Quick Start Guide

Quick Start Guide

A stream usually begins at a source, so this is also how we start an Akka Stream. Before we create one, we import the full complement of streaming tools:

```
1. import akka.stream._
2. import akka.stream.scaladsl._
```

If you want to execute the code samples while you read through the quick start guide, you will also need the following imports:

```
1. import akka.{ NotUsed, Done }
2. import akka.actor.ActorSystem
3. import akka.util.ByteString
4. import scala.concurrent._
5. import scala.concurrent.duration._
6. import java.nio.file.Paths
```

Now we will start with a rather simple source, emitting the integers 1 to 100:

```
1. val source: Source[Int, NotUsed] = Source(1 to 100)
```

The `Source` type is parameterized with two types: the first one is the type of element that this source emits and the second one may signal that running the source produces some auxiliary value (e.g. a network source may provide information about the bound port or the peer's address). Where no auxiliary information is produced, the type `akka.NotUsed` is used—and a simple range of integers surely falls into this category.

Having created this source means that we have a description of how to emit the first 100 natural numbers, but this source is not yet active. In order to get those numbers out we have to run it:

```
1. source.runForeach(i => println(i))(materializer)
```

This line will complement the source with a consumer function—in this example we simply print out the numbers to the console—and pass this little stream setup to an Actor that runs it. This activation is signaled by having "run" be part of the method name; there are other methods that run Akka Streams, and they all follow this pattern.

You may wonder where the Actor gets created that runs the stream, and you are probably also asking yourself what this `materializer` means. In order to get this value we first need to create an Actor system:

```
1. implicit val system = ActorSystem("QuickStart")
2. implicit val materializer = ActorMaterializer()
```

There are other ways to create a materializer, e.g. from an `ActorContext` when using streams from within Actors. The `Materializer` is a factory for stream execution engines, it is the thing that makes streams run—you don't need to worry about any of the details just now apart from that you need one for calling any of the `run` methods on a `Source`. The materializer is picked up implicitly if it is omitted from the run method call arguments, which we will do in the

Contents

```
val factorials = source.scan(BigInt(1))((acc, next) => acc * next)

val result: Future[IOResult] =
  factorials
    .map(num => ByteString(s"$num\n"))
    .runWith(FileIO.toPath(Paths.get("factorials.txt")))
```

First we use the `scan` combinator to run a computation over the whole stream: starting with the number 1 (`BigInt(1)`) we multiple by each of the incoming numbers, one after the other; the scan operation emits the initial value and then every calculation result. This yields the series of factorial numbers which we stash away as a `Source` for later reuse—it is important to keep in mind that nothing is actually computed yet, this is just a description of what we want to have computed once we run the stream. Then we convert the resulting series of numbers into a stream of `ByteString` objects describing lines in a text file. This stream is then run by attaching a file as the receiver of the data. In the terminology of Akka Streams this is called a `Sink`. `IOResult` is a type that IO operations return in Akka Streams in order to tell you how many bytes or elements were consumed and whether the stream terminated normally or exceptionally.

Reusable Pieces

One of the nice parts of Akka Streams—and something that other stream libraries do not offer—is that not only sources can be reused like blueprints, all other elements can be as well. We can take the file-writing `Sink`, prepend the processing steps necessary to get the `ByteString` elements from incoming strings and package that up as a reusable piece as well. Since the language for writing these streams always flows from left to right (just like plain English), we need a starting point that is like a source but with an "open" input. In Akka Streams this is called a `Flow`:

```
def lineSink(filename: String): Sink[String, Future[IOResult]] =
  Flow[String]
    .map(s => ByteString(s + "\n"))
    .toMat(FileIO.toPath(Paths.get(filename)))(Keep.right)
```

Starting from a flow of strings we convert each to `ByteString` and then feed to the already known file-writing `Sink`. The resulting blueprint is a `Sink[String, Future[IOResult]]`, which means that it accepts strings as its input and when materialized it will create auxiliary information of type `Future[IOResult]` (when chaining operations on a `Source` or `Flow` the type of the auxiliary information—called the "materialized value"—is given by the leftmost starting point; since we want to retain what the `FileIO.toFile` sink has to offer, we need to say `Keep.right`).

We can use the new and shiny `Sink` we just created by attaching it to our `factorials` source—after a small adaptation to turn the numbers into strings:

```
factorials.map(_.toString).runWith(lineSink("factorial2.txt"))
```

Time-Based Processing

Before we start looking at a more involved example we explore the streaming nature of what Akka Streams can do. Starting from the `factorials` source we transform the stream by zipping it together with another stream, represented by a `Source` that emits the number 0 to 100: the first number emitted by the `factorials` source is the factorial of zero, the second is the factorial of one, and so on. We combine these two by forming strings like `"3! = 6"`.

```
val done: Future[Done] =
  factorials
    .zipWith(Source(0 to 100))((num, idx) => s"$idx! = $num")
    .throttle(1, 1.second, 1, ThrottleMode.shaping)
    .runForeach(println)
```

All operations so far have been time-independent and could have been performed in the same fashion on strict collections of elements. The next line demonstrates that we are in fact dealing with streams that can flow at a certain

does not crash with an OutOfMemoryError, even though you will also notice that running the streams happens in the background, asynchronously (this is the reason for the auxiliary information to be provided as a `Future`). The secret that makes this work is that Akka Streams implicitly implement pervasive flow control, all combinators respect back-pressure. This allows the throttle combinator to signal to all its upstream sources of data that it can only accept elements at a certain rate—when the incoming rate is higher than one per second the throttle combinator will assert *back-pressure* upstream.

This is basically all there is to Akka Streams in a nutshell—glossing over the fact that there are dozens of sources and sinks and many more stream transformation combinators to choose from, see also *Overview of built-in stages and their semantics*.

Reactive Tweets

A typical use case for stream processing is consuming a live stream of data that we want to extract or aggregate some other data from. In this example we'll consider consuming a stream of tweets and extracting information concerning Akka from them.

We will also consider the problem inherent to all non-blocking streaming solutions: *"What if the subscriber is too slow to consume the live stream of data?"*. Traditionally the solution is often to buffer the elements, but this can—and usually will—cause eventual buffer overflows and instability of such systems. Instead Akka Streams depend on internal backpressure signals that allow to control what should happen in such scenarios.

Here's the data model we'll be working with throughout the quickstart examples:

```
1. final case class Author(handle: String)
2.
3. final case class Hashtag(name: String)
4.
5. final case class Tweet(author: Author, timestamp: Long, body: String) {
6.   def hashtags: Set[Hashtag] =
7.     body.split(" ").collect { case t if t.startsWith("#") => Hashtag(t) }.toSet
8. }
9.
10. val akka = Hashtag("#akka")
```

Note

If you would like to get an overview of the used vocabulary first instead of diving head-first into an actual example you can have a look at the *Core concepts* and *Defining and running streams* sections of the docs, and then come back to this quickstart to see it all pieced together into a simple example application.

Transforming and consuming simple streams

The example application we will be looking at is a simple Twitter feed stream from which we'll want to extract certain information, like for example finding all twitter handles of users who tweet about `#akka`.

In order to prepare our environment by creating an `ActorSystem` and `ActorMaterializer`, which will be responsible for materializing and running the streams we are about to create:

```
1. implicit val system = ActorSystem("reactive-tweets")
2. implicit val materializer = ActorMaterializer()
```

The `ActorMaterializer` can optionally take `ActorMaterializerSettings` which can be used to define materialization properties, such as default buffer sizes (see also *Buffers for asynchronous stages*), the dispatcher to be used by the pipeline etc. These can be overridden with `withAttributes` on `Flow`, `Source`, `Sink` and `Graph`.

Let's assume we have a stream of tweets readily available. In Akka this is expressed as a `Source[Out, M]`:

```
1. val tweets: Source[Tweet, NotUsed]
```

The operations should look familiar to anyone who has used the Scala Collections library, however they operate on streams and not collections of data (which is a very important distinction, as some operations only make sense in streaming and vice versa):

```
1. val authors: Source[Author, NotUsed] =
2.   tweets
3.     .filter(_.hashtags.contains(akka))
4.     .map(_.author)
```

Finally in order to *materialize* and run the stream computation we need to attach the Flow to a `Sink` that will get the Flow running. The simplest way to do this is to call `runWith(sink)` on a `Source`. For convenience a number of common Sinks are predefined and collected as methods on the `Sink` companion object. For now let's simply print each author:

```
1. authors.runWith(Sink.foreach(println))
```

or by using the shorthand version (which are defined only for the most popular Sinks such as `Sink.fold` and `Sink.foreach`):

```
1. authors.runForeach(println)
```

Materializing and running a stream always requires a `Materializer` to be in implicit scope (or passed in explicitly, like this: `.run(materializer)`).

The complete snippet looks like this:

```
1. implicit val system = ActorSystem("reactive-tweets")
2. implicit val materializer = ActorMaterializer()
3.
4. val authors: Source[Author, NotUsed] =
5.   tweets
6.     .filter(_.hashtags.contains(akka))
7.     .map(_.author)
8.
9. authors.runWith(Sink.foreach(println))
```

Flattening sequences in streams

In the previous section we were working on 1:1 relationships of elements which is the most common case, but sometimes we might want to map from one element to a number of elements and receive a "flattened" stream, similarly like `flatMap` works on Scala Collections. In order to get a flattened stream of hashtags from our stream of tweets we can use the `mapConcat` combinator:

```
1. val hashtags: Source[Hashtag, NotUsed] = tweets.mapConcat(_.hashtags.toList)
```

Note

The name `flatMap` was consciously avoided due to its proximity with for-comprehensions and monadic composition. It is problematic for two reasons: first, flattening by concatenation is often undesirable in bounded stream processing due to the risk of deadlock (with merge being the preferred strategy), and second, the monad laws would not hold for our implementation of flatMap (due to the liveness issues).

Please note that the `mapConcat` requires the supplied function to return a strict collection (`f:Out=>immutable.Seq[T]`), whereas `flatMap` would have to operate on streams all the way through.

source stream into two streams which will handle the writing to these different files.

Elements that can be used to form such "fan-out" (or "fan-in") structures are referred to as "junctions" in Akka Streams. One of these that we'll be using in this example is called `Broadcast`, and it simply emits elements from its input port to all of its output ports.

Akka Streams intentionally separate the linear stream structures (Flows) from the non-linear, branching ones (Graphs) in order to offer the most convenient API for both of these cases. Graphs can express arbitrarily complex stream setups at the expense of not reading as familiarly as collection transformations.

Graphs are constructed using `GraphDSL` like this:

```
val writeAuthors: Sink[Author, Unit] = ???
val writeHashtags: Sink[Hashtag, Unit] = ???
val g = RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  import GraphDSL.Implicits._

  val bcast = b.add(Broadcast[Tweet](2))
  tweets ~> bcast.in
  bcast.out(0) ~> Flow[Tweet].map(_.author) ~> writeAuthors
  bcast.out(1) ~> Flow[Tweet].mapConcat(_.hashtags.toList) ~> writeHashtags
  ClosedShape
})
g.run()
```

As you can see, inside the `GraphDSL` we use an implicit graph builder `b` to mutably construct the graph using the `~>` "edge operator" (also read as "connect" or "via" or "to"). The operator is provided implicitly by importing `GraphDSL.Implicits._`.

`GraphDSL.create` returns a `Graph`, in this example a `Graph[ClosedShape, Unit]` where `ClosedShape` means that it is *a fully connected graph* or "closed" - there are no unconnected inputs or outputs. Since it is closed it is possible to transform the graph into a `RunnableGraph` using `RunnableGraph.fromGraph`. The runnable graph can then be `run()` to materialize a stream out of it.

Both `Graph` and `RunnableGraph` are *immutable, thread-safe, and freely shareable*.

A graph can also have one of several other shapes, with one or more unconnected ports. Having unconnected ports expresses a graph that is a *partial graph*. Concepts around composing and nesting graphs in large structures are explained in detail in *Modularity, Composition and Hierarchy*. It is also possible to wrap complex computation graphs as Flows, Sinks or Sources, which will be explained in detail in *Constructing Sources, Sinks and Flows from Partial Graphs*.

Back-pressure in action

One of the main advantages of Akka Streams is that they *always* propagate back-pressure information from stream Sinks (Subscribers) to their Sources (Publishers). It is not an optional feature, and is enabled at all times. To learn more about the back-pressure protocol used by Akka Streams and all other Reactive Streams compatible implementations read *Back-pressure explained*.

A typical problem applications (not using Akka Streams) like this often face is that they are unable to process the incoming data fast enough, either temporarily or by design, and will start buffering incoming data until there's no more space to buffer, resulting in either `OutOfMemoryError` s or other severe degradations of service responsiveness. With Akka Streams buffering can and must be handled explicitly. For example, if we are only interested in the "*most recent tweets, with a buffer of 10 elements*" this can be expressed using the `buffer` element:

```
tweets
  .buffer(10, OverflowStrategy.dropHead)
  .map(slowComputation)
  .runWith(Sink.ignore)
```

The `buffer` element takes an explicit and required `OverflowStrategy`, which defines how the buffer should react

So far we've been only processing data using Flows and consuming it into some kind of external Sink - be it by printing values or storing them in some external system. However sometimes we may be interested in some value that can be obtained from the materialized processing pipeline. For example, we want to know how many tweets we have processed. While this question is not as obvious to give an answer to in case of an infinite stream of tweets (one way to answer this question in a streaming setting would be to create a stream of counts described as "*up until now*, we've processed N tweets"), but in general it is possible to deal with finite streams and come up with a nice result such as a total count of elements.

First, let's write such an element counter using `Sink.fold` and see how the types look like:

```
val count: Flow[Tweet, Int, NotUsed] = Flow[Tweet].map(_ => 1)

val sumSink: Sink[Int, Future[Int]] = Sink.fold[Int, Int](0)(_ + _)

val counterGraph: RunnableGraph[Future[Int]] =
  tweets
    .via(count)
    .toMat(sumSink)(Keep.right)

val sum: Future[Int] = counterGraph.run()

sum.foreach(c => println(s"Total tweets processed: $c"))
```

First we prepare a reusable `Flow` that will change each incoming tweet into an integer of value `1`. We'll use this in order to combine those with a `Sink.fold` that will sum all `Int` elements of the stream and make its result available as a `Future[Int]`. Next we connect the `tweets` stream to `count` with `via`. Finally we connect the Flow to the previously prepared Sink using `toMat`.

Remember those mysterious `Mat` type parameters on `Source[+Out, +Mat]`, `Flow[-In, +Out, +Mat]` and `Sink[-In, +Mat]`? They represent the type of values these processing parts return when materialized. When you chain these together, you can explicitly combine their materialized values. In our example we used the `Keep.right` predefined function, which tells the implementation to only care about the materialized type of the stage currently appended to the right. The materialized type of `sumSink` is `Future[Int]` and because of using `Keep.right`, the resulting `RunnableGraph` has also a type parameter of `Future[Int]`.

This step does *not* yet materialize the processing pipeline, it merely prepares the description of the Flow, which is now connected to a Sink, and therefore can be `run()`, as indicated by its type: `RunnableGraph[Future[Int]]`. Next we call `run()` which uses the implicit `ActorMaterializer` to materialize and run the Flow. The value returned by calling `run()` on a `RunnableGraph[T]` is of type `T`. In our case this type is `Future[Int]` which, when completed, will contain the total length of our `tweets` stream. In case of the stream failing, this future would complete with a Failure.

A `RunnableGraph` may be reused and materialized multiple times, because it is just the "blueprint" of the stream. This means that if we materialize a stream, for example one that consumes a live stream of tweets within a minute, the materialized values for those two materializations will be different, as illustrated by this example:

```
val sumSink = Sink.fold[Int, Int](0)(_ + _)
val counterRunnableGraph: RunnableGraph[Future[Int]] =
  tweetsInMinuteFromNow
    .filter(_.hashtags contains akka)
    .map(t => 1)
    .toMat(sumSink)(Keep.right)

// materialize the stream once in the morning
val morningTweetsCount: Future[Int] = counterRunnableGraph.run()
// and once in the evening, reusing the flow
val eveningTweetsCount: Future[Int] = counterRunnableGraph.run()
```

Many elements in Akka Streams provide materialized values which can be used for obtaining either results of computation or steering these elements which will be discussed in detail in *Stream Materialization*. Summing up this

Note

`runWith()` is a convenience method that automatically ignores the materialized value of any other stages except those appended by the `runWith()` itself. In the above example it translates to using `Keep.right` as the combiner for materialized values.