DISPATCHES FROM THE TRENCHES

STREAMS IN PRODUCTION

PRESENTED BY

RAMON J. ROMERO Y VIGIL

Github Presentation:
https://github.com/RamonJRV/akkaTipsAndTricks

Linkedin:
https://www.linkedin.com/in/ramonjromeroyvigil

Stackoverflow:
https://stackoverflow.com/users/1876739/ramon-j-romero-y-vigil

THANK YOU!!!

- iHeartRadio
- Akka.io & Contributors
- Work-Bench

PURPOSE

- Good practices for Akka Streams & HTTP
  - Avoid pain
- Follow up presentation to:
  - Adam Warski: Implementing the Reactive Manifesto with Akka
  - Lance Arlaus: Intro to Akka Streams & HTTP

…THE MINIMUM NUMBER OF ACTIONS IT WILL TAKE, FOR US TO WIN THE WAR…

The Imitation Game

AKKA STREAM: SOURCE —> 0-N FLOWS —> SINK

```scala
val linesFromStdin : Source[String, _] =
  Source fromIterator io.Source.stdin.getLines

val strToIntFlow = Flow[String].map[Int](strVal => strVal.toInt)

def multInt(i : Int) = i * 2
val multIntFlow = Flow[Int] map multInt

val resultSink = Sink.seq[Int]

implicit val actorSystem = akka.actor.ActorSystem("StreamIntro")
implicit val actorMaterializer = akka.stream.ActorMaterializer()
import actorSystem.dispatcher


val seqFut : Future[Seq[Int]] = linesFromStdin.via(strToIntFlow)
                                              .via(multIntFlow)
                                              .runWith(resultSink)



seqFut onSuccess { case seq =>
  println(s"Sequence is: $seq")
}
```

AKKA STREAM: SOURCE —>

```
val linesFromStdin : Source[String, _] =
  Source fromIterator io.Source.stdin.getLines
```

- Many Publisher Types for a Source
  - Iterators
  - Iterables
  - Actors
  - Files
  - Ports

—> 0-N FLOWS —>

```
        In        Out
Flow[String].map[Int](strVal => strVal.toInt)

            Flow[Int] map multInt
```

```
INPUT --> Flow --> Output
```

```
val resultSink = Sink.seq[Int]
```

- Many Subscriber Types
  - Sequence
  - foreach
  - Ignore

AKKA STREAMS

- Streams
    - Concurrency via Actors
    - Backpressure
    - Composition

AKKA STREAM : MATERIALIZATION

```
implicit val actorSystem = ActorSystem("StreamIntro")
implicit val actorMaterializer = ActorMaterializer()

val seqFut : Future[Seq[Int]] =
  linesFromStdin.via(strToIntFlow)
                .via(multIntFlow)
                .runWith(resultSink)
```

- Each stream element has it’s own Actor
- Demand is propagated from the sink
  - back pressure
- Streams “materialize” into values

AKKA HTTP

```
//  HttpRequest  -->  Route  -->  HttpResponse
val httpHandler : Route =
  (get & path("/mult" / Segment)) { (intAsStr : String) =>
    val intVal = intAsStr.toInt
    complete(HttpResponse(entity = multInt(intVal).toString))
  }


//The entire server
Http().bindAndHandle(httpHandler, "localhost", 80)


//client code
val reqVal = 24

val resp : Future[HttpResponse] =
  Http().singleRequest(HttpRequest(uri="/mult", entity=s"$reqVal"))
```

AKKA HTTP : ROUTE

```
//  HttpRequest  -->  Route  -->  HttpResponse
val httpHandler : Route =
  (get & path("/mult" / Segment)) { (intAsStr : String) =>
    val intVal = intAsStr.toInt
    complete(HttpResponse(entity = multInt(intVal).toString))
  }
```

- Like grep for HttpRequest objects

AKKA HTTP : ROUTE

```
val httpHandler : Route =
  (get & path("/mult" / Segment)) { (intAsStr : String) =>
    val intVal = intAsStr.toInt
    complete(HttpResponse(entity = multInt(intVal).toString))
  }
```

- Like grep for HttpRequest objects
  - Only match Get Requests

```
val httpHandler : Route =
  (get & path("/mult" / Segment)) { (intAsStr : String) =>
    val intVal = intAsStr.toInt
    complete(HttpResponse(entity = multInt(intVal).toString))
  }
```

- Like grep for HttpRequest objects
  - Only match Get Requests
  - AND Paths that look like “/mult/123”

```
val httpHandler : Route =
  (get & path("/mult" / Segment)) { (intAsStr : String) =>
    val intVal = intAsStr.toInt
    complete(HttpResponse(entity = multInt(intVal).toString))
  }
```

- Like grep for HttpRequest objects
  - Only match Get Reqs
  - AND Paths that look like “/mult/123”

- Complete with an answer

AKKA HTTP : SERVER

```
//  HttpRequest  -->  Route  -->  HttpResponse
val httpHandler : Route =
  (get & path("/mult" / Segment)) { (intAsStr : String) =>
    val intVal = intAsStr.toInt
    complete(HttpResponse(entity = multInt(intVal).toString))
  }


//The entire server
Http().bindAndHandle(httpHandler, "localhost", 80)


//client code
val reqVal = 24

val resp : Future[HttpResponse] =
  Http().singleRequest(HttpRequest(uri="/mult", entity=s"$reqVal"))
```

```
//The entire server
Http().bindAndHandle(httpHandler, "localhost", 80)
```

- A server is a Source[Connection, _]
    - A Connection is a Source[HttpRequest, _]
- Server takes a Flow[HttpRequest, HttpResponse]
    - materialized for each Connection

AKKA HTTP : CLIENT

```
val resp : Future[HttpResponse] =
  Http().singleRequest(HttpRequest(uri=s”/mult/$reqVal"))
```

AKKA STREAMS AND HTTP

- Streams
  - Concurrency via Actors
  - Backpressure
  - Composition
- Http
  - Servers as Streams

SEPARATE BUSINESS LOGIC FROM AKKA

- Makes unit testing & debugging easier
- Allows for different concurrency models
- Cleaner code

MIXED LOGIC

```scala
val closed = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
  import GraphDSL.Implicits._

  type FileInputType = (Int, Array[String])

  …

  //Flow body contains business logic
  val filterFileInputs = Flow[FileInputType] filter {
    case (r, s) => {
      println(s"sink ${(r >= 3)} $r")
      r >= 3
    }
  }

  //Even the structure has business logic
  fileSource ~> merge ~> afterMerge ~> broadcast ~> filterFileInputs ~> ignore
                merge <~  toRetry    <~ broadcast
  ClosedShape
})
```

MIXED LOGIC : BUZ LOGIC INSIDE AKKA FLOW

```
val closed = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
  import GraphDSL.Implicits._

  type FileInputType = (Int, Array[String])

  …

  //Flow body contains business logic
  val filterFileInputs = Flow[FileInputType] filter {
    case (r, s) => {
      println(s"sink ${(r >= 3)} $r")
      r >= 3
    }
  }

  //Even the structure has business logic
  fileSource ~> merge ~> afterMerge ~> broadcast ~> filterFileInputs ~> ignore
                merge <~  toRetry   <~ broadcast
  ClosedShape
})
```

MIXED LOGIC

```
val closed = RunnableGraph.fromGraph(GraphDSL.create() { implicit builder =>
  import GraphDSL.Implicits._

  type FileInputType = (Int, Array[String])

  …

  //Flow body contains business logic
  val filterFileInputs = Flow[FileInputType] filter {
    case (r, s) => {
      println(s"sink ${(r >= 3)} $r")
      r >= 3
    }
  }

  //Even the structure has business logic
  fileSource ~> merge ~> afterMerge ~> broadcast ~> filterFileInputs ~> ignore
                merge <~  toRetry   <~ broadcast
  ClosedShape
})
```

MIXED LOGIC : GRAPHS ARE VISUAL STREAMS

```
fileSource ~> merge ~> afterMerge ~> broadcast ~> filterFileInputs ~> ignore
              merge <~  toRetry   <~ broadcast
```

```
~> merge ~> afterMerge ~> broadcast ~> filterFileInputs
   merge <~  toRetry    <~ broadcast
```

- Visual recursion

SEPARATE LOGIC

```scala
object SeperateBizLogic {
  type FileInputType = (Int, Array[String])
  val emptyInputType = (0, Array.empty[String])

  @scala.annotation.tailrec
  def recursiveRetry(fileInput : FileInputType) : FileInputType =
    fileInput match {
      case (r,_) if r >= 3  => fileInput
      case (r,a)            => recursiveRetry((r+1, a))
    }
}


object AkkaBizExtension {
  import SeperateBizLogic._

  val stream = Source.single(emptyInputType)
                     .via(Flow[FileInputType] map recursiveRetry)
                     .to(Sink.ignore)
}
```

SEPARATE LOGIC : CLEAN LOGIC WITH NO CONCURRENCY LIBRARY

```
object SeperateBizLogic {
  type FileInputType = (Int, Array[String])
  val emptyInputType = (0, Array.empty[String])

  @scala.annotation.tailrec
  def recursiveRetry(fileInput : FileInputType) : FileInputType =
    fileInput match {
      case (r,_) if r >= 3  => fileInput
      case (r,a)            => recursiveRetry((r+1, a))
    }
}


object AkkaBizExtension {
  import SeperateBizLogic._

  val stream = Source.single(emptyInputType)
                     .via(Flow[FileInputType] map recursiveRetry)
                     .to(Sink.ignore)
}
```

SEPARATE LOGIC : CLEAN STREAMS WITH NO “GUTS”

```scala
object SeperateBizLogic {
  type FileInputType = (Int, Array[String])
  val emptyInputType = (0, Array.empty[String])

  @scala.annotation.tailrec
  def recursiveRetry(fileInput : FileInputType) : FileInputType =
    fileInput match {
      case (r,_) if r >= 3  => fileInput
      case (r,a)            => recursiveRetry((r+1, a))
    }
}


object AkkaBizExtension {
  import SeperateBizLogic._

  val stream = Source.single(emptyInputType)
                     .via(Flow[FileInputType] map recursiveRetry)
                     .to(Sink.ignore)
}
```

SEPARATE LOGIC : CLEAN STREAMS WITH NO “GUTS”

```
Source.single(emptyInputType)
      .via(Flow[FileInputType] map recursiveRetry)
      .runWith(Sink.ignore)
```

- Clean implementation
- Not much to test/debug

CONSIDER FUTURES FIRST

- They can look like Streams
- No back-pressure performance hit
- Composition without the verbiage

FUTURES CAN LOOK LIKE STREAMS

```
type LoginId = String
type UniqueId = java.util.UUID

def dbLookupLoginToUniqueId(loginId : LoginId) : Future[UniqueId] = ???

def authenticatorLookupIdActive(uniqueId : UniqueId,
                                date : Date) : Future[Boolean] = ???


def loginWasActive(loginId : LoginId, date : Date) : Future[Boolean] =
  for {
    uniqueId  <- dbLookupLoginToUniqueId(loginId)
    wasActive <- authenticatorLookupIdActive(uniqueId, date)
  } yield wasActive
```

FUTURES CAN LOOK LIKE STREAMS

```
type LoginId = String
type UniqueId = java.util.UUID

def dbLookupLoginToUniqueId(loginId : LoginId) : Future[UniqueId] = ???

def authenticatorLookupIdActive(uniqueId : UniqueId,
                                date : Date) : Future[Boolean] = ???


def loginWasActive(loginId : LoginId, date : Date) : Future[Boolean] =
  for {
    uniqueId  <- dbLookupLoginToUniqueId(loginId)
    wasActive <- authenticatorLookupIdActive(uniqueId, date)
  } yield wasActive
```

FOR COMPREHENSION: “TIME AFTER TIME” - CYNDI LAUPER

```
for {
   uniqueId  <- dbLookupLoginToUniqueId(loginId)
   wasActive <- authenticatorLookupIdActive(uniqueId, date)
 } yield wasActive
```

FOR COMPREHENSION: SOURCE —>

```
for {
   uniqueId  <- dbLookupLoginToUniqueId(loginId)
   wasActive <- authenticatorLookupIdActive(uniqueId, date)
 } yield wasActive
```

FOR COMPREHENSION: —> 0-N FLOWS —>

```
for {
   uniqueId  <- dbLookupLoginToUniqueId(loginId)
   wasActive <- authenticatorLookupIdActive(uniqueId, date)
 } yield wasActive
```

FOR COMPREHENSION: —> SINK

```
for {
   uniqueId  <- dbLookupLoginToUniqueId(loginId)
   wasActive <- authenticatorLookupIdActive(uniqueId, date)
 } yield wasActive
```

FOR COMPREHENSION: SOURCE —> 0-N FLOWS —> SINK

```
for {
   uniqueId  <- dbLookupLoginToUniqueId(loginId)
   wasActive <- authenticatorLookupIdActive(uniqueId, date)
 } yield wasActive
```

- Monads are fun!!!

MANY FUTURES DISTILLED INTO A SINGLE FUTURE

```
val allIds : Iterable[LoginId] = ???

val someDate : Date = ???

def loginWasActiveOnDate(loginId : LoginId) : Future[Boolean] =
  loginWasActive(loginId,someDate)


// Iterable(Future[Boolean], Future[Boolean], Future[Boolean], ...)

val allIdsActive : Iterable[Future[Boolean]] =
  allIds map loginWasActiveOnDate


// Future[Iterable(Boolean, Boolean, Boolean, ...)

val idsAreActive : Future[Iterable[Boolean]] =
  Future sequence allIdsActive
```

MANY FUTURES DISTILLED INTO A SINGLE FUTURE

```
val allIdsActive : Iterable[Future[Boolean]] =
  allIds map loginWasActiveOnDate

val idsAreActive : Future[Iterable[Boolean]] =
  Future sequence allIdsActive
```

Future.sequence

```
Iterable[Future[Boolean]]

Future[Iterable[Boolean]]
```

STATE IS POSSIBLE IN STREAMS

- Streams come with a lot of functionality
- Read the documentation

Source : Foo Bar Foo Baz

Result : Map()

Map(Foo -> 1)

Map(Foo -> 1, Bar -> 1)

Map(Foo -> 2, Bar -> 1)

Map(Foo -> 2, Bar -> 1, Baz -> 1)

HOW CAN WE KEEP STATE?

Source : Foo Bar Foo Baz

Result : Map()

Map(Foo -> 1)

Map(Foo -> 1, Bar -> 1)

Map(Foo -> 2, Bar -> 1)

Map(Foo -> 2, Bar -> 1, Baz -> 1)

RUNNING STATE

```
object BizLogic {
  type Word = String
  type Count = Int

  type WordCounter = immutable.Map[Word, Count]
  val emptyCounter : WordCounter = immutable.Map.empty[Word, Count]

  // Increments a running counter for the inputed book.
  def incrementCounter(counter : WordCounter, word : Word) : WordCounter =
    counter.updated(word, counter.getOrElse(word, 0) + 1)
}


object StreamState {
  import WordCounter._

  //  Word  ->  flowCounter  ->  WordCounter
  val flowCounter : Flow[Word, WordCounter, _] =
    Flow[Word].scan(emptyCounter)(incrementCounter)
}
```

RUNNING STATE

```
object BizLogic {
  type Word = String
  type Count = Int

  type WordCounter = immutable.Map[Word, Count]
  val emptyCounter : WordCounter = immutable.Map.empty[Word, Count]

  // Increments a running counter for the inputed book.
               updater                accumulator              newVal
  def incrementCounter(counter : WordCounter, word : Word) :
WordCounter =
    counter.updated(word, counter.getOrElse(word, 0) + 1)
}


object StreamState {
  import WordCounter._

  //  Word  ->  flowCounter  ->  WordCounter
  val flowCounter : Flow[Word, WordCounter, _] =
    Flow[Word].scan(emptyCounter)(incrementCounter)
}
```

FLOW.SCAN

```
//  Word  ->  flowCounter  ->  WordCounter
val flowCounter : Flow[Word, WordCounter, _] =
  Flow[Word].scan(emptyCounter)(incrementCounter)
```

- scan
  - keeps the most recently returned value
  - calls the updater function on (accum, update)

FLOW.SCAN

```
val flowCounter : Flow[Word, WordCounter, _] =
  Flow[Word].scan(emptyCounter)(incrementCounter)
```

- scan
  - keeps the most recently returned value
  - calls the updater function on (accum, newVal)
  - starts with the “zero” argument

FLOW.SCAN

```
//  Word  ->  flowCounter  ->  WordCounter
val flowCounter : Flow[Word, WordCounter, _] =
  Flow[Word].scan(emptyCounter)(incrementCounter)
```

- scan
    - keeps the most recently returned value
    - calls the updater function on (accum, newVal)
    - starts with the “zero” argument
    - forwards the updated accumulator

```
val flowCounter : Flow[Word, WordCounter, _] =
  Flow[Word].scan(emptyCounter)(incrementCounter)
```

- scan
  - keeps the most recently returned value
  - calls the updater function on (accum, newVal)
  - starts with the “zero” argument
  - forwards the updated accumulator

EXHAUSTIVE STATE

```
public class FlatmapExample {

  static int maxSize = 1024;

  static final ByteString delim = ByteString.fromString("\r\n");

  static final Flow<String, String, NotUsed> pathsToContents =
    Flow.of(String.class)
        .flatMapConcat(path -> FileIO.fromFile(new File(path))
                                     .via(Framing.delimiter(delim, maxSize))
                                     .map(byteStr -> byteStr.utf8String()));

}
```

EXHAUSTIVE STATE: PATH —> SOURCE[FILE CONTENTS]

```
public class FlatmapExample {

  static int maxSize = 1024;

  static final ByteString delim = ByteString.fromString("\r\n");

  static final Flow<String, String, NotUsed> pathsToContents =
    Flow.of(String.class)
 .flatMapConcat(path -> FileIO.fromFile(new File(path))
                              .via(Framing.delimiter(delim, maxSize))
                              .map(byteStr -> byteStr.utf8String()));

}
```

EXHAUSTIVE STATE: RINSE & REPEAT

```
public class FlatmapExample {

  static int maxSize = 1024;

  static final ByteString delim = ByteString.fromString("\r\n");

  static final Flow<String, String, NotUsed> pathsToContents =
    Flow.of(String.class)
 .flatMapConcat(path -> FileIO.fromFile(new File(path))
                             .via(Framing.delimiter(delim, maxSize))
                             .map(byteStr -> byteStr.utf8String()));

}
```

MIX IN THE APPROPRIATE CONCURRENCY MODULE

- Ok to switch between Actors, Futures, Streams, Routes
- Right tool for the right job

CAN ALWAYS GO BACK TO ACTORS

```scala
class RequestHandlerActor extends Actor {
  override def receive = {
    case _ : HttpRequest =>
      sender() ! HttpResponse(entity = "actor responds nicely")
  }
}

object MixActorsWithRoutes {

  def internalError(ex : Throwable) =
    complete((InternalServerError, s"Actor not playing nice: ${ex.getMessage}"))

  def actorRoute(requestRef : ActorRef)(implicit timeout : Timeout) : Route =
    extractRequest { request =>
      onComplete((requestRef ? request).mapTo[HttpResponse]) {
        case Success(response) => complete(response)
        case Failure(ex)       => internalError(ex)
      }
    }

}
```

CAN ALWAYS GO BACK TO ACTORS

```
class RequestHandlerActor extends Actor {
  override def receive = {
    case _ : HttpRequest =>
      sender() ! HttpResponse(entity = "actor responds nicely")
  }
}
```

```
object MixActorsWithRoutes {

  def internalError(ex : Throwable) =
    complete((InternalServerError, s"Actor not playing nice: ${ex.getMessage}"))

  def actorRoute(requestRef : ActorRef)(implicit timeout : Timeout) : Route =
    extractRequest { request =>
      onComplete((requestRef ? request).mapTo[HttpResponse]) {
        case Success(response) => complete(response)
        case Failure(ex)       => internalError(ex)
      }
    }

}
```

SCALA & AKKA ARE THE WAY FORWARD FOR MULTI CORE/BOX

- Concurrency can be made easier
- Functional Programming & Akka do so elegantly