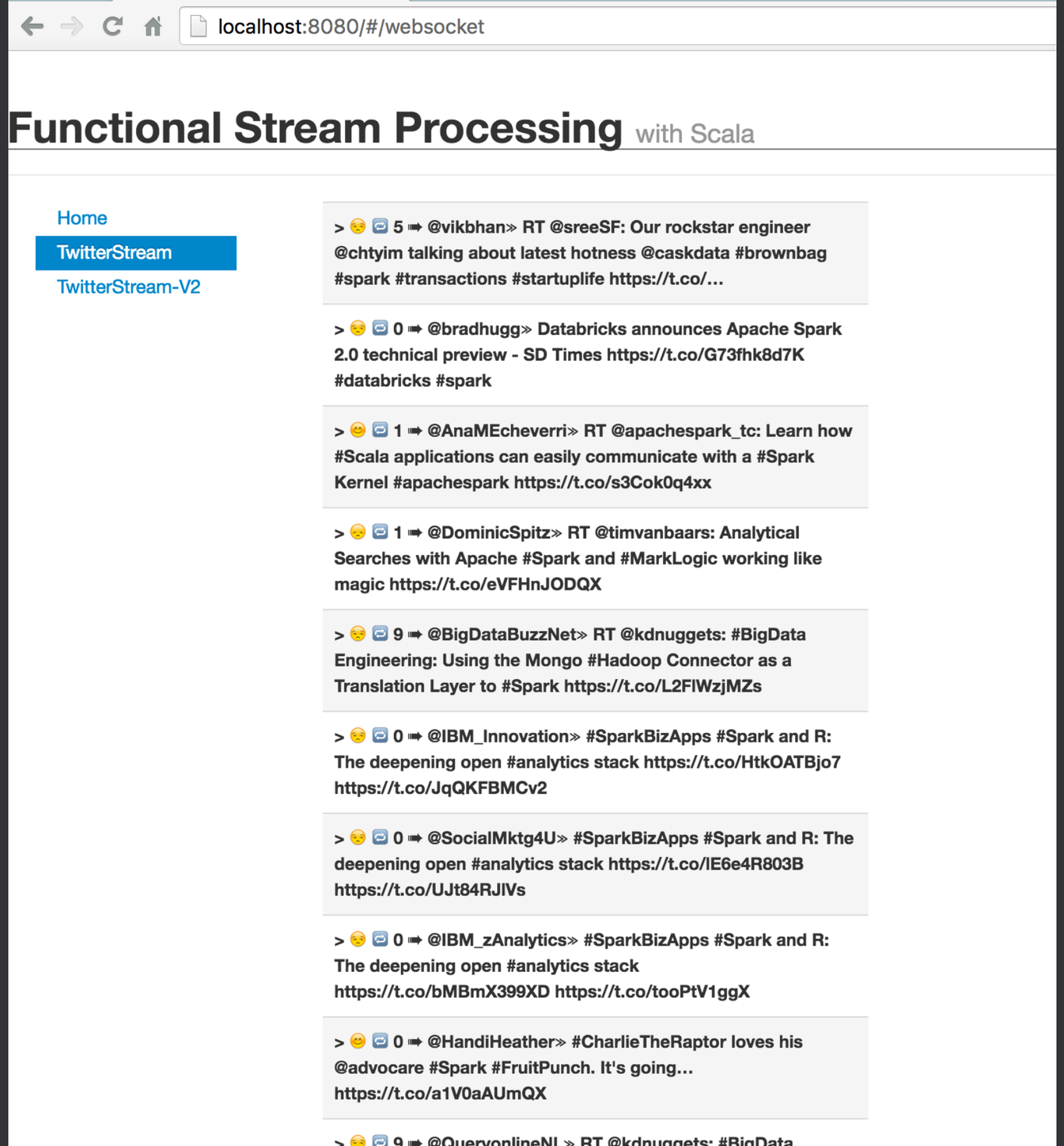
Source |> Channel

```
def twitterSource(query: Query): Process[Task, String] =
  tweetsP(query) map(status ⇒ Tweet(status))
  through analysisChannel // ... to analysis channel
  map (_.toString)
```

Source |> Channel |> Sink

```
// http4s Websocket Route
case r @ GET -> Root / "websocket" ⇒
  val query = new Query("#spark")
  val src1: Process[Task, Text] = twitterSource(query)
    .observe(io.stdOutLines)
    .map(Text(_))

  WS(Exchange(src1, Process.halt))
```


localhost:8080/#/websocket
Functional Stream Processing with Scala
Home
TwitterStream
TwitterStream-V2
> 😒 🔁 5 ➡ @vikbhan» RT @sreeSF: Our rockstar engineer @chtyim talking about latest hotness @caskdata #brownbag #spark #transactions #startuplife https://t.co/…
> 😒 🔁 0 ➡ @bradhugg» Databricks announces Apache Spark 2.0 technical preview - SD Times https://t.co/G73fhk8d7K #databricks #spark
> 😊 🔁 1 ➡ @AnaMEcheverri» RT @apachespark_tc: Learn how #Scala applications can easily communicate with a #Spark Kernel #apachespark https://t.co/s3Cok0q4xx
> 😞 🔁 1 ➡ @DominicSpitz» RT @timvanbaars: Analytical Searches with Apache #Spark and #MarkLogic working like magic https://t.co/eVFHnJODQX
> 😒 🔁 9 ➡ @BigDataBuzzNet» RT @kdnuggets: #BigData Engineering: Using the Mongo #Hadoop Connector as a Translation Layer to #Spark https://t.co/L2FlWzjMZs
> 😒 🔁 0 ➡ @IBM_Innovation» #SparkBizApps #Spark and R: The deepening open #analytics stack https://t.co/HtkOATBjo7 https://t.co/JqQKFBMCv2
> 😒 🔁 0 ➡ @SocialMktg4U» #SparkBizApps #Spark and R: The deepening open #analytics stack https://t.co/lE6e4R803B https://t.co/UJt84RJlVs
> 😒 🔁 0 ➡ @IBM_zAnalytics» #SparkBizApps #Spark and R: The deepening open #analytics stack https://t.co/bMBmX399XD https://t.co/tooPtV1ggX
> 😊 🔁 0 ➡ @HandiHeather» #CharlieTheRaptor loves his @advocare #Spark #FruitPunch. It's going… https://t.co/a1V0aAUmQX

Example

Sentiment Analysis of Tweets

using Twitter's Streaming API

async.boundedQueue

```
val twitterStream: TwitterStream = ???
val tweetsQueue: Queue[Status]
  = async.boundedQueue[Status](size)

def stream(q: Queue[Status]) = Task{
  twitterStream.
    addListener(twitterStatusListener(q))
  twitterStream.sample()
}
```

async.boundedQueue

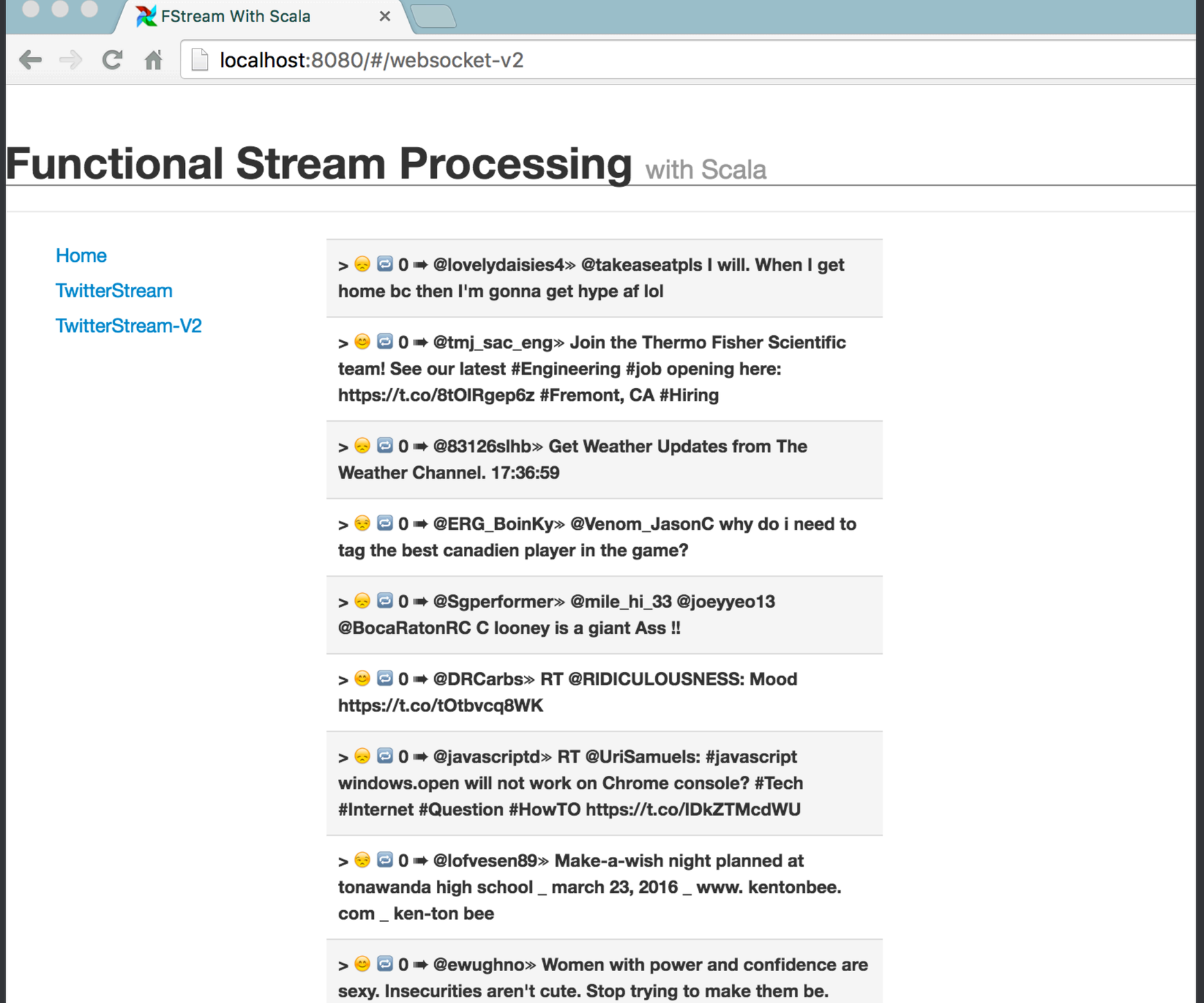
- Producer

```
Process.eval_(stream(tweetsQueue))
       .run
       .runAsync { _ ⇒ println("All input data was written") }
```

async.boundedQueue

- Consumer

```
tweetsQueue.dequeue.map(status ⇒
  Tweet(
    author = Author(status.getUser.getScreenName),
    retweetCount = status.getRetweetCount,
    body = status.getText)
  ) through analysisChannel map (_.toString)
```


FStream With Scala
localhost:8080/#/websocket-v2
Functional Stream Processing with Scala
Home
TwitterStream
TwitterStream-V2
> 😞 🔁 0 ➡ @lovelydaisies4» @takeaseatpls I will. When I get home bc then I'm gonna get hype af lol
> 😊 🔁 0 ➡ @tmj_sac_eng» Join the Thermo Fisher Scientific team! See our latest #Engineering #job opening here: https://t.co/8tOlRgep6z #Fremont, CA #Hiring
> 😞 🔁 0 ➡ @83126slhb» Get Weather Updates from The Weather Channel. 17:36:59
> 😒 🔁 0 ➡ @ERG_BoinKy» @Venom_JasonC why do i need to tag the best canadien player in the game?
> 😞 🔁 0 ➡ @Sgperformer» @mile_hi_33 @joeyyeo13 @BocaRatonRC C looney is a giant Ass !!
> 😊 🔁 0 ➡ @DRCarbs» RT @RIDICULOUSNESS: Mood https://t.co/tOtbvcq8WK
> 😞 🔁 0 ➡ @javascriptd» RT @UriSamuels: #javascript windows.open will not work on Chrome console? #Tech #Internet #Question #HowTO https://t.co/IDkZTMcdWU
> 😒 🔁 0 ➡ @lofvesen89» Make-a-wish night planned at tonawanda high school _ march 23, 2016 _ www. kentonbee. com _ ken-ton bee
> 😊 🔁 0 ➡ @ewughno» Women with power and confidence are sexy. Insecurities aren't cute. Stop trying to make them be.

Notable Constructs

Tee

- `type Tee[-I,-I2,+O] = Process[Env[I,I2]#T, O]`
  - Deterministic
  - Left and then Right and then Left ...

```
scala> import tee._
scala> import Process._
scala> val p1: Process[Task, Int] = integerStream
scala> val p2: Process[Task, String] = emitAll(Seq("A", "B", "C"))

scala> val teeExample1: Process[Nothing, Any] = (p1 tee p2)(interleave)
scala> teeExample1.runLog.run
res2: Vector[Any] = Vector(1, A, 2, B, 3, C)

scala> val teeExample2: Process[Nothing, Int] = (p1 tee p2)(passL)
scala> teeExample2.take(5).runLog.run
res3: Vector[Int] = Vector(1, 2, 3, 4, 5)
```

Wye

- `type Wye[-I,-I2,+O] = Process[Env[I,I2]#Y, O]`
  - Non-deterministic
  - Left, right or both

Next Version: FS2

- **Functional Streams for Scala**
- New algebra.
- Improved abstraction.
- Better performance.

“There are two types of

libraries: the ones people hate

and the ones nobody use"

— Unknown

Thank You

- Creadits: **Edward Kmett, Runar Bjarnasson & Paul Chuisano** et al.
- Sides and code samples will be posted at **http://github.com/adilakhter/scalaitaly-functional-stream-processing**.