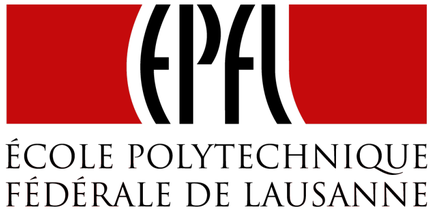


Akka Streams

Programming Reactive Systems

Konrad Malawski, Julien Richard-Foy

Akka Streams

Akka Streams provides a *high-level API* for streams processing.

Akka Streams *implements the Reactive Streams protocol* on all of its layers.

Canonical Example

```
import akka.actor._ // untyped Actor System
import akka.stream.scaladsl.{ Source, Flow, Sink }
implicit val system = ActorSystem()
implicit val mat = ActorMaterializer()

val eventuallyResult: Future[Int] =
  Source(1 to 10)
    .map(_ * 2)
    .runFold(0)((acc, x) => acc + x)
```

Canonical Example

```
import akka.actor._ // untyped Actor System
import akka.stream.scaladsl.{ Source, Flow, Sink }
implicit val system = ActorSystem()
implicit val mat = ActorMaterializer()

val eventuallyResult: Future[Int] =
  Source(1 to 10)
    .map(_ * 2)
    .runWith(
      Sink.fold(0)((acc: Int, x: Int) => acc + x)
    )
```

Canonical Example (modularity + composition)

```
import akka.actor._ // untyped Actor System
import akka.stream.scaladsl.{ Source, Flow, Sink }
implicit val system = ActorSystem()
implicit val mat = ActorMaterializer()

val numbers  = Source(1 to 10)
val doubling = Flow.fromFunction((x: Int) => x * 2)
val sum      = Sink.fold(0)((acc: Int, x: Int) => acc + x)

val eventuallyResult: Future[Int] =
  numbers.via(doubling).runWith(sum)
```

Shapes of processing stages

In Akka Streams the "steps" of the processing pipeline (the Graph), are referred to as *stages*.

The term *operator* is used for the fluent DSL's API, such as `map`, `filter` etc.

The term "Stage" is more general, as it also means various fan-in and fan-out shapes.

Akka Stream's main shapes we will be dealing with are:

- Source – has exactly 1 output
- Flow – has exactly 1 input, and 1 output
- Sink – has exactly 1 input

Akka Streams - Reactive Streams correspondence

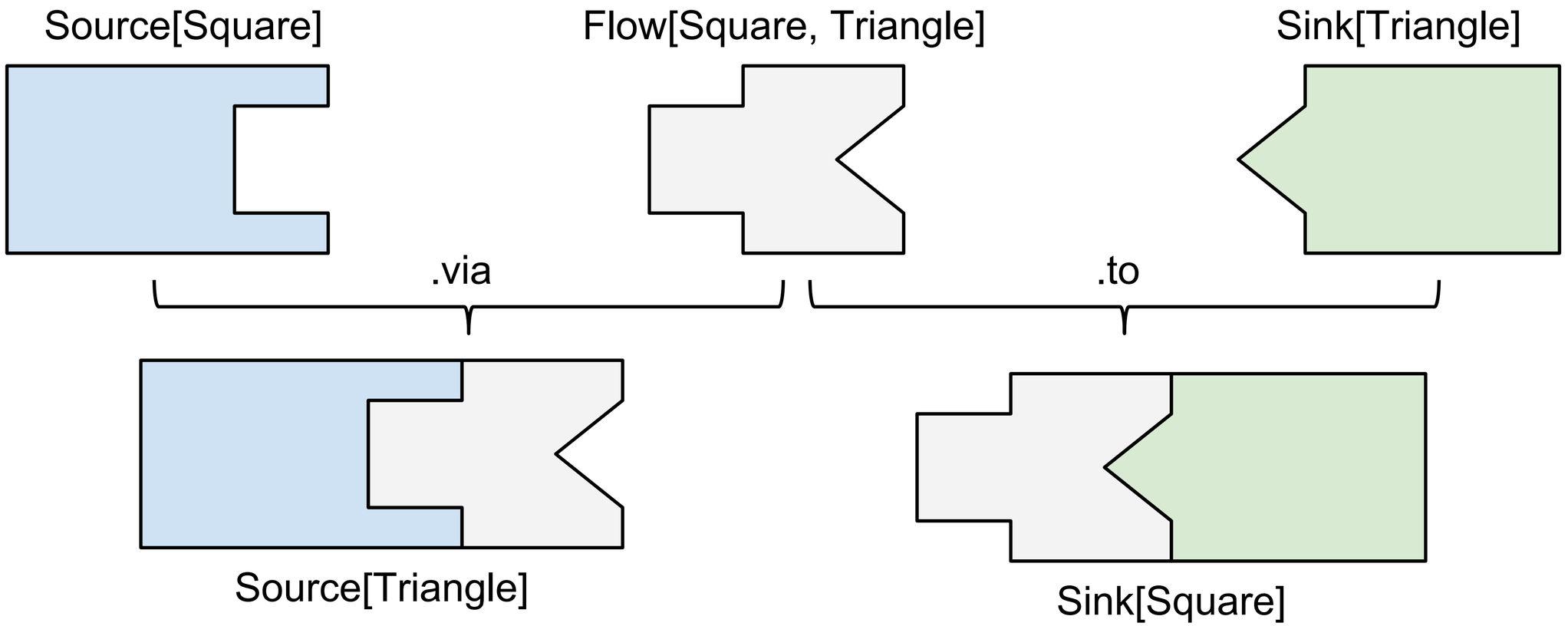
Akka Streams implement the Reactive Streams protocol, that means all stages adhere to the semantics we learnt about in the previous video.

Since Reactive Streams is an SPI, Akka Streams hides the raw SPI types from end-users (unless asked to expose them).

In general though there is a 1:1 correspondence:

- `Source[O, _]` (AS) is equivalent to `Publisher[O]` (RS)
- `Sink[I, _]` (AS) is equivalent to `Subscriber[I]` (RS)
- `Flow[I, O, _]` (AS) is equivalent to `Processor[I, O]` (RS)

Composing Stages



Materializing (running) streams

"Running a stream" is performed by "materializing" it.

Materialization can be explained as the Sources, Flows and Sinks being the "description" of *what* needs to be done, and by materializing it we pass this description to an execution engine – the `Materializer`.

By default, we will be using the `ActorMaterializer`.

Operator API similarity to Scala collections

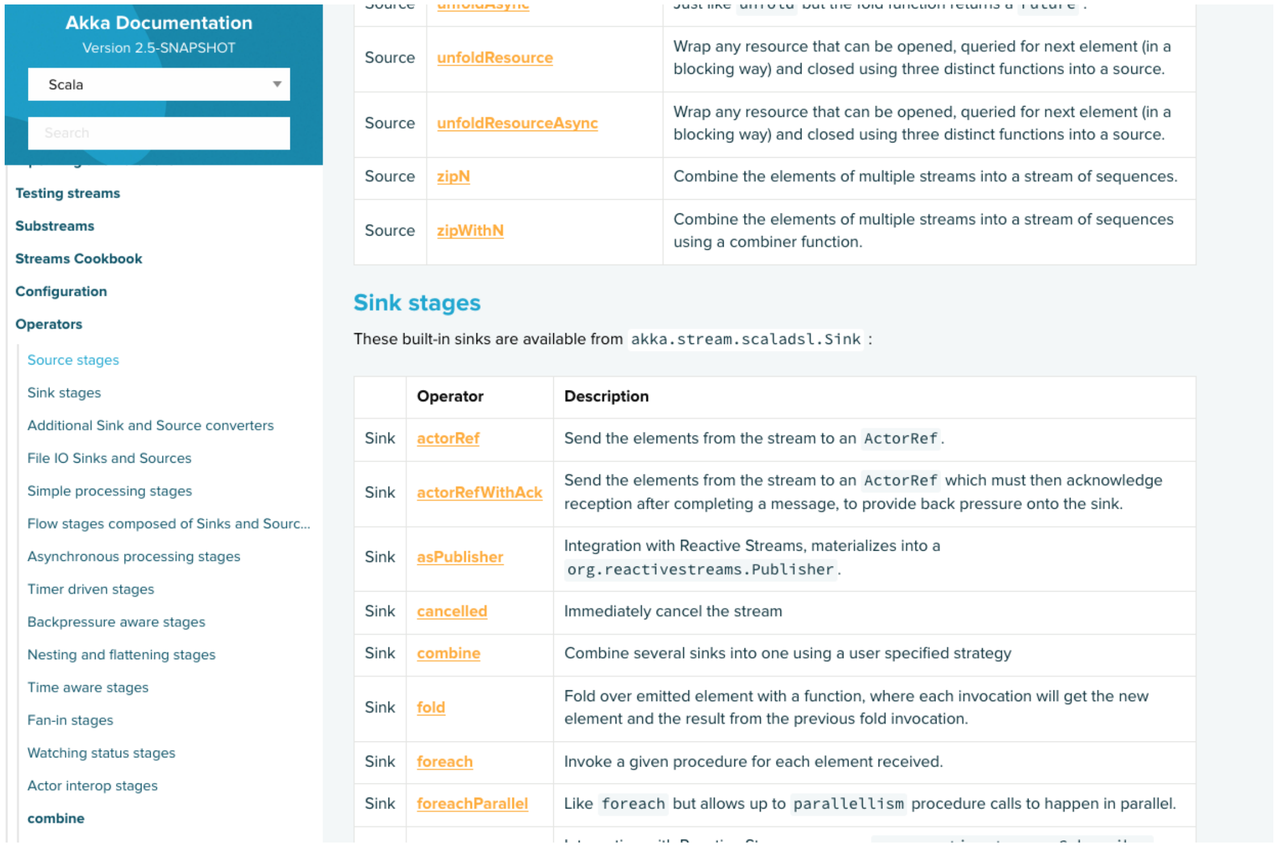
Simple operations in Akka Streams are intentionally looking similar to Scala collections.

For example, consider this Scala code:

```
tweets.filter(_.date < now).map(_.author)
```

The same lines of code would work if the tweets was of type: `Source[Tweet, _]`, or any of the collections in Scala (e.g. `List`, `Seq`, `Set`...).

Reference docs for all operators



https://doc.akka.io/docs/akka/snapshot/stream/operators/index.html

Stream execution model / resource sharing

While the looks may be (almost) the same, the internals and execution mechanisms of those APIs could not be more different.

Consider the following code, printing out the thread on which the execution is happening:

```
tweets.wireTap(_ => println(Thread.currentThread.getName))
```

Could return, either of:

```
Followers-akka.actor.default-dispatcher-2
Followers-akka.actor.default-dispatcher-2
```

or

```
Followers-akka.actor.default-dispatcher-1
Followers-akka.actor.default-dispatcher-2
Followers-akka.actor.default-dispatcher-1
```

Concurrency is not parallelism

> *In programming, concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations. Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once. – Andrew Gerrand*

Stages are by definition concurrent, yet depending on how they are fused and executed may or may not be running in parallel.

To introduce parallelism, use the `.async` operator, or `*Async` versions of operators.

https://blog.golang.org/concurrency-is-not-parallelism

Materialized Values

The last parameter of all stages is the materialized value, we obtain it when we *run* (*materialize*) the stream:

```
val sink: Sink[Int, Future[Int]] =
  Sink.fold(0)((acc, x) => acc + x)

val result: Future[Int] =
  Source(1 to 10)
    .runWith(sink) // materializes sink's mat-value
```

Usually we keep the left-hand value when combining stages. The `runWith` method keeps the right-hand side;

Alternatively the `source.to(sink).run()` would keep the Source's mat value.

Accessing Materialized values in-line

You can change or access a materialized value by using the `.mapMaterializedValue(mat => newMat)` operator.

Sometimes you may not care about materialized values, and then you can simply ignore them.

Summary

- Akka Streams *basic building blocks* are `Source`, `Flow` and `Sink`
- The Akka Streams DSL feels *similar to the Scala collections* API for basic operations
- Stream stages are by construction concurrent, however *fusing and actual runtime (materializer) determine their parallelism*
- *Materialized values* can be used to communicate from “within” the stream with the outside of it in a type-safe and thread-safe way