Akka Stream Recap – Code

```
// part 1 - actor system
// heavy data structure that allocate some threads under the hood
// which allocated to actors in order to run
implicit val system = ActorSystem("MyActorSystem")
```

מאפשר הרצה של רכבים מהספרייה של Akka

```
implicit val materializer = ActorMaterializer
```

יש צורך לעשות import לספרייה הבאה:

```
import akka.stream.scaladsl._
```

דוגמה לגרף:

```
val source = Source(1 to 100)
val flow = Flow[Int].map { x => x.toString } // Int is the type the Flow
processed
val flow2 = Flow[String].map { x => x } // String is the type the Flow
processed
val sink = Sink.foreach[String](println)
val sink2 = Sink.fold[String,String]("")(_ + _)
val graph = source.via(flow).via(flow2).to(sink2)
graph.run()
```

זה שקול ל:

```
val graph = source.map(x => x.toString).map(x => x).to(sink2)
```

דוגמה לשימוש ב-Materializer Value:

```
  val source = Source(1 to 5)
  val flow = Flow[Int].map { x => x.toString } // Int is the type the Flow
processed
  val flow2 = Flow[String].map { x => x } // String is the type the Flow
processed
  val sink2 = Sink.fold[String,String]("")(_ + " " +  _)
  val g = source
    .viaMat(flow)(Keep.right)
    .viaMat(flow2)(Keep.right)
    .toMat(sink2)(Keep.right)
  val sum = g.run()
  sum.onComplete {
    case Success(value) => println(s"the value is $value")
    case Failure(ex) => ()
  }
//Outputs: the value is  1 2 3 4 5
```

דרך Actors: (אם לא משתמשים ב-mapAsyn או asyn אז כל התהליך מתבצע בעזרת Actor **אחד**)

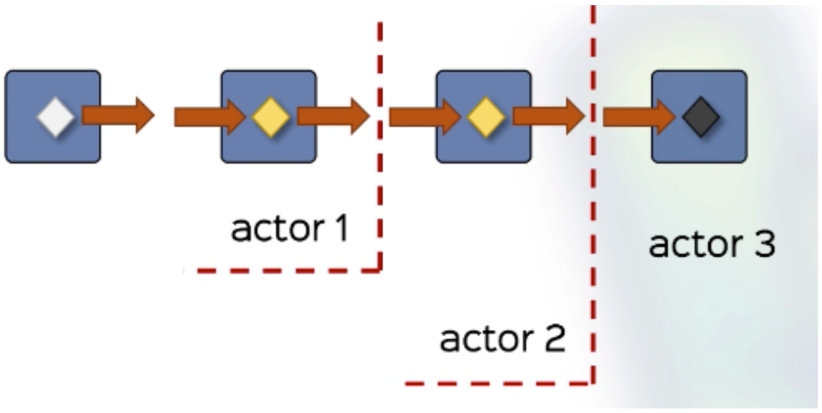
```
val source = Source(1 to 5)
val flow1 = Flow[Int].map( _ + 1 ) // Int is the type the Flow processed
val flow2 = Flow[Int].map( _ * 10 )
val sink = Sink.foreach[Int](println)
source.via(flow1).via(flow2).to(sink).run()

//part 2 - create actors
// kind of human that talk with each other that talk through messages
// and then they process and then replay or not replay back
// (*) actors are uniquely identified
// (*) messages are asynchronous
// (*) each actor may respond differently
// (*) you cant acces to actor data, communication by messages
class MyActor extends Actor {
  override def receive: Receive = {
    case x: Int =>
```

```
      // flow operations
      val flow1 = x + 1
      val flow2 = flow1 * 10
      // sink operations
    println(flow2)
  }
}
val myActor = system.actorOf(Props[MyActor],"my-actor")
(1 to 5).foreach( myActor ! _ )
```

זה לא טוב אם הרכיבים לוקחים הרבה זמן
ולכן הפתרון הוא:

```
source
  .via(flow1).async // Run on one actor
  .via(flow2).async // Run on second actor
  .to(sink)
  .run()
```

כך זה נראה:



סדר הגעה מובטח:

```
Source(1 to 3)
  .map( ele => {println(s"flow1: $ele"); ele})
  .map( ele => {println(s"flow2: $ele"); ele})
  .map( ele => {println(s"flow3: $ele"); ele})
  .runWith(Sink.ignore)
```

פלט:

```
flow1: 1
flow2: 1
flow3: 1
flow1: 2
flow2: 2
flow3: 2
flow1: 3
flow2: 3
flow3: 3
```

סדר הגעה לא מובטח אבל באותו רכיב הסדר מובטח

```
Source(1 to 3)
  .map( ele => {println(s"flow1: $ele"); ele}).async
  .map( ele => {println(s"flow2: $ele"); ele}).async
  .map( ele => {println(s"flow3: $ele"); ele}).async
  .runWith(Sink.ignore)
```

פלט:

```
flow1: 1
flow1: 2
flow2: 1
flow2: 2
```

```
flow3: 1
flow3: 2
flow1: 3
flow2: 3
flow3: 3
```

Backpressure

זה לא Backpressure כי זה מתבצע באותו Actor

```scala
val fastConsumer = Source(Stream.from(1))
val slowProducer = Sink.foreach[Int](ele => {
  Thread.sleep(1000)
  println(ele)
})
fastConsumer.runWith(slowProducer)
```

לעומת זאת זה כן:

```scala
fastConsumer.async.runWith(slowProducer)
```

עוד דוגמא:

```scala
val fastConsumer = Source(1 to 100)
val slowProducer = Sink.foreach[Int](ele => {
  Thread.sleep(1000)
  println(s"Sink got $ele")
})
val flow = Flow[Int].map { ele =>
  println(s"Incoming: $ele");
  ele
}
fastConsumer.async
  .via(flow).async
  .to(slowProducer).run()
```

פלט:

```
Incoming: 1
Incoming: 2
Incoming: 3
Incoming: 4
Incoming: 5
Incoming: 6
Incoming: 7
Incoming: 8
Incoming: 9
Incoming: 10
Incoming: 11
Incoming: 12
Incoming: 13
Incoming: 14
Incoming: 15
Incoming: 16
Sink got 1
Sink got 2
Sink got 3
Sink got 4
Sink got 5
Sink got 6
Sink got 7
Incoming: 17
…
```

בגלל שה-Sink איטי הוא מסמן ל-Flow אז במקום לשלוח את ההודעות ל-Sink הוא עושה Buffer להודעות כאשר ה-Buffer הדיפולטיבי מוגדר להיות 16 ואז הוא נותן לעוד הודעות לעבור בהמשך

הסדר פעולות ל-Backpressure (לפי הסדר):

1. לנסות להאט את הקצב אם אפשרי
2. לעשות Buffer לאיברים (אם יש Flow כמו בדוגמה השנייה זה לא אפשרי)
3. לזרוק איברים אם ה-Buffer עובר על הערך שמוגדר לו
4. לזרוק שגיאה Failure

אנחנו בתור מתכנים יכולים להגיע עד שלב (3.)

דוגמה למימוש Buffer (מסתמך על הקוד הקודם) – המימוש של Akka זה עם OverflowStrrategy.backpressure

```
val bufferedFlow = flow.buffer(10, OverflowStrategy.dropHead)
fastConsumer.async
  .via(bufferedFlow).async
  .to(slowProducer).run()
```

```
Incoming: 1
Incoming: 2
Incoming: 3
…
Incoming: 99
Incoming: 100
Sink got 1
Sink got 2
Sink got 3
…
Sink got 16
Sink got 91
Sink got 92
…
```

להעביר מספר איברים כל כמה יחידות זמן

```
import scala.concurrent.duration._
Source(1 to 100).throttle(5, 1 seconds)
.to(Sink.foreach[Int](println)).run()
```

GraphDSL:

```
val source = Source(1 to 100)
val sink = Sink.foreach[(Int,Int)](println)
val flow1 = Flow[Int].map { x => x}
val flow2 = Flow[Int].map { x => x}
val graph = RunnableGraph.fromGraph(
  GraphDSL.create() { implicit builder: GraphDSL.Builder[NotUsed] =>
    import GraphDSL.Implicits._

    // components
    val broadcast = builder.add(Broadcast[Int](2)) // fan-out operator (2
outputs 1 input)
    val zip = builder.add(Zip[Int, Int]) // fan-in operator (1 outputs 2
input)

    source ~> broadcast

    broadcast.out(0) ~> flow1 ~> zip.in0
    broadcast.out(1) ~> flow2 ~> zip.in1

    zip.out ~> sink
    ClosedShape
    // shape
  } // graph
) // runnable graph
  .run()
```

פלט:

(1,1)
...
(100,100)

Actor כ-Source:

```
val sourceActor: Source[Int, ActorRef] = Source.actorRef[Int](10,
OverflowStrategy.dropHead)
val sink = Sink.foreach[Int](println)
val materializedVRef: ActorRef = sourceActor.to(sink).run()
materializedVRef ! 10
// termination the stream
materializedVRef ! akka.actor.Status.Success("Complete")
```

Actor כ-Flow:

```

class MyActor extends Actor {
  override def receive: Receive = {
    case s:String => sender() ! s + s
    case i: Int => sender() ! i*2
  }
}
val myActor = system.actorOf(Props[MyActor],"my-actor")
val source = Source(1 to 100)
// actor as flow
import scala.concurrent.duration._
implicit val timeout = Timeout(2 seconds)
val actorFlow = Flow[Int].ask[Int](4)(myActor)

source.via(actorFlow).to(Sink.foreach(println)).run()
```

או לחלופין:

```
source.ask[Int](4)(myActor).to(Sink.foreach(println)).run()
```

Actor כ-Sink:

```
class SinkActor extends Actor {
  override def receive: Receive = {
    case StreamInit =>
      println("started...")
      sender() ! StreamAck
    case StreamAck =>
    case StreamComplete =>
      println("stream complete")
      context.stop(self)
    case StreamFail =>
          println("warning ....!!! log error")
    case message =>
      println(s"got message: $message")
      sender() ! StreamAck
  }
}
val sinkActor = system.actorOf(Props[SinkActor],"sinkActor")
val sink = Sink.actorRefWithAck[Int](
  sinkActor,
  onInitMessage = StreamInit,
  onCompleteMessage = StreamComplete,
  ackMessage = StreamAck ,
  onFailureMessage = StreamFail
)
Source(1 to 100).to(sink).run()
```

אפשריות לטיפול בשגיאות:

```

/** (1) - logging */
val sourceWithException = Source(1 to 10).map { n =>
  if (n == 8) throw new Exception("n == 8")
  else n
}
sourceWithException.log("trackingElements").to(Sink.ignore).run()
/* Result:
all the upstream (from sink to source) is canceled
all the downstream is processed
will do the stream and then will throw exception and stop the stream
 */

/** (2) - gracefully terminating a stream */
sourceWithException.recover {
  case _: Exception => Int.MinValue
}
  // Result after the recover the stream will be stopped
  .log("trackingElements").to(Sink.foreach[Int](println)).run()
// will do the stream and then will stop the stream
// 1, 2, ..., 7, -214..

/** (3) - recover with another stream */
sourceWithException.recoverWithRetries(3, {
  case _ => Source(20 to 30)
  // if source was sourceWithException then its will do [1,7]*3 and then
  // will throw an exception
}).log("recoverWithRetries").to(Sink.foreach[Int](println)).run()

/** (4) - backoff supervision  */
// withBackoff triggered on a complete or failure will restart
// onFailureBackoff will do only on failure

import scala.concurrent.duration._

RestartSource.onFailuresWithBackoff(
  1 seconds,
  30 seconds,
  0.2) { () =>
  val random = Random.nextInt(20)
  // Source with 50% change to failure
  Source(1 to 10).map { x => if (x == random) throw new Exception else x }
}.to(Sink.foreach[Int](println)).run()

/** (5)  - supervision strategy */
sourceWithException.withAttributes(ActorAttributes.supervisionStrategy {
  case _: Exception => Supervision.Resume
  // resume - skip the faulty element
  // stop - stop the streat
  // restart - clearly internal state + resume
}).to(Sink.foreach[Int](println)).run() // Will drop 8 and continue

sourceWithException

.to(Sink.foreach[Int](println)).withAttributes(ActorAttributes.withSupervis
ionStrategy {
  case _: Exception => Supervision.Resume
}).run // same result */

val flow = Flow[Int].map { x =>
  println(s"flow got x = $x");
  if (x == 9) throw new Exception
  else
    x
}
sourceWithException // Wil throw exception on 8
  .via(flow) // will not handle x = 8 and will print till sink got x = 7
  .to(Sink.foreach[Int](x => println(s"sink got x = $x")))
  .run()
// But if:
sourceWithException // Wil throw exception on 8
  .via(flow) // will not handle x = 8 and will print till sink got x = 7
and then
  // from x = 9 until 10
  .to(Sink.foreach[Int](x => println(s"sink got x = $x")))
  .withAttributes(ActorAttributes.supervisionStrategy { case _ => Resume })
  .run()
```

עוד דוגמה:

```
implicit val system = ActorSystem("MyActorSystem")
implicit val materializer = ActorMaterializer

val flow = Flow[Int].map { x =>
  println(x)
  if (x == 9) throw new Exception
  else
    x
}
import scala.concurrent.duration._
val sourceWithException = RestartSource.withBackoff(7 seconds, 30 seconds,
0.2)( () => Source(1 to 35)).map { n =>
  if (n == 8) throw new Exception("n == 8")
  else n
}


sourceWithException.async
  .via(flow).async
  .to(Sink.foreach[Int](x => { Thread.sleep(1000); println(s"sink got x =
$x") }))
  .withAttributes(ActorAttributes.supervisionStrategy { case _ => Resume })
  .run()

// Result:
1
2
3
```

```
4
5
6
7
9
10
11
12
13
14
15
16
17
18
sink got x = 1
sink got x = 2
sink got x = 3
sink got x = 4
sink got x = 5
sink got x = 6
sink got x = 7
19
20
21
22
23
24
25
26
sink got x = 10
sink got x = 11
sink got x = 12
sink got x = 13
sink got x = 14
sink got x = 15
sink got x = 16
sink got x = 17
27
28
29
30
31
32
33
34
sink got x = 18
sink got x = 19
sink got x = 20
sink got x = 21
sink got x = 22
sink got x = 23
sink got x = 24
sink got x = 25
35
1
2
3
```

```
4
5
6
7
sink got x = 26
```

KillSwitch

```
val sharedKilledSwitch = KillSwitches.shared("redButton")
// source 1
Source(Stream.from(1)).via(sharedKilledSwitch.flow).to(Sink.foreach[Int](pr
intln)).run()
// source 2
Source(Stream.from(2)).via(sharedKilledSwitch.flow).to(Sink.foreach[Int](pr
intln)).run()

import scala.concurrent.duration._
system.scheduler.scheduleOnce(3 seconds) {
  sharedKilledSwitch.shutdown()
```