Akka Stream and HTTP Experimental Java Documentation

Release 1.0-RC2

Typesafe Inc

April 30, 2015

CHAPTER

ONE

STREAMS

1.1 Introduction

1.1.1 Motivation

The way we consume services from the internet today includes many instances of streaming data, both downloading from a service as well as uploading to it or peer-to-peer data transfers. Regarding data as a stream of elements instead of in its entirety is very useful because it matches the way computers send and receive them (for example via TCP), but it is often also a necessity because data sets frequently become too large to be handled as a whole. We spread computations or analyses over large clusters and call it "big data", where the whole principle of processing them is by feeding those data sequentially—as a stream—through some CPUs.

Actors can be seen as dealing with streams as well: they send and receive series of messages in order to transfer knowledge (or data) from one place to another. We have found it tedious and error-prone to implement all the proper measures in order to achieve stable streaming between actors, since in addition to sending and receiving we also need to take care to not overflow any buffers or mailboxes in the process. Another pitfall is that Actor messages can be lost and must be retransmitted in that case lest the stream have holes on the receiving side. When dealing with streams of elements of a fixed given type, Actors also do not currently offer good static guarantees that no wiring errors are made: type-safety could be improved in this case.

For these reasons we decided to bundle up a solution to these problems as an Akka Streams API. The purpose is to offer an intuitive and safe way to formulate stream processing setups such that we can then execute them efficiently and with bounded resource usage—no more OutOfMemoryErrors. In order to achieve this our streams need to be able to limit the buffering that they employ, they need to be able to slow down producers if the consumers cannot keep up. This feature is called back-pressure and is at the core of the Reactive Streams initiative of which Akka is a founding member. For you this means that the hard problem of propagating and reacting to back-pressure has been incorporated in the design of Akka Streams already, so you have one less thing to worry about; it also means that Akka Streams interoperate seamlessly with all other Reactive Streams implementations (where Reactive Streams interfaces define the interoperability SPI while implementations like Akka Streams offer a nice user API).

Relationship with Reactive Streams

The Akka Streams API is completely decoupled from the Reactive Streams interfaces. While Akka Streams focus on the formulation of transformations on data streams the scope of Reactive Streams is just to define a common mechanism of how to move data across an asynchronous boundary without losses, buffering or resource exhaustion.

The relationship between these two is that the Akka Streams API is geared towards end-users while the Akka Streams implementation uses the Reactive Streams interfaces internally to pass data between the different processing stages. For this reason you will not find any resemblance between the Reactive Streams interfaces and the Akka Streams API. This is in line with the expectations of the Reactive Streams project, whose primary purpose is to define interfaces such that different streaming implementation can interoperate; it is not the purpose of Reactive Streams to describe an end-user API.

1.1.2 How to read these docs

Stream processing is a different paradigm to the Actor Model or to Future composition, therefore it may take some careful study of this subject until you feel familiar with the tools and techniques. The documentation is here to help and for best results we recommend the following approach:

- Read the *Quick Start Guide: Reactive Tweets* to get a feel for how streams look like and what they can do.
- The top-down learners may want to peruse the *Design Principles behind Akka Streams* at this point.
- The bottom-up learners may feel more at home rummaging through the *stream-cookbook-java*.
- For a complete overview of the built-in processing stages you can look at the table in *Overview of built-in stages and their semantics*
- The other sections can be read sequentially or as needed during the previous steps, each digging deeper into specific topics.

1.2 Quick Start Guide: 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:

```
public static class Author {
  public final String handle;

  public Author(String handle) {
    this.handle = handle;
  }

  // ...

}

public static class Hashtag {
  public final String name;

  public Hashtag(String name) {
    this.name = name;
  }

  // ...
}

public static class Tweet {
  public final Author author;
  public final long timestamp;
  public final String body;

  public Tweet(Author author, long timestamp, String body) {
    this.author = author;
    this.timestamp = timestamp;
    this.body = body;
  }
```

```
  public Set<Hashtag> hashtags() {
    return Arrays.asList(body.split(" ")).stream()
      .filter(a -> a.startsWith("#"))
      .map(a -> new Hashtag(a))
      .collect(Collectors.toSet());
  }

  // ...
}

public static final Hashtag AKKA = new Hashtag("#akka");
```

1.2.1 Transforming and consuming simple streams

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

```
final ActorSystem system = ActorSystem.create("reactive-tweets");
final FlowMaterializer mat = ActorFlowMaterializer.create(system);
```

The `ActorFlowMaterializer` can optionally take `ActorFlowMaterializerSettings` which can be used to define materialization properties, such as default buffer sizes (see also *Buffers in Akka Streams*), the dispatcher to be used by the pipeline etc. These can be overridden `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`:

```
Source<Tweet, BoxedUnit> tweets;
```

Streams always start flowing from a `Source` then can continue through `Flow` elements or more advanced graph elements to finally be consumed by a `Sink`. The first type parameter—`Tweet` in this case—designates the kind of elements produced by the source while the second one describes the object that is created during materialization (see below)—`BoxedUnit` (from the `scala.runtime` package) means that no value is produced, it is the generic equivalent of `void`. Both Sources and Flows provide stream operations that can be used to transform the flowing data, a `Sink` however does not since its the "end of stream" and its behavior depends on the type of `Sink` used.

In our case let's say we want to find all twitter handles of users which tweet about `#akka`, the operations should look familiar to anyone who has used the Scala Collections library, however they operate on streams and not collections of data:

```
final Source<Author, BoxedUnit> authors =
  tweets
    .filter(t -> t.hashtags().contains(AKKA))
    .map(t -> t.author);
```

Finally in order to *materialize* and run the stream computation we need to attach the Flow to a `Sink<T>` that will get the flow running. The simplest way to do this is to call `runWith(sink)` on a `Source<Out>`. For convenience a number of common Sinks are predefined and collected as static methods on the Sink class. For now let's simply print each author:

```
authors.runWith(Sink.foreach(a -> System.out.println(a)), mat);
```

or by using the shorthand version (which are defined only for the most popular sinks such as `FoldSink` and `ForeachSink`):

```
authors.runForeach(a -> System.out.println(a), mat);
```

Materializing and running a stream always requires a `FlowMaterializer` to be passed in explicitly, like this: `.run(mat)`.

1.2.2 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:

```
final Source<Hashtag, BoxedUnit> hashtags =
  tweets.mapConcat(t -> new ArrayList<Hashtag>(t.hashtags()));
```

Note: The name `flatMap` was consciously avoided due to its proximity with for-comprehensions and monadic composition. It is problematic for two reasons: firstly, flattening by concatenation is often undesirable in bounded stream processing due to the risk of deadlock (with merge being the preferred strategy), and secondly, 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 (`Out f -> java.util.List<T>`), whereas `flatMap` would have to operate on streams all the way through.

1.2.3 Broadcasting a stream

Now let's say we want to persist all hashtags, as well as all author names from this one live stream. For example we'd like to write all author handles into one file, and all hashtags into another file on disk. This means we have to split the source stream into 2 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 (FlowGraphs) 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. 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*. FlowGraphs are constructed like this:

```
Sink<Author, BoxedUnit> writeAuthors;
Sink<Hashtag, BoxedUnit> writeHashtags;
FlowGraph.factory().closed(b -> {
  final UniformFanOutShape<Tweet, Tweet> bcast = b.graph(Broadcast.create(2));
  final Flow<Tweet, Author, BoxedUnit> toAuthor = Flow.of(Tweet.class).map(t -> t.author);
  final Flow<Tweet, Hashtag, BoxedUnit> toTags =
      Flow.of(Tweet.class).mapConcat(t -> new ArrayList<Hashtag>(t.hashtags()));

  b.from(tweets).via(bcast).via(toAuthor).to(writeAuthors);
             b.from(bcast).via(toTags).to(writeHashtags);
}).run(mat);
```

As you can see, we use graph builder to mutably construct the graph using the `addEdge` method. Once we have the FlowGraph in the value `g` *it is immutable, thread-safe, and freely shareable*. A graph can can be `run()` directly - assuming all ports (sinks/sources) within a flow have been connected properly. It is possible to construct `PartialFlowGraph` s where this is not required but this will be covered in detail in *Constructing and combining Partial Flow Graphs*.

As all Akka Streams elements, `Broadcast` will properly propagate back-pressure to its upstream element.

1.2.4 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(t -> slowComputation(t))
  .runWith(Sink.ignore(), mat);
```

The `buffer` element takes an explicit and required `OverflowStrategy`, which defines how the buffer should react when it receives another element element while it is full. Strategies provided include dropping the oldest element (`dropHead`), dropping the entire buffer, signalling failures etc. Be sure to pick and choose the strategy that fits your use case best.

1.2.5 Materialized values

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 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 `FoldSink` and then we'll see how it is possible to obtain materialized values from a `MaterializedMap` which is returned by materializing an Akka stream. We'll split execution into multiple lines for the sake of explaining the concepts of `Materializable` elements and `MaterializedType`

```
final Sink<Integer, Future<Integer>> sumSink =
  Sink.<Integer, Integer>fold(0, (acc, elem) -> acc + elem);

final RunnableFlow<Future<Integer>> counter =
    tweets.map(t -> 1).to(sumSink, Keep.right());

final Future<Integer> sum = counter.run(mat);

sum.foreach(new Foreach<Integer>() {
  public void each(Integer c) {
    System.out.println("Total tweets processed: " + c);
  }
}, system.dispatcher());
```

First, we prepare the `FoldSink` which will be used to sum all `Integer` elements of the stream. Next we connect the `tweets` stream though a `map` step which converts each tweet into the number `1`, finally we connect the flow `to` the previously prepared Sink. Notice that 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: `RunnableFlow`. Next we call `run()` which uses the implicit `ActorFlowMaterializer` to materialize and run the flow. The value returned by calling `run()` on a `RunnableFlow` or `FlowGraph` is `MaterializedMap`, which can be used to retrieve materialized values from the running stream.

In order to extract an materialized value from a running stream it is possible to call `get(Materializable)` on a materialized map obtained from materializing a flow or graph. Since `FoldSink` implements `Materializable` and implements the `MaterializedType` as `Future<Integer>` we can use it to obtain the `Future` 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.

The reason we have to `get` the value out from the materialized map, is because a `RunnableFlow` 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:

```
final Sink<Integer, Future<Integer>> sumSink =
  Sink.<Integer, Integer>fold(0, (acc, elem) -> acc + elem);
final RunnableFlow<Future<Integer>> counterRunnableFlow =
  tweetsInMinuteFromNow
    .filter(t -> t.hashtags().contains(AKKA))
    .map(t -> 1)
    .to(sumSink, Keep.right());

// materialize the stream once in the morning
final Future<Integer> morningTweetsCount = counterRunnableFlow.run(mat);
// and once in the evening, reusing the blueprint
final Future<Integer> eveningTweetsCount = counterRunnableFlow.run(mat);
```

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 section, now we know what happens behind the scenes when we run this one-liner, which is equivalent to the multi line version above:

```
final Future<Integer> sum = tweets.map(t -> 1).runWith(sumSink, mat);
```

1.3 Design Principles behind Akka Streams

It took quite a while until we were reasonably happy with the look and feel of the API and the architecture of the implementation, and while being guided by intuition the design phase was very much exploratory research. This section details the findings and codifies them into a set of principles that have emerged during the process.

Note: As detailed in the introduction keep in mind that the Akka Streams API is completely decoupled from the Reactive Streams interfaces which are just an implementation detail for how to pass stream data between individual processing stages.

1.3.1 What shall users of Akka Streams expect?

Akka is built upon a conscious decision to offer APIs that are minimal and consistent—as opposed to easy or intuitive. The credo is that we favor explicitness over magic, and if we provide a feature then it must work always, no exceptions. Another way to say this is that we minimize the number of rules a user has to learn instead of trying to keep the rules close to what we think users might expect.

From this follows that the principles implemented by Akka Streams are:

- all features are explicit in the API, no magic
- supreme compositionality: combined pieces retain the function of each part
- exhaustive model of the domain of distributed bounded stream processing

This means that we provide all the tools necessary to express any stream processing topology, that we model all the essential aspects of this domain (back-pressure, buffering, transformations, failure recovery, etc.) and that whatever the user builds is reusable in a larger context.

Resulting Implementation Constraints

Compositionality entails reusability of partial stream topologies, which led us to the lifted approach of describing data flows as (partial) FlowGraphs that can act as composite sources, flows (a.k.a. pipes) and sinks of data. These building blocks shall then be freely shareable, with the ability to combine them freely to form larger flows. The representation of these pieces must therefore be an immutable blueprint that is materialized in an explicit step in order to start the stream processing. The resulting stream processing engine is then also immutable in the sense of having a fixed topology that is prescribed by the blueprint. Dynamic networks need to be modeled by explicitly using the Reactive Streams interfaces for plugging different engines together.

The process of materialization may be parameterized, e.g. instantiating a blueprint for handling a TCP connection's data with specific information about the connection's address and port information. Additionally, materialization will often create specific objects that are useful to interact with the processing engine once it is running, for example for shutting it down or for extracting metrics. This means that the materialization function takes a set of parameters from the outside and it produces a set of results. Compositionality demands that these two sets cannot interact, because that would establish a covert channel by which different pieces could communicate, leading to problems of initialization order and inscrutable runtime failures.

Another aspect of materialization is that we want to support distributed stream processing, meaning that both the parameters and the results need to be location transparent—either serializable immutable values or ActorRefs. Using for example Futures would restrict materialization to the local JVM. There may be cases for which this will typically not be a severe restriction (like opening a TCP connection), but the principle remains.

1.3.2 Interoperation with other Reactive Streams implementations

Akka Streams fully implement the Reactive Streams specification and interoperate with all other conformant implementations. We chose to completely separate the Reactive Streams interfaces (which we regard to be an SPI) from the user-level API. In order to obtain a `Publisher` or `Subscriber` from an Akka Stream topology, a corresponding `Sink.publisher` or `Source.subscriber` element must be used.

All stream Processors produced by the default materialization of Akka Streams are restricted to having a single Subscriber, additional Subscribers will be rejected. The reason for this is that the stream topologies described using our DSL never require fan-out behavior from the Publisher sides of the elements, all fan-out is done using explicit elements like `Broadcast[T]`.

This means that `Sink.fanoutPublisher` must be used where multicast behavior is needed for interoperation with other Reactive Streams implementations.

1.3.3 What shall users of streaming libraries expect?

We expect libraries to be built on top of Akka Streams, in fact Akka HTTP is one such example that lives within the Akka project itself. In order to allow users to profit from the principles that are described for Akka Streams above, the following rules are established:

- libraries shall provide their users with reusable pieces, allowing full compositionality
- libraries may optionally and additionally provide facilities that consume and materialize flow descriptions

The reasoning behind the first rule is that compositionality would be destroyed if different libraries only accepted flow descriptions and expected to materialize them: using two of these together would be impossible because materialization can only happen once. As a consequence, the functionality of a library must be expressed such that materialization can be done by the user, outside of the library's control.

The second rule allows a library to additionally provide nice sugar for the common case, an example of which is the Akka HTTP API that provides a `handleWith` method for convenient materialization.

Note: One important consequence of this is that a reusable flow description cannot be bound to "live" resources, any connection to or allocation of such resources must be deferred until materialization time. Examples of "live" resources are already existing TCP connections, a multicast Publisher, etc.; a TickSource does not fall into this category if its timer is created only upon materialization (as is the case for our implementation).

Resulting Implementation Constraints

Akka Streams must enable a library to express any stream processing utility in terms of immutable blueprints. The most common building blocks are

- Source: something with exactly one output stream
- Sink: something with exactly one input stream
- Flow: something with exactly one input and one output stream
- BidirectionalFlow: something with exactly two input streams and two output streams that conceptually behave like two Flows of opposite direction
- Graph: a packaged stream processing topology that exposes a certain set of input and output ports, characterized by an object of type `Shape`.

Note: A source that emits a stream of streams is still just a normal Source, the kind of elements that are produced does not play a role in the static stream topology that is being expressed.

1.3.4 The difference between Error and Failure

The starting point for this discussion is the definition given by the Reactive Manifesto. Translated to streams this means that an error is accessible within the stream as a normal data element, while a failure means that the stream itself has failed and is collapsing. In concrete terms, on the Reactive Streams interface level data elements (including errors) are signaled via `onNext` while failures raise the `onError` signal.

Note: Unfortunately the method name for signaling *failure* to a Subscriber is called `onError` for historical reasons. Always keep in mind that the Reactive Streams interfaces (Publisher/Subscription/Subscriber) are modeling the low-level infrastructure for passing streams between execution units, and errors on this level are precisely the failures that we are talking about on the higher level that is modeled by Akka Streams.

There is only limited support for treating `onError` in Akka Streams compared to the operators that are available for the transformation of data elements, which is intentional in the spirit of the previous paragraph. Since `onError` signals that the stream is collapsing, its ordering semantics are not the same as for stream completion: transformation stages of any kind will just collapse with the stream, possibly still holding elements in implicit or explicit buffers. This means that data elements emitted before a failure can still be lost if the `onError` overtakes them.

The ability for failures to propagate faster than data elements is essential for tearing down streams that are back-pressured—especially since back-pressure can be the failure mode (e.g. by tripping upstream buffers which then abort because they cannot do anything else; or if a dead-lock occurred).

The semantics of stream recovery

A recovery element (i.e. any transformation that absorbs an `onError` signal and turns that into possibly more data elements followed normal stream completion) acts as a bulkhead that confines a stream collapse to a given region of the flow topology. Within the collapsed region buffered elements may be lost, but the outside is not affected by the failure.

This works in the same fashion as a `try–catch` expression: it marks a region in which exceptions are caught, but the exact amount of code that was skipped within this region in case of a failure might not be known precisely—the placement of statements matters.

1.3.5 The finer points of stream materialization

Note: This is not yet implemented as stated here, this document illustrates intent.

It is commonly necessary to parameterize a flow so that it can be materialized for different arguments, an example would be the handler Flow that is given to a server socket implementation and materialized for each incoming connection with information about the peer's address. On the other hand it is frequently necessary to retrieve specific objects that result from materialization, for example a `Future[Unit]` that signals the completion of a `ForeachSink`.

It might be tempting to allow different pieces of a flow topology to access the materialization results of other pieces in order to customize their behavior, but that would violate composability and reusability as argued above. Therefore the arguments and results of materialization need to be segregated:

- The FlowMaterializer is configured with a (type-safe) mapping from keys to values, which is exposed to the processing stages during their materialization.
- The values in this mapping may act as channels, for example by using a Promise/Future pair to communicate a value; another possibility for such information-passing is of course to explicitly model it as a stream of configuration data elements within the graph itself.
- The materialized values obtained from the processing stages are combined as prescribed by the user, but can of course be dependent on the values in the argument mapping.

To avoid having to use `Future` values as key bindings, materialization itself may become fully asynchronous. This would allow for example the use of the bound server port within the rest of the flow, and only if the port was actually bound successfully. The downside is that some APIs will then return `Future[MaterializedMap]`, which means that others will have to accept this in turn in order to keep the usage burden as low as possible.

1.4 Basics and working with Flows

1.4.1 Core concepts

Akka Streams is a library to process and transfer a sequence of elements using bounded buffer space. This latter property is what we refer to as *boundedness* and it is the defining feature of Akka Streams. Translated to everyday terms it is possible to express a chain (or as we see later, graphs) of processing entities, each executing independently (and possibly concurrently) from the others while only buffering a limited number of elements at any given time. This property of bounded buffers is one of the differences from the actor model, where each actor usually has an unbounded, or a bounded, but dropping mailbox. Akka Stream processing entities have bounded "mailboxes" that do not drop.

Before we move on, let's define some basic terminology which will be used throughout the entire documentation:

Stream An active process that involves moving and transforming data.

Element An element is the processing unit of streams. All operations transform and transfer elements from upstream to downstream. Buffer sizes are always expressed as number of elements independently form the actual size of the elements.

Back-pressure A means of flow-control, a way for consumers of data to notify a producer about their current availability, effectively slowing down the upstream producer to match their consumption speeds. In the context of Akka Streams back-pressure is always understood as *non-blocking* and *asynchronous*.

Processing Stage The common name for all building blocks that build up a Flow or FlowGraph. Examples of a processing stage would be operations like `map()`, `filter()`, stages added by `transform()` like `PushStage`, `PushPullStage`, `StatefulStage` and graph junctions like `Merge` or `Broadcast`. For the full list of built-in processing stages see *Overview of built-in stages and their semantics*

Defining and running streams

Linear processing pipelines can be expressed in Akka Streams using the following core abstractions:

Source A processing stage with *exactly one output*, emitting data elements whenever downstream processing stages are ready to receive them.

Sink A processing stage with *exactly one input*, requesting and accepting data elements possibly slowing down the upstream producer of elements

Flow A processing stage which has *exactly one input and output*, which connects its up- and downstreams by transforming the data elements flowing through it.

RunnableFlow A Flow that has both ends "attached" to a Source and Sink respectively, and is ready to be `run()`.

It is possible to attach a `Flow` to a `Source` resulting in a composite source, and it is also possible to prepend a `Flow` to a `Sink` to get a new sink. After a stream is properly terminated by having both a source and a sink, it will be represented by the `RunnableFlow` type, indicating that it is ready to be executed.

It is important to remember that even after constructing the `RunnableFlow` by connecting all the source, sink and different processing stages, no data will flow through it until it is materialized. Materialization is the process of allocating all resources needed to run the computation described by a Flow (in Akka Streams this will often involve starting up Actors). Thanks to Flows being simply a description of the processing pipeline they are *immutable, thread-safe, and freely shareable*, which means that it is for example safe to share and send them between actors, to have one actor prepare the work, and then have it be materialized at some completely different place in the code.

```
final Source<Integer, BoxedUnit> source =
    Source.from(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
// note that the Future is scala.concurrent.Future
final Sink<Integer, Future<Integer>> sink =
    Sink.fold(0, (aggr, next) -> aggr + next);

// connect the Source to the Sink, obtaining a RunnableFlow
final RunnableFlow<Future<Integer>> runnable =
    source.to(sink, Keep.right());

// materialize the flow
final Future<Integer> sum = runnable.run(mat);
```

After running (materializing) the `RunnableFlow` we get a special container object, the `MaterializedMap`. Both sources and sinks are able to put specific objects into this map. Whether they put something in or not is implementation dependent. For example a `FoldSink` will make a `Future` available in this map which will represent the result of the folding process over the stream. In general, a stream can expose multiple materialized values, but it is quite common to be interested in only the value of the Source or the Sink in the stream. For this reason there is a convenience method called `runWith()` available for `Sink`, `Source` or `Flow` requiring, respectively, a supplied `Source` (in order to run a `Sink`), a `Sink` (in order to run a `Source`) or both a `Source` and a `Sink` (in order to run a `Flow`, since it has neither attached yet).

```
final Source<Integer, BoxedUnit> source =
    Source.from(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
final Sink<Integer, Future<Integer>> sink =
    Sink.fold(0, (aggr, next) -> aggr + next);

// materialize the flow, getting the Sinks materialized value
final Future<Integer> sum = source.runWith(sink, mat);
```

It is worth pointing out that since processing stages are *immutable*, connecting them returns a new processing stage, instead of modifying the existing instance, so while constructing long flows, remember to assign the new value to a variable or run it:

```
final Source<Integer, BoxedUnit> source =
    Source.from(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
source.map(x -> 0); // has no effect on source, since it's immutable
source.runWith(Sink.fold(0, (agg, next) -> agg + next), mat); // 55

// returns new Source<Integer>, with `map()` appended
final Source<Integer, BoxedUnit> zeroes = source.map(x -> 0);
final Sink<Integer, Future<Integer>> fold =
    Sink.fold(0, (agg, next) -> agg + next);
zeroes.runWith(fold, mat); // 0
```

Note: By default Akka Streams elements support **exactly one** downstream processing stage. Making fan-out (supporting multiple downstream processing stages) an explicit opt-in feature allows default stream elements to be less complex and more efficient. Also it allows for greater flexibility on *how exactly* to handle the multicast scenarios, by providing named fan-out elements such as broadcast (signals all down-stream elements) or balance (signals one of available down-stream elements).

In the above example we used the `runWith` method, which both materializes the stream and returns the materialized value of the given sink or source.

Since a stream can be materialized multiple times, the `MaterializedMap` returned is different for each materialization. In the example below we create two running materialized instance of the stream that we described in the `runnable` variable, and both materializations give us a different `Future` from the map even though we used the same `sink` to refer to the future:

```
// connect the Source to the Sink, obtaining a RunnableFlow
final Sink<Integer, Future<Integer>> sink =
  Sink.fold(0, (aggr, next) -> aggr + next);
final RunnableFlow<Future<Integer>> runnable =
    Source.from(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)).to(sink, Keep.right());

// get the materialized value of the FoldSink
final Future<Integer> sum1 = runnable.run(mat);
final Future<Integer> sum2 = runnable.run(mat);

// sum1 and sum2 are different Futures!
```

Defining sources, sinks and flows

The objects `Source` and `Sink` define various ways to create sources and sinks of elements. The following examples show some of the most useful constructs (refer to the API documentation for more details):

```
// Create a source from an Iterable
List<Integer> list = new LinkedList<Integer>();
list.add(1);
list.add(2);
list.add(3);
Source.from(list);

// Create a source form a Future
Source.from(Futures.successful("Hello Streams!"));

// Create a source from a single element
Source.single("only one element");

// an empty source
Source.empty();

// Sink that folds over the stream and returns a Future
// of the final result in the MaterializedMap
Sink.fold(0, (Integer aggr, Integer next) -> aggr + next);

// Sink that returns a Future in the MaterializedMap,
// containing the first element of the stream
Sink.head();

// A Sink that consumes a stream without doing anything with the elements
Sink.ignore();

// A Sink that executes a side-effecting call for every element of the stream
Sink.foreach(System.out::println);
```

There are various ways to wire up different parts of a stream, the following examples show some of the available options:

```
// Explicitly creating and wiring up a Source, Sink and Flow
Source.from(Arrays.asList(1, 2, 3, 4))
  .via(Flow.of(Integer.class).map(elem -> elem * 2))
  .to(Sink.foreach(System.out::println));

// Starting from a Source
final Source<Integer, BoxedUnit> source = Source.from(Arrays.asList(1, 2, 3, 4))
    .map(elem -> elem * 2);
source.to(Sink.foreach(System.out::println));

// Starting from a Sink
final Sink<Integer, BoxedUnit> sink = Flow.of(Integer.class)
    .map(elem -> elem * 2).to(Sink.foreach(System.out::println));
Source.from(Arrays.asList(1, 2, 3, 4)).to(sink);
```

Illegal stream elements

In accordance to the Reactive Streams specification (Rule 2.13) Akka Streams do not allow `null` to be passed through the stream as an element. In case you want to model the concept of absence of a value we recommend using `akka.japi.Option` (for Java 6 and 7) or `java.util.Optional` which is available since Java 8.

Back-pressure explained

Akka Streams implement an asynchronous non-blocking back-pressure protocol standardised by the Reactive Streams specification, which Akka is a founding member of.

The user of the library does not have to write any explicit back-pressure handling code — it is built in and dealt with automatically by all of the provided Akka Streams processing stages. It is possible however to add explicit buffer stages with overflow strategies that can influence the behaviour of the stream. This is especially important in complex processing graphs which may even contain loops (which *must* be treated with very special care, as explained in *Graph cycles, liveness and deadlocks*).

The back pressure protocol is defined in terms of the number of elements a downstream `Subscriber` is able to receive and buffer, referred to as `demand`. The source of data, referred to as `Publisher` in Reactive Streams terminology and implemented as `Source` in Akka Streams, guarantees that it will never emit more elements than the received total demand for any given `Subscriber`.

Note: The Reactive Streams specification defines its protocol in terms of `Publisher` and `Subscriber`. These types are **not** meant to be user facing API, instead they serve as the low level building blocks for different Reactive Streams implementations.

Akka Streams implements these concepts as `Source`, `Flow` (referred to as `Processor` in Reactive Streams) and `Sink` without exposing the Reactive Streams interfaces directly. If you need to integrate with other Reactive Stream libraries read *Integrating with Reactive Streams*.

The mode in which Reactive Streams back-pressure works can be colloquially described as "dynamic push / pull mode", since it will switch between push and pull based back-pressure models depending on the downstream being able to cope with the upstream production rate or not.

To illustrate this further let us consider both problem situations and how the back-pressure protocol handles them:

Slow Publisher, fast Subscriber

This is the happy case of course – we do not need to slow down the Publisher in this case. However signalling rates are rarely constant and could change at any point in time, suddenly ending up in a situation where the Subscriber

is now slower than the Publisher. In order to safeguard from these situations, the back-pressure protocol must still be enabled during such situations, however we do not want to pay a high penalty for this safety net being enabled.

The Reactive Streams protocol solves this by asynchronously signalling from the Subscriber to the Publisher `Request(int n)` signals. The protocol guarantees that the Publisher will never signal *more* elements than the signalled demand. Since the Subscriber however is currently faster, it will be signalling these Request messages at a higher rate (and possibly also batching together the demand - requesting multiple elements in one Request signal). This means that the Publisher should not ever have to wait (be back-pressured) with publishing its incoming elements.

As we can see, in this scenario we effectively operate in so called push-mode since the Publisher can continue producing elements as fast as it can, since the pending demand will be recovered just-in-time while it is emitting elements.

Fast Publisher, slow Subscriber

This is the case when back-pressuring the `Publisher` is required, because the `Subscriber` is not able to cope with the rate at which its upstream would like to emit data elements.

Since the `Publisher` is not allowed to signal more elements than the pending demand signalled by the `Subscriber`, it will have to abide to this back-pressure by applying one of the below strategies:

- not generate elements, if it is able to control their production rate,
- try buffering the elements in a *bounded* manner until more demand is signalled,
- drop elements until more demand is signalled,
- tear down the stream if unable to apply any of the above strategies.

As we can see, this scenario effectively means that the `Subscriber` will *pull* the elements from the Publisher – this mode of operation is referred to as pull-based back-pressure.

Stream Materialization

When constructing flows and graphs in Akka Streams think of them as preparing a blueprint, an execution plan. Stream materialization is the process of taking a stream description (the graph) and allocating all the necessary resources it needs in order to run. In the case of Akka Streams this often means starting up Actors which power the processing, but is not restricted to that - it could also mean opening files or socket connections etc. – depending on what the stream needs.

Materialization is triggered at so called "terminal operations". Most notably this includes the various forms of the `run()` and `runWith()` methods defined on flow elements as well as a small number of special syntactic sugars for running with well-known sinks, such as `runForeach(el -> )` (being an alias to `runWith(Sink.foreach(el -> ))`.

Materialization is currently performed synchronously on the materializing thread. Tha actual stream processing is handled by *Actors actor-java* started up during the streams materialization, which will be running on the thread pools they have been configured to run on - which defaults to the dispatcher set in `MaterializationSettings` while constructing the `ActorFlowMaterializer`.

Note: Reusing *instances* of linear computation stages (Source, Sink, Flow) inside FlowGraphs is legal, yet will materialize that stage multiple times.

Combining materialized values

Since every processing stage in Akka Streams can provide a materialized value after being materialized, it is necessary to somehow express how these values should be composed to a final value when we plug these stages together. For this, many combinator methods have variants that take an additional argument, a function, that will

be used to combine the resulting values. Some examples of using these combiners are illustrated in the example below.

```
// An empty source that can be shut down explicitly from the outside
Source<Integer, Promise<BoxedUnit>> source = Source.<Integer>lazyEmpty();

// A flow that internally throttles elements to 1/second, and returns a Cancellable
// which can be used to shut down the stream
Flow<Integer, Integer, Cancellable> flow = throttler;

// A sink that returns the first element of a stream in the returned Future
Sink<Integer, Future<Integer>> sink = Sink.head();


// By default, the materialized value of the leftmost stage is preserved
RunnableFlow<Promise<BoxedUnit>> r1 = source.via(flow).to(sink);

// Simple selection of materialized values by using Keep.right
RunnableFlow<Cancellable> r2 = source.via(flow, Keep.right()).to(sink);
RunnableFlow<Future<Integer>> r3 = source.via(flow).to(sink, Keep.right());

// Using runWith will always give the materialized values of the stages added
// by runWith() itself
Future<Integer> r4 = source.via(flow).runWith(sink, mat);
Promise<BoxedUnit> r5 = flow.to(sink).runWith(source, mat);
Pair<Promise<BoxedUnit>, Future<Integer>> r6 = flow.runWith(source, sink, mat);

// Using more complext combinations
RunnableFlow<Pair<Promise<BoxedUnit>, Cancellable>> r7 =
source.via(flow, Keep.both()).to(sink);

RunnableFlow<Pair<Promise<BoxedUnit>, Future<Integer>>> r8 =
source.via(flow).to(sink, Keep.both());

RunnableFlow<Pair<Pair<Promise<BoxedUnit>, Cancellable>, Future<Integer>>> r9 =
source.via(flow, Keep.both()).to(sink, Keep.both());

RunnableFlow<Pair<Cancellable, Future<Integer>>> r10 =
source.via(flow, Keep.right()).to(sink, Keep.both());

// It is also possible to map over the materialized values. In r9 we had a
// doubly nested pair, but we want to flatten it out


RunnableFlow<Cancellable> r11 =
r9.mapMaterialized( (nestedTuple) -> {
  Promise<BoxedUnit> p = nestedTuple.first().first();
  Cancellable c = nestedTuple.first().second();
  Future<Integer> f = nestedTuple.second();

  // Picking the Cancellable, but we could  also construct a domain class here
  return c;
});
```

Note: In Graphs it is possible to access the materialized value from inside the stream processing graph. For details see *Accessing the materialized value inside the Graph*

1.4.2 Stream ordering

In Akka Streams almost all computation stages *preserve input order* of elements. This means that if inputs `{IA1,IA2,...,IAn}` "cause" outputs `{OA1,OA2,...,OAk}` and inputs `{IB1,IB2,...,IBm}` "cause"

outputs {OB1,OB2,...,OBl} and all of IAi happened before all IBi then OAi happens before OBi.

This property is even uphold by async operations such as mapAsync, however an unordered version exists called mapAsyncUnordered which does not preserve this ordering.

However, in the case of Junctions which handle multiple input streams (e.g. Merge) the output order is, in general, *not defined* for elements arriving on different input ports. That is a merge-like operation may emit Ai before emitting Bi, and it is up to its internal logic to decide the order of emitted elements. Specialized elements such as Zip however *do guarantee* their outputs order, as each output element depends on all upstream elements having been signalled already – thus the ordering in the case of zipping is defined by this property.

If you find yourself in need of fine grained control over order of emitted elements in fan-in scenarios consider using MergePreferred or FlexiMerge – which gives you full control over how the merge is performed.

1.5 Working with Graphs

In Akka Streams computation graphs are not expressed using a fluent DSL like linear computations are, instead they are written in a more graph-resembling DSL which aims to make translating graph drawings (e.g. from notes taken from design discussions, or illustrations in protocol specifications) to and from code simpler. In this section we'll dive into the multiple ways of constructing and re-using graphs, as well as explain common pitfalls and how to avoid them.

Graphs are needed whenever you want to perform any kind of fan-in ("multiple inputs") or fan-out ("multiple outputs") operations. Considering linear Flows to be like roads, we can picture graph operations as junctions: multiple flows being connected at a single point. Some graph operations which are common enough and fit the linear style of Flows, such as concat (which concatenates two streams, such that the second one is consumed after the first one has completed), may have shorthand methods defined on Flow or Source themselves, however you should keep in mind that those are also implemented as graph junctions.

1.5.1 Constructing Flow Graphs

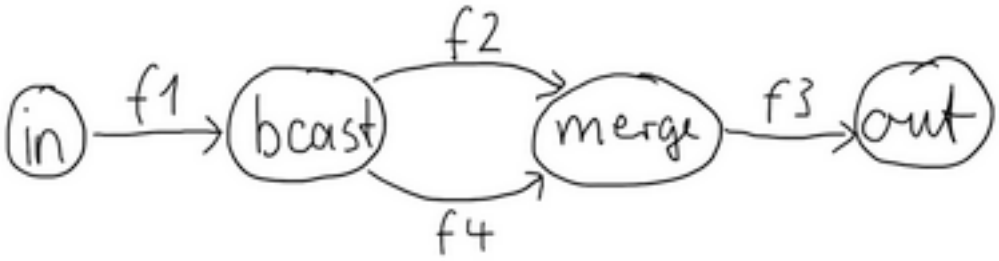
Flow graphs are built from simple Flows which serve as the linear connections within the graphs as well as junctions which serve as fan-in and fan-out points for Flows. Thanks to the junctions having meaningful types based on their behaviour and making them explicit elements these elements should be rather straightforward to use.

Akka Streams currently provide these junctions (for a detailed list see *Overview of built-in stages and their semantics*):

- **Fan-out**
- Broadcast[T] – *(1 input, N outputs)* given an input element emits to each output
- Balance[T] – *(1 input, N outputs)* given an input element emits to one of its output ports
- UnZip[A,B] – *(1 input, 2 outputs)* splits a stream of (A,B) tuples into two streams, one of type A and one of type B
- FlexiRoute[In] – *(1 input, N outputs)* enables writing custom fan out elements using a simple DSL
- **Fan-in**
- Merge[In] – *(N inputs , 1 output)* picks randomly from inputs pushing them one by one to its output
- MergePreferred[In] – like Merge but if elements are available on preferred port, it picks from it, otherwise randomly from others
- ZipWith[A,B,...,Out] – *(N inputs, 1 output)* which takes a function of N inputs that given a value for each input emits 1 output element
- Zip[A,B] – *(2 inputs, 1 output)* is a ZipWith specialised to zipping input streams of A and B into an (A,B) tuple stream
- Concat[A] – *(2 inputs, 1 output)* concatenates two streams (first consume one, then the second one)

- `FlexiMerge[Out]` – *(N inputs, 1 output)* enables writing custom fan-in elements using a simple DSL

One of the goals of the FlowGraph DSL is to look similar to how one would draw a graph on a whiteboard, so that it is simple to translate a design from whiteboard to code and be able to relate those two. Let's illustrate this by translating the below hand drawn graph into Akka Streams:



Such graph is simple to translate to the Graph DSL since each linear element corresponds to a `Flow`, and each circle corresponds to either a `Junction` or a `Source` or `Sink` if it is beginning or ending a `Flow`.

```
final Source<Integer, BoxedUnit> in = Source.from(Arrays.asList(1, 2, 3, 4, 5));
final Sink<List<String>, Future<List<String>>> sink = Sink.head();
final Flow<Integer, Integer, BoxedUnit> f1 =
    Flow.of(Integer.class).map(elem -> elem + 10);
final Flow<Integer, Integer, BoxedUnit> f2 =
    Flow.of(Integer.class).map(elem -> elem + 20);
final Flow<Integer, String, BoxedUnit> f3 =
    Flow.of(Integer.class).map(elem -> elem.toString());
final Flow<Integer, Integer, BoxedUnit> f4 =
    Flow.of(Integer.class).map(elem -> elem + 30);

final RunnableFlow<Future<List<String>>> result = FlowGraph.factory()
    .closed(
        sink,
        (builder, out) -> {
          final UniformFanOutShape<Integer, Integer> bcast =
              builder.graph(Broadcast.create(2));
          final UniformFanInShape<Integer, Integer> merge =
              builder.graph(Merge.create(2));

          builder.from(in).via(f1).via(bcast).via(f2).via(merge)
              .via(f3.grouped(1000)).to(out);
          builder.from(bcast).via(f4).to(merge);
        });
```

Note: Junction *reference equality* defines *graph node equality* (i.e. the same merge *instance* used in a FlowGraph refers to the same location in the resulting graph).

By looking at the snippets above, it should be apparent that the `builder` object is *mutable*. The reason for this design choice is to enable simpler creation of complex graphs, which may even contain cycles. Once the FlowGraph has been constructed though, the `RunnableFlow` instance *is immutable, thread-safe, and freely shareable*. The same is true of all flow pieces—sources, sinks, and flows—once they are constructed. This means that you can safely re-use one given Flow in multiple places in a processing graph.

We have seen examples of such re-use already above: the merge and broadcast junctions were imported into the graph using `builder.graph(...)`, an operation that will make a copy of the blueprint that is passed to it and return the inlets and outlets of the resulting copy so that they can be wired up. Another alternative is to pass existing graphs—of any shape—into the factory method that produces a new graph. The difference between these approaches is that importing using `b.graph(...)` ignores the materialized value of the imported graph while importing via the factory method allows its inclusion; for more details see *stream-materialization-scala*.

In the example below we prepare a graph that consists of two parallel streams, in which we re-use the same instance of `Flow`, yet it will properly be materialized as two connections between the corresponding Sources and

Sinks:

```
final Sink<Integer, Future<Integer>> topHeadSink = Sink.head();
final Sink<Integer, Future<Integer>> bottomHeadSink = Sink.head();
final Flow<Integer, Integer, BoxedUnit> sharedDoubler =
    Flow.of(Integer.class).map(elem -> elem * 2);

final RunnableFlow<Pair<Future<Integer>, Future<Integer>>> g = FlowGraph
    .factory().closed(
        topHeadSink,    // import this sink into the graph
        bottomHeadSink, // and this as well
        Keep.both(),
        (b, top, bottom) -> {
          final UniformFanOutShape<Integer, Integer> bcast = b
              .graph(Broadcast.create(2));

          b.from(Source.single(1)).via(bcast).via(sharedDoubler).to(top);
                                  b.from(bcast).via(sharedDoubler).to(bottom);
        });
```

1.5.2 Constructing and combining Partial Flow Graphs

Sometimes it is not possible (or needed) to construct the entire computation graph in one place, but instead construct all of its different phases in different places and in the end connect them all into a complete graph and run it.

This can be achieved using `FlowGraph.factory().partial()` instead of `FlowGraph.factory().closed()`, which will return a `Graph` instead of a `RunnableFlow`. The reason of representing it as a different type is that a `RunnableFlow` requires all ports to be connected, and if they are not it will throw an exception at construction time, which helps to avoid simple wiring errors while working with graphs. A partial flow graph however allows you to return the set of yet to be connected ports from the code block that performs the internal wiring.

Let's imagine we want to provide users with a specialized element that given 3 inputs will pick the greatest int value of each zipped triple. We'll want to expose 3 input ports (unconnected sources) and one output port (unconnected sink).

```
final Graph<FanInShape2<Integer, Integer, Integer>, BoxedUnit> zip =
  ZipWith.create((Integer left, Integer right) -> Math.max(left, right));

final Graph<UniformFanInShape<Integer, Integer>, BoxedUnit> pickMaxOfThree =
    FlowGraph.factory().partial(builder -> {
      final FanInShape2<Integer, Integer, Integer> zip1 = builder.graph(zip);
      final FanInShape2<Integer, Integer, Integer> zip2 = builder.graph(zip);

      builder.edge(zip1.out(), zip2.in0());
      // return the shape, which has three inputs and one output
      return new UniformFanInShape<Integer, Integer>(zip2.out(),
          new Inlet[] {zip1.in0(), zip1.in1(), zip2.in1()});
    });

final Sink<Integer, Future<Integer>> resultSink = Sink.<Integer>head();

final RunnableFlow<Future<Integer>> g = FlowGraph.factory()
    .closed(resultSink, (builder, sink) -> {
      // import the partial flow graph explicitly
      final UniformFanInShape<Integer, Integer> pm = builder.graph(pickMaxOfThree);

      builder.from(Source.single(1)).to(pm.in(0));
      builder.from(Source.single(2)).to(pm.in(1));
      builder.from(Source.single(3)).to(pm.in(2));
      builder.from(pm.out()).to(sink);
```

```
    });

final Future<Integer> max = g.run(mat);
```

As you can see, first we construct the partial graph that describes how to compute the maximum of two input streams, then we reuse that twice while constructing the partial graph that extends this to three input streams, then we import it (all of its nodes and connections) explicitly to the `FlowGraph` instance in which all the undefined elements are rewired to real sources and sinks. The graph can then be run and yields the expected result.

Warning: Please note that a `FlowGraph` is not able to provide compile time type-safety about whether or not all elements have been properly connected—this validation is performed as a runtime check during the graph's instantiation.
A partial flow graph also verifies that all ports are either connected or part of the returned `Shape`.

1.5.3 Constructing Sources, Sinks and Flows from Partial Graphs

Instead of treating a `PartialFlowGraph` as simply a collection of flows and junctions which may not yet all be connected it is sometimes useful to expose such a complex graph as a simpler structure, such as a `Source`, `Sink` or `Flow`.

In fact, these concepts can be easily expressed as special cases of a partially connected graph:

- `Source` is a partial flow graph with *exactly one* output, that is it returns a `SourceShape`.
- `Sink` is a partial flow graph with *exactly one* input, that is it returns a `SinkShape`.
- `Flow` is a partial flow graph with *exactly one* input and *exactly one* output, that is it returns a `FlowShape`.

Being able to hide complex graphs inside of simple elements such as Sink / Source / Flow enables you to easily create one complex element and from there on treat it as simple compound stage for linear computations.

In order to create a Source from a partial flow graph `Source` provides a special apply method that takes a function that must return an `Outlet`. This unconnected sink will become "the sink that must be attached before this Source can run". Refer to the example below, in which we create a Source that zips together two numbers, to see this graph construction in action:

```
// first create an indefinite source of integer numbers
class Ints implements Iterator<Integer> {
  private int next = 0;
  @Override
  public boolean hasNext() {
    return true;
  }
  @Override
  public Integer next() {
    return next++;
  }
}
final Source<Integer, BoxedUnit> ints = Source.fromIterator(() -> new Ints());

final Source<Pair<Integer, Integer>, BoxedUnit> pairs = Source.factory().create(
    builder -> {
      final FanInShape2<Integer, Integer, Pair<Integer, Integer>> zip =
          builder.graph(Zip.create());

      builder.from(ints.filter(i -> i % 2 == 0)).to(zip.in0());
      builder.from(ints.filter(i -> i % 2 == 1)).to(zip.in1());

      return zip.out();
    });
```

```
final Future<Pair<Integer, Integer>> firstPair =
    pairs.runWith(Sink.<Pair<Integer, Integer>>head(), mat);
```

Similarly the same can be done for a `Sink<T>`, in which case the returned value must be an `Inlet<T>`. For defining a `Flow<T>` we need to expose both an undefined source and sink:

```
final Flow<Integer, Pair<Integer, String>, BoxedUnit> pairs = Flow.factory().create(
    b -> {
      final UniformFanOutShape<Integer, Integer> bcast = b.graph(Broadcast.create(2));
      final FanInShape2<Integer, String, Pair<Integer, String>> zip =
          b.graph(Zip.create());

      b.from(bcast).to(zip.in0());
      b.from(bcast).via(Flow.of(Integer.class).map(i -> i.toString())).to(zip.in1());

      return new Pair<>(bcast.in(), zip.out());
    });

Source.single(1).via(pairs).runWith(Sink.<Pair<Integer, String>>head(), mat);
```

1.5.4 Bidirectional Flows

A graph topology that is often useful is that of two flows going in opposite directions. Take for example a codec stage that serializes outgoing messages and deserializes incoming octet streams. Another such stage could add a framing protocol that attaches a length header to outgoing data and parses incoming frames back into the original octet stream chunks. These two stages are meant to be composed, applying one atop the other as part of a protocol stack. For this purpose exists the special type `BidiFlow` which is a graph that has exactly two open inlets and two open outlets. The corresponding shape is called `BidiShape` and is defined like this:

```
/**
 * A bidirectional flow of elements that consequently has two inputs and two
 * outputs, arranged like this:
 *
 * {{{
 *        +------+
 *  In1 ~>|      |~> Out1
 *        | bidi |
 * Out2 <~|      |<~ In2
 *        +------+
 * }}}
 */
final case class BidiShape[-In1, +Out1, -In2, +Out2](in1: Inlet[In1],
                                                     out1: Outlet[Out1],
                                                     in2: Inlet[In2],
                                                     out2: Outlet[Out2]) extends Shape {
  // implementation details elided ...
}
```

A bidirectional flow is defined just like a unidirectional `Flow` as demonstrated for the codec mentioned above:

```
static interface Message {}
static class Ping implements Message {
  final int id;
  public Ping(int id) { this.id = id; }
  @Override
  public boolean equals(Object o) {
    if (o instanceof Ping) {
      return ((Ping) o).id == id;
    } else return false;
  }
  @Override
```

```
  public int hashCode() {
    return id;
  }
}
static class Pong implements Message {
  final int id;
  public Pong(int id) { this.id = id; }
  @Override
  public boolean equals(Object o) {
    if (o instanceof Pong) {
      return ((Pong) o).id == id;
    } else return false;
  }
  @Override
  public int hashCode() {
    return id;
  }
}

public static ByteString toBytes(Message msg) {
  // implementation details elided ...
}

public static Message fromBytes(ByteString bytes) {
  // implementation details elided ...
}

public final BidiFlow<Message, ByteString, ByteString, Message, BoxedUnit> codecVerbose =
    BidiFlow.factory().create(b -> {
      final FlowShape<Message, ByteString> top =
          b.graph(Flow.<Message> empty().map(BidiFlowDocTest::toBytes));
      final FlowShape<ByteString, Message> bottom =
          b.graph(Flow.<ByteString> empty().map(BidiFlowDocTest::fromBytes));
      return new BidiShape<>(top, bottom);
    });

public final BidiFlow<Message, ByteString, ByteString, Message, BoxedUnit> codec =
    BidiFlow.fromFunctions(BidiFlowDocTest::toBytes, BidiFlowDocTest::fromBytes);
```

The first version resembles the partial graph constructor, while for the simple case of a functional 1:1 transformation there is a concise convenience method as shown on the last line. The implementation of the two functions is not difficult either:

```
public static ByteString toBytes(Message msg) {
  if (msg instanceof Ping) {
    final int id = ((Ping) msg).id;
    return new ByteStringBuilder().putByte((byte) 1)
        .putInt(id, ByteOrder.LITTLE_ENDIAN).result();
  } else {
    final int id = ((Pong) msg).id;
    return new ByteStringBuilder().putByte((byte) 2)
        .putInt(id, ByteOrder.LITTLE_ENDIAN).result();
  }
}

public static Message fromBytes(ByteString bytes) {
  final ByteIterator it = bytes.iterator();
  switch(it.getByte()) {
  case 1:
    return new Ping(it.getInt(ByteOrder.LITTLE_ENDIAN));
  case 2:
    return new Pong(it.getInt(ByteOrder.LITTLE_ENDIAN));
  default:
```

```
    throw new RuntimeException("message format error");
  }
}
```

In this way you could easily integrate any other serialization library that turns an object into a sequence of bytes.

The other stage that we talked about is a little more involved since reversing a framing protocol means that any received chunk of bytes may correspond to zero or more messages. This is best implemented using a `PushPullStage` (see also *Using PushPullStage*).

```
public static ByteString addLengthHeader(ByteString bytes) {
  final int len = bytes.size();
  return new ByteStringBuilder()
    .putInt(len, ByteOrder.LITTLE_ENDIAN)
    .append(bytes)
    .result();
}

public static class FrameParser extends PushPullStage<ByteString, ByteString> {
  // this holds the received but not yet parsed bytes
  private ByteString stash = ByteString.empty();
  // this holds the current message length or -1 if at a boundary
  private int needed = -1;

  @Override
  public SyncDirective onPull(Context<ByteString> ctx) {
    return run(ctx);
  }

  @Override
  public SyncDirective onPush(ByteString bytes, Context<ByteString> ctx) {
    stash = stash.concat(bytes);
    return run(ctx);
  }

  @Override
  public TerminationDirective onUpstreamFinish(Context<ByteString> ctx) {
    if (stash.isEmpty()) return ctx.finish();
    else return ctx.absorbTermination(); // we still have bytes to emit
  }

  private SyncDirective run(Context<ByteString> ctx) {
    if (needed == -1) {
      // are we at a boundary? then figure out next length
      if (stash.size() < 4) return pullOrFinish(ctx);
      else {
        needed = stash.iterator().getInt(ByteOrder.LITTLE_ENDIAN);
        stash = stash.drop(4);
        return run(ctx); // cycle back to possibly already emit the next chunk
      }
    } else if (stash.size() < needed) {
      // we are in the middle of a message, need more bytes
      return pullOrFinish(ctx);
    } else {
      // we have enough to emit at least one message, so do it
      final ByteString emit = stash.take(needed);
      stash = stash.drop(needed);
      needed = -1;
      return ctx.push(emit);
    }
  }

  /*
```

```
   * After having called absorbTermination() we cannot pull any more, so if we need
   * more data we will just have to give up.
   */
  private SyncDirective pullOrFinish(Context<ByteString> ctx) {
    if (ctx.isFinishing()) return ctx.finish();
    else return ctx.pull();
  }
}

public final BidiFlow<ByteString, ByteString, ByteString, ByteString, BoxedUnit> framing =
    BidiFlow.factory().create(b -> {
      final FlowShape<ByteString, ByteString> top =
          b.graph(Flow.<ByteString> empty().map(BidiFlowDocTest::addLengthHeader));
      final FlowShape<ByteString, ByteString> bottom =
          b.graph(Flow.<ByteString> empty().transform(() -> new FrameParser()));
      return new BidiShape<>(top, bottom);
    });
```

With these implementations we can build a protocol stack and test it:

```
/* construct protocol stack
 *         +------------------------------------+
 *         | stack                              |
 *         |                                    |
 *         |  +-------+            +---------+  |
 *    ~>   O~~o       |     ~>     |         o~~O    ~>
 * Message |  | codec | ByteString | framing |  | ByteString
 *    <~   O~~o       |     <~     |         o~~O    <~
 *         |  +-------+            +---------+  |
 *         +------------------------------------+
 */
final BidiFlow<Message, ByteString, ByteString, Message, BoxedUnit> stack =
    codec.atop(framing);

// test it by plugging it into its own inverse and closing the right end
final Flow<Message, Message, BoxedUnit> pingpong =
    Flow.<Message> empty().collect(new PFBuilder<Message, Message>()
        .match(Ping.class, p -> new Pong(p.id))
        .build()
        );
final Flow<Message, Message, BoxedUnit> flow =
    stack.atop(stack.reversed()).join(pingpong);
final Future<List<Message>> result = Source
    .from(Arrays.asList(0, 1, 2))
    .<Message> map(id -> new Ping(id))
    .via(flow)
    .grouped(10)
    .runWith(Sink.<List<Message>> head(), mat);
final FiniteDuration oneSec = Duration.create(1, TimeUnit.SECONDS);
assertArrayEquals(
    new Message[] { new Pong(0), new Pong(1), new Pong(2) },
    Await.result(result, oneSec).toArray(new Message[0]));
```

This example demonstrates how `BidiFlow` subgraphs can be hooked together and also turned around with the `.reversed()` method. The test simulates both parties of a network communication protocol without actually having to open a network connection—the flows can just be connected directly.

1.5.5 Accessing the materialized value inside the Graph

In certain cases it might be necessary to feed back the materialized value of a Graph (partial, closed or backing a Source, Sink, Flow or BidiFlow). This is possible by using `builder.matValue` which gives an `Outlet` that can be used in the graph as an ordinary source or outlet, and which will eventually emit the materialized value. If

the materialized value is needed at more than one place, it is possible to call `matValue` any number of times to acquire the necessary number of outlets.

```
final Sink<Integer, Future<Integer>> foldSink = Sink.<Integer, Integer>fold(0, (a, b) -> {
  return a + b;
});

final Flow<Future<Integer>, Integer, BoxedUnit> flatten = Flow.<Future<Integer>>empty()
  .mapAsync(4, x -> {
    return x;
  });

final Flow<Integer, Integer, Future<Integer>> foldingFlow = Flow.factory().create(foldSink,
  (b, fold) -> {
     return new Pair<>(
       fold.inlet(),
       b.from(b.matValue()).via(flatten).out()
     );
});
```

Be careful not to introduce a cycle where the materialized value actually contributes to the materialized value. The following example demonstrates a case where the materialized `Future` of a fold is fed back to the fold itself.

```
// This cannot produce any value:
final Source<Integer, Future<Integer>> cyclicSource = Source.factory().create(foldSink,
  (b, fold) -> {
    // - Fold cannot complete until its upstream mapAsync completes
    // - mapAsync cannot complete until the materialized Future produced by
    //   fold completes
    // As a result this Source will never emit anything, and its materialited
    // Future will never complete
    b.from(b.matValue()).via(flatten).to(fold);
    return b.from(b.matValue()).via(flatten).out();
  });
```

1.5.6 Graph cycles, liveness and deadlocks

By default `FlowGraph` does not allow (or to be precise, its builder does not allow) the creation of cycles. The reason for this is that cycles need special considerations to avoid potential deadlocks and other liveness issues. This section shows several examples of problems that can arise from the presence of feedback arcs in stream processing graphs.

The first example demonstrates a graph that contains a naive cycle (the presence of cycles is enabled by calling `allowCycles()` on the builder). The graph takes elements from the source, prints them, then broadcasts those elements to a consumer (we just used `Sink.ignore` for now) and to a feedback arc that is merged back into the main stream via a `Merge` junction.

```
// WARNING! The graph below deadlocks!
final Flow<Integer, Integer, BoxedUnit> printFlow =
  Flow.of(Integer.class).map(s -> {
    System.out.println(s);
    return s;
  });

FlowGraph.factory().closed(b -> {
  final UniformFanInShape<Integer, Integer> merge = b.graph(Merge.create(2));
  final UniformFanOutShape<Integer, Integer> bcast = b.graph(Broadcast.create(2));

  b.from(source).via(merge).via(printFlow).via(bcast).to(Sink.ignore());
               b.to(merge)              .from(bcast);
});
```

Running this we observe that after a few numbers have been printed, no more elements are logged to the console - all processing stops after some time. After some investigation we observe that:

- through merging from `source` we increase the number of elements flowing in the cycle
- by broadcasting back to the cycle we do not decrease the number of elements in the cycle

Since Akka Streams (and Reactive Streams in general) guarantee bounded processing (see the "Buffering" section for more details) it means that only a bounded number of elements are buffered over any time span. Since our cycle gains more and more elements, eventually all of its internal buffers become full, backpressuring `source` forever. To be able to process more elements from `source` elements would need to leave the cycle somehow.

If we modify our feedback loop by replacing the `Merge` junction with a `MergePreferred` we can avoid the deadlock. `MergePreferred` is unfair as it always tries to consume from a preferred input port if there are elements available before trying the other lower priority input ports. Since we feed back through the preferred port it is always guaranteed that the elements in the cycles can flow.

```
// WARNING! The graph below stops consuming from "source" after a few steps
FlowGraph.factory().closed(b -> {
  final MergePreferredShape<Integer> merge = b.graph(MergePreferred.create(1));
  final UniformFanOutShape<Integer, Integer> bcast = b.graph(Broadcast.create(2));

  b.from(source).via(merge).via(printFlow).via(bcast).to(Sink.ignore());
               b.to(merge.preferred())   .from(bcast);
});
```

If we run the example we see that the same sequence of numbers are printed over and over again, but the processing does not stop. Hence, we avoided the deadlock, but `source` is still back-pressured forever, because buffer space is never recovered: the only action we see is the circulation of a couple of initial elements from `source`.

Note: What we see here is that in certain cases we need to choose between boundedness and liveness. Our first example would not deadlock if there would be an infinite buffer in the loop, or vice versa, if the elements in the cycle would be balanced (as many elements are removed as many are injected) then there would be no deadlock.

To make our cycle both live (not deadlocking) and fair we can introduce a dropping element on the feedback arc. In this case we chose the `buffer()` operation giving it a dropping strategy `OverflowStrategy.dropHead`.

```
FlowGraph.factory().closed(b -> {
  final UniformFanInShape<Integer, Integer> merge = b.graph(Merge.create(2));
  final UniformFanOutShape<Integer, Integer> bcast = b.graph(Broadcast.create(2));
  final FlowShape<Integer, Integer> droppyFlow = b.graph(
      Flow.of(Integer.class).buffer(10, OverflowStrategy.dropHead()));

  b.from(source).via(merge).via(printFlow).via(bcast).to(Sink.ignore());
              b.to(merge).via(droppyFlow).from(bcast);
});
```

If we run this example we see that

- The flow of elements does not stop, there are always elements printed
- We see that some of the numbers are printed several times over time (due to the feedback loop) but on average the numbers are increasing in the long term

This example highlights that one solution to avoid deadlocks in the presence of potentially unbalanced cycles (cycles where the number of circulating elements are unbounded) is to drop elements. An alternative would be to define a larger buffer with `OverflowStrategy.fail` which would fail the stream instead of deadlocking it after all buffer space has been consumed.

As we discovered in the previous examples, the core problem was the unbalanced nature of the feedback loop. We circumvented this issue by adding a dropping element, but now we want to build a cycle that is balanced from the beginning instead. To achieve this we modify our first graph by replacing the `Merge` junction with a `ZipWith`. Since `ZipWith` takes one element from `source` *and* from the feedback arc to inject one element into the cycle, we maintain the balance of elements.

```
// WARNING! The graph below never processes any elements
FlowGraph.factory().closed(b -> {
  final FanInShape2<Integer, Integer, Integer>
    zip = b.graph(ZipWith.create((Integer left, Integer right) -> left));
  final UniformFanOutShape<Integer, Integer> bcast = b.graph(Broadcast.create(2));

  b.from(source).to(zip.in0());
  b.from(zip.out()).via(printFlow).via(bcast).to(Sink.ignore());
    b.to(zip.in1())              .from(bcast);
});
```

Still, when we try to run the example it turns out that no element is printed at all! After some investigation we realize that:

- In order to get the first element from `source` into the cycle we need an already existing element in the cycle
- In order to get an initial element in the cycle we need an element from `source`

These two conditions are a typical "chicken-and-egg" problem. The solution is to inject an initial element into the cycle that is independent from `source`. We do this by using a `Concat` junction on the backwards arc that injects a single element using `Source.single`.

```
FlowGraph.factory().closed(b -> {
  final FanInShape2<Integer, Integer, Integer>
    zip = b.graph(ZipWith.create((Integer left, Integer right) -> left));
  final UniformFanOutShape<Integer, Integer> bcast = b.graph(Broadcast.create(2));
  final UniformFanInShape<Integer, Integer> concat = b.graph(Concat.create());

  b.from(source).to(zip.in0());
  b.from(zip.out()).via(printFlow).via(bcast).to(Sink.ignore());
    b.to(zip.in1()).via(concat).from(Source.single(1));
                   b.to(concat)  .from(bcast);
});
```

When we run the above example we see that processing starts and never stops. The important takeaway from this example is that balanced cycles often need an initial "kick-off" element to be injected into the cycle.

1.6 Buffers and working with rate

Akka Streams processing stages are asynchronous and pipelined by default which means that a stage, after handing out an element to its downstream consumer is able to immediately process the next message. To demonstrate what we mean by this, let's take a look at the following example:

```
Source.from(Arrays.asList(1, 2, 3))
  .map(i -> {System.out.println("A: " + i); return i;})
  .map(i -> {System.out.println("B: " + i); return i;})
  .map(i -> {System.out.println("C: " + i); return i;})
  .runWith(Sink.ignore(), mat);
```

Running the above example, one of the possible outputs looks like this:

```
A: 1
A: 2
B: 1
A: 3
B: 2
C: 1
B: 3
C: 2
C: 3
```

Note that the order is *not* `A:1, B:1, C:1, A:2, B:2, C:2,` which would correspond to a synchronous execution model where an element completely flows through the processing pipeline before the next element enters the flow. The next element is processed by a stage as soon as it emitted the previous one.

While pipelining in general increases throughput, in practice there is a cost of passing an element through the asynchronous (and therefore thread crossing) boundary which is significant. To amortize this cost Akka Streams uses a *windowed*, *batching* backpressure strategy internally. It is windowed because as opposed to a Stop-And-Wait protocol multiple elements might be "in-flight" concurrently with requests for elements. It is also batching because a new element is not immediately requested once an element has been drained from the window-buffer but multiple elements are requested after multiple elements has been drained. This batching strategy reduces the communication cost of propagating the backpressure signal through the asynchronous boundary.

While this internal protocol is mostly invisible to the user (apart form its throughput increasing effects) there are situations when these details get exposed. In all of our previous examples we always assumed that the rate of the processing chain is strictly coordinated through the backpressure signal causing all stages to process no faster than the throughput of the connected chain. There are tools in Akka Streams however that enable the rates of different segments of a processing chain to be "detached" or to define the maximum throughput of the stream through external timing sources. These situations are exactly those where the internal batching buffering strategy suddenly becomes non-transparent.

1.6.1 Buffers in Akka Streams

Internal buffers and their effect

As we have explained, for performance reasons Akka Streams introduces a buffer for every processing stage. The purpose of these buffers is solely optimization, in fact the size of 1 would be the most natural choice if there would be no need for throughput improvements. Therefore it is recommended to keep these buffer sizes small, and increase them only to a level that throughput requirements of the application require. Default buffer sizes can be set through configuration:

```
akka.stream.materializer.max-input-buffer-size = 16
```

Alternatively they can be set by passing a `ActorFlowMaterializerSettings` to the materializer:

```
final FlowMaterializer materializer = ActorFlowMaterializer.create(
  ActorFlowMaterializerSettings.create(system)
    .withInputBuffer(64, 64), system);
```

If buffer size needs to be set for segments of a `Flow` only, it is possible by defining a separate `Flow` with these attributes:

```
final Flow<Integer, Integer, BoxedUnit> flow1 =
  Flow.of(Integer.class)
  .map(elem -> elem * 2) // the buffer size of this map is 1
  .withAttributes(OperationAttributes.inputBuffer(1, 1));
final Flow<Integer, Integer, BoxedUnit> flow2 =
  flow1.via(
    Flow.of(Integer.class)
    .map(elem -> elem / 2)); // the buffer size of this map is the default
```

Here is an example of a code that demonstrate some of the issues caused by internal buffers:

```
final FiniteDuration oneSecond =
    FiniteDuration.create(1, TimeUnit.SECONDS);
final Source<String, Cancellable> msgSource =
    Source.from(oneSecond, oneSecond, "message!");
final Source<String, Cancellable> tickSource =
    Source.from(oneSecond.mul(3), oneSecond.mul(3), "tick");
final Flow<String, Integer, BoxedUnit> conflate =
    Flow.of(String.class).conflate(
        first -> 1, (count, elem) -> count + 1);
```

```
FlowGraph.factory().closed(b -> {
  final FanInShape2<String, Integer, Integer> zipper =
      b.graph(ZipWith.create((String tick, Integer count) -> count));

  b.from(msgSource).via(conflate).to(zipper.in1());
  b.from(tickSource).to(zipper.in0());
  b.from(zipper.out()).to(Sink.foreach(elem -> System.out.println(elem)));
}).run(mat);
```

Running the above example one would expect the number *3* to be printed in every 3 seconds (the `conflate` step here is configured so that it counts the number of elements received before the downstream `ZipWith` consumes them). What is being printed is different though, we will see the number *1*. The reason for this is the internal buffer which is by default 16 elements large, and prefetches elements before the `ZipWith` starts consuming them. It is possible to fix this issue by changing the buffer size of `ZipWith` (or the whole graph) to 1. We will still see a leading 1 though which is caused by an initial prefetch of the `ZipWith` element.

Note: In general, when time or rate driven processing stages exhibit strange behavior, one of the first solutions to try should be to decrease the input buffer of the affected elements to 1.

Explicit user defined buffers

The previous section explained the internal buffers of Akka Streams used to reduce the cost of crossing elements through the asynchronous boundary. These are internal buffers which will be very likely automatically tuned in future versions. In this section we will discuss *explicit* user defined buffers that are part of the domain logic of the stream processing pipeline of an application.

The example below will ensure that 1000 jobs (but not more) are dequeued from an external (imaginary) system and stored locally in memory - relieving the external system:

```
// Getting a stream of jobs from an imaginary external system as a Source
final Source<Job, BoxedUnit> jobs = inboundJobsConnector;
jobs.buffer(1000, OverflowStrategy.backpressure());
```

The next example will also queue up 1000 jobs locally, but if there are more jobs waiting in the imaginary external systems, it makes space for the new element by dropping one element from the *tail* of the buffer. Dropping from the tail is a very common strategy but it must be noted that this will drop the *youngest* waiting job. If some "fairness" is desired in the sense that we want to be nice to jobs that has been waiting for long, then this option can be useful.

```
jobs.buffer(1000, OverflowStrategy.dropTail());
```

Here is another example with a queue of 1000 jobs, but it makes space for the new element by dropping one element from the *head* of the buffer. This is the *oldest* waiting job. This is the preferred strategy if jobs are expected to be resent if not processed in a certain period. The oldest element will be retransmitted soon, (in fact a retransmitted duplicate might be already in the queue!) so it makes sense to drop it first.

```
jobs.buffer(1000, OverflowStrategy.dropHead());
```

Compared to the dropping strategies above, dropBuffer drops all the 1000 jobs it has enqueued once the buffer gets full. This aggressive strategy is useful when dropping jobs is preferred to delaying jobs.

```
jobs.buffer(1000, OverflowStrategy.dropBuffer());
```

If our imaginary external job provider is a client using our API, we might want to enforce that the client cannot have more than 1000 queued jobs otherwise we consider it flooding and terminate the connection. This is easily achievable by the error strategy which simply fails the stream once the buffer gets full.

```
jobs.buffer(1000, OverflowStrategy.fail());
```

1.6.2 Rate transformation

Understanding conflate

TODO

Understanding expand

TODO

1.7 Custom stream processing

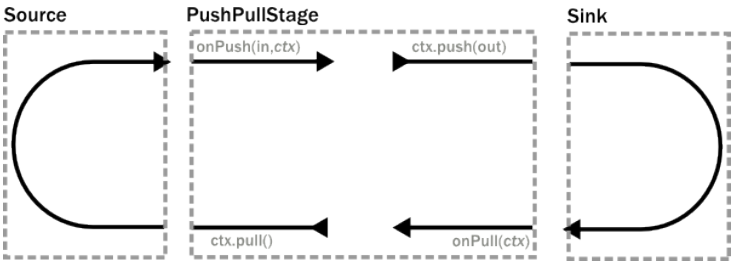
While the processing vocabulary of Akka Streams is quite rich (see the *stream-cookbook-java* for examples) it is sometimes necessary to define new transformation stages either because some functionality is missing from the stock operations, or for performance reasons. In this part we show how to build custom processing stages and graph junctions of various kinds.

1.7.1 Custom linear processing stages

To extend the available transformations on a `Flow` or `Source` one can use the `transform()` method which takes a factory function returning a `Stage`. Stages come in different flavors swhich we will introduce in this page.

Using PushPullStage

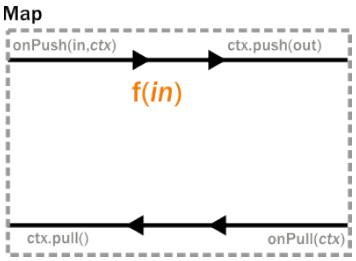
The most elementary transformation stage is the `PushPullStage` which can express a large class of algorithms working on streams. A `PushPullStage` can be illustrated as a box with two "input" and two "output ports" as it is seen in the illustration below.



The "input ports" are implemented as event handlers `onPush(elem,ctx)` and `onPull(ctx)` while "output ports" correspond to methods on the `Context` object that is handed as a parameter to the event handlers. By calling exactly one "output port" method we wire up these four ports in various ways which we demonstrate shortly.

> **Warning:** There is one very important rule to remember when working with a `Stage`. **Exactly one** method should be called on the **currently passed** `Context` **exactly once** and as the **last statement of the handler** where the return type of the called method **matches the expected return type of the handler**. Any violation of this rule will almost certainly result in unspecified behavior (in other words, it will break in spectacular ways). Exceptions to this rule are the query methods `isHolding()` and `isFinishing()`

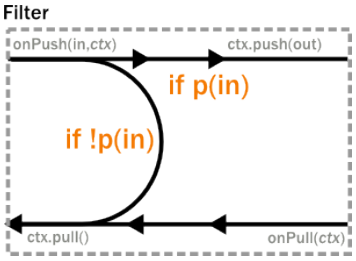
To illustrate these concepts we create a small `PushPullStage` that implements the `map` transformation.



Map calls `ctx.push()` from the `onPush()` handler and it also calls `ctx.pull()` form the `onPull` handler resulting in the conceptual wiring above, and fully expressed in code below:

```
public class Map<A, B> extends PushPullStage<A, B> {
  private final Function<A, B> f;
  public Map(Function<A, B> f) {
    this.f = f;
  }

  @Override public SyncDirective onPush(A elem, Context<B> ctx) {
    return ctx.push(f.apply(elem));
  }

  @Override public SyncDirective onPull(Context<B> ctx) {
    return ctx.pull();
  }
}
```

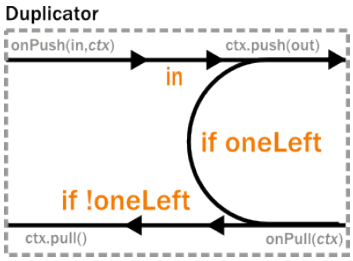
Map is a typical example of a one-to-one transformation of a stream. To demonstrate a many-to-one stage we will implement filter. The conceptual wiring of `Filter` looks like this:



As we see above, if the given predicate matches the current element we are propagating it downwards, otherwise we return the "ball" to our upstream so that we get the new element. This is achieved by modifying the map example by adding a conditional in the `onPush` handler and decide between a `ctx.pull()` or `ctx.push()` call (and of course not having a mapping `f` function).

```
public class Filter<A> extends PushPullStage<A, A> {
  private final Predicate<A> p;
  public Filter(Predicate<A> p) {
    this.p = p;
  }

  @Override public SyncDirective onPush(A elem, Context<A> ctx) {
    if (p.test(elem)) return ctx.push(elem);
    else return ctx.pull();
  }

  @Override public SyncDirective onPull(Context<A> ctx) {
    return ctx.pull();
  }
}
```

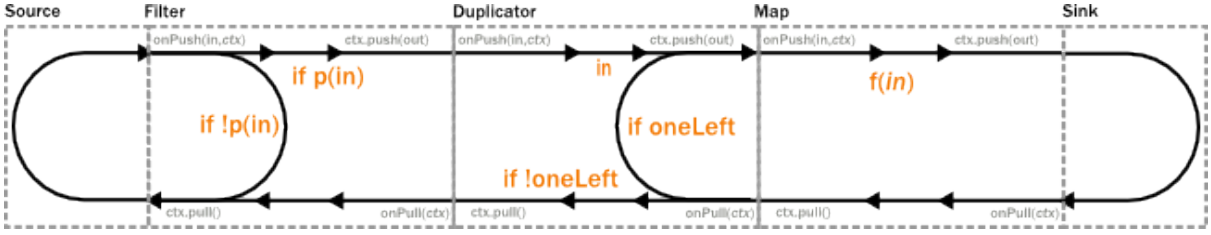
To complete the picture we define a one-to-many transformation as the next step. We chose a straightforward example stage that emits every upstream element twice downstream. The conceptual wiring of this stage looks like this:



This is a stage that has state: the last element it has seen, and a flag `oneLeft` that indicates if we have duplicated this last element already or not. Looking at the code below, the reader might notice that our `onPull` method is more complex than it is demonstrated by the figure above. The reason for this is completion handling, which we will explain a little bit later. For now it is enough to look at the `if(!ctx.isFinishing)` block which corresponds to the logic we expect by looking at the conceptual picture.

```
class Duplicator<A> extends PushPullStage<A, A> {
  private A lastElem = null;
  private boolean oneLeft = false;

  @Override public SyncDirective onPush(A elem, Context<A> ctx) {
    lastElem = elem;
    oneLeft = true;
    return ctx.push(elem);
  }

  @Override public SyncDirective onPull(Context<A> ctx) {
    if (!ctx.isFinishing()) {
      // the main pulling logic is below as it is demonstrated on the illustration
      if (oneLeft) {
        oneLeft = false;
        return ctx.push(lastElem);
      } else
        return ctx.pull();
    } else {
      // If we need to emit a final element after the upstream
      // finished
      if (oneLeft) return ctx.pushAndFinish(lastElem);
      else return ctx.finish();
    }
  }

  @Override public TerminationDirective onUpstreamFinish(Context<A> ctx) {
    return ctx.absorbTermination();
  }

}
```

Finally, to demonstrate all of the stages above, we put them together into a processing chain, which conceptually would correspond to the following structure:



In code this is only a few lines, using the `transform` method to inject our custom processing into a stream:

```
final RunnableFlow<Future<List<Integer>>> runnable =
  Source
    .from(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
```

```
    .transform(() -> new Filter<Integer>(elem -> elem % 2 == 0))
    .transform(() -> new Duplicator<Integer>())
    .transform(() -> new Map<Integer, Integer>(elem -> elem / 2))
    .to(sink, Keep.right());
```

Completion handling

Completion handling usually (but not exclusively) comes into the picture when processing stages need to emit a few more elements after their upstream source has been completed. We have seen an example of this in our `Duplicator` class where the last element needs to be doubled even after the upstream neighbor stage has been completed. Since the `onUpstreamFinish()` handler expects a `TerminationDirective` as the return type we are only allowed to call `ctx.finish()`, `ctx.fail()` or `ctx.absorbTermination()`. Since the first two of these available methods will immediately terminate, our only option is `absorbTermination()`. It is also clear from the return type of `onUpstreamFinish` that we cannot call `ctx.push()` but we need to emit elements somehow! The trick is that after calling `absorbTermination()` the `onPull()` handler will be called eventually, and at the same time `ctx.isFinishing` will return true, indicating that `ctx.pull()` cannot be called anymore. Now we are free to emit additional elementss and call `ctx.finish()` or `ctx.pushAndFinish()` eventually to finish processing.

Note: The reason for this slightly complex termination sequence is that the underlying `onComplete` signal of Reactive Streams may arrive without any pending demand, i.e. without respecting backpressure. This means that our push/pull structure that was illustrated in the figure of our custom processing chain does not apply to termination. Our neat model that is analogous to a ball that bounces back-and-forth in a pipe (it bounces back on `Filter`, `Duplicator` for example) cannot describe the termination signals. By calling `absorbTermination()` the execution environment checks if the conceptual token was *above* the current stage at that time (which means that it will never come back, so the environment immediately calls `onPull`) or it was *below* (which means that it will come back eventually, so the environment does not need to call anything yet).

Using PushStage

Many one-to-one and many-to-one transformations do not need to override the `onPull()` handler at all since all they do is just propagate the pull upwards. For such transformations it is better to extend PushStage directly. For example our `Map` and `Filter` would look like this:

```
public class Map2<A, B> extends PushStage<A, B> {
  private final Function<A, B> f;
  public Map2(Function<A, B> f) {
    this.f = f;
  }

  @Override public SyncDirective onPush(A elem, Context<B> ctx) {
    return ctx.push(f.apply(elem));
  }
}

public class Filter2<A> extends PushStage<A, A> {
  private final Predicate<A> p;
  public Filter2(Predicate<A> p) {
    this.p = p;
  }

  @Override public SyncDirective onPush(A elem, Context<A> ctx) {
    if (p.test(elem)) return ctx.push(elem);
    else return ctx.pull();
  }
}
```

The reason to use `PushStage` is not just cosmetic: internal optimizations rely on the fact that the onPull method only calls `ctx.pull()` and allow the environment do process elements faster than without this knowledge. By extending `PushStage` the environment can be sure that `onPull()` was not overridden since it is `final` on `PushStage`.

Using StatefulStage

On top of `PushPullStage` which is the most elementary and low-level abstraction and `PushStage` that is a convenience class that also informs the environment about possible optimizations `StatefulStage` is a new tool that builds on `PushPullStage` directly, adding various convenience methods on top of it. It is possible to explicitly maintain state-machine like states using its `become()` method to encapsulates states explicitly. There is also a handy `emit()` method that simplifies emitting multiple values given as an iterator. To demonstrate this feature we reimplemented `Duplicator` in terms of a `StatefulStage`:

```
public class Duplicator2<A> extends StatefulStage<A, A> {
  @Override public StageState<A, A> initial() {
    return new StageState<A, A>() {
      @Override public SyncDirective onPush(A elem, Context<A> ctx) {
        return emit(Arrays.asList(elem, elem).iterator(), ctx);
      }
    };
  }
}
```

Using DetachedStage

TODO

1.7.2 Custom graph processing junctions

To extend available fan-in and fan-out structures (graph stages) Akka Streams include `FlexiMerge` and `FlexiRoute` which provide an intuitive DSL which allows to describe which upstream or downstream stream elements should be pulled from or emitted to.

Using FlexiMerge

`FlexiMerge` can be used to describe a fan-in element which contains some logic about which upstream stage the merge should consume elements. It is recommended to create your custom fan-in stage as a separate class, name it appropriately to the behavior it is exposing and reuse it this way – similarly as you would use built-in fan-in stages.

The first flexi merge example we are going to implement is a so-called "preferring merge", in which one of the input ports is *preferred*, e.g. if the merge could pull from the preferred or another secondary input port, it will pull from the preferred port, only pulling from the secondary ports once the preferred one does not have elements available.

Implementing a custom merge stage is done by extending the `FlexiMerge` trait, exposing its input ports and finally defining the logic which will decide how this merge should behave. First we need to create the ports which are used to wire up the fan-in element in a `FlowGraph`. These input ports *must* be properly typed and their names should indicate what kind of port it is.

```
public class PreferringMerge extends FlexiMerge<Integer, Integer, FanInShape3<Integer, Integer, I
  public PreferringMerge() {
    super(
            new FanInShape3<Integer, Integer, Integer, Integer>("PreferringMerge"),
            OperationAttributes.name("PreferringMerge")
    );
  }
```

```
  @Override
  public MergeLogic<Integer, Integer> createMergeLogic(FanInShape3<Integer, Integer, Integer, Inte
    return new MergeLogic<Integer, Integer>() {

      @Override
      public State<Integer, Integer> initialState() {
        return new State<Integer, Integer>(readPreferred(s.in0(), s.in1(), s.in2())) {
          @Override
          public State<Integer, Integer> onInput(MergeLogicContext<Integer> ctx,
                                                 InPort inputHandle, Integer element) {
            ctx.emit(element);
            return sameState();
          }
        };
      }
    };
  }
}
```

Next we implement the `createMergeLogic` method, which will be used as factory of merges `MergeLogic`. A new `MergeLogic` object will be created for each materialized stream, so it is allowed to be stateful.

The `MergeLogic` defines the behaviour of our merge stage, and may be *stateful* (for example to buffer some elements internally).

> **Warning:** While a `MergeLogic` instance *may* be stateful, the `FlexiMerge` instance *must not* hold any mutable state, since it may be shared across several materialized `FlowGraph` instances.

Next we implement the `initialState` method, which returns the behaviour of the merge stage. A `MergeLogic#State` defines the behaviour of the merge by signaling which input ports it is interested in consuming, and how to handle the element once it has been pulled from its upstream. Signalling which input port we are interested in pulling data from is done by using an appropriate *read condition*. Available *read conditions* include:

- `Read(input)` - reads from only the given input,
- `ReadAny(inputs)` – reads from any of the given inputs,
- `ReadPreferred(preferred)(secondaries)` – reads from the preferred input if elements available, otherwise from one of the secondaries,
- `ReadAll(inputs)` – reads from *all* given inputs (like `Zip`), and offers an `ReadAllInputs` as the `element` passed into the state function, which allows to obtain the pulled element values in a type-safe way.

In our case we use the `ReadPreferred` read condition which has the exact semantics which we need to implement our preferring merge – it pulls elements from the preferred input port if there are any available, otherwise reverting to pulling from the secondary inputs. The context object passed into the state function allows us to interact with the connected streams, for example by emitting an `element`, which was just pulled from the given `input`, or signalling completion or failure to the merges downstream stage.

The state function must always return the next behaviour to be used when an element should be pulled from its upstreams, we use the special `SameState` object which signals `FlexiMerge` that no state transition is needed.

> **Note:** As response to an input element it is allowed to emit at most one output element.

Implementing Zip-like merges

More complex fan-in junctions may require not only multiple States but also sharing state between those states. As `MergeLogic` is allowed to be stateful, it can be easily used to hold the state of the merge junction.

We now implement the equivalent of the built-in `Zip` junction by using the property that a the MergeLogic can be stateful and that each read is followed by a state transition (much like in Akka FSM or `Actor#become`).

```
public class Zip2<A, B> extends FlexiMerge<A, Pair<A, B>, FanInShape2<A, B, Pair<A, B>>> {
  public Zip2() {
    super(new FanInShape2<A, B, Pair<A, B>>("Zip2"), OperationAttributes.name("Zip2"));
  }

  @Override
  public MergeLogic<A, Pair<A, B>> createMergeLogic(final FanInShape2<A, B, Pair<A, B>> s) {
    return new MergeLogic<A, Pair<A, B>>() {

      private A lastInA = null;

      private final State<A, Pair<A, B>> readA = new State<A, Pair<A, B>>(read(s.in0())) {
        @Override
        public State<B, Pair<A, B>> onInput(MergeLogicContext<Pair<A, B>> ctx, InPort inputHandle
          lastInA = element;
          return readB;
        }
      };

      private final State<B, Pair<A, B>> readB = new State<B, Pair<A, B>>(read(s.in1())) {
        @Override
        public State<A, Pair<A, B>> onInput(MergeLogicContext<Pair<A, B>> ctx, InPort inputHandle
          ctx.emit(new Pair<A, B>(lastInA, element));
          return readA;
        }
      };

      @Override
      public State<A, Pair<A, B>> initialState() {
        return readA;
      }

      @Override
      public CompletionHandling<Pair<A, B>> initialCompletionHandling() {
        return eagerClose();
      }

    };
  }
}
```

The above style of implementing complex flexi merges is useful when we need fine grained control over consuming from certain input ports. Sometimes however it is simpler to strictly consume all of a given set of inputs. In the `Zip` rewrite below we use the `ReadAll` read condition, which behaves slightly differently than the other read conditions, as the element it is emitting is of the type `ReadAllInputs` instead of directly handing over the pulled elements:

```
public class Zip<A, B> extends FlexiMerge<FlexiMerge.ReadAllInputs, Pair<A, B>, FanInShape2<A, B,
  public Zip() {
    super(new FanInShape2<A, B, Pair<A, B>>("Zip"), OperationAttributes.name("Zip"));
  }

  @Override
  public MergeLogic<ReadAllInputs, Pair<A, B>> createMergeLogic(final FanInShape2<A, B, Pair<A, B
    return new MergeLogic<ReadAllInputs, Pair<A, B>>() {
      @Override
      public State<ReadAllInputs, Pair<A, B>> initialState() {
        return new State<ReadAllInputs, Pair<A, B>>(readAll(s.in0(), s.in1())) {
          @Override
          public State<ReadAllInputs, Pair<A, B>> onInput(
```

```
                  MergeLogicContext<Pair<A, B>> ctx,
                  InPort input,
                  ReadAllInputs inputs) {
            final A a = inputs.get(s.in0());
            final B b = inputs.get(s.in1());

            ctx.emit(new Pair<A, B>(a, b));

            return this;
          }
        };
      }

      @Override
      public CompletionHandling<Pair<A, B>> initialCompletionHandling() {
        return eagerClose();
      }

    };
  }
}
```

Thanks to being handed a `ReadAllInputs` instance instead of the elements directly it is possible to pick elements in a type-safe way based on their input port.

Connecting your custom junction is as simple as creating an instance and connecting Sources and Sinks to its ports (notice that the merged output port is named `out`):

```
final Sink<Pair<Integer, String>, Future<Pair<Integer, String>>> head =
        Sink.<Pair<Integer, String>>head();

final Future<Pair<Integer, String>> future = FlowGraph.factory().closed(head,
        (builder, headSink) -> {
          final FanInShape2<Integer, String, Pair<Integer, String>> zip = builder.graph(new Zip<I
          builder.from(Source.single(1)).to(zip.in0());
          builder.from(Source.single("A")).to(zip.in1());
          builder.from(zip.out()).to(headSink);

        }).run(mat);
```

Completion handling

Completion handling in `FlexiMerge` is defined by an `CompletionHandling` object which can react on completion and failure signals from its upstream input ports. The default strategy is to remain running while at-least-one upstream input port which are declared to be consumed in the current state is still running (i.e. has not signalled completion or failure).

Customising completion can be done via overriding the `MergeLogic#initialCompletionHandling` method, or from within a `State` by calling `ctx.changeCompletionHandling(handling)`. Other than the default completion handling (as late as possible) `FlexiMerge` also provides an `eagerClose` completion handling which completes (or fails) its downstream as soon as at least one of its upstream inputs completes (or fails).

In the example below the we implement an `ImportantWithBackups` fan-in stage which can only keep operating while the `important` and at-least-one of the `replica` inputs are active. Therefore in our custom completion strategy we have to investigate which input has completed or failed and act accordingly. If the important input completed or failed we propagate this downstream completing the stream, on the other hand if the first replicated input fails, we log the exception and instead of failing the downstream swallow this exception (as one failed replica is still acceptable). Then we change the completion strategy to `eagerClose` which will propagate any future completion or failure event right to this stages downstream effectively shutting down the stream.

```
public class ImportantWithBackups<T> extends FlexiMerge<T, T, FanInShape3<T, T, T, T>> {
  public ImportantWithBackups() {
    super(
            new FanInShape3<T, T, T, T>("ImportantWithBackup"),
            OperationAttributes.name("ImportantWithBackup")
    );
  }


  @Override
  public MergeLogic<T, T> createMergeLogic(final FanInShape3<T, T, T, T> s) {
    return new MergeLogic<T, T>() {

      @Override
      public CompletionHandling<T> initialCompletionHandling() {
        return new CompletionHandling<T>() {
          @Override
          public State<T, T> onUpstreamFinish(MergeLogicContextBase<T> ctx,
                                              InPort input) {
            if (input == s.in0()) {
              System.out.println("Important input completed, shutting down.");
              ctx.finish();
              return sameState();
            } else {
              System.out.printf("Replica %s completed, " +
                      "no more replicas available, " +
                      "applying eagerClose completion handling.\n", input);

              ctx.changeCompletionHandling(eagerClose());
              return sameState();
            }
          }

          @Override
          public State<T, T> onUpstreamFailure(MergeLogicContextBase<T> ctx,
                                               InPort input, Throwable cause) {
            if (input == s.in0()) {
              ctx.fail(cause);
              return sameState();
            } else {
              System.out.printf("Replica %s failed, " +
                      "no more replicas available, " +
                      "applying eagerClose completion handling.\n", input);

              ctx.changeCompletionHandling(eagerClose());
              return sameState();
            }
          }
        };

      }

      @Override
      public State<T, T> initialState() {
        return new State<T, T>(readAny(s.in0(), s.in1(), s.in2())) {
          @Override
          public State<T, T> onInput(MergeLogicContext<T> ctx,
                                     InPort input, T element) {
            ctx.emit(element);
            return sameState();
          }
        };
      }
```

```
    };
  }
}
```

In case you want to change back to the default completion handling, it is available as `MergeLogic#defaultCompletionHandling`.

It is not possible to emit elements from the completion handling, since completion handlers may be invoked at any time (without regard to downstream demand being available).

Using FlexiRoute

Similarily to using `FlexiMerge`, implementing custom fan-out stages requires extending the `FlexiRoute` class and with a `RouteLogic` object which determines how the route should behave.

The first flexi route stage that we are going to implement is `Unzip`, which consumes a stream of pairs and splits it into two streams of the first and second elements of each pair.

A `FlexiRoute` has exactly-one input port (in our example, type parameterized as `Pair<A,B>`), and may have multiple output ports, all of which must be created beforehand (they can not be added dynamically). First we need to create the ports which are used to wire up the fan-in element in a `FlowGraph`.

```
public class Unzip<A, B> extends FlexiRoute<Pair<A, B>, FanOutShape2<Pair<A, B>, A, B>> {
  public Unzip() {
    super(new FanOutShape2<Pair<A, B>, A, B>("Unzip"), OperationAttributes.name("Unzip"));
  }
  @Override
  public RouteLogic<Pair<A, B>> createRouteLogic(final FanOutShape2<Pair<A, B>, A, B> s) {
    return new RouteLogic<Pair<A, B>>() {
      @Override
      public State<BoxedUnit, Pair<A, B>> initialState() {
        return new State<BoxedUnit, Pair<A, B>>(demandFromAll(s.out0(), s.out1())) {
          @Override
          public State<BoxedUnit, Pair<A, B>> onInput(RouteLogicContext<Pair<A, B>> ctx, BoxedUnit
                                                      Pair<A, B> element) {
            ctx.emit(s.out0(), element.first());
            ctx.emit(s.out1(), element.second());
            return sameState();
          }
        };
      }

      @Override
      public CompletionHandling<Pair<A, B>> initialCompletionHandling() {
        return eagerClose();
      }

    };
  }
}
```

Next we implement `RouteLogic#initialState` by providing a State that uses the `DemandFromAll` *demand condition* to signal to flexi route that elements can only be emitted from this stage when demand is available from all given downstream output ports. Other available demand conditions are:

- `DemandFrom(output)` - triggers when the given output port has pending demand,
- `DemandFromAny(outputs)` - triggers when any of the given output ports has pending demand,
- `DemandFromAll(outputs)` - triggers when *all* of the given output ports has pending demand.

Since the `Unzip` junction we're implementing signals both downstreams stages at the same time, we use `DemandFromAll`, unpack the incoming pair in the state function and signal its first element to the `left` stream,

and the second element of the pair to the `right` stream. Notice that since we are emitting values of different types (`A` and `B`), the output type parameter of this `State` must be set to `Any`. This type can be utilised more efficiently when a junction is emitting the same type of element to its downstreams e.g. in all *strictly routing* stages.

The state function must always return the next behaviour to be used when an element should be emitted, we use the special `SameState` object which signals `FlexiRoute` that no state transition is needed.

Warning: While a `RouteLogic` instance *may* be stateful, the `FlexiRoute` instance *must not* hold any mutable state, since it may be shared across several materialized `FlowGraph` instances.

Note: It is only allowed to *emit* at most one element to each output in response to *onInput*, *IllegalStateException* is thrown.

Completion handling

Completion handling in `FlexiRoute` is handled similarily to `FlexiMerge` (which is explained in depth in *Completion handling*), however in addition to reacting to its upstreams *completion* or *failure* it can also react to its downstream stages *cancelling* their subscriptions. The default completion handling for `FlexiRoute` (defined in `RouteLogic#defaultCompletionHandling`) is to continue running until all of its downstreams have cancelled their subscriptions, or the upstream has completed / failed.

In order to customise completion handling we can override overriding the `RouteLogic#initialCompletionHandling` method, or call `ctx.changeCompletionHandling(handling)` from within a `State`. Other than the default completion handling (as late as possible) `FlexiRoute` also provides an `eagerClose` completion handling which completes all its downstream streams as well as cancels its upstream as soon as *any* of its downstream stages cancels its subscription.

In the example below we implement a custom completion handler which completes the entire stream eagerly if the `important` downstream cancels, otherwise (if any other downstream cancels their subscription) the `ImportantRoute` keeps running.

```
public class ImportantRoute<T> extends FlexiRoute<T, FanOutShape3<T, T, T, T>> {

  public ImportantRoute() {
    super(new FanOutShape3<T, T, T, T>("ImportantRoute"), OperationAttributes.name("ImportantRoute
  }

  @Override
  public RouteLogic<T> createRouteLogic(FanOutShape3<T, T, T, T> s) {
    return new RouteLogic<T>() {

      @Override
      public CompletionHandling<T> initialCompletionHandling() {
        return new CompletionHandling<T>() {
          @Override
          public State<T, T> onDownstreamFinish(RouteLogicContextBase<T> ctx,
              OutPort output) {
            if (output == s.out0()) {
              // finish all downstreams, and cancel the upstream
              ctx.finish();
              return sameState();
            } else {
              return sameState();
            }
          }

          @Override
          public void onUpstreamFinish(RouteLogicContextBase<T> ctx) {
          }
```

```
          @Override
          public void onUpstreamFailure(RouteLogicContextBase<T> ctx, Throwable t) {
          }
        };

      }

      @Override
      public State<OutPort, T> initialState() {
        return new State<OutPort, T>(demandFromAny(s.out0(), s.out1(), s.out2())) {
          @SuppressWarnings("unchecked")
          @Override
          public State<T, T> onInput(RouteLogicContext<T> ctx, OutPort preferred, T element) {
            ctx.emit((Outlet<T>) preferred, element);
            return sameState();
          }
        };
      }
    };
  }
}
```

Notice that State changes are only allowed in reaction to downstream cancellations, and not in the upstream completion/failure cases. This is because since there is only one upstream, there is nothing else to do than possibly flush buffered elements and continue with shutting down the entire stream.

It is not possible to emit elements from the completion handling, since completion handlers may be invoked at any time (without regard to downstream demand being available).

1.8 Integration

1.8.1 Integrating with Actors

For piping the elements of a stream as messages to an ordinary actor you can use the `Sink.actorRef`. Messages can be sent to a stream via the `ActorRef` that is materialized by `Source.actorRef`.

For more advanced use cases the `ActorPublisher` and `ActorSubscriber` traits are provided to support implementing Reactive Streams `Publisher` and `Subscriber` with an `Actor`.

These can be consumed by other Reactive Stream libraries or used as a Akka Streams `Source` or `Sink`.

Warning: `AbstractActorPublisher` and `AbstractActorSubscriber` cannot be used with remote actors, because if signals of the Reactive Streams protocol (e.g. `request`) are lost the the stream may deadlock.

Note: These Actors are designed to be implemented using Java 8 lambda expressions. In case you need to stay on a JVM prior to 8, Akka provides `UntypedActorPublisher` and `UntypedActorSubscriber` which can be used easily from any language level.

Source.actorRef

Messages sent to the actor that is materialized by `Source.actorRef` will be emitted to the stream if there is demand from downstream, otherwise they will be buffered until request for demand is received.

Depending on the defined `OverflowStrategy` it might drop elements if there is no space available in the buffer.

The stream can be completed successfully by sending `akka.actor.PoisonPill` or `akka.actor.Status.Success` to the actor reference.

The stream can be completed with failure by sending `akka.actor.Status.Failure` to the actor reference.

The actor will be stopped when the stream is completed, failed or cancelled from downstream, i.e. you can watch it to get notified when that happens.

Sink.actorRef

The sink sends the elements of the stream to the given *ActorRef*. If the target actor terminates the stream will be cancelled. When the stream is completed successfully the given `onCompleteMessage` will be sent to the destination actor. When the stream is completed with failure a `akka.actor.Status.Failure` message will be sent to the destination actor.

> **Warning:** There is no back-pressure signal from the destination actor, i.e. if the actor is not consuming the messages fast enough the mailbox of the actor will grow. For potentially slow consumer actors it is recommended to use a bounded mailbox with zero *mailbox-push-timeout-time* or use a rate limiting stage in front of this stage.

ActorPublisher

Extend `akka.stream.actor.AbstractActorPublisher` to implement a stream publisher that keeps track of the subscription life cycle and requested elements.

Here is an example of such an actor. It dispatches incoming jobs to the attached subscriber:

```
public static class JobManagerProtocol {
  final public static class Job {
    public final String payload;

    public Job(String payload) {
      this.payload = payload;
    }

  }

  public static class JobAcceptedMessage {
    @Override
    public String toString() {
      return "JobAccepted";
    }
  }
  public static final JobAcceptedMessage JobAccepted = new JobAcceptedMessage();

  public static class JobDeniedMessage {
    @Override
    public String toString() {
      return "JobDenied";
    }
  }
  public static final JobDeniedMessage JobDenied = new JobDeniedMessage();
}
public static class JobManager extends AbstractActorPublisher<JobManagerProtocol.Job> {

  public static Props props() { return Props.create(JobManager.class); }

  private final int MAX_BUFFER_SIZE = 100;
  private final List<JobManagerProtocol.Job> buf = new ArrayList<>();

  public JobManager() {
```

```
    receive(ReceiveBuilder.
      match(JobManagerProtocol.Job.class, job -> buf.size() == MAX_BUFFER_SIZE, job -> {
        sender().tell(JobManagerProtocol.JobDenied, self());
      }).
      match(JobManagerProtocol.Job.class, job -> {
        sender().tell(JobManagerProtocol.JobAccepted, self());

        if (buf.isEmpty() && totalDemand() > 0)
          onNext(job);
        else {
          buf.add(job);
          deliverBuf();
        }
      }).
      match(ActorPublisherMessage.Request.class, request -> deliverBuf()).
      match(ActorPublisherMessage.Cancel.class, cancel -> context().stop(self())).
      build());
  }

  void deliverBuf() {
    while (totalDemand() > 0) {
        /*
         * totalDemand is a Long and could be larger than
         * what buf.splitAt can accept
         */
      if (totalDemand() <= Integer.MAX_VALUE) {
        final List<JobManagerProtocol.Job> took =
          buf.subList(0, Math.min(buf.size(), (int) totalDemand()));
        took.forEach(this::onNext);
        buf.removeAll(took);
        break;
      } else {
        final List<JobManagerProtocol.Job> took =
          buf.subList(0, Math.min(buf.size(), Integer.MAX_VALUE));
        took.forEach(this::onNext);
        buf.removeAll(took);
      }
    }
  }
}
```

You send elements to the stream by calling `onNext`. You are allowed to send as many elements as have been requested by the stream subscriber. This amount can be inquired with `totalDemand`. It is only allowed to use `onNext` when `isActive` and `totalDemand>0`, otherwise `onNext` will throw `IllegalStateException`.

When the stream subscriber requests more elements the `ActorPublisherMessage.Request` message is delivered to this actor, and you can act on that event. The `totalDemand` is updated automatically.

When the stream subscriber cancels the subscription the `ActorPublisherMessage.Cancel` message is delivered to this actor. After that subsequent calls to `onNext` will be ignored.

You can complete the stream by calling `onComplete`. After that you are not allowed to call `onNext`, `onError` and `onComplete`.

You can terminate the stream with failure by calling `onError`. After that you are not allowed to call `onNext`, `onError` and `onComplete`.

If you suspect that this `AbstractActorPublisher` may never get subscribed to, you can override the `subscriptionTimeout` method to provide a timeout after which this Publisher should be considered canceled. The actor will be notified when the timeout triggers via an `ActorPublisherMessage.SubscriptionTimeoutExceeded` message and MUST then perform cleanup and stop itself.

If the actor is stopped the stream will be completed, unless it was not already terminated with failure, completed or canceled.

More detailed information can be found in the API documentation.

This is how it can be used as input `Source` to a `Flow`:

```
final Source<JobManagerProtocol.Job, ActorRef> jobManagerSource =
  Source.actorPublisher(JobManager.props());

final ActorRef ref = jobManagerSource
  .map(job -> job.payload.toUpperCase())
  .map(elem -> {
    System.out.println(elem);
    return elem;
  })
  .to(Sink.ignore())
  .run(mat);

ref.tell(new JobManagerProtocol.Job("a"), ActorRef.noSender());
ref.tell(new JobManagerProtocol.Job("b"), ActorRef.noSender());
ref.tell(new JobManagerProtocol.Job("c"), ActorRef.noSender());
```

You can only attach one subscriber to this publisher. Use a `Broadcast` element or attach a `Sink.fanoutPublisher` to enable multiple subscribers.

ActorSubscriber

Extend `akka.stream.actor.AbstractActorSubscriber` to make your class a stream subscriber with full control of stream back pressure. It will receive `ActorSubscriberMessage.OnNext`, `ActorSubscriberMessage.OnComplete` and `ActorSubscriberMessage.OnError` messages from the stream. It can also receive other, non-stream messages, in the same way as any actor.

Here is an example of such an actor. It dispatches incoming jobs to child worker actors:

```
public static class WorkerPoolProtocol {

  public static class Msg {
    public final int id;
    public  final ActorRef replyTo;

    public Msg(int id, ActorRef replyTo) {
      this.id = id;
      this.replyTo = replyTo;
    }

    @Override
    public String toString() {
      return String.format("Msg(%s, %s)", id, replyTo);
    }
  }
  public static Msg msg(int id, ActorRef replyTo) {
    return new Msg(id, replyTo);
  }


  public static class Work {
    public final int id;
    public Work(int id) { this.id = id; }

    @Override
    public String toString() {
      return String.format("Work(%s)", id);
```

```
    }
  }
  public static Work work(int id) {
    return new Work(id);
  }


  public static class Reply {
    public final int id;
    public Reply(int id) { this.id = id; }

    @Override
    public String toString() {
      return String.format("Reply(%s)", id);
    }
  }
  public static Reply reply(int id) {
    return new Reply(id);
  }


  public static class Done {
    public final int id;
    public Done(int id) { this.id = id; }

    @Override
    public String toString() {
      return String.format("Done(%s)", id);
    }

    @Override
    public boolean equals(Object o) {
      if (this == o) {
        return true;
      }
      if (o == null || getClass() != o.getClass()) {
        return false;
      }

      Done done = (Done) o;

      if (id != done.id) {
        return false;
      }

      return true;
    }

    @Override
    public int hashCode() {
      return id;
    }
  }
  public static Done done(int id) {
    return new Done(id);
  }

}

public static class WorkerPool extends AbstractActorSubscriber {

  public static Props props() { return Props.create(WorkerPool.class); }
```

```
  final int MAX_QUEUE_SIZE = 10;
  final Map<Integer, ActorRef> queue = new HashMap<>();

  final Router router;

  @Override
  public RequestStrategy requestStrategy() {
    return new MaxInFlightRequestStrategy(MAX_QUEUE_SIZE) {
      @Override
      public int inFlightInternally() {
        return queue.size();
      }
    };
  }

  public WorkerPool() {
    final List<Routee> routees = new ArrayList<>();
    for (int i = 0; i < 3; i++)
      routees.add(new ActorRefRoutee(context().actorOf(Props.create(Worker.class))));
    router = new Router(new RoundRobinRoutingLogic(), routees);

    receive(ReceiveBuilder.
    match(ActorSubscriberMessage.OnNext.class, on -> on.element() instanceof WorkerPoolProtocol.Ms
      onNext -> {
        WorkerPoolProtocol.Msg msg = (WorkerPoolProtocol.Msg) onNext.element();
        queue.put(msg.id, msg.replyTo);

        if (queue.size() > MAX_QUEUE_SIZE)
          throw new RuntimeException("queued too many: " + queue.size());

        router.route(WorkerPoolProtocol.work(msg.id), self());
    }).
    match(WorkerPoolProtocol.Reply.class, reply -> {
      int id = reply.id;
      queue.get(id).tell(WorkerPoolProtocol.done(id), self());
      queue.remove(id);
    }).
    build());
  }
}

static class Worker extends AbstractActor {
  public Worker() {
    receive(ReceiveBuilder.
      match(WorkerPoolProtocol.Work.class, work -> {
        // ...
        sender().tell(WorkerPoolProtocol.reply(work.id), self());
      }).build());
  }
}
```

Subclass must define the `RequestStrategy` to control stream back pressure. After each incoming message the `AbstractActorSubscriber` will automatically invoke the `RequestStrategy.requestDemand` and propagate the returned demand to the stream.

- The provided `WatermarkRequestStrategy` is a good strategy if the actor performs work itself.
- The provided `MaxInFlightRequestStrategy` is useful if messages are queued internally or delegated to other actors.
- You can also implement a custom `RequestStrategy` or call `request` manually together with `ZeroRequestStrategy` or some other strategy. In that case you must also call `request` when the actor is started or when it is ready, otherwise it will not receive any elements.

More detailed information can be found in the API documentation.

This is how it can be used as output `Sink` to a `Flow`:

```
final int N = 117;
final List<Integer> data = new ArrayList<>(N);
for (int i = 0; i < N; i++) {
  data.add(i);
}

Source.from(data)
  .map(i -> WorkerPoolProtocol.msg(i, replyTo))
  .runWith(Sink.<WorkerPoolProtocol.Msg>actorSubscriber(WorkerPool.props()), mat);
```

1.8.2 Integrating with External Services

Stream transformations and side effects involving external non-stream based services can be performed with `mapAsync` or `mapAsyncUnordered`.

For example, sending emails to the authors of selected tweets using an external email service:

```
public Future<Email> send(Email email) {
  // ...
}
```

We start with the tweet stream of authors:

```
final Source<Author, BoxedUnit> authors = tweets
  .filter(t -> t.hashtags().contains(AKKA))
  .map(t -> t.author);
```

Assume that we can lookup their email address using:

```
public Future<Optional<String>> lookupEmail(String handle)
```

Transforming the stream of authors to a stream of email addresses by using the `lookupEmail` service can be done with `mapAsync`:

```
final Source<String, BoxedUnit> emailAddresses = authors
  .mapAsync(4, author -> addressSystem.lookupEmail(author.handle))
  .filter(o -> o.isPresent())
  .map(o -> o.get());
```

Finally, sending the emails:

```
final RunnableFlow<BoxedUnit> sendEmails = emailAddresses
  .mapAsync(4, address ->
    emailServer.send(new Email(address, "Akka", "I like your tweet")))
  .to(Sink.ignore());

sendEmails.run(mat);
```

`mapAsync` is applying the given function that is calling out to the external service to each of the elements as they pass through this processing step. The function returns a `Future` and the value of that future will be emitted downstreams. The number of Futures that shall run in parallel is given as the first argument to `mapAsync`. These Futures may complete in any order, but the elements that are emitted downstream are in the same order as received from upstream.

That means that back-pressure works as expected. For example if the `emailServer.send` is the bottleneck it will limit the rate at which incoming tweets are retrieved and email addresses looked up.

The final piece of this pipeline is to generate the demand that pulls the tweet authors information through the emailing pipeline: we attach a `Sink.ignore` which makes it all run. If our email process would return some

interesting data for further transformation then we would of course not ignore it but send that result stream onwards for further processing or storage.

Note that `mapAsync` preserves the order of the stream elements. In this example the order is not important and then we can use the more efficient `mapAsyncUnordered`:

```
final Source<Author, BoxedUnit> authors =
  tweets
    .filter(t -> t.hashtags().contains(AKKA))
    .map(t -> t.author);

final Source<String, BoxedUnit> emailAddresses =
  authors
    .mapAsyncUnordered(4, author -> addressSystem.lookupEmail(author.handle))
    .filter(o -> o.isPresent())
    .map(o -> o.get());

final RunnableFlow<BoxedUnit> sendEmails =
  emailAddresses
    .mapAsyncUnordered(4, address ->
      emailServer.send(new Email(address, "Akka", "I like your tweet")))
    .to(Sink.ignore());

sendEmails.run(mat);
```

In the above example the services conveniently returned a `Future` of the result. If that is not the case you need to wrap the call in a `Future`. If the service call involves blocking you must also make sure that you run it on a dedicated execution context, to avoid starvation and disturbance of other tasks in the system.

```
final MessageDispatcher blockingEc = system.dispatchers().lookup("blocking-dispatcher");

final RunnableFlow sendTextMessages =
  phoneNumbers
    .mapAsync(4, phoneNo  ->
      Futures.future(() ->
        smsServer.send(new TextMessage(phoneNo, "I like your tweet")),
        blockingEc)
    )
    .to(Sink.ignore());

sendTextMessages.run(mat);
```

The configuration of the `"blocking-dispatcher"` may look something like:

```
blocking-dispatcher {
  executor = "thread-pool-executor"
  thread-pool-executor {
    core-pool-size-min    = 10
    core-pool-size-max    = 10
  }
}
```

An alternative for blocking calls is to perform them in a `map` operation, still using a dedicated dispatcher for that operation.

```
final Flow<String, Boolean, BoxedUnit> send =
  Flow.of(String.class)
  .map(phoneNo -> smsServer.send(new TextMessage(phoneNo, "I like your tweet")))
  .withAttributes(ActorOperationAttributes.dispatcher("blocking-dispatcher"));
final RunnableFlow<?> sendTextMessages =
  phoneNumbers.via(send).to(Sink.ignore());

sendTextMessages.run(mat);
```

However, that is not exactly the same as `mapAsync`, since the `mapAsync` may run several calls concurrently, but `map` performs them one at a time.

For a service that is exposed as an actor, or if an actor is used as a gateway in front of an external service, you can use `ask`:

```
final Source<Tweet, BoxedUnit> akkaTweets = tweets.filter(t -> t.hashtags().contains(AKKA));

final RunnableFlow saveTweets =
  akkaTweets
    .mapAsync(4, tweet -> ask(database, new Save(tweet), 300))
    .to(Sink.ignore());
```

Note that if the `ask` is not completed within the given timeout the stream is completed with failure. If that is not desired outcome you can use `recover` on the `ask` `Future`.

Illustrating ordering and parallelism

Let us look at another example to get a better understanding of the ordering and parallelism characteristics of `mapAsync` and `mapAsyncUnordered`.

Several `mapAsync` and `mapAsyncUnordered` futures may run concurrently. The number of concurrent futures are limited by the downstream demand. For example, if 5 elements have been requested by downstream there will be at most 5 futures in progress.

`mapAsync` emits the future results in the same order as the input elements were received. That means that completed results are only emitted downstream when earlier results have been completed and emitted. One slow call will thereby delay the results of all successive calls, even though they are completed before the slow call.

`mapAsyncUnordered` emits the future results as soon as they are completed, i.e. it is possible that the elements are not emitted downstream in the same order as received from upstream. One slow call will thereby not delay the results of faster successive calls as long as there is downstream demand of several elements.

Here is a fictive service that we can use to illustrate these aspects.

```
static class SometimesSlowService {
  private final ExecutionContext ec;

  public SometimesSlowService(ExecutionContext ec) {
    this.ec = ec;
  }

  private final AtomicInteger runningCount = new AtomicInteger();

  public Future<String> convert(String s) {
    System.out.println("running: " + s + "(" + runningCount.incrementAndGet() + ")");
    return Futures.future(() -> {
      if (!s.isEmpty() && Character.isLowerCase(s.charAt(0)))
        Thread.sleep(500);
      else
        Thread.sleep(20);
      System.out.println("completed: " + s + "(" + runningCount.decrementAndGet() + ")");
      return s.toUpperCase();
    }, ec);
  }
}
```

Elements starting with a lower case character are simulated to take longer time to process.

Here is how we can use it with `mapAsync`:

```
final MessageDispatcher blockingEc = system.dispatchers().lookup("blocking-dispatcher");
final SometimesSlowService service = new SometimesSlowService(blockingEc);
```

```
final ActorFlowMaterializer mat = ActorFlowMaterializer.create(
  ActorFlowMaterializerSettings.create(system).withInputBuffer(4, 4), system);

Source.from(Arrays.asList("a", "B", "C", "D", "e", "F", "g", "H", "i", "J"))
  .map(elem -> { System.out.println("before: " + elem); return elem; })
  .mapAsync(4, service::convert)
  .runForeach(elem -> System.out.println("after: " + elem), mat);
```

The output may look like this:

```
before: a
before: B
before: C
before: D
running: a (1)
running: B (2)
before: e
running: C (3)
before: F
running: D (4)
before: g
before: H
completed: C (3)
completed: B (2)
completed: D (1)
completed: a (0)
after: A
after: B
running: e (1)
after: C
after: D
running: F (2)
before: i
before: J
running: g (3)
running: H (4)
completed: H (2)
completed: F (3)
completed: e (1)
completed: g (0)
after: E
after: F
running: i (1)
after: G
after: H
running: J (2)
completed: J (1)
completed: i (0)
after: I
after: J
```

Note that `after` lines are in the same order as the `before` lines even though elements are `completed` in a different order. For example `H` is `completed` before `g`, but still emitted afterwards.

The numbers in parenthesis illustrates how many calls that are in progress at the same time. Here the downstream demand and thereby the number of concurrent calls are limited by the buffer size (4) of the `ActorFlowMaterializerSettings`.

Here is how we can use the same service with `mapAsyncUnordered`:

```
final MessageDispatcher blockingEc = system.dispatchers().lookup("blocking-dispatcher");
final SometimesSlowService service = new SometimesSlowService(blockingEc);

final ActorFlowMaterializer mat = ActorFlowMaterializer.create(
```

```
  ActorFlowMaterializerSettings.create(system).withInputBuffer(4, 4), system);

Source.from(Arrays.asList("a", "B", "C", "D", "e", "F", "g", "H", "i", "J"))
  .map(elem -> { System.out.println("before: " + elem); return elem; })
  .mapAsyncUnordered(4, service::convert)
  .runForeach(elem -> System.out.println("after: " + elem), mat);
```

The output may look like this:

```
before: a
before: B
before: C
before: D
running: a (1)
running: B (2)
before: e
running: C (3)
before: F
running: D (4)
before: g
before: H
completed: B (3)
completed: C (1)
completed: D (2)
after: B
after: D
running: e (2)
after: C
running: F (3)
before: i
before: J
completed: F (2)
after: F
running: g (3)
running: H (4)
completed: H (3)
after: H
completed: a (2)
after: A
running: i (3)
running: J (4)
completed: J (3)
after: J
completed: e (2)
after: E
completed: g (1)
after: G
completed: i (0)
after: I
```

Note that `after` lines are not in the same order as the `before` lines. For example `H` overtakes the slow `G`.

The numbers in parenthesis illustrates how many calls that are in progress at the same time. Here the downstream demand and thereby the number of concurrent calls are limited by the buffer size (4) of the `ActorFlowMaterializerSettings`.

1.8.3 Integrating with Reactive Streams

Reactive Streams defines a standard for asynchronous stream processing with non-blocking back pressure. It makes it possible to plug together stream libraries that adhere to the standard. Akka Streams is one such library.

An incomplete list of other implementations:

- Reactor (1.1+)
- RxJava
- Ratpack
- Slick

The two most important interfaces in Reactive Streams are the `Publisher` and `Subscriber`.

```
import org.reactivestreams.Publisher;
import org.reactivestreams.Subscriber;
```

Let us assume that a library provides a publisher of tweets:

```
Publisher<Tweet> tweets();
```

and another library knows how to store author handles in a database:

```
Subscriber<Author> storage();
```

Using an Akka Streams `Flow` we can transform the stream and connect those:

```
final Flow<Tweet, Author, BoxedUnit> authors = Flow.of(Tweet.class)
  .filter(t -> t.hashtags().contains(AKKA))
  .map(t -> t.author);

  Source.from(rs.tweets())
    .via(authors)
    .to(Sink.create(rs.storage()));
```

The `Publisher` is used as an input `Source` to the flow and the `Subscriber` is used as an output `Sink`.

A `Flow` can also be materialized to a `Subscriber`, `Publisher` pair:

```
final Pair<Subscriber<Tweet>, Publisher<Author>> ports =
  authors.runWith(
    Source.<Tweet>subscriber(),
    Sink.<Author>publisher(),
    mat);

final Subscriber<Tweet> subscriber = ports.first();
final Publisher<Author> publisher = ports.second();
```

A publisher can be connected to a subscriber with the `subscribe` method.

It is also possible to expose a `Source` as a `Publisher` by using the Publisher-`Sink`:

```
final Publisher<Author> authorPublisher =
  Source.from(rs.tweets()).via(authors).runWith(Sink.publisher(), mat);

authorPublisher.subscribe(rs.storage());
```

A publisher that is created with `Sink.publisher` only supports one subscriber. A second subscription attempt will be rejected with an `IllegalStateException`.

A publisher that supports multiple subscribers can be created with `Sink.fanoutPublisher` instead:

```
Subscriber<Author> storage();
Subscriber<Author> alert();
```

```
final Publisher<Author> authorPublisher =
  Source.from(rs.tweets())
    .via(authors)
    .runWith(Sink.fanoutPublisher(8, 16), mat);

authorPublisher.subscribe(rs.storage());
authorPublisher.subscribe(rs.alert());
```

The buffer size controls how far apart the slowest subscriber can be from the fastest subscriber before slowing down the stream.

To make the picture complete, it is also possible to expose a `Sink` as a `Subscriber` by using the Subscriber-`Source`:

```
final Subscriber<Author> storage = rs.storage();

final Subscriber<Tweet> tweetSubscriber =
  authors
    .to(Sink.create(storage))
    .runWith(Source.subscriber(), mat);

rs.tweets().subscribe(tweetSubscriber);
```

1.9 Error Handling

Strategies for how to handle exceptions from processing stream elements can be defined when materializing the stream. The error handling strategies are inspired by actor supervision strategies, but the semantics have been adapted to the domain of stream processing.

1.9.1 Supervision Strategies

There are three ways to handle exceptions from application code:

- `Stop` - The stream is completed with failure.
- `Resume` - The element is dropped and the stream continues.
- `Restart` - The element is dropped and the stream continues after restarting the stage. Restarting a stage means that any accumulated state is cleared. This is typically performed by creating a new instance of the stage.

By default the stopping strategy is used for all exceptions, i.e. the stream will be completed with failure when an exception is thrown.

```
final FlowMaterializer mat = ActorFlowMaterializer.create(system);
final Source<Integer, BoxedUnit> source = Source.from(Arrays.asList(0, 1, 2, 3, 4, 5))
  .map(elem -> 100 / elem);
final Sink<Integer, Future<Integer>> fold =
  Sink.fold(0, (acc, elem) -> acc + elem);
final Future<Integer> result = source.runWith(fold, mat);
// division by zero will fail the stream and the
// result here will be a Future completed with Failure(ArithmeticException)
```

The default supervision strategy for a stream can be defined on the settings of the materializer.

```
final Function<Throwable, Supervision.Directive> decider = exc -> {
  if (exc instanceof ArithmeticException)
    return Supervision.resume();
  else
    return Supervision.stop();
};
final FlowMaterializer mat = ActorFlowMaterializer.create(
  ActorFlowMaterializerSettings.create(system).withSupervisionStrategy(decider),
  system);
final Source<Integer, BoxedUnit> source = Source.from(Arrays.asList(0, 1, 2, 3, 4, 5))
  .map(elem -> 100 / elem);
final Sink<Integer, Future<Integer>> fold =
  Sink.fold(0, (acc, elem) -> acc + elem);
final Future<Integer> result = source.runWith(fold, mat);
```

```
// the element causing division by zero will be dropped
// result here will be a Future completed with Success(228)
```

Here you can see that all `ArithmeticException` will resume the processing, i.e. the elements that cause the division by zero are effectively dropped.

Note: Be aware that dropping elements may result in deadlocks in graphs with cycles, as explained in *Graph cycles, liveness and deadlocks*.

The supervision strategy can also be defined for all operators of a flow.

```
final FlowMaterializer mat = ActorFlowMaterializer.create(system);
final Function<Throwable, Supervision.Directive> decider = exc -> {
  if (exc instanceof ArithmeticException)
    return Supervision.resume();
  else
    return Supervision.stop();
};
final Flow<Integer, Integer, BoxedUnit> flow =
    Flow.of(Integer.class).filter(elem -> 100 / elem < 50).map(elem -> 100 / (5 - elem))
    .withAttributes(ActorOperationAttributes.withSupervisionStrategy(decider));
final Source<Integer, BoxedUnit> source = Source.from(Arrays.asList(0, 1, 2, 3, 4, 5))
  .via(flow);
final Sink<Integer, Future<Integer>> fold =
  Sink.fold(0, (acc, elem) -> acc + elem);
final Future<Integer> result = source.runWith(fold, mat);
// the elements causing division by zero will be dropped
// result here will be a Future completed with Success(150)
```

`Restart` works in a similar way as `Resume` with the addition that accumulated state, if any, of the failing processing stage will be reset.

```
final FlowMaterializer mat = ActorFlowMaterializer.create(system);
final Function<Throwable, Supervision.Directive> decider = exc -> {
  if (exc instanceof IllegalArgumentException)
    return Supervision.restart();
  else
    return Supervision.stop();
};
final Flow<Integer, Integer, BoxedUnit> flow =
  Flow.of(Integer.class).scan(0, (acc, elem) -> {
    if (elem < 0) throw new IllegalArgumentException("negative not allowed");
    else return acc + elem;
  })
  .withAttributes(ActorOperationAttributes.withSupervisionStrategy(decider));
final Source<Integer, BoxedUnit> source = Source.from(Arrays.asList(1, 3, -1, 5, 7))
  .via(flow);
final Future<List<Integer>> result = source.grouped(1000)
  .runWith(Sink.<List<Integer>>head(), mat);
// the negative element cause the scan stage to be restarted,
// i.e. start from 0 again
// result here will be a Future completed with Success(List(0, 1, 4, 0, 5, 12))
```

1.9.2 Errors from mapAsync

Stream supervision can also be applied to the futures of `mapAsync`.

Let's say that we use an external service to lookup email addresses and we would like to discard those that cannot be found.

We start with the tweet stream of authors:

```
final Source<Author, BoxedUnit> authors = tweets
  .filter(t -> t.hashtags().contains(AKKA))
  .map(t -> t.author);
```

Assume that we can lookup their email address using:

```
public Future<String> lookupEmail(String handle)
```

The `Future` is completed with `Failure` if the email is not found.

Transforming the stream of authors to a stream of email addresses by using the `lookupEmail` service can be done with `mapAsync` and we use `Supervision.getResumingDecider` to drop unknown email addresses:

```
final OperationAttributes resumeAttrib =
  ActorOperationAttributes.withSupervisionStrategy(Supervision.getResumingDecider());
final Flow<Author, String, BoxedUnit> lookupEmail =
    Flow.of(Author.class)
    .mapAsync(4, author -> addressSystem.lookupEmail(author.handle))
    .withAttributes(resumeAttrib);
final Source<String, BoxedUnit> emailAddresses = authors.via(lookupEmail);
```

If we would not use `Resume` the default stopping strategy would complete the stream with failure on the first `Future` that was completed with `Failure`.

1.10 Working with streaming IO

Akka Streams provides a way of handling File IO and TCP connections with Streams. While the general approach is very similar to the Actor based TCP handling using Akka IO, by using Akka Streams you are freed of having to manually react to back-pressure signals, as the library does it transparently for you.

1.10.1 Streaming TCP

Accepting connections: Echo Server

In order to implement a simple EchoServer we `bind` to a given address, which returns a `Source[IncomingConnection]`, which will emit an `IncomingConnection` element for each new connection that the Server should handle:

```
// IncomingConnection and ServerBinding imported from Tcp
final Source<IncomingConnection, Future<ServerBinding>> connections =
    Tcp.get(system).bind("127.0.0.1", 8889);
```

Next, we simply handle *each* incoming connection using a `Flow` which will be used as the processing stage to handle and emit ByteStrings from and to the TCP Socket. Since one `ByteString` does not have to necessarily correspond to exactly one line of text (the client might be sending the line in chunks) we use the `parseLines` recipe from the *cookbook-parse-lines-java* Akka Streams Cookbook recipe to chunk the inputs up into actual lines of text. In this example we simply add exclamation marks to each incoming text message and push it through the flow:

```
connections.runForeach(connection -> {
  System.out.println("New connection from: " + connection.remoteAddress());

  final Flow<ByteString, ByteString, BoxedUnit> echo = Flow.of(ByteString.class)
    .transform(() -> RecipeParseLines.parseLines("\n", 256))
    .map(s -> s + "!!!\n")
    .map(s -> ByteString.fromString(s));

  connection.handleWith(echo, mat);
}, mat);
```

Notice that while most building blocks in Akka Streams are reusable and freely shareable, this is *not* the case for the incoming connection Flow, since it directly corresponds to an existing, already accepted connection its handling can only ever be materialized *once*.

Closing connections is possible by cancelling the *incoming connection* `Flow` from your server logic (e.g. by connecting its downstream to an `CancelledSink` and its upstream to a *completed* `Source`). It is also possible to shut down the servers socket by cancelling the `connections:Source[IncomingConnection]`.

We can then test the TCP server by sending data to the TCP Socket using `netcat`:

```
$ echo -n "Hello World" | netcat 127.0.0.1 8889
Hello World!!!
```

Connecting: REPL Client

In this example we implement a rather naive Read Evaluate Print Loop client over TCP. Let's say we know a server has exposed a simple command line interface over TCP, and would like to interact with it using Akka Streams over TCP. To open an outgoing connection socket we use the `outgoingConnection` method:

```
final Flow<ByteString, ByteString, Future<OutgoingConnection>> connection =
    Tcp.get(system).outgoingConnection("127.0.0.1", 8889);

final PushStage<String, ByteString> replParser = new PushStage<String, ByteString>() {
  @Override public SyncDirective onPush(String elem, Context<ByteString> ctx) {
    if (elem.equals("q"))
      return ctx.pushAndFinish(ByteString.fromString("BYE\n"));
    else
      return ctx.push(ByteString.fromString(elem + "\n"));
  }
};

final Flow<ByteString, ByteString, BoxedUnit> repl = Flow.of(ByteString.class)
  .transform(() -> RecipeParseLines.parseLines("\n", 256))
  .map(text -> {System.out.println("Server: " + text); return "next";})
  .map(elem -> readLine("> "))
  .transform(() -> replParser);

connection.join(repl).run(mat);
```

The `repl` flow we use to handle the server interaction first prints the servers response, then awaits on input from the command line (this blocking call is used here just for the sake of simplicity) and converts it to a `ByteString` which is then sent over the wire to the server. Then we simply connect the TCP pipeline to this processing stage–at this point it will be materialized and start processing data once the server responds with an *initial message*.

A resilient REPL client would be more sophisticated than this, for example it should split out the input reading into a separate mapAsync step and have a way to let the server write more data than one ByteString chunk at any given time, these improvements however are left as exercise for the reader.

Avoiding deadlocks and liveness issues in back-pressured cycles

When writing such end-to-end back-pressured systems you may sometimes end up in a situation of a loop, in which *either side is waiting for the other one to start the conversation*. One does not need to look far to find examples of such back-pressure loops. In the two examples shown previously, we always assumed that the side we are connecting to would start the conversation, which effectively means both sides are back-pressured and can not get the conversation started. There are multiple ways of dealing with this which are explained in depth in *Graph cycles, liveness and deadlocks*, however in client-server scenarios it is often the simplest to make either side simply send an initial message.

Note: In case of back-pressured cycles (which can occur even between different systems) sometimes you have to decide which of the sides has start the conversation in order to kick it off. This can be often done by injecting an initial message from one of the sides–a conversation starter.

To break this back-pressure cycle we need to inject some initial message, a "conversation starter". First, we need to decide which side of the connection should remain passive and which active. Thankfully in most situations finding the right spot to start the conversation is rather simple, as it often is inherent to the protocol we are trying to implement using Streams. In chat-like applications, which our examples resemble, it makes sense to make the Server initiate the conversation by emitting a "hello" message:

```
connections.runForeach(connection -> {
  // server logic, parses incoming commands
  final PushStage<String, String> commandParser = new PushStage<String, String>() {
    @Override public SyncDirective onPush(String elem, Context<String> ctx) {
      if (elem.equals("BYE"))
        return ctx.finish();
      else
        return ctx.push(elem + "!");
    }
  };

  final String welcomeMsg = "Welcome to: " + connection.localAddress() +
      " you are: " + connection.remoteAddress() + "!\n";

  final Source<ByteString, BoxedUnit> welcome =
      Source.single(ByteString.fromString(welcomeMsg));
  final Flow<ByteString, ByteString, BoxedUnit> echoFlow =
      Flow.of(ByteString.class)
        .transform(() -> RecipeParseLines.parseLines("\n", 256))
        .transform(() -> commandParser)
        .map(s ->  s + "\n")
        .map(s -> ByteString.fromString(s));

  final Flow<ByteString, ByteString, BoxedUnit> serverLogic =
      Flow.factory().create(builder -> {
        final UniformFanInShape<ByteString, ByteString> concat =
            builder.graph(Concat.create());
        final FlowShape<ByteString, ByteString> echo = builder.graph(echoFlow);

        builder
          .from(welcome).to(concat)
          .from(echo).to(concat);

        return new Pair<>(echo.inlet(), concat.out());
  });

  connection.handleWith(serverLogic, mat);
}, mat);
```

The way we constructed a `Flow` using a `PartialFlowGraph` is explained in detail in *Constructing Sources, Sinks and Flows from Partial Graphs*, however the basic concepts is rather simple– we can encapsulate arbitrarily complex logic within a `Flow` as long as it exposes the same interface, which means exposing exactly one `UndefinedSink` and exactly one `UndefinedSource` which will be connected to the TCP pipeline. In this example we use a `Concat` graph processing stage to inject the initial message, and then continue with handling all incoming data using the echo handler. You should use this pattern of encapsulating complex logic in Flows and attaching those to `StreamIO` in order to implement your custom and possibly sophisticated TCP servers.

In this example both client and server may need to close the stream based on a parsed command command - `BYE` in the case of the server, and `q` in the case of the client. This is implemented by using a custom `PushStage` (see *Using PushPullStage*) which completes the stream once it encounters such command.

1.10.2 Streaming File IO

Akka Streams provide simple Sources and Sinks that can work with `ByteString` instances to perform IO operations on files.

Note: Since the current version of Akka (`2.3.x`) needs to support JDK6, the currently provided File IO implementations are not able to utilise Asynchronous File IO operations, as these were introduced in JDK7 (and newer). Once Akka is free to require JDK8 (from `2.4.x`) these implementations will be updated to make use of the new NIO APIs (i.e. `AsynchronousFileChannel`).

Streaming data from a file is as easy as defining a *SynchronousFileSource* given a target file, and an optional `chunkSize` which determines the buffer size determined as one "element" in such stream:

```
final File file = new File("example.csv");
  SynchronousFileSource.create(file)
    .runForeach(chunk -> System.out.println(chunk.utf8String()), mat);
```

Please note that these processing stages are backed by Actors and by default are configured to run on a pre-configured threadpool-backed dispatcher dedicated for File IO. This is very important as it isolates the blocking file IO operations from the rest of the ActorSystem allowing each dispatcher to be utilised in the most efficient way. If you want to configure a custom dispatcher for file IO operations globally, you can do so by changing the `akka.strea.file-io-dispatcher`, or for a specific stage by spefifying a custom Dispatcher in code, like this:

```
SynchronousFileSink.create(file)
  .withAttributes(ActorOperationAttributes.dispatcher("custom-file-io-dispatcher"));
```

1.11 Pipelining and Parallelism

Akka Streams processing stages (be it simple operators on Flows and Sources or graph junctions) are executed concurrently by default. This is realized by mapping each of the processing stages to a dedicated actor internally.

We will illustrate through the example of pancake cooking how streams can be used for various processing patterns, exploiting the available parallelism on modern computers. The setting is the following: both Patrik and Roland like to make pancakes, but they need to produce sufficient amount in a cooking session to make all of the children happy. To increase their pancake production throughput they use two frying pans. How they organize their pancake processing is markedly different.

1.11.1 Pipelining

Roland uses the two frying pans in an asymmetric fashion. The first pan is only used to fry one side of the pancake then the half-finished pancake is flipped into the second pan for the finishing fry on the other side. Once the first frying pan becomes available it gets a new scoop of batter. As an effect, most of the time there are two pancakes being cooked at the same time, one being cooked on its first side and the second being cooked to completion. This is how this setup would look like implemented as a stream:

```
Flow<ScoopOfBatter, HalfCookedPancake, BoxedUnit> fryingPan1 =
  Flow.of(ScoopOfBatter.class).map(batter -> new HalfCookedPancake());

Flow<HalfCookedPancake, Pancake, BoxedUnit> fryingPan2 =
  Flow.of(HalfCookedPancake.class).map(halfCooked -> new Pancake());

// With the two frying pans we can fully cook pancakes
Flow<ScoopOfBatter, Pancake, BoxedUnit> pancakeChef = fryingPan1.via(fryingPan2);
```

The two `map` stages in sequence (encapsulated in the "frying pan" flows) will be executed in a pipelined way, basically doing the same as Roland with his frying pans:

1. A `ScoopOfBatter` enters `fryingPan1`

2. `fryingPan1` emits a HalfCookedPancake once `fryingPan2` becomes available
3. `fryingPan2` takes the HalfCookedPancake
4. at this point fryingPan1 already takes the next scoop, without waiting for fryingPan2 to finish

The benefit of pipelining is that it can be applied to any sequence of processing steps that are otherwise not parallelisable (for example because the result of a processing step depends on all the information from the previous step). One drawback is that if the processing times of the stages are very different then some of the stages will not be able to operate at full throughput because they will wait on a previous or subsequent stage most of the time. In the pancake example frying the second half of the pancake is usually faster than frying the first half, `fryingPan2` will not be able to operate at full capacity [1].

Stream processing stages have internal buffers to make communication between them more efficient. For more details about the behavior of these and how to add additional buffers refer to *stream-rate-scala*.

1.11.2 Parallel processing

Patrik uses the two frying pans symmetrically. He uses both pans to fully fry a pancake on both sides, then puts the results on a shared plate. Whenever a pan becomes empty, he takes the next scoop from the shared bowl of batter. In essence he parallelizes the same process over multiple pans. This is how this setup will look like if implemented using streams:

```
Flow<ScoopOfBatter, Pancake, BoxedUnit> fryingPan =
  Flow.of(ScoopOfBatter.class).map(batter -> new Pancake());

Flow<ScoopOfBatter, Pancake, BoxedUnit> pancakeChef =
  Flow.factory().create(b -> {
    final UniformFanInShape<Pancake, Pancake> mergePancakes =
      b.graph(Merge.create(2));
    final UniformFanOutShape<ScoopOfBatter, ScoopOfBatter> dispatchBatter =
      b.graph(Balance.create(2));

    // Using two frying pans in parallel, both fully cooking a pancake from the batter.
    // We always put the next scoop of batter to the first frying pan that becomes available.
    b.from(dispatchBatter.out(0)).via(fryingPan).to(mergePancakes.in(0));
    // Notice that we used the "fryingPan" flow without importing it via builder.add().
    // Flows used this way are auto-imported, which in this case means that the two
    // uses of "fryingPan" mean actually different stages in the graph.
    b.from(dispatchBatter.out(1)).via(fryingPan).to(mergePancakes.in(1));

    return new Pair(dispatchBatter.in(), mergePancakes.out());
  });
```

The benefit of parallelizing is that it is easy to scale. In the pancake example it is easy to add a third frying pan with Patrik's method, but Roland cannot add a third frying pan, since that would require a third processing step, which is not practically possible in the case of frying pancakes.

One drawback of the example code above that it does not preserve the ordering of pancakes. This might be a problem if children like to track their "own" pancakes. In those cases the `Balance` and `Merge` stages should be replaced by strict-round robing balancing and merging stages that put in and take out pancakes in a strict order.

A more detailed example of creating a worker pool can be found in the cookbook: *cookbook-balance-scala*

1.11.3 Combining pipelining and parallel processing

The two concurrency patterns that we demonstrated as means to increase throughput are not exclusive. In fact, it is rather simple to combine the two approaches and streams provide a nice unifying language to express and compose them.

[1] Roland's reason for this seemingly suboptimal procedure is that he prefers the temperature of the second pan to be slightly lower than the first in order to achieve a more homogeneous result.

First, let's look at how we can parallelize pipelined processing stages. In the case of pancakes this means that we will employ two chefs, each working using Roland's pipelining method, but we use the two chefs in parallel, just like Patrik used the two frying pans. This is how it looks like if expressed as streams:

```
Flow<ScoopOfBatter, Pancake, BoxedUnit> pancakeChef =
  Flow.factory().create(b -> {
    final UniformFanInShape<Pancake, Pancake> mergePancakes =
      b.graph(Merge.create(2));
    final UniformFanOutShape<ScoopOfBatter, ScoopOfBatter> dispatchBatter =
      b.graph(Balance.create(2));

    // Using two pipelines, having two frying pans each, in total using
    // four frying pans
    b.from(dispatchBatter.out(0))
      .via(fryingPan1)
      .via(fryingPan2)
      .to(mergePancakes.in(0));

    b.from(dispatchBatter.out(1))
      .via(fryingPan1)
      .via(fryingPan2)
      .to(mergePancakes.in(1));

    return new Pair(dispatchBatter.in(), mergePancakes.out());
  });
```

The above pattern works well if there are many independent jobs that do not depend on the results of each other, but the jobs themselves need multiple processing steps where each step builds on the result of the previous one. In our case individual pancakes do not depend on each other, they can be cooked in parallel, on the other hand it is not possible to fry both sides of the same pancake at the same time, so the two sides have to be fried in sequence.

It is also possible to organize parallelized stages into pipelines. This would mean employing four chefs:

- the first two chefs prepare half-cooked pancakes from batter, in parallel, then putting those on a large enough flat surface.
- the second two chefs take these and fry their other side in their own pans, then they put the pancakes on a shared plate.

This is again straightforward to implement with the streams API:

```
Flow<ScoopOfBatter, HalfCookedPancake, BoxedUnit> pancakeChefs1 =
  Flow.factory().create(b -> {
    final UniformFanInShape<HalfCookedPancake, HalfCookedPancake> mergeHalfCooked =
      b.graph(Merge.create(2));
    final UniformFanOutShape<ScoopOfBatter, ScoopOfBatter> dispatchBatter =
      b.graph(Balance.create(2));

    // Two chefs work with one frying pan for each, half-frying the pancakes then putting
    // them into a common pool
    b.from(dispatchBatter.out(0)).via(fryingPan1).to(mergeHalfCooked.in(0));
    b.from(dispatchBatter.out(1)).via(fryingPan1).to(mergeHalfCooked.in(1));

    return new Pair(dispatchBatter.in(), mergeHalfCooked.out());
  });

Flow<HalfCookedPancake, Pancake, BoxedUnit> pancakeChefs2 =
  Flow.factory().create(b -> {
    final UniformFanInShape<Pancake, Pancake> mergePancakes =
      b.graph(Merge.create(2));
    final UniformFanOutShape<HalfCookedPancake, HalfCookedPancake> dispatchHalfCooked =
      b.graph(Balance.create(2));

    // Two chefs work with one frying pan for each, finishing the pancakes then putting
    // them into a common pool
```

```
    b.from(dispatchHalfCooked.out(0)).via(fryingPan2).to(mergePancakes.in(0));
    b.from(dispatchHalfCooked.out(1)).via(fryingPan2).to(mergePancakes.in(1));

    return new Pair(dispatchHalfCooked.in(), mergePancakes.out());
  });

Flow<ScoopOfBatter, Pancake, BoxedUnit> kitchen =
    pancakeChefs1.via(pancakeChefs2);
```

This usage pattern is less common but might be usable if a certain step in the pipeline might take wildly different times to finish different jobs. The reason is that there are more balance-merge steps in this pattern compared to the parallel pipelines. This pattern rebalances after each step, while the previous pattern only balances at the entry point of the pipeline. This only matters however if the processing time distribution has a large deviation.

1.12 Testing streams

Akka Streams comes with an `akka-stream-testkit` module that provides tools which can be used for controlling and asserting various parts of the stream pipeline.

1.12.1 Probe Sink

Using probe as a *Sink* allows manual control over demand and assertions over elements coming downstream. Streams testkit provides a sink that materializes to a `TestSubscriber.Probe`.

```
Source
  .from(Arrays.asList(1, 2, 3, 4))
  .filter(elem -> elem % 2 == 0)
  .map(elem -> elem * 2)
  .runWith(TestSink.probe(system), mat)
  .request(2)
  .expectNext(4, 8)
  .expectComplete();
```

1.12.2 Probe Source

A source that materializes to `TestPublisher.Probe` can be used for asserting demand or controlling when stream is completed or ended with an error.

```
TestSource.<Integer>probe(system)
  .to(Sink.cancelled(), Keep.left())
  .run(mat)
  .expectCancellation();
```

TODO

List by example various operations on probes. Using probes without a sink.

1.13 Overview of built-in stages and their semantics

All stages by default backpressure if the computation they encapsulate is not fast enough to keep up with the rate of incoming elements from the preceding stage. There are differences though how the different stages handle when some of their downstream stages backpressure them. This table provides a summary of all built-in stages and their semantics.

All stages stop and propagate the failure downstream as soon as any of their upstreams emit a failure unless supervision is used. This happens to ensure reliable teardown of streams and cleanup when failures happen.

Failures are meant to be to model unrecoverable conditions, therefore they are always eagerly propagated. For in-band error handling of normal errors (dropping elements if a map fails for example) you should use the upervision support, or explicitly wrap your element types in a proper container that can express error or success states (for example `Try` in Scala).

Custom components are not covered by this table since their semantics are defined by the user.

1.13.1 Simple processing stages

These stages are all expressible as a `PushPullStage`. These stages can transform the rate of incoming elements since there are stages that emit multiple elements for a single input (e.g. *mapConcat') or consume multiple elements before emitting one output (e.g. ''filter'*). However, these rate transformations are data-driven, i.e. it is the incoming elements that define how the rate is affected. This is in contrast with *detached-stages-overview* which can change their processing behavior depending on being backpressured by downstream or not.

Stage	Emits when	Backpressures when	Completes when
map	the mapping function returns an element	downstream backpressures	upstream completes
mapConcat	the mapping function returns an element or there are still remaining elements from the previously calculated collection	downstream backpressures or there are still available elements from the previously calculated collection	upstream completes and all remaining elements has been emitted
filter	the given predicate returns true for the element	the given predicate returns true for the element and downstream backpressures	upstream completes
collect	the provided partial function is defined for the element	the partial function is defined for the element and downstream backpressures	upstream completes
grouped	the specified number of elements has been accumulated or upstream completed	a group has been assembled and downstream backpressures	upstream completes
scan	the function scanning the element returns a new element	downstream backpressures	upstream completes
drop	the specified number of elements has been dropped already	the specified number of elements has been dropped and downstream backpressures	upstream completes
take	the specified number of elements to take has not yet been reached	downstream backpressures	the defined number of elements has been taken or upstream completes

1.13.2 Asynchronous processing stages

These stages encapsulate an asynchronous computation, properly handling backpressure while taking care of the asynchronous operation at the same time (usually handling the completion of a Future).

It is currently not possible to build custom asynchronous processing stages

Stage	Emits when	Backpressures when	Completes when
mapAsync	the Future returned by the provided function finishes for the next element in sequence	the number of futures reaches the configured parallelism and the downstream backpressures	upstream completes and all futures has been completed and all elements has been emitted [2]
mapAsyncUnordered	any of the Futures returned by the provided function complete	the number of futures reaches the configured parallelism and the downstream backpressures	upstream completes and all futures has been completed and all elements has been emitted [1]

1.13.3 Timer driven stages

These stages process elements using timers, delaying, dropping or grouping elements for certain time durations.

Stage	Emits when	Backpressures when	Completes when
takeWithin	an upstream element arrives	downstream backpressures	upstream completes or timer fires
dropWithin	after the timer fired and a new upstream element arrives	downstream backpressures	upstream completes
groupedWithin	the configured time elapses since the last group has been emitted	the group has been assembled (the duration elapsed) and downstream backpressures	upstream completes

It is currently not possible to build custom timer driven stages

1.13.4 Backpressure aware stages

These stages are all expressible as a `DetachedStage`. These stages are aware of the backpressure provided by their downstreams and able to adapt their behavior to that signal.

Stage	Emits when	Backpressures when	Completes when
conflate	downstream stops backpressuring and there is a conflated element available	never [3]	upstream completes
expand	downstream stops backpressuring	downstream backpressures	upstream completes
buffer (Backpressure)	downstream stops backpressuring and there is a pending element in the buffer	buffer is full	upstream completes and buffered elements has been drained
buffer (DropX)	downstream stops backpressuring and there is a pending element in the buffer	never [2]	upstream completes and buffered elements has been drained
buffer (Fail)	downstream stops backpressuring and there is a pending element in the buffer	fails the stream instead of backpressuring when buffer is full	upstream completes and buffered elements has been drained

1.13.5 Nesting and flattening stages

These stages either take a stream and turn it into a stream of streams (nesting) or they take a stream that contains nested streams and turn them into a stream of elements instead (flattening).

It is currently not possible to build custom nesting or flattening stages

[2] If a Future fails, the stream also fails (unless a different supervision strategy is applied)
[3] Except if the encapsulated computation is not fast enough

Stage	Emits when	Backpressures when	Completes when
prefixAndTail	the configured number of prefix elements are available. Emits this prefix, and the rest as a substream	downstream backpressures or substream backpressures	prefix elements has been consumed and substream has been consumed
groupBy	an element for which the grouping function returns a group that has not yet been created. Emits the new group	there is an element pending for a group whose substream backpressures	upstream completes [4]
splitWhen	an element for which the provided predicate is true, opening and emitting a new substream for subsequent elements	there is an element pending for the next substream, but the previous is not fully consumed yet, or the substream backpressures	upstream completes [3]
flatten (Concat)	the current consumed substream has an element available	downstream backpressures	upstream completes and all consumed substreams complete

1.13.6 Fan-in stages

Most of these stages can be expressible as a `FlexiMerge`. These stages take multiple streams as their input and provide a single output combining the elements from all of the inputs in different ways.

The custom fan-in stages that can be built currently are limited

Stage	Emits when	Backpressures when	Completes when
merge	one of the inputs has an element available	downstream backpressures	all upstreams complete
mergePreferred	one of the inputs has an element available, preferring a defined input if multiple have elements available	downstream backpressures	all upstreams complete
zip	all of the inputs has an element available	downstream backpressures	any upstream completes
zipWith	all of the inputs has an element available	downstream backpressures	any upstream completes
concat	the current stream has an element available; if the current input completes, it tries the next one	downstream backpressures	all upstreams complete

1.13.7 Fan-out stages

Most of these stages can be expressible as a `FlexiRoute`. These have one input and multiple outputs. They might route the elements between different outputs, or emit elements on multiple outputs at the same time.

The custom fan-out stages that can be built currently are limited

Stage	Emits when	Backpressures when	Completes when
unzip	all of the outputs stops backpressuring and there is an input element available	any of the outputs backpressures	upstream completes
broadcast	all of the outputs stops backpressuring and there is an input element available	any of the outputs backpressures	upstream completes
balance	any of the outputs stops backpressuring; emits the element to the first available output	all of the outputs backpressure	upstream completes

[4] Until the end of stream it is not possible to know whether new substreams will be needed or not

1.14 Configuration

```
#####################################
# Akka Stream Reference Config File #
#####################################

akka {
  stream {

    # Default flow materializer settings
    materializer {

      # Initial size of buffers used in stream elements
      initial-input-buffer-size = 4
      # Maximum size of buffers used in stream elements
      max-input-buffer-size = 16

      # Fully qualified config path which holds the dispatcher configuration
      # to be used by FlowMaterialiser when creating Actors.
      # When this value is left empty, the default-dispatcher will be used.
      dispatcher = ""

      # Cleanup leaked publishers and subscribers when they are not used within a given
      # deadline
      subscription-timeout {
        # when the subscription timeout is reached one of the following strategies on
        # the "stale" publisher:
        # cancel - cancel it (via `onError` or subscribing to the publisher and
        #          `cancel()`ing the subscription right away
        # warn   - log a warning statement about the stale element (then drop the
        #          reference to it)
        # noop   - do nothing (not recommended)
        mode = cancel

        # time after which a subscriber / publisher is considered stale and eligible
        # for cancelation (see `akka.stream.subscription-timeout.mode`)
        timeout = 5s
      }

      # Enable additional troubleshooting logging at DEBUG log level
      debug-logging = off

      # Maximum number of elements emitted in batch if downstream signals large demand
      output-burst-limit = 1000
    }

    # Fully qualified config path which holds the dispatcher configuration
    # to be used by FlowMaterialiser when creating Actors for IO operations,
    # such as FileSource, FileSink and others.
    file-io-dispatcher = "akka.stream.default-file-io-dispatcher"

    default-file-io-dispatcher {
      type = "Dispatcher"
      executor = "thread-pool-executor"
      throughput = 1

      thread-pool-executor {
        core-pool-size-min = 2
        core-pool-size-factor = 2.0
        core-pool-size-max = 16
      }
    }
```

```
    }
}
```