Akka Stream and HTTP Experimental Scala 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 *Streams Cookbook*.
- 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:

```
final case class Author(handle: String)

final case class Hashtag(name: String)

final case class Tweet(author: Author, timestamp: Long, body: String) {
  def hashtags: Set[Hashtag] =
    body.split(" ").collect { case t if t.startsWith("#") => Hashtag(t) }.toSet
}

val akka = 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:

```
implicit val system = ActorSystem("reactive-tweets")
implicit val materializer = ActorFlowMaterializer()
```

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[Out, M]`:

```
val tweets: Source[Tweet, Unit]
```

Streams always start flowing from a `Source[Out,M1]` then can continue through `Flow[In,Out,M2]` elements or more advanced graph elements to finally be consumed by a `Sink[In,M3]` (ignore the type parameters `M1`, `M2` and `M3` for now, they are not relevant to the types of the elements produced/consumed by these classes).

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:

```
val authors: Source[Author, Unit] =
  tweets
    .filter(_.hashtags.contains(akka))
    .map(_.author)
```

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

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

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

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

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

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:

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

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

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

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:

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

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

Note: The `~>` (read as "edge", "via" or "to") operator is only available if `FlowGraph.Implicits._` are imported. Without this import you can still construct graphs using the `builder.addEdge(from,[through,]to)` method.

As you can see, inside the `FlowGraph` we use an implicit graph builder to mutably construct the graph using the `~>` "edge operator" (also read as "connect" or "via" or "to"). 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 partial graphs 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(slowComputation)
  .runWith(Sink.ignore)
```

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 errors 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 see how the types look like:

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

val counter: RunnableFlow[Future[Int]] = tweets.map(t => 1).toMat(sumSink)(Keep.right)

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

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

First, we prepare the `FoldSink` which will be used to sum all `Int` 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 using `toMat` the previously prepared Sink. Remember those mysterious type parameters on `Source` `Flow` and `Sink`? They represent the type of values these processing parts return when materialized. When you chain these together, you can explicitly combine their materialized values: in our example we used the `Keep.right` predefined function, which tells the implementation to only care about the materialized type of the stage currently appended to the right. As you can notice, the materialized type of sumSink is `Future[Int]` and because of using `Keep.right`, the resulting `RunnableFlow` has also a type parameter of `Future[Int]`.

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

A `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:

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

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

Many elements in Akka Streams provide materialized values which can be used for obtaining either results of computation or steering these elements which will be discussed in detail in *Stream Materialization*. Summing up this section, now we know what happens behind the scenes when we run this one-liner, which is equivalent to the multi line version above:

```
val sum: Future[Int] = tweets.map(t => 1).runWith(sumSink)
```

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

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.

```
val source = Source(1 to 10)
val sink = Sink.fold[Int, Int](0)(_ + _)

// connect the Source to the Sink, obtaining a RunnableFlow
val runnable: RunnableFlow[Future[Int]] = source.toMat(sink)(Keep.right)

// materialize the flow and get the value of the FoldSink
val sum: Future[Int] = runnable.run()
```

After running (materializing) the `RunnableFlow[T]` we get back the materialized value of type T. Every stream processing stage can produce a materialized value, and it is the responsibility of the user to combine them to a new type. In the above example we used `toMat` to indicate that we want to transform the materialized value of the source and sink, and we used the convenience function `Keep.right` to say that we are only interested in the materialized value of the sink. In our example the `FoldSink` materializes a value of type `Future` 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).

```
val source = Source(1 to 10)
val sink = Sink.fold[Int, Int](0)(_ + _)

// materialize the flow, getting the Sinks materialized value
val sum: Future[Int] = source.runWith(sink)
```

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:

```
val source = Source(1 to 10)
source.map(_ => 0) // has no effect on source, since it's immutable
source.runWith(Sink.fold(0)(_ + _)) // 55

val zeroes = source.map(_ => 0) // returns new Source[Int], with `map()` appended
zeroes.runWith(Sink.fold(0)(_ + _)) // 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 materialized value will also be calculated anew for each such materialization, usually leading to different values being returned each time. 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
val sink = Sink.fold[Int, Int](0)(_ + _)
val runnable: RunnableFlow[Future[Int]] =
  Source(1 to 10).toMat(sink)(Keep.right)

// get the materialized value of the FoldSink
val sum1: Future[Int] = runnable.run()
val sum2: Future[Int] = runnable.run()

// 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
Source(List(1, 2, 3))

// Create a source from a Future
Source(Future.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 as its materialized value
Sink.fold[Int, Int](0)(_ + _)

// Sink that returns a Future as its materialized value,
// 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[String](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(1 to 6).via(Flow[Int].map(_ * 2)).to(Sink.foreach(println(_)))

// Starting from a Source
val source = Source(1 to 6).map(_ * 2)
source.to(Sink.foreach(println(_)))

// Starting from a Sink
val sink: Sink[Int, Unit] = Flow[Int].map(_ * 2).to(Sink.foreach(println(_)))
Source(1 to 6).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 `scala.Option` or `scala.util.Either`.

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(n:Int)` 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-scala* 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
val source: Source[Int, Promise[Unit]] = Source.lazyEmpty[Int]

// A flow that internally throttles elements to 1/second, and returns a Cancellable
// which can be used to shut down the stream
val flow: Flow[Int, Int, Cancellable] = throttler

// A sink that returns the first element of a stream in the returned Future
val sink: Sink[Int, Future[Int]] = Sink.head[Int]

// By default, the materialized value of the leftmost stage is preserved
val r1: RunnableFlow[Promise[Unit]] = source.via(flow).to(sink)

// Simple selection of materialized values by using Keep.right
val r2: RunnableFlow[Cancellable] = source.viaMat(flow)(Keep.right).to(sink)
val r3: RunnableFlow[Future[Int]] = source.via(flow).toMat(sink)(Keep.right)

// Using runWith will always give the materialized values of the stages added
// by runWith() itself
val r4: Future[Int] = source.via(flow).runWith(sink)
val r5: Promise[Unit] = flow.to(sink).runWith(source)
val r6: (Promise[Unit], Future[Int]) = flow.runWith(source, sink)

// Using more complext combinations
val r7: RunnableFlow[(Promise[Unit], Cancellable)] =
  source.viaMat(flow)(Keep.both).to(sink)

val r8: RunnableFlow[(Promise[Unit], Future[Int])] =
  source.via(flow).toMat(sink)(Keep.both)

val r9: RunnableFlow[((Promise[Unit], Cancellable), Future[Int])] =
  source.viaMat(flow)(Keep.both).toMat(sink)(Keep.both)

val r10: RunnableFlow[(Cancellable, Future[Int])] =
  source.viaMat(flow)(Keep.right).toMat(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
val r11: RunnableFlow[(Promise[Unit], Cancellable, Future[Int])] =
  r9.mapMaterialized {
    case ((promise, cancellable), future) =>
      (promise, cancellable, future)
  }

// Now we can use pattern matching to get the resulting materialized values
val (promise, cancellable, future) = r11.run()

// Type inference works as expected
promise.success(0)
cancellable.cancel()
```

```
future.map(_ + 3)

// The result of r11 can be also achieved by using the Graph API
val r12: RunnableFlow[(Promise[Unit], Cancellable, Future[Int])] =
  FlowGraph.closed(source, flow, sink)((_, _, _)) { implicit builder =>
    (src, f, dst) =>
      import FlowGraph.Implicits._
      src ~> f ~> dst
  }
```

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

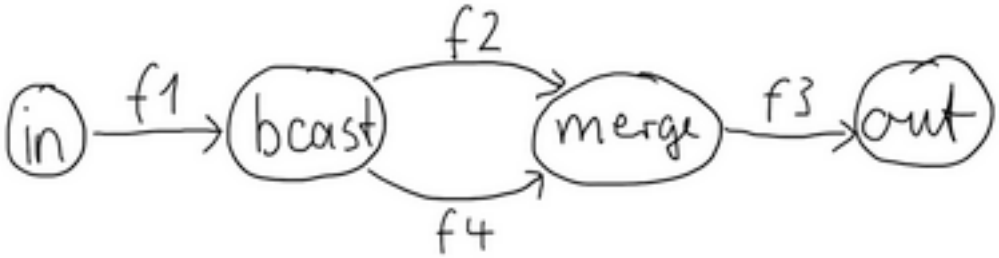
Flow graphs are built from simple Flows which serve as the linear connections within the graphs as well as junctions which serve as fan-in and fan-out points for Flows. Thanks to the junctions having meaningful types based on their behaviour and making them explicit elements these elements should be 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`. Junctions must always be created with defined type parameters, as otherwise the `Nothing` type will be inferred.

```
val g = FlowGraph.closed() { implicit builder: FlowGraph.Builder[Unit] =>
  import FlowGraph.Implicits._
  val in = Source(1 to 10)
  val out = Sink.ignore

  val bcast = builder.add(Broadcast[Int](2))
  val merge = builder.add(Merge[Int](2))

  val f1, f2, f3, f4 = Flow[Int].map(_ + 10)

  in ~> f1 ~> bcast ~> f2 ~> merge ~> f3 ~> out
              bcast ~> f4 ~> 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).

Notice the `import FlowGraph.Implicits._` which brings into scope the `~>` operator (read as "edge", "via" or "to") and its inverted counterpart `<~` (for noting down flows in the opposite direction where appropriate). It is also possible to construct graphs without the `~>` operator in case you prefer to use the graph builder explicitly:

```
val g = FlowGraph.closed() { builder: FlowGraph.Builder[Unit] =>
  val in = Source(1 to 10)
  val out = Sink.ignore

  val broadcast = builder.add(Broadcast[Int](2))
  val merge = builder.add(Merge[Int](2))

  val f1 = Flow[Int].map(_ + 10)
  val f3 = Flow[Int].map(_.toString)
  val f2 = Flow[Int].map(_ + 20)

  builder.addEdge(builder.add(in), broadcast.in)
  builder.addEdge(broadcast.out(0), f1, merge.in(0))
  builder.addEdge(broadcast.out(1), f2, merge.in(1))
  builder.addEdge(merge.out, f3, builder.add(out))
}
```

By looking at the snippets above, it should be apparent that the `FlowGraph.Builder` object is *mutable*. It is used (implicitly) by the `~>` operator, also making it a mutable operation as well. 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 `FlowGraph` 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.add(...)`, 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.add(...)` ignores the materialized value of the imported graph while importing via the factory method allows its inclusion; for more details see *Stream Materialization*.

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:

```
val topHeadSink = Sink.head[Int]
val bottomHeadSink = Sink.head[Int]
val sharedDoubler = Flow[Int].map(_ * 2)

FlowGraph.closed(topHeadSink, bottomHeadSink)((_, _)) { implicit builder =>
  (topHS, bottomHS) =>
  import FlowGraph.Implicits._
  val broadcast = builder.add(Broadcast[Int](2))
  Source.single(1) ~> broadcast.in

  broadcast.out(0) ~> sharedDoubler ~> topHS.inlet
  broadcast.out(1) ~> sharedDoubler ~> bottomHS.inlet
}
```

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.partial` instead of `FlowGraph.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).

```
val pickMaxOfThree = FlowGraph.partial() { implicit b =>
  import FlowGraph.Implicits._

  val zip1 = b.add(ZipWith[Int, Int, Int](math.max _))
  val zip2 = b.add(ZipWith[Int, Int, Int](math.max _))
  zip1.out ~> zip2.in0

  UniformFanInShape(zip2.out, zip1.in0, zip1.in1, zip2.in1)
}

val resultSink = Sink.head[Int]

val g = FlowGraph.closed(resultSink) { implicit b =>
  sink =>
    import FlowGraph.Implicits._

    // importing the partial graph will return its shape (inlets & outlets)
    val pm3 = b.add(pickMaxOfThree)

    Source.single(1) ~> pm3.in(0)
    Source.single(2) ~> pm3.in(1)
    Source.single(3) ~> pm3.in(2)
    pm3.out ~> sink.inlet
}

val max: Future[Int] = g.run()
Await.result(max, 300.millis) should equal(3)
```

As you can see, first we construct the partial graph that contains all the zipping and comparing of stream elements. This partial graph will have three inputs and one output, wherefore we use the `UniformFanInShape`. Then we import it (all of its nodes and connections) explicitly into the closed graph built in the second step 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 partial flow graph 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[T]`. 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:

```
val pairs = Source() { implicit b =>
  import FlowGraph.Implicits._

  // prepare graph elements
  val zip = b.add(Zip[Int, Int]())
  def ints = Source(() => Iterator.from(1))

  // connect the graph
  ints.filter(_ % 2 != 0) ~> zip.in0
  ints.filter(_ % 2 == 0) ~> zip.in1

  // expose port
  zip.out
}

val firstPair: Future[(Int, Int)] = pairs.runWith(Sink.head)
```

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 inlet and an outlet:

```
val pairUpWithToString = Flow() { implicit b =>
  import FlowGraph.Implicits._

  // prepare graph elements
  val broadcast = b.add(Broadcast[Int](2))
  val zip = b.add(Zip[Int, String]())

  // connect the graph
  broadcast.out(0).map(identity) ~> zip.in0
  broadcast.out(1).map(_.toString) ~> zip.in1

  // expose ports
  (broadcast.in, zip.out)
}

pairUpWithToString.runWith(Source(List(1)), Sink.head)
```

1.5.4 Building reusable Graph components

It is possible to build reusable, encapsulated components of arbitrary input and output ports using the graph DSL.

As an example, we will build a graph junction that represents a pool of workers, where a worker is expressed as a `Flow[I,O,_]`, i.e. a simple transformation of jobs of type `I` to results of type `O` (as you have seen already, this flow can actually contain a complex graph inside). Our reusable worker pool junction will not preserve the order of the incoming jobs (they are assumed to have a proper ID field) and it will use a `Balance` junction to schedule jobs to available workers. On top of this, our junction will feature a "fastlane", a dedicated port where jobs of higher priority can be sent.

Altogether, our junction will have two input ports of type `I` (for the normal and priority jobs) and an output port of type `O`. To represent this interface, we need to define a custom `Shape`. The following lines show how to do that.

```
// A shape represents the input and output ports of a reusable
// processing module
case class PriorityWorkerPoolShape[In, Out](
  jobsIn: Inlet[In],
  priorityJobsIn: Inlet[In],
  resultsOut: Outlet[Out]) extends Shape {

  // It is important to provide the list of all input and output
```

```
  // ports with a stable order. Duplicates are not allowed.
  override val inlets: immutable.Seq[Inlet[_]] =
    jobsIn :: priorityJobsIn :: Nil
  override val outlets: immutable.Seq[Outlet[_]] =
    resultsOut :: Nil

  // A Shape must be able to create a copy of itself. Basically
  // it means a new instance with copies of the ports
  override def deepCopy() = PriorityWorkerPoolShape(
    new Inlet[In](jobsIn.toString),
    new Inlet[In](priorityJobsIn.toString),
    new Outlet[Out](resultsOut.toString))

  // A Shape must also be able to create itself from existing ports
  override def copyFromPorts(
    inlets: immutable.Seq[Inlet[_]],
    outlets: immutable.Seq[Outlet[_]]) = {
    assert(inlets.size == this.inlets.size)
    assert(outlets.size == this.outlets.size)
    // This is why order matters when overriding inlets and outlets
    PriorityWorkerPoolShape(inlets(0), inlets(1), outlets(0))
  }
}
```

In general a custom `Shape` needs to be able to provide all its input and output ports, be able to copy itself, and also be able to create a new instance from given ports. There are some predefined shapes provided to avoid unnecessary boilerplate

- `SourceShape`, `SinkShape`, `FlowShape` for simpler shapes,
- `UniformFanInShape` and `UniformFanOutShape` for junctions with multiple input (or output) ports of the same type,
- `FanInShape1`, `FanInShape2`, ..., `FanOutShape1`, `FanOutShape2`, ... for junctions with multiple input (or output) ports of different types.

Since our shape has two input ports and one output port, we can just use the `FanInShape` DSL to define our custom shape:

```
import FanInShape.Name
import FanInShape.Init

class PriorityWorkerPoolShape2[In, Out](_init: Init[Out] = Name("PriorityWorkerPool"))
  extends FanInShape[Out](_init) {
  protected override def construct(i: Init[Out]) = new PriorityWorkerPoolShape2(i)

  val jobsIn = newInlet[In]("jobsIn")
  val priorityJobsIn = newInlet[In]("priorityJobsIn")
  // Outlet[Out] with name "out" is automatically created
}
```

Now that we have a `Shape` we can wire up a Graph that represents our worker pool. First, we will merge incoming normal and priority jobs using `MergePreferred`, then we will send the jobs to a `Balance` junction which will fan-out to a configurable number of workers (flows), finally we merge all these results together and send them out through our only output port. This is expressed by the following code:

```
object PriorityWorkerPool {
  def apply[In, Out](
    worker: Flow[In, Out, Any],
    workerCount: Int): Graph[PriorityWorkerPoolShape[In, Out], Unit] = {

    FlowGraph.partial() { implicit b ⇒
      import FlowGraph.Implicits._
```

```
      val priorityMerge = b.add(MergePreferred[In](1))
      val balance = b.add(Balance[In](workerCount))
      val resultsMerge = b.add(Merge[Out](workerCount))

      // After merging priority and ordinary jobs, we feed them to the balancer
      priorityMerge ~> balance

      // Wire up each of the outputs of the balancer to a worker flow
      // then merge them back
      for (i <- 0 until workerCount)
        balance.out(i) ~> worker ~> resultsMerge.in(i)

      // We now expose the input ports of the priorityMerge and the output
      // of the resultsMerge as our PriorityWorkerPool ports
      // -- all neatly wrapped in our domain specific Shape
      PriorityWorkerPoolShape(
        jobsIn = priorityMerge.in(0),
        priorityJobsIn = priorityMerge.preferred,
        resultsOut = resultsMerge.out)
    }

  }

}
```

All we need to do now is to use our custom junction in a graph. The following code simulates some simple workers and jobs using plain strings and prints out the results. Actually we used *two* instances of our worker pool junction using `add()` twice.

```
val worker1 = Flow[String].map("step 1 " + _)
val worker2 = Flow[String].map("step 2 " + _)

FlowGraph.closed() { implicit b =>
  import FlowGraph.Implicits._

  val priorityPool1 = b.add(PriorityWorkerPool(worker1, 4))
  val priorityPool2 = b.add(PriorityWorkerPool(worker2, 2))

  Source(1 to 100).map("job: " + _) ~> priorityPool1.jobsIn
  Source(1 to 100).map("priority job: " + _) ~> priorityPool1.priorityJobsIn

  priorityPool1.resultsOut ~> priorityPool2.jobsIn
  Source(1 to 100).map("one-step, priority " + _) ~> priorityPool2.priorityJobsIn

  priorityPool2.resultsOut ~> Sink.foreach(println)
}.run()
```

1.5.5 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:

```
trait Message
case class Ping(id: Int) extends Message
case class Pong(id: Int) extends Message

def toBytes(msg: Message): ByteString = {
  // implementation details elided ...
}

def fromBytes(bytes: ByteString): Message = {
  // implementation details elided ...
}

val codecVerbose = BidiFlow() { b =>
  // construct and add the top flow, going outbound
  val outbound = b.add(Flow[Message].map(toBytes))
  // construct and add the bottom flow, going inbound
  val inbound = b.add(Flow[ByteString].map(fromBytes))
  // fuse them together into a BidiShape
  BidiShape(outbound, inbound)
}

// this is the same as the above
val codec = BidiFlow(toBytes _, 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:

```
def toBytes(msg: Message): ByteString = {
  implicit val order = ByteOrder.LITTLE_ENDIAN
  msg match {
    case Ping(id) => ByteString.newBuilder.putByte(1).putInt(id).result()
    case Pong(id) => ByteString.newBuilder.putByte(2).putInt(id).result()
  }
}

def fromBytes(bytes: ByteString): Message = {
  implicit val order = ByteOrder.LITTLE_ENDIAN
  val it = bytes.iterator
  it.getByte match {
    case 1     => Ping(it.getInt)
    case 2     => Pong(it.getInt)
    case other => throw new RuntimeException(s"parse error: expected 1|2 got $other")
  }
}
```

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*).

```
val framing = BidiFlow() { b =>
  implicit val order = ByteOrder.LITTLE_ENDIAN

  def addLengthHeader(bytes: ByteString) = {
    val len = bytes.length
    ByteString.newBuilder.putInt(len).append(bytes).result()
  }

  class FrameParser extends PushPullStage[ByteString, ByteString] {
    // this holds the received but not yet parsed bytes
    var stash = ByteString.empty
    // this holds the current message length or -1 if at a boundary
    var needed = -1

    override def onPush(bytes: ByteString, ctx: Context[ByteString]) = {
      stash ++= bytes
      run(ctx)
    }
    override def onPull(ctx: Context[ByteString]) = run(ctx)
    override def onUpstreamFinish(ctx: Context[ByteString]) =
      if (stash.isEmpty) ctx.finish()
      else ctx.absorbTermination() // we still have bytes to emit

    private def run(ctx: Context[ByteString]): SyncDirective =
      if (needed == -1) {
        // are we at a boundary? then figure out next length
        if (stash.length < 4) pullOrFinish(ctx)
        else {
          needed = stash.iterator.getInt
          stash = stash.drop(4)
          run(ctx) // cycle back to possibly already emit the next chunk
        }
      } else if (stash.length < needed) {
        // we are in the middle of a message, need more bytes
        pullOrFinish(ctx)
      } else {
        // we have enough to emit at least one message, so do it
        val emit = stash.take(needed)
        stash = stash.drop(needed)
        needed = -1
        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 def pullOrFinish(ctx: Context[ByteString]) =
      if (ctx.isFinishing) ctx.finish()
      else ctx.pull()
  }

  val outbound = b.add(Flow[ByteString].map(addLengthHeader))
  val inbound = b.add(Flow[ByteString].transform(() => new FrameParser))
  BidiShape(outbound, inbound)
}
```

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    <~
 *         |  +-------+            +---------+  |
 *         +------------------------------------+
 */
val stack = codec.atop(framing)

// test it by plugging it into its own inverse and closing the right end
val pingpong = Flow[Message].collect { case Ping(id) => Pong(id) }
val flow = stack.atop(stack.reversed).join(pingpong)
val result = Source((0 to 9).map(Ping)).via(flow).grouped(20).runWith(Sink.head)
Await.result(result, 1.second) should ===((0 to 9).map(Pong))
```

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.

1.5.6 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.

```
import FlowGraph.Implicits._
val foldFlow: Flow[Int, Int, Future[Int]] = Flow(Sink.fold[Int, Int](0)(_ + _)) {
  implicit builder ⇒
    fold ⇒
      (fold.inlet, builder.matValue.mapAsync(4)(identity).outlet)
}
```

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.

```
import FlowGraph.Implicits._
// This cannot produce any value:
val cyclicFold: Source[Int, Future[Int]] = Source(Sink.fold[Int, Int](0)(_ + _)) {
  implicit builder =>
    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
      builder.matValue.mapAsync(4)(identity) ~> fold
      builder.matValue.mapAsync(4)(identity).outlet
}
```

1.5.7 Graph cycles, liveness and deadlocks

Cycles in bounded flow graphs 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 naïve 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.

Note: The graph DSL allows the connection arrows to be reversed, which is particularly handy when writing cycles—as we will see there are cases where this is very helpful.

```
// WARNING! The graph below deadlocks!
FlowGraph.closed() { implicit b =>
  import FlowGraph.Implicits._

  val merge = b.add(Merge[Int](2))
  val bcast = b.add(Broadcast[Int](2))

  source ~> merge ~> Flow[Int].map { s => println(s); s } ~> bcast ~> Sink.ignore
            merge                    <~                      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.closed() { implicit b =>
  import FlowGraph.Implicits._

  val merge = b.add(MergePreferred[Int](1))
  val bcast = b.add(Broadcast[Int](2))

  source ~> merge ~> Flow[Int].map { s => println(s); s } ~> bcast ~> Sink.ignore
            merge.preferred              <~                  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.closed() { implicit b =>
  import FlowGraph.Implicits._

  val merge = b.add(Merge[Int](2))
  val bcast = b.add(Broadcast[Int](2))
```

```
  source ~> merge ~> Flow[Int].map { s => println(s); s } ~> bcast ~> Sink.ignore
      merge <~ Flow[Int].buffer(10, OverflowStrategy.dropHead) <~ 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.closed() { implicit b =>
  import FlowGraph.Implicits._

  val zip = b.add(ZipWith[Int, Int, Int]((left, right) => right))
  val bcast = b.add(Broadcast[Int](2))

  source ~> zip.in0
  zip.out.map { s => println(s); s } ~> bcast ~> Sink.ignore
  zip.in1             <~                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.closed() { implicit b =>
  import FlowGraph.Implicits._

  val zip = b.add(ZipWith((left: Int, right: Int) => left))
  val bcast = b.add(Broadcast[Int](2))
  val concat = b.add(Concat[Int]())
  val start = Source.single(0)

  source ~> zip.in0
  zip.out.map { s => println(s); s } ~> bcast ~> Sink.ignore
  zip.in1 <~ concat <~ start
             concat         <~          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(1 to 3)
  .map { i => println(s"A: $i"); i }
  .map { i => println(s"B: $i"); i }
  .map { i => println(s"C: $i"); i }
  .runWith(Sink.ignore)
```

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:

```
val materializer = ActorFlowMaterializer(
  ActorFlowMaterializerSettings(system)
    .withInputBuffer(
```

```
      initialSize = 64,
      maxSize = 64))
```

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

```
val section = Flow[Int].map(_ * 2)
  .withAttributes(OperationAttributes.inputBuffer(initial = 1, max = 1))
val flow = section.via(Flow[Int].map(_ / 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:

```
import scala.concurrent.duration._
case class Tick()

FlowGraph.closed() { implicit b =>
  import FlowGraph.Implicits._

  val zipper = b.add(ZipWith[Tick, Int, Int]((tick, count) => count))

  Source(initialDelay = 3.second, interval = 3.second, Tick()) ~> zipper.in0

  Source(initialDelay = 1.second, interval = 1.second, "message!")
    .conflate(seed = (_) => 1)((count, _) => count + 1) ~> zipper.in1

  zipper.out ~> Sink.foreach(println)
}
```

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
val jobs: Source[Job, Unit] = 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

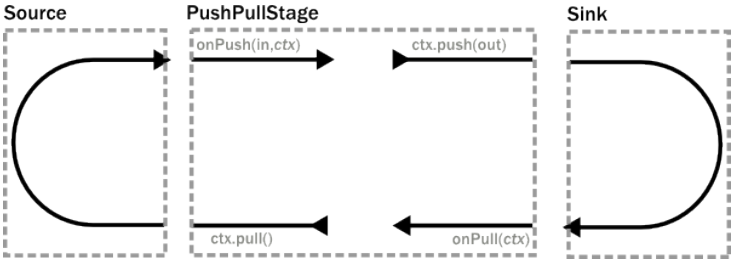
While the processing vocabulary of Akka Streams is quite rich (see the *Streams Cookbook* for examples) it is sometimes necessary to define new transformation stages either because some functionality is missing from the stock operations, or for performance reasons. In this part we show how to build custom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

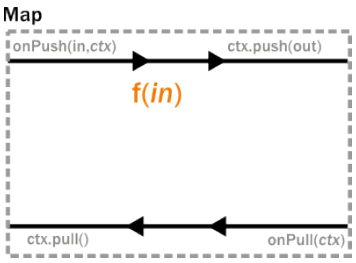
The most elementary transformation stage is the `PushPullStage` which can express a large class of algorithms working on streams. A `PushPullStage` can be illustrated as a box with two "input" and two "output ports" as it is seen in the illustration below.



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()`

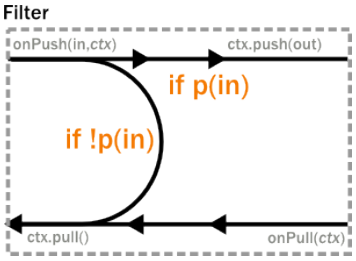
To illustrate these concepts we create a small `PushPullStage` that implements the `map` transformation.



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:

```
class Map[A, B](f: A => B) extends PushPullStage[A, B] {
  override def onPush(elem: A, ctx: Context[B]): SyncDirective =
    ctx.push(f(elem))

  override def onPull(ctx: Context[B]): SyncDirective =
    ctx.pull()
}
```

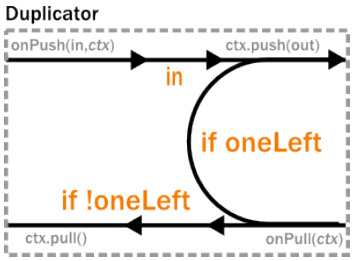
Map is a typical example of a one-to-one transformation of a stream. To demonstrate a many-to-one stage we will implement filter. The conceptual wiring of `Filter` looks like this:



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).

```
class Filter[A](p: A => Boolean) extends PushPullStage[A, A] {
  override def onPush(elem: A, ctx: Context[A]): SyncDirective =
    if (p(elem)) ctx.push(elem)
    else ctx.pull()

  override def onPull(ctx: Context[A]): SyncDirective =
    ctx.pull()
}
```

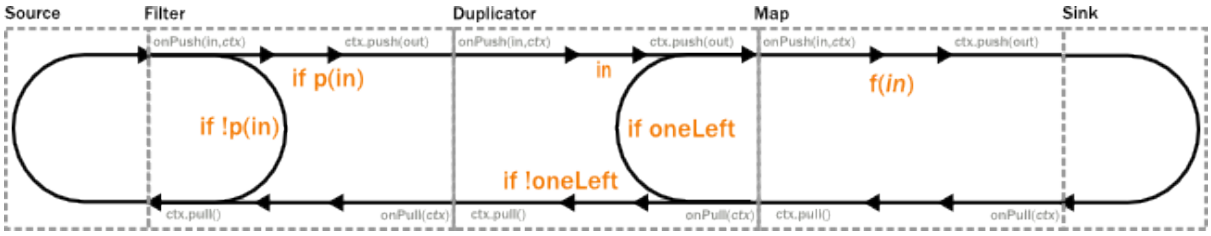
To complete the picture we define a one-to-many transformation as the next step. We chose a straightforward example stage that emits every upstream element twice downstream. The conceptual wiring of this stage looks like this:



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 var lastElem: A = _
  private var oneLeft = false

  override def onPush(elem: A, ctx: Context[A]): SyncDirective = {
    lastElem = elem
    oneLeft = true
    ctx.push(elem)
  }

  override def onPull(ctx: Context[A]): SyncDirective =
    if (!ctx.isFinishing) {
      // the main pulling logic is below as it is demonstrated on the illustration
      if (oneLeft) {
        oneLeft = false
        ctx.push(lastElem)
      } else
        ctx.pull()
    } else {
      // If we need to emit a final element after the upstream
      // finished
      if (oneLeft) ctx.pushAndFinish(lastElem)
      else ctx.finish()
    }

  override def onUpstreamFinish(ctx: Context[A]): TerminationDirective =
    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:

```
val resultFuture = Source(1 to 10)
  .transform(() => new Filter(_ % 2 == 0))
  .transform(() => new Duplicator())
  .transform(() => new Map(_ / 2))
  .runWith(sink)
```

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:

```
class Map[A, B](f: A => B) extends PushStage[A, B] {
  override def onPush(elem: A, ctx: Context[B]): SyncDirective =
    ctx.push(f(elem))
}

class Filter[A](p: A => Boolean) extends PushStage[A, A] {
  override def onPush(elem: A, ctx: Context[A]): SyncDirective =
    if (p(elem)) ctx.push(elem)
    else 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`:

```
class Duplicator[A]() extends StatefulStage[A, A] {
  override def initial: StageState[A, A] = new StageState[A, A] {
    override def onPush(elem: A, ctx: Context[A]): SyncDirective =
```

```
      emit(List(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.

```
class PreferringMergeShape[A](_init: Init[A] = Name("PreferringMerge"))
  extends FanInShape[A](_init) {
  val preferred = newInlet[A]("preferred")
  val secondary1 = newInlet[A]("secondary1")
  val secondary2 = newInlet[A]("secondary2")
  protected override def construct(i: Init[A]) = new PreferringMergeShape(i)
}
```

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.

```
class PreferringMerge extends FlexiMerge[Int, PreferringMergeShape[Int]](
  new PreferringMergeShape, OperationAttributes.name("ImportantWithBackups")) {
  import akka.stream.scaladsl.FlexiMerge._

  override def createMergeLogic(p: PortT) = new MergeLogic[Int] {
    override def initialState =
      State[Int](ReadPreferred(p.preferred, p.secondary1, p.secondary2)) {
        (ctx, input, element) =>
          ctx.emit(element)
          SameState
      }
  }
}
```

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`).

```
import akka.stream.FanInShape._
class ZipPorts[A, B](_init: Init[(A, B)] = Name("Zip"))
  extends FanInShape[(A, B)](_init) {
  val left = newInlet[A]("left")
  val right = newInlet[B]("right")
  protected override def construct(i: Init[(A, B)]) = new ZipPorts(i)
}
  class Zip[A, B] extends FlexiMerge[(A, B), ZipPorts[A, B]](
    new ZipPorts, OperationAttributes.name("Zip2State")) {
    import FlexiMerge._

    override def createMergeLogic(p: PortT) = new MergeLogic[(A, B)] {
      var lastInA: A = _

      val readA: State[A] = State[A](Read(p.left)) { (ctx, input, element) =>
        lastInA = element
        readB
      }

      val readB: State[B] = State[B](Read(p.right)) { (ctx, input, element) =>
        ctx.emit((lastInA, element))
        readA
```

```
      }

      override def initialState: State[_] = readA

      override def initialCompletionHandling = 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:

```
import akka.stream.FanInShape._
class ZipPorts[A, B](_init: Init[(A, B)] = Name("Zip"))
  extends FanInShape[(A, B)](_init) {
  val left = newInlet[A]("left")
  val right = newInlet[B]("right")
  protected override def construct(i: Init[(A, B)]) = new ZipPorts(i)
}
  class Zip[A, B] extends FlexiMerge[(A, B), ZipPorts[A, B]](
    new ZipPorts, OperationAttributes.name("Zip1State")) {
    import FlexiMerge._
    override def createMergeLogic(p: PortT) = new MergeLogic[(A, B)] {
      override def initialState =
        State(ReadAll(p.left, p.right)) { (ctx, _, inputs) =>
          val a = inputs(p.left)
          val b = inputs(p.right)
          ctx.emit((a, b))
          SameState
        }

      override def initialCompletionHandling = 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`):

```
FlowGraph.closed(Sink.head[(Int, String)]) { implicit b =>
  o =>
  import FlowGraph.Implicits._

  val zip = b.add(new Zip[Int, String])

  Source.single(1)   ~> zip.left
  Source.single("1") ~> zip.right
                        zip.out ~> o.inlet
}
```

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.

```
class ImportantWithBackupShape[A](_init: Init[A] = Name("Zip"))
  extends FanInShape[A](_init) {
  val important = newInlet[A]("important")
  val replica1 = newInlet[A]("replica1")
  val replica2 = newInlet[A]("replica2")
  protected override def construct(i: Init[A]) =
    new ImportantWithBackupShape(i)
}
class ImportantWithBackups[A] extends FlexiMerge[A, ImportantWithBackupShape[A]](
  new ImportantWithBackupShape, OperationAttributes.name("ImportantWithBackups")) {
  import FlexiMerge._

  override def createMergeLogic(p: PortT) = new MergeLogic[A] {
    import p.important
    override def initialCompletionHandling =
      CompletionHandling(
        onUpstreamFinish = (ctx, input) => input match {
          case `important` =>
            log.info("Important input completed, shutting down.")
            ctx.finish()
            SameState

          case replica =>
            log.info("Replica {} completed, " +
              "no more replicas available, " +
              "applying eagerClose completion handling.", replica)

            ctx.changeCompletionHandling(eagerClose)
            SameState
        },
        onUpstreamFailure = (ctx, input, cause) => input match {
          case `important` =>
            ctx.fail(cause)
            SameState

          case replica =>
            log.error(cause, "Replica {} failed, " +
              "no more replicas available, " +
              "applying eagerClose completion handling.", replica)

            ctx.changeCompletionHandling(eagerClose)
            SameState
        })

    override def initialState =
      State[A](ReadAny(p.important, p.replica1, p.replica2)) {
        (ctx, input, element) =>
          ctx.emit(element)
          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 tuple.

A `FlexiRoute` has exactly-one input port (in our example, type parameterized as `(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`.

```
import FanOutShape._
class UnzipShape[A, B](_init: Init[(A, B)] = Name[(A, B)]("Unzip"))
  extends FanOutShape[(A, B)](_init) {
  val outA = newOutlet[A]("outA")
  val outB = newOutlet[B]("outB")
  protected override def construct(i: Init[(A, B)]) = new UnzipShape(i)
}
class Unzip[A, B] extends FlexiRoute[(A, B), UnzipShape[A, B]](
  new UnzipShape, OperationAttributes.name("Unzip")) {
  import FlexiRoute._

  override def createRouteLogic(p: PortT) = new RouteLogic[(A, B)] {
    override def initialState =
      State[Any](DemandFromAll(p.outA, p.outB)) {
        (ctx, _, element) =>
          val (a, b) = element
          ctx.emit(p.outA)(a)
          ctx.emit(p.outB)(b)
          SameState
      }

    override def initialCompletionHandling = 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 tuple in the state function and signal its first element to the `left` stream, and the second element of the tuple to the `right` stream. Notice that since we are emitting values of different types (`A` and `B`), the 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 similarly 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.

```
class ImportantRouteShape[A](_init: Init[A] = Name[A]("ImportantRoute")) extends FanOutShape[A](_
  val important = newOutlet[A]("important")
  val additional1 = newOutlet[A]("additional1")
  val additional2 = newOutlet[A]("additional2")
  protected override def construct(i: Init[A]) = new ImportantRouteShape(i)
}
class ImportantRoute[A] extends FlexiRoute[A, ImportantRouteShape[A]](
  new ImportantRouteShape, OperationAttributes.name("ImportantRoute")) {
  import FlexiRoute._
  override def createRouteLogic(p: PortT) = new RouteLogic[A] {
    import p.important
    private val select = (p.important | p.additional1 | p.additional2)

    override def initialCompletionHandling =
      CompletionHandling(
        // upstream:
        onUpstreamFinish = (ctx) => (),
        onUpstreamFailure = (ctx, thr) => (),
        // downstream:
        onDownstreamFinish = (ctx, output) => output match {
          case `important` =>
            // finish all downstreams, and cancel the upstream
            ctx.finish()
            SameState
          case _ =>
            SameState
        })

    override def initialState =
      State(DemandFromAny(p.important, p.additional1, p.additional2)) {
        (ctx, output, element) =>
          ctx.emit(select(output))(element)
```

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

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/mixin `akka.stream.actor.ActorPublisher` in your `Actor` to make it 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:

```
object JobManager {
  def props: Props = Props[JobManager]

  final case class Job(payload: String)
  case object JobAccepted
  case object JobDenied
}

class JobManager extends ActorPublisher[JobManager.Job] {
  import akka.stream.actor.ActorPublisherMessage._
  import JobManager._

  val MaxBufferSize = 100
  var buf = Vector.empty[Job]

  def receive = {
    case job: Job if buf.size == MaxBufferSize =>
      sender() ! JobDenied
    case job: Job =>
      sender() ! JobAccepted
      if (buf.isEmpty && totalDemand > 0)
        onNext(job)
      else {
        buf :+= job
        deliverBuf()
      }
    case Request(_) =>
      deliverBuf()
    case Cancel =>
      context.stop(self)
  }

  @tailrec final def deliverBuf(): Unit =
    if (totalDemand > 0) {
      /*
       * totalDemand is a Long and could be larger than
       * what buf.splitAt can accept
       */
      if (totalDemand <= Int.MaxValue) {
        val (use, keep) = buf.splitAt(totalDemand.toInt)
        buf = keep
        use foreach onNext
      } else {
        val (use, keep) = buf.splitAt(Int.MaxValue)
        buf = keep
        use foreach onNext
        deliverBuf()
      }
    }
}
```

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 `ActorPublisher` 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`:

```
val jobManagerSource = Source.actorPublisher[JobManager.Job](JobManager.props)
val ref = Flow[JobManager.Job]
  .map(_.payload.toUpperCase)
  .map { elem => println(elem); elem }
  .to(Sink.ignore)
  .runWith(jobManagerSource)

ref ! JobManager.Job("a")
ref ! JobManager.Job("b")
ref ! JobManager.Job("c")
```

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

ActorSubscriber

Extend/mixin `akka.stream.actor.ActorSubscriber` in your `Actor` to make it 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:

```
object WorkerPool {
  case class Msg(id: Int, replyTo: ActorRef)
  case class Work(id: Int)
  case class Reply(id: Int)
  case class Done(id: Int)

  def props: Props = Props(new WorkerPool)
}

class WorkerPool extends ActorSubscriber {
  import WorkerPool._
  import ActorSubscriberMessage._

  val MaxQueueSize = 10
  var queue = Map.empty[Int, ActorRef]

  val router = {
```

```
    val routees = Vector.fill(3) {
      ActorRefRoutee(context.actorOf(Props[Worker]))
    }
    Router(RoundRobinRoutingLogic(), routees)
  }

  override val requestStrategy = new MaxInFlightRequestStrategy(max = MaxQueueSize) {
    override def inFlightInternally: Int = queue.size
  }

  def receive = {
    case OnNext(Msg(id, replyTo)) =>
      queue += (id -> replyTo)
      assert(queue.size <= MaxQueueSize, s"queued too many: ${queue.size}")
      router.route(Work(id), self)
    case Reply(id) =>
      queue(id) ! Done(id)
      queue -= id
  }
}

class Worker extends Actor {
  import WorkerPool._
  def receive = {
    case Work(id) =>
      // ...
      sender() ! Reply(id)
  }
}
```

Subclass must define the `RequestStrategy` to control stream back pressure. After each incoming message the `ActorSubscriber` 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`:

```
val N = 117
Source(1 to N).map(WorkerPool.Msg(_, replyTo))
  .runWith(Sink.actorSubscriber(WorkerPool.props))
```

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:

```
def send(email: Email): Future[Unit] = {
  // ...
}
```

We start with the tweet stream of authors:

```
val authors: Source[Author, Unit] =
  tweets
    .filter(_.hashtags.contains(akka))
    .map(_.author)
```

Assume that we can lookup their email address using:

```
def lookupEmail(handle: String): Future[Option[String]] =
```

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

```
val emailAddresses: Source[String, Unit] =
  authors
    .mapAsync(4)(author => addressSystem.lookupEmail(author.handle))
    .collect { case Some(emailAddress) => emailAddress }
```

Finally, sending the emails:

```
val sendEmails: RunnableFlow[Unit] =
  emailAddresses
    .mapAsync(4)(address => {
      emailServer.send(
        Email(to = address, title = "Akka", body = "I like your tweet"))
    })
    .to(Sink.ignore)

sendEmails.run()
```

`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`:

```
val authors: Source[Author, Unit] =
  tweets.filter(_.hashtags.contains(akka)).map(_.author)

val emailAddresses: Source[String, Unit] =
  authors
    .mapAsyncUnordered(4)(author => addressSystem.lookupEmail(author.handle))
    .collect { case Some(emailAddress) => emailAddress }

val sendEmails: RunnableFlow[Unit] =
  emailAddresses
    .mapAsyncUnordered(4)(address => {
      emailServer.send(
        Email(to = address, title = "Akka", body = "I like your tweet"))
    })
    .to(Sink.ignore)

sendEmails.run()
```

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.

```
val blockingExecutionContext = system.dispatchers.lookup("blocking-dispatcher")

val sendTextMessages: RunnableFlow[Unit] =
  phoneNumbers
    .mapAsync(4)(phoneNo => {
      Future {
        smsServer.send(
          TextMessage(to = phoneNo, body = "I like your tweet"))
      }(blockingExecutionContext)
    })
    .to(Sink.ignore)

sendTextMessages.run()
```

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.

```
val send = Flow[String]
  .map { phoneNo =>
    smsServer.send(TextMessage(to = phoneNo, body = "I like your tweet"))
  }
  .withAttributes(ActorOperationAttributes.dispatcher("blocking-dispatcher"))
val sendTextMessages: RunnableFlow[Unit] =
  phoneNumbers.via(send).to(Sink.ignore)

sendTextMessages.run()
```

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`:

```
val akkaTweets: Source[Tweet, Unit] = tweets.filter(_.hashtags.contains(akka))

implicit val timeout = Timeout(3.seconds)
val saveTweets: RunnableFlow[Unit] =
  akkaTweets
    .mapAsync(4)(tweet => database ? Save(tweet))
    .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.

```
class SometimesSlowService(implicit ec: ExecutionContext) {

  private val runningCount = new AtomicInteger

  def convert(s: String): Future[String] = {
    println(s"running: $s (${runningCount.incrementAndGet()})")
    Future {
      if (s.nonEmpty && s.head.isLower)
        Thread.sleep(500)
      else
        Thread.sleep(20)
      println(s"completed: $s (${runningCount.decrementAndGet()})")
      s.toUpperCase
    }
  }
}
```

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

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

```
implicit val blockingExecutionContext = system.dispatchers.lookup("blocking-dispatcher")
val service = new SometimesSlowService

implicit val mat = ActorFlowMaterializer(
  ActorFlowMaterializerSettings(system).withInputBuffer(initialSize = 4, maxSize = 4))

Source(List("a", "B", "C", "D", "e", "F", "g", "H", "i", "J"))
  .map(elem => { println(s"before: $elem"); elem })
  .mapAsync(4)(service.convert)
  .runForeach(elem => println(s"after: $elem"))
```

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`:

```
implicit val blockingExecutionContext = system.dispatchers.lookup("blocking-dispatcher")
val service = new SometimesSlowService

implicit val mat = ActorFlowMaterializer(
  ActorFlowMaterializerSettings(system).withInputBuffer(initialSize = 4, maxSize = 4))

Source(List("a", "B", "C", "D", "e", "F", "g", "H", "i", "J"))
  .map(elem => { println(s"before: $elem"); elem })
  .mapAsyncUnordered(4)(service.convert)
  .runForeach(elem => println(s"after: $elem"))
```

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:

```
def tweets: Publisher[Tweet]
```

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

```
def storage: Subscriber[Author]
```

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

```
val authors = Flow[Tweet]
  .filter(_.hashtags.contains(akka))
  .map(_.author)

Source(tweets).via(authors).to(Sink(storage)).run()
```

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:

```
val (in: Subscriber[Tweet], out: Publisher[Author]) =
  authors.runWith(Source.subscriber[Tweet], Sink.publisher[Author])

tweets.subscribe(in)
out.subscribe(storage)
```

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`:

```
val authorPublisher: Publisher[Author] =
  Source(tweets).via(authors).runWith(Sink.publisher)

authorPublisher.subscribe(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:

```
def storage: Subscriber[Author]
def alert: Subscriber[Author]
```

```
val authorPublisher: Publisher[Author] =
  Source(tweets).via(authors)
    .runWith(Sink.fanoutPublisher(initialBufferSize = 8, maximumBufferSize = 16))

authorPublisher.subscribe(storage)
authorPublisher.subscribe(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`:

```
val tweetSubscriber: Subscriber[Tweet] =
  authors.to(Sink(storage)).runWith(Source.subscriber[Tweet])

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.

```
implicit val mat = ActorFlowMaterializer()
val source = Source(0 to 5).map(100 / _)
val result = source.runWith(Sink.fold(0)(_ + _))
// 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.

```
val decider: Supervision.Decider = {
  case _: ArithmeticException => Supervision.Resume
  case _                      => Supervision.Stop
}
implicit val mat = ActorFlowMaterializer(
  ActorFlowMaterializerSettings(system).withSupervisionStrategy(decider))
val source = Source(0 to 5).map(100 / _)
val result = source.runWith(Sink.fold(0)(_ + _))
// 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.

```
implicit val mat = ActorFlowMaterializer()
val decider: Supervision.Decider = {
  case _: ArithmeticException => Supervision.Resume
  case _                      => Supervision.Stop
}
val flow = Flow[Int]
  .filter(100 / _ < 50).map(elem => 100 / (5 - elem))
  .withAttributes(ActorOperationAttributes.supervisionStrategy(decider))
val source = Source(0 to 5).via(flow)

val result = source.runWith(Sink.fold(0)(_ + _))
// 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.

```
implicit val mat = ActorFlowMaterializer()
val decider: Supervision.Decider = {
  case _: IllegalArgumentException => Supervision.Restart
  case _                           => Supervision.Stop
}
val flow = Flow[Int]
  .scan(0) { (acc, elem) =>
    if (elem < 0) throw new IllegalArgumentException("negative not allowed")
    else acc + elem
  }
  .withAttributes(ActorOperationAttributes.supervisionStrategy(decider))
val source = Source(List(1, 3, -1, 5, 7)).via(flow)
val result = source.grouped(1000).runWith(Sink.head)
// 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(Vector(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:

```
val authors: Source[Author, Unit] =
  tweets
    .filter(_.hashtags.contains(akka))
    .map(_.author)
```

Assume that we can lookup their email address using:

```
def lookupEmail(handle: String): Future[String] =
```

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.resumingDecider` to drop unknown email addresses:

```
import ActorOperationAttributes.supervisionStrategy
import Supervision.resumingDecider

val emailAddresses: Source[String, Unit] =
  authors.via(
    Flow[Author].mapAsync(4)(author => addressSystem.lookupEmail(author.handle))
      .withAttributes(supervisionStrategy(resumingDecider)))
```

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:

```
val connections: Source[IncomingConnection, Future[ServerBinding]] =
  Tcp().bind("127.0.0.1", 8888)
```

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 *Parsing lines from a stream of ByteStrings* 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 =>
  println(s"New connection from: ${connection.remoteAddress}")

  val echo = Flow[ByteString]
    .transform(() => RecipeParseLines.parseLines("\n", maximumLineBytes = 256))
    .map(_ + "!!!\n")
    .map(ByteString(_))

  connection.handleWith(echo)
}
```

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 8888
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:

```
  val connection = Tcp().outgoingConnection("127.0.0.1", 8888)

  val replParser = new PushStage[String, ByteString] {
    override def onPush(elem: String, ctx: Context[ByteString]): SyncDirective = {
      elem match {
        case "q" ⇒ ctx.pushAndFinish(ByteString("BYE\n"))
        case _   ⇒ ctx.push(ByteString(s"$elem\n"))
      }
    }
  }

  val repl = Flow[ByteString]
    .transform(() => RecipeParseLines.parseLines("\n", maximumLineBytes = 256))
    .map(text => println("Server: " + text))
    .map(_ => readLine("> "))
    .transform(() ⇒ replParser)

  connection.join(repl).run()
}
```

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 =>

  val serverLogic = Flow() { implicit b =>
    import FlowGraph.Implicits._

    // server logic, parses incoming commands
    val commandParser = new PushStage[String, String] {
      override def onPush(elem: String, ctx: Context[String]): SyncDirective = {
        elem match {
          case "BYE" ⇒ ctx.finish()
          case _     ⇒ ctx.push(elem + "!")
        }
      }
    }

    import connection._
    val welcomeMsg = s"Welcome to: $localAddress, you are: $remoteAddress!\n"

    val welcome = Source.single(ByteString(welcomeMsg))
    val echo = b.add(Flow[ByteString]
      .transform(() => RecipeParseLines.parseLines("\n", maximumLineBytes = 256))
      .transform(() ⇒ commandParser)
      .map(_ + "\n")
      .map(ByteString(_)))

    val concat = b.add(Concat[ByteString]())
    // first we emit the welcome message,
    welcome ~> concat.in(0)
    // then we continue using the echo-logic Flow
    echo.outlet ~> concat.in(1)

    (echo.inlet, concat.out)
  }

  connection.handleWith(serverLogic)
}
```

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:

```
import akka.stream.io._
val file = new File("example.csv")

SynchronousFileSource(file)
  .runForeach((chunk: ByteString) ⇒ handle(chunk))
```

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

If you would like to keep all sink and source factories defined on the `Source` and `Sink` objects instead of using the separate objects contained in `akka.stream.io` to create these you can import an *implicit coversion* that makes these operations available as shown below:

```
import akka.stream.io.Implicits._

Source.synchronousFile(file) to Sink.outputStream(() ⇒ System.out)
```

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:

```
// Takes a scoop of batter and creates a pancake with one side cooked
val fryingPan1: Flow[ScoopOfBatter, HalfCookedPancake, Unit] =
  Flow[ScoopOfBatter].map { batter => HalfCookedPancake() }

// Finishes a half-cooked pancake
val fryingPan2: Flow[HalfCookedPancake, Pancake, Unit] =
  Flow[HalfCookedPancake].map { halfCooked => Pancake() }

// With the two frying pans we can fully cook pancakes
val pancakeChef: Flow[ScoopOfBatter, Pancake, Unit] =
  Flow[ScoopOfBatter].via(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 *Buffers and working with rate*.

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:

```
val fryingPan: Flow[ScoopOfBatter, Pancake, Unit] =
  Flow[ScoopOfBatter].map { batter => Pancake() }

val pancakeChef: Flow[ScoopOfBatter, Pancake, Unit] = Flow() { implicit builder =>
  val dispatchBatter = builder.add(Balance[ScoopOfBatter](2))
  val mergePancakes = builder.add(Merge[Pancake](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.
  dispatchBatter.out(0) ~> fryingPan ~> 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.
```

[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.

```
  dispatchBatter.out(1) ~> fryingPan ~> mergePancakes.in(1)

  (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: *Balancing jobs to a fixed pool of workers*

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.

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:

```
val pancakeChef: Flow[ScoopOfBatter, Pancake, Unit] = Flow() { implicit builder =>

  val dispatchBatter = builder.add(Balance[ScoopOfBatter](2))
  val mergePancakes = builder.add(Merge[Pancake](2))

  // Using two pipelines, having two frying pans each, in total using
  // four frying pans
  dispatchBatter.out(0) ~> fryingPan1 ~> fryingPan2 ~> mergePancakes.in(0)
  dispatchBatter.out(1) ~> fryingPan1 ~> fryingPan2 ~> mergePancakes.in(1)

  (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:

```
val pancakeChefs1: Flow[ScoopOfBatter, HalfCookedPancake, Unit] = Flow() { implicit builder =>
  val dispatchBatter = builder.add(Balance[ScoopOfBatter](2))
  val mergeHalfPancakes = builder.add(Merge[HalfCookedPancake](2))

  // Two chefs work with one frying pan for each, half-frying the pancakes then putting
  // them into a common pool
  dispatchBatter.out(0) ~> fryingPan1 ~> mergeHalfPancakes.in(0)
  dispatchBatter.out(1) ~> fryingPan1 ~> mergeHalfPancakes.in(1)
```

```
  (dispatchBatter.in, mergeHalfPancakes.out)
}

val pancakeChefs2: Flow[HalfCookedPancake, Pancake, Unit] = Flow() { implicit builder =>
  val dispatchHalfPancakes = builder.add(Balance[HalfCookedPancake](2))
  val mergePancakes = builder.add(Merge[Pancake](2))

  // Two chefs work with one frying pan for each, finishing the pancakes then putting
  // them into a common pool
  dispatchHalfPancakes.out(0) ~> fryingPan2 ~> mergePancakes.in(0)
  dispatchHalfPancakes.out(1) ~> fryingPan2 ~> mergePancakes.in(1)

  (dispatchHalfPancakes.in, mergePancakes.out)
}

val kitchen: Flow[ScoopOfBatter, Pancake, Unit] = 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(1 to 4)
  .filter(_ % 2 == 0)
  .map(_ * 2)
  .runWith(TestSink.probe[Int])
  .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.probe[Int]
  .toMat(Sink.cancelled)(Keep.left)
  .run()
  .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 Streams Cookbook

1.14.1 Introduction

This is a collection of patterns to demonstrate various usage of the Akka Streams API by solving small targeted problems in the format of "recipes". The purpose of this page is to give inspiration and ideas how to approach various small tasks involving streams. The recipes in this page can be used directly as-is, but they are most powerful as starting points: customization of the code snippets is warmly encouraged.

This part also serves as supplementary material for the main body of documentation. It is a good idea to have this page open while reading the manual and look for examples demonstrating various streaming concepts as they appear in the main body of documentation.

If you need a quick reference of the available processing stages used in the recipes see *Overview of built-in stages and their semantics*.

1.14.2 Working with Flows

In this collection we show simple recipes that involve linear flows. The recipes in this section are rather general, more targeted recipes are available as separate sections ("Working with rate", "Working with IO").

Logging elements of a stream

Situation: During development it is sometimes helpful to see what happens in a particular section of a stream.

The simplest solution is to simply use a `map` operation and use `println` to print the elements received to the console. While this recipe is rather simplistic, it is often suitable for a quick debug session.

```
val loggedSource = mySource.map { elem => println(elem); elem }
```

If a proper logging solution is needed another approach is to create a `PushStage` and override all upstream event handlers, emitting log information through an Akka `LoggingAdapter`. This small stage does not influence the elements flowing in the stream, it just emits them unmodified by calling `ctx.push(elem)` in its `onPush` event handler logic.

```
// customise log levels
mySource.log("before-map")
  .withAttributes(OperationAttributes.logLevels(onElement = Logging.WarningLevel))
  .map(analyse)

// or provide custom logging adapter
implicit val adapter = Logging(system, "customLogger")
mySource.log("custom")
```

Flattening a stream of sequences

Problem: A stream is given as a stream of sequence of elements, but a stream of elements needed instead, streaming all the nested elements inside the sequences separately.

The `mapConcat` operation can be used to implement a one-to-many transformation of elements using a mapper function in the form of `In => immutable.Seq[Out]`. In this case we want to map a `Seq` of elements to the elements in the collection itself, so we can just call `mapConcat(identity)`.

```
val myData: Source[List[Message], Unit] = someDataSource
val flattened: Source[Message, Unit] = myData.mapConcat(identity)
```

Draining a stream to a strict collection

Situation: A finite sequence of elements is given as a stream, but a scala collection is needed instead.

In this recipe we will use the `grouped` stream operation that groups incoming elements into a stream of limited size collections (it can be seen as the almost opposite version of the "Flattening a stream of sequences" recipe we showed before). By using a `grouped(MaxAllowedSeqSize).runWith(Sink.head)` we first create a stream of groups with maximum size of `MaxAllowedSeqSize` and then we take the first element of this stream. What we get is a `Future` containing a sequence with all the elements of the original up to `MaxAllowedSeqSize` size (further elements are dropped).

```
val strict: Future[immutable.Seq[Message]] =
  myData.grouped(MaxAllowedSeqSize).runWith(Sink.head)
```

Calculating the digest of a ByteString stream

Problem: A stream of bytes is given as a stream of `ByteStrings` and we want to calculate the cryptographic digest of the stream.

This recipe uses a `PushPullStage` to host a mutable `MessageDigest` class (part of the Java Cryptography API) and update it with the bytes arriving from the stream. When the stream starts, the `onPull` handler of the stage is called, which just bubbles up the `pull` event to its upstream. As a response to this pull, a ByteString chunk will arrive (`onPush`) which we use to update the digest, then it will pull for the next chunk.

Eventually the stream of `ByteStrings` depletes and we get a notification about this event via `onUpstreamFinish`. At this point we want to emit the digest value, but we cannot do it in this handler directly. Instead we call `ctx.absorbTermination` signalling to our context that we do not yet want to finish. When the environment decides that we can emit further elements `onPull` is called again, and we see `ctx.isFinishing` returning true (since the upstream source has been depleted already). Since we only want to emit a final element it is enough to call `ctx.pushAndFinish` passing the digest ByteString to be emitted.

```
import akka.stream.stage._
def digestCalculator(algorithm: String) = new PushPullStage[ByteString, ByteString] {
  val digest = MessageDigest.getInstance(algorithm)

  override def onPush(chunk: ByteString, ctx: Context[ByteString]): SyncDirective = {
    digest.update(chunk.toArray)
    ctx.pull()
  }

  override def onPull(ctx: Context[ByteString]): SyncDirective = {
    if (ctx.isFinishing) ctx.pushAndFinish(ByteString(digest.digest()))
    else ctx.pull()
  }

  override def onUpstreamFinish(ctx: Context[ByteString]): TerminationDirective = {
    // If the stream is finished, we need to emit the last element in the onPull block.
    // It is not allowed to directly emit elements from a termination block
    // (onUpstreamFinish or onUpstreamFailure)
    ctx.absorbTermination()
  }
}

val digest: Source[ByteString, Unit] = data.transform(() => digestCalculator("SHA-256"))
```

Parsing lines from a stream of ByteStrings

Problem: A stream of bytes is given as a stream of `ByteStrings` containing lines terminated by line ending characters (or, alternatively, containing binary frames delimited by a special delimiter byte sequence) which needs to be parsed.

We express our solution as a `StatefulStage` because it has support for emitting multiple elements easily through its `emit(iterator, ctx)` helper method. Since an incoming ByteString chunk might contain multiple lines (frames) this feature comes in handy.

To create the parser we only need to hook into the `onPush` handler. We maintain a buffer of bytes (expressed as a `ByteString`) by simply concatenating incoming chunks with it. Since we don't want to allow unbounded size lines (records) we always check if the buffer size is larger than the allowed `maximumLineBytes` value, and terminate the stream if this invariant is violated.

After we updated the buffer, we try to find the terminator sequence as a subsequence of the current buffer. To be efficient, we also maintain a pointer `nextPossibleMatch` into the buffer so that we only search that part of the buffer where new matches are possible.

The search for a match is done in two steps: first we try to search for the first character of the terminator sequence in the buffer. If we find a match, we do a full subsequence check to see if we had a false positive or not. The parsing logic is recursive to be able to parse multiple lines (records) contained in the decoding buffer.

```
def parseLines(separator: String, maximumLineBytes: Int) =
  new StatefulStage[ByteString, String] {
    private val separatorBytes = ByteString(separator)
    private val firstSeparatorByte = separatorBytes.head
    private var buffer = ByteString.empty
    private var nextPossibleMatch = 0

    def initial = new State {
      override def onPush(chunk: ByteString, ctx: Context[String]): SyncDirective = {
        buffer ++= chunk
        if (buffer.size > maximumLineBytes)
          ctx.fail(new IllegalStateException(s"Read ${buffer.size} bytes " +
            s"which is more than $maximumLineBytes without seeing a line terminator"))
        else emit(doParse(Vector.empty).iterator, ctx)
      }

      @tailrec
      private def doParse(parsedLinesSoFar: Vector[String]): Vector[String] = {
        val possibleMatchPos = buffer.indexOf(firstSeparatorByte, from = nextPossibleMatch)
        if (possibleMatchPos == -1) {
          // No matching character, we need to accumulate more bytes into the buffer
          nextPossibleMatch = buffer.size
          parsedLinesSoFar
        } else if (possibleMatchPos + separatorBytes.size > buffer.size) {
          // We have found a possible match (we found the first character of the terminator
          // sequence) but we don't have yet enough bytes. We remember the position to
          // retry from next time.
          nextPossibleMatch = possibleMatchPos
          parsedLinesSoFar
        } else {
          if (buffer.slice(possibleMatchPos, possibleMatchPos + separatorBytes.size)
            == separatorBytes) {
            // Found a match
            val parsedLine = buffer.slice(0, possibleMatchPos).utf8String
            buffer = buffer.drop(possibleMatchPos + separatorBytes.size)
            nextPossibleMatch -= possibleMatchPos + separatorBytes.size
            doParse(parsedLinesSoFar :+ parsedLine)
          } else {
            nextPossibleMatch += 1
            doParse(parsedLinesSoFar)
          }
        }

      }
    }
  }
```

Implementing reduce-by-key

Situation: Given a stream of elements, we want to calculate some aggregated value on different subgroups of the elements.

The "hello world" of reduce-by-key style operations is *wordcount* which we demonstrate below. Given a stream of words we first create a new stream `wordStreams` that groups the words according to the `identity` function, i.e. now we have a stream of streams, where every substream will serve identical words.

To count the words, we need to process the stream of streams (the actual groups containing identical words). By mapping over the groups and using `fold` (remember that `fold` automatically materializes and runs the stream it is used on) we get a stream with elements of `Future[String,Int]`. Now all we need is to flatten this stream, which can be achieved by calling `mapAsynch(identity)`.

There is one tricky issue to be noted here. The careful reader probably noticed that we put a `buffer` between the `mapAsync()` operation that flattens the stream of futures and the actual stream of futures. The reason for this is that the substreams produced by `groupBy()` can only complete when the original upstream source completes. This means that `mapAsync()` cannot pull for more substreams because it still waits on folding futures to finish, but these futures never finish if the additional group streams are not consumed. This typical deadlock situation is resolved by this buffer which either able to contain all the group streams (which ensures that they are already running and folding) or fails with an explicit failure instead of a silent deadlock.

```
// split the words into separate streams first
val wordStreams: Source[(String, Source[String, Unit]), Unit] = words.groupBy(identity)

// add counting logic to the streams
val countedWords: Source[Future[(String, Int)], Unit] = wordStreams.map {
  case (word, wordStream) =>
    wordStream.runFold((word, 0)) {
      case ((w, count), _) => (w, count + 1)
    }
}

// get a stream of word counts
val counts: Source[(String, Int), Unit] =
  countedWords
    .buffer(MaximumDistinctWords, OverflowStrategy.fail)
    .mapAsync(4)(identity)
```

By extracting the parts specific to *wordcount* into

- a `groupKey` function that defines the groups
- a `foldZero` that defines the zero element used by the fold on the substream given the group key
- a `fold` function that does the actual reduction

we get a generalized version below:

```
def reduceByKey[In, K, Out](
  maximumGroupSize: Int,
  groupKey: (In) => K,
  foldZero: (K) => Out)(fold: (Out, In) => Out): Flow[In, (K, Out), Unit] = {

  val groupStreams = Flow[In].groupBy(groupKey)
  val reducedValues = groupStreams.map {
    case (key, groupStream) =>
      groupStream.runFold((key, foldZero(key))) {
        case ((key, aggregated), elem) => (key, fold(aggregated, elem))
      }
  }

  reducedValues.buffer(maximumGroupSize, OverflowStrategy.fail).mapAsync(4)(identity)
}
```

```
val wordCounts = words.via(reduceByKey(
  MaximumDistinctWords,
  groupKey = (word: String) => word,
  foldZero = (key: String) => 0)(fold = (count: Int, elem: String) => count + 1))
```

Note: Please note that the reduce-by-key version we discussed above is sequential, in other words it is **NOT** a parallelization pattern like mapReduce and similar frameworks.

Sorting elements to multiple groups with groupBy

Situation: The `groupBy` operation strictly partitions incoming elements, each element belongs to exactly one group. Sometimes we want to map elements into multiple groups simultaneously.

To achieve the desired result, we attack the problem in two steps:

- first, using a function `topicMapper` that gives a list of topics (groups) a message belongs to, we transform our stream of `Message` to a stream of `(Message, Topic)` where for each topic the message belongs to a separate pair will be emitted. This is achieved by using `mapConcat`
- Then we take this new stream of message topic pairs (containing a separate pair for each topic a given message belongs to) and feed it into groupBy, using the topic as the group key.

```
  val topicMapper: (Message) => immutable.Seq[Topic] = ???

val messageAndTopic: Source[(Message, Topic), Unit] = elems.mapConcat { msg: Message =>
  val topicsForMessage = topicMapper(msg)
  // Create a (Msg, Topic) pair for each of the topics
  // the message belongs to
  topicsForMessage.map(msg -> _)
}

val multiGroups: Source[(Topic, Source[String, Unit]), Unit] = messageAndTopic.groupBy(_._2).map
  case (topic, topicStream) =>
    // chopping of the topic from the (Message, Topic) pairs
    (topic, topicStream.map(_._1))
}
```

1.14.3 Working with Graphs

In this collection we show recipes that use stream graph elements to achieve various goals.

Triggering the flow of elements programmatically

Situation: Given a stream of elements we want to control the emission of those elements according to a trigger signal. In other words, even if the stream would be able to flow (not being backpressured) we want to hold back elements until a trigger signal arrives.

This recipe solves the problem by simply zipping the stream of `Message` elments with the stream of `Trigger` signals. Since `Zip` produces pairs, we simply map the output stream selecting the first element of the pair.

```
val graph = FlowGraph.closed() { implicit builder =>
  import FlowGraph.Implicits._
  val zip = builder.add(Zip[Message, Trigger]())
  elements ~> zip.in0
  triggerSource ~> zip.in1
  zip.out ~> Flow[(Message, Trigger)].map { case (msg, trigger) => msg } ~> sink
}
```

Alternatively, instead of using a `Zip`, and then using `map` to get the first element of the pairs, we can avoid creating the pairs in the first place by using `ZipWith` which takes a two argument function to produce the output element. If this function would return a pair of the two argument it would be exactly the behavior of `Zip` so `ZipWith` is a generalization of zipping.

```
val graph = FlowGraph.closed() { implicit builder =>
  import FlowGraph.Implicits._
  val zip = builder.add(ZipWith((msg: Message, trigger: Trigger) => msg))

  elements ~> zip.in0
  triggerSource ~> zip.in1
  zip.out ~> sink
}
```

Balancing jobs to a fixed pool of workers

Situation: Given a stream of jobs and a worker process expressed as a `Flow` create a pool of workers that automatically balances incoming jobs to available workers, then merges the results.

We will express our solution as a function that takes a worker flow and the number of workers to be allocated and gives a flow that internally contains a pool of these workers. To achieve the desired result we will create a `Flow` from a graph.

The graph consists of a `Balance` node which is a special fan-out operation that tries to route elements to available downstream consumers. In a `for` loop we wire all of our desired workers as outputs of this balancer element, then we wire the outputs of these workers to a `Merge` element that will collect the results from the workers.

To convert the graph to a `Flow` we need to define special graph nodes that will correspond to the input and output ports of the resulting `Flow`. This is achieved by defining a pair of undefined sink and source which we return from the builder block.

```
def balancer[In, Out](worker: Flow[In, Out, Any], workerCount: Int): Flow[In, Out, Unit] = {
  import FlowGraph.Implicits._

  Flow() { implicit b =>
    val balancer = b.add(Balance[In](workerCount, waitForAllDownstreams = true))
    val merge = b.add(Merge[Out](workerCount))

    for (_ <- 1 to workerCount) {
      // for each worker, add an edge from the balancer to the worker, then wire
      // it to the merge element
      balancer ~> worker ~> merge
    }

    (balancer.in, merge.out)
  }
}

val processedJobs: Source[Result, Unit] = myJobs.via(balancer(worker, 3))
```

1.14.4 Working with rate

This collection of recipes demonstrate various patterns where rate differences between upstream and downstream needs to be handled by other strategies than simple backpressure.

Dropping elements

Situation: Given a fast producer and a slow consumer, we want to drop elements if necessary to not slow down the producer too much.

This can be solved by using the most versatile rate-transforming operation, `conflate`. Conflate can be thought as a special `fold` operation that collapses multiple upstream elements into one aggregate element if needed to keep the speed of the upstream unaffected by the downstream.

When the upstream is faster, the fold process of the `conflate` starts. This folding needs a zero element, which is given by a `seed` function that takes the current element and produces a zero for the folding process. In our case this is `identity` so our folding state starts form the message itself. The folder function is also special: given the aggregate value (the last message) and the new element (the freshest element) our aggregate state becomes simply the freshest element. This choice of functions results in a simple dropping operation.

```
val droppyStream: Flow[Message, Message, Unit] =
  Flow[Message].conflate(seed = identity)((lastMessage, newMessage) => newMessage)
```

Dropping broadcast

Situation: The default `Broadcast` graph element is properly backpressured, but that means that a slow downstream consumer can hold back the other downstream consumers resulting in lowered throughput. In other words the rate of `Broadcast` is the rate of its slowest downstream consumer. In certain cases it is desirable to allow faster consumers to progress independently of their slower siblings by dropping elements if necessary.

One solution to this problem is to append a `buffer` element in front of all of the downstream consumers defining a dropping strategy instead of the default `Backpressure`. This allows small temporary rate differences between the different consumers (the buffer smooths out small rate variances), but also allows faster consumers to progress by dropping from the buffer of the slow consumers if necessary.

```
val graph = FlowGraph.closed(mySink1, mySink2, mySink3)((_, _, _)) { implicit b =>
  (sink1, sink2, sink3) =>
    import FlowGraph.Implicits._

    val bcast = b.add(Broadcast[Int](3))
    myElements ~> bcast

    bcast.buffer(10, OverflowStrategy.dropHead) ~> sink1
    bcast.buffer(10, OverflowStrategy.dropHead) ~> sink2
    bcast.buffer(10, OverflowStrategy.dropHead) ~> sink3
}
```

Collecting missed ticks

Situation: Given a regular (stream) source of ticks, instead of trying to backpressure the producer of the ticks we want to keep a counter of the missed ticks instead and pass it down when possible.

We will use `conflate` to solve the problem. Conflate takes two functions:

- A seed function that produces the zero element for the folding process that happens when the upstream is faster than the downstream. In our case the seed function is a constant function that returns 0 since there were no missed ticks at that point.
- A fold function that is invoked when multiple upstream messages needs to be collapsed to an aggregate value due to the insufficient processing rate of the downstream. Our folding function simply increments the currently stored count of the missed ticks so far.

As a result, we have a stream of `Int` where the number represents the missed ticks. A number 0 means that we were able to consume the tick fast enough (i.e. zero means: 1 non-missed tick + 0 missed ticks)

```
// tickStream is a Source[Tick]
val missedTicks: Source[Int, Unit] =
  tickStream.conflate(seed = (_) => 0)(
    (missedTicks, tick) => missedTicks + 1)
```

Create a stream processor that repeats the last element seen

Situation: Given a producer and consumer, where the rate of neither is known in advance, we want to ensure that none of them is slowing down the other by dropping earlier unconsumed elements from the upstream if necessary, and repeating the last value for the downstream if necessary.

We have two options to implement this feature. In both cases we will use `DetachedStage` to build our custom element (`DetachedStage` is specifically designed for rate translating elements just like `conflate`, `expand` or `buffer`). In the first version we will use a provided initial value `initial` that will be used to feed the downstream if no upstream element is ready yet. In the `onPush()` handler we just overwrite the `currentValue` variable and immediately relieve the upstream by calling `pull()` (remember, implementations of `DetachedStage` are not allowed to call `push()` as a response to `onPush()` or call `pull()` as a response of `onPull()`). The downstream `onPull` handler is very similar, we immediately relieve the downstream by emitting `currentValue`.

```
import akka.stream.stage._
class HoldWithInitial[T](initial: T) extends DetachedStage[T, T] {
  private var currentValue: T = initial

  override def onPush(elem: T, ctx: DetachedContext[T]): UpstreamDirective = {
    currentValue = elem
    ctx.pull()
  }

  override def onPull(ctx: DetachedContext[T]): DownstreamDirective = {
    ctx.push(currentValue)
  }

}
```

While it is relatively simple, the drawback of the first version is that it needs an arbitrary initial element which is not always possible to provide. Hence, we create a second version where the downstream might need to wait in one single case: if the very first element is not yet available.

We introduce a boolean variable `waitingFirstValue` to denote whether the first element has been provided or not (alternatively an `Option` can be used for `currentValue` of if the element type is a subclass of AnyRef a null can be used with the same purpose). In the downstream `onPull()` handler the difference from the previous version is that we call `hold()` if the first element is not yet available and thus blocking our downstream. The upstream `onPush()` handler sets `waitingFirstValue` to false, and after checking if `hold()` has been called it either releaves the upstream producer, or both the upstream producer and downstream consumer by calling `pushAndPull()`

```
import akka.stream.stage._
class HoldWithWait[T] extends DetachedStage[T, T] {
  private var currentValue: T = _
  private var waitingFirstValue = true

  override def onPush(elem: T, ctx: DetachedContext[T]): UpstreamDirective = {
    currentValue = elem
    waitingFirstValue = false
    if (ctx.isHoldingDownstream) ctx.pushAndPull(currentValue)
    else ctx.pull()
  }

  override def onPull(ctx: DetachedContext[T]): DownstreamDirective = {
    if (waitingFirstValue) ctx.holdDownstream()
    else ctx.push(currentValue)
  }

}
```

Globally limiting the rate of a set of streams

Situation: Given a set of independent streams that we cannot merge, we want to globally limit the aggregate throughput of the set of streams.

One possible solution uses a shared actor as the global limiter combined with mapAsync to create a reusable `Flow` that can be plugged into a stream to limit its rate.

As the first step we define an actor that will do the accounting for the global rate limit. The actor maintains a timer, a counter for pending permit tokens and a queue for possibly waiting participants. The actor has an `open` and `closed` state. The actor is in the `open` state while it has still pending permits. Whenever a request for permit arrives as a `WantToPass` message to the actor the number of available permits is decremented and we notify the sender that it can pass by answering with a `MayPass` message. If the amount of permits reaches zero, the actor transitions to the `closed` state. In this state requests are not immediately answered, instead the reference of the sender is added to a queue. Once the timer for replenishing the pending permits fires by sending a `ReplenishTokens` message, we increment the pending permits counter and send a reply to each of the waiting senders. If there are more waiting senders than permits available we will stay in the `closed` state.

```
object Limiter {
  case object WantToPass
  case object MayPass

  case object ReplenishTokens

  def props(maxAvailableTokens: Int, tokenRefreshPeriod: FiniteDuration, tokenRefreshAmount: Int)
    Props(new Limiter(maxAvailableTokens, tokenRefreshPeriod, tokenRefreshAmount))
}

class Limiter(
  val maxAvailableTokens: Int,
  val tokenRefreshPeriod: FiniteDuration,
  val tokenRefreshAmount: Int) extends Actor {
  import Limiter._
  import context.dispatcher
  import akka.actor.Status

  private var waitQueue = immutable.Queue.empty[ActorRef]
  private var permitTokens = maxAvailableTokens
  private val replenishTimer = system.scheduler.schedule(
    initialDelay = tokenRefreshPeriod,
    interval = tokenRefreshPeriod,
    receiver = self,
    ReplenishTokens)

  override def receive: Receive = open

  val open: Receive = {
    case ReplenishTokens =>
      permitTokens = math.min(permitTokens + tokenRefreshAmount, maxAvailableTokens)
    case WantToPass =>
      permitTokens -= 1
      sender() ! MayPass
      if (permitTokens == 0) context.become(closed)
  }

  val closed: Receive = {
    case ReplenishTokens =>
      permitTokens = math.min(permitTokens + tokenRefreshAmount, maxAvailableTokens)
      releaseWaiting()
    case WantToPass =>
      waitQueue = waitQueue.enqueue(sender())
  }
```

```
  private def releaseWaiting(): Unit = {
    val (toBeReleased, remainingQueue) = waitQueue.splitAt(permitTokens)
    waitQueue = remainingQueue
    permitTokens -= toBeReleased.size
    toBeReleased foreach (_ ! MayPass)
    if (permitTokens > 0) context.become(open)
  }

  override def postStop(): Unit = {
    replenishTimer.cancel()
    waitQueue foreach (_ ! Status.Failure(new IllegalStateException("limiter stopped")))
  }
}
```

To create a Flow that uses this global limiter actor we use the `mapAsync` function with the combination of the `ask` pattern. We also define a timeout, so if a reply is not received during the configured maximum wait period the returned future from `ask` will fail, which will fail the corresponding stream as well.

```
def limitGlobal[T](limiter: ActorRef, maxAllowedWait: FiniteDuration): Flow[T, T, Unit] = {
  import akka.pattern.ask
  import akka.util.Timeout
  Flow[T].mapAsync(4)((element: T) => {
    import system.dispatcher
    implicit val triggerTimeout = Timeout(maxAllowedWait)
    val limiterTriggerFuture = limiter ? Limiter.WantToPass
    limiterTriggerFuture.map((_) => element)
  })

}
```

Note: The global actor used for limiting introduces a global bottleneck. You might want to assign a dedicated dispatcher for this actor.

1.14.5 Working with IO

Chunking up a stream of ByteStrings into limited size ByteStrings

Situation: Given a stream of ByteStrings we want to produce a stream of ByteStrings containing the same bytes in the same sequence, but capping the size of ByteStrings. In other words we want to slice up ByteStrings into smaller chunks if they exceed a size threshold.

This can be achieved with a single `PushPullStage`. The main logic of our stage is in `emitChunkOrPull()` which implements the following logic:

- if the buffer is empty, we pull for more bytes
- if the buffer is nonEmpty, we split it according to the `chunkSize`. This will give a next chunk that we will emit, and an empty or nonempty remaining buffer.

Both `onPush()` and `onPull()` calls `emitChunkOrPull()` the only difference is that the push handler also stores the incoming chunk by appending to the end of the buffer.

```
import akka.stream.stage._

class Chunker(val chunkSize: Int) extends PushPullStage[ByteString, ByteString] {
  private var buffer = ByteString.empty

  override def onPush(elem: ByteString, ctx: Context[ByteString]): SyncDirective = {
    buffer ++= elem
    emitChunkOrPull(ctx)
  }
```

```
  override def onPull(ctx: Context[ByteString]): SyncDirective = emitChunkOrPull(ctx)

  private def emitChunkOrPull(ctx: Context[ByteString]): SyncDirective = {
    if (buffer.isEmpty) ctx.pull()
    else {
      val (emit, nextBuffer) = buffer.splitAt(chunkSize)
      buffer = nextBuffer
      ctx.push(emit)
    }
  }

}

val chunksStream = rawBytes.transform(() => new Chunker(ChunkLimit))
```

Limit the number of bytes passing through a stream of ByteStrings

Situation: Given a stream of ByteStrings we want to fail the stream if more than a given maximum of bytes has been consumed.

This recipe uses a `PushStage` to implement the desired feature. In the only handler we override, `onPush()` we just update a counter and see if it gets larger than `maximumBytes`. If a violation happens we signal failure, otherwise we forward the chunk we have received.

```
import akka.stream.stage._
class ByteLimiter(val maximumBytes: Long) extends PushStage[ByteString, ByteString] {
  private var count = 0

  override def onPush(chunk: ByteString, ctx: Context[ByteString]): SyncDirective = {
    count += chunk.size
    if (count > maximumBytes) ctx.fail(new IllegalStateException("Too much bytes"))
    else ctx.push(chunk)
  }
}

val limiter = Flow[ByteString].transform(() => new ByteLimiter(SizeLimit))
```

Compact ByteStrings in a stream of ByteStrings

Situation: After a long stream of transformations, due to their immutable, structural sharing nature ByteStrings may refer to multiple original ByteString instances unnecessarily retaining memory. As the final step of a transformation chain we want to have clean copies that are no longer referencing the original ByteStrings.

The recipe is a simple use of map, calling the `compact()` method of the `ByteString` elements. This does copying of the underlying arrays, so this should be the last element of a long chain if used.

```
val compacted: Source[ByteString, Unit] = data.map(_.compact)
```

Injecting keep-alive messages into a stream of ByteStrings

Situation: Given a communication channel expressed as a stream of ByteStrings we want to inject keep-alive messages but only if this does not interfere with normal traffic.

All this recipe needs is the `MergePreferred` element which is a version of a merge that is not fair. In other words, whenever the merge can choose because multiple upstream producers have elements to produce it will always choose the preferred upstream effectively giving it an absolute priority.

```
val keepAliveStream: Source[ByteString, Unit] = ticks
  .conflate(seed = (tick) => keepaliveMessage)((msg, newTick) => msg)

val graph = FlowGraph.closed() { implicit builder =>
  import FlowGraph.Implicits._
  val unfairMerge = builder.add(MergePreferred[ByteString](1))

  dataStream ~> unfairMerge.preferred
  // If data is available then no keepalive is injected
  keepAliveStream ~> unfairMerge ~> sink
}
```

1.15 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
      }
    }
  }
}
```

CHAPTER

TWO

AKKA HTTP

The purpose of the Akka HTTP layer is to expose Actors to the web via HTTP and to enable them to consume HTTP services as a client. It is not an HTTP framework, it is an Actor-based toolkit for interacting with web services and clients. This toolkit is structured into several layers:

- `akka-http-core`: A complete implementation of the HTTP standard, both as client and server.
- `akka-http`: A convenient and powerful routing DSL for expressing web services.
- `akka-http-testkit`: A test harness and set of utilities for verifying your web service implementations.

Additionally there are several integration modules with bindings for different serialization and deserialization schemes for HTTP entities.

2.1 HTTP Client & Server

2.1.1 Model

The akka HTTP model contains a mostly immutable, case-class based model of the major HTTP data structures, like HTTP requests, responses and common headers. It also includes a parser for the latter, which is able to construct the more structured header models from raw unstructured header name/value pairs.

Overview

Since akka-http-core provides the central HTTP data structures you will find the following import in quite a few places around the code base (and probably your own code as well):

```
import akka.http.scaladsl.model._
```

This brings in scope all of the relevant things that are defined here and that you'll want to work with, mainly:

- `HttpRequest` and `HttpResponse`, the central message model
- `headers`, the package containing all the predefined HTTP header models and supporting types
- Supporting types like `Uri`, `HttpMethods`, `MediaTypes`, `StatusCodes`, etc.

A common pattern is that the model of a certain entity is represented by an immutable type (class or trait), while the actual instances of the entity defined by the HTTP spec live in an accompanying object carrying the name of the type plus a trailing plural 's'.

For example:

- Defined HttpMethod instances live in the HttpMethods object.
- Defined HttpCharset instances live in the HttpCharsets object.
- Defined HttpEncoding instances live in the HttpEncodings object.
- Defined HttpProtocol instances live in the HttpProtocols object.

- Defined MediaType instances live in the MediaTypes object.
- Defined StatusCode instances live in the StatusCodes object.

HttpRequest / HttpResponse

`HttpRequest` and `HttpResponse` are the basic case classes representing HTTP messages.

An `HttpRequest` consists of

- method (GET, POST, etc.)
- URI
- protocol
- headers
- entity (body data)

Here are some examples how to construct an `HttpRequest`:

```
import HttpMethods._

// construct simple GET request to `homeUri`
val homeUri = Uri("/abc")
HttpRequest(GET, uri = homeUri)

// construct simple GET request to "/index" which is converted to Uri automatically
HttpRequest(GET, uri = "/index")

// construct simple POST request containing entity
val data = ByteString("abc")
HttpRequest(POST, uri = "/receive", entity = data)

// customize every detail of HTTP request
import HttpProtocols._
import MediaTypes._
val userData = ByteString("abc")
val authorization = headers.Authorization(BasicHttpCredentials("user", "pass"))
HttpRequest(
  PUT,
  uri = "/user",
  entity = HttpEntity(`text/plain`, userData),
  headers = List(authorization),
  protocol = `HTTP/1.0`)
```

All parameters of `HttpRequest` have default values set, so e.g. `headers` don't need to be specified if there are none. Many of the parameters types (like `HttpEntity` and `Uri`) define implicit conversions for common use cases to simplify the creation of request and response instances.

An `HttpResponse` consists of

- status code
- protocol
- headers
- entity

Here are some examples how to construct an `HttpResponse`:

```
import StatusCodes._

// simple OK response without data created using the integer status code
HttpResponse(200)
```

```
// 404 response created using the named StatusCode constant
HttpResponse(NotFound)

// 404 response with a body explaining the error
HttpResponse(404, entity = "Unfortunately, the resource couldn't be found.")

// A redirecting response containing an extra header
val locationHeader = headers.Location("http://example.com/other")
HttpResponse(Found, headers = List(locationHeader))
```

Aside from the simple `HttpEntity` constructors to create an entity from a fixed `ByteString` shown here, subclasses of `HttpEntity` allow data to be specified as a stream of data which is explained in the next section.

HttpEntity

An `HttpEntity` carries the content of a request together with its content-type which is needed to interpret the raw byte data.

Akka HTTP provides several kinds of entities to support static and streaming data for the different kinds of ways to transport streaming data with HTTP. There are four subtypes of HttpEntity:

HttpEntity.Strict An entity which wraps a static `ByteString`. It represents a standard, non-chunked HTTP message with `Content-Length` set.

HttpEntity.Default A streaming entity which needs a predefined length and a `Producer[ByteString]` to produce the body data of the message. It represents a standard, non-chunked HTTP message with `Content-Length` set. It is an error if the provided `Producer[ByteString]` doesn't produce exactly as many bytes as specified. On the wire, a Strict entity and a Default entity cannot be distinguished. However, they offer a valuable alternative in the API to distinguish between strict and streamed data.

HttpEntity.Chunked A streaming entity of unspecified length that uses Chunked Transfer Coding for transmitting data. Data is represented by a `Producer[ChunkStreamPart]`. A `ChunkStreamPart` is either a non-empty `Chunk` or a `LastChunk` containing optional trailer headers. The stream must consist of 0..n `Chunked` parts and can be terminated by an optional `LastChunk` part (which carries optional trailer headers).

HttpEntity.CloseDelimited A streaming entity of unspecified length that is delimited by closing the connection ("Connection: close"). Note, that this entity type can only be used in an `HttpResponse`.

HttpEntity.IndefiniteLength A streaming entity of unspecified length that can be used as a `BodyPart` entity.

Entity types `Strict`, `Default`, and `Chunked` are a subtype of `HttpEntity.Regular` which allows to use them for requests and responses. In contrast, `HttpEntity.CloseDelimited` can only be used for responses.

Streaming entity types (i.e. all but `Strict`) cannot be shared or serialized. To create a strict, sharable copy of an entity or message use `HttpEntity.toStrict` or `HttpMessage.toStrict` which returns a Future of the object with the body data collected into a `ByteString`.

The `HttpEntity` companion object contains several helper constructors to create entities from common types easily.

You can pattern match over the subtypes of `HttpEntity` if you want to provide special handling for each of the subtypes. However, in many cases a recipient of an *HttpEntity* doesn't care about of which subtype an entity is (and how data is transported exactly on the HTTP layer). Therefore, a general `HttpEntity.dataBytes` is provided which allows access to the data of an entity regardless of its concrete subtype.

Note:

When to use which subtype?

- Use Strict if the amount of data is small and it is already in the heap (or even available as a `ByteString`)
- Use Default if the data is generated by a streaming data source and the size of the data is fixed

- Use Chunked to support a data stream of unknown length
- Use CloseDelimited for a response as an alternative to Chunked e.g. if chunked transfer encoding isn't supported by a client.
- Use IndefiniteLength instead of CloseDelimited in a BodyPart.

Header model

Akka HTTP contains a rich model of the common HTTP headers. Parsing and rendering is done automatically so that applications don't need to care for the actual syntax of headers. Headers not modelled explicitly are represented as a `RawHeader`.

See these examples of how to deal with headers:

```
import akka.http.scaladsl.model.headers._

// create a ``Location`` header
val loc = Location("http://example.com/other")

// create an ``Authorization`` header with HTTP Basic authentication data
val auth = Authorization(BasicHttpCredentials("joe", "josepp"))

// a method that extracts basic HTTP credentials from a request
case class User(name: String, pass: String)
def credentialsOfRequest(req: HttpRequest): Option[User] =
  for {
    Authorization(BasicHttpCredentials(user, pass)) <- req.header[headers.Authorization]
  } yield User(user, pass)
```

Parsing / Rendering

Parsing and rendering of HTTP data structures is heavily optimized and for most types there's currently no public API provided to parse (or render to) Strings or byte arrays.

2.1.2 Client API

(todo)

2.1.3 Server API

The Akka HTTP server is an embedded, stream-based, fully asynchronous, low-overhead HTTP/1.1 server implemented on top of *Streams*.

It sports the following features:

- Full support for HTTP persistent connections
- Full support for HTTP pipelining
- Full support for asynchronous HTTP streaming including "chunked" transfer encoding accessible through an idiomatic reactive streams API
- Optional SSL/TLS encryption

Design Philosophy

Akka HTTP server is scoped with a clear focus on the essential functionality of an HTTP/1.1 server:

- Connection management
- Parsing messages and headers
- Timeout management (for requests and connections)
- Response ordering (for transparent pipelining support)

All non-core features of typical HTTP servers (like request routing, file serving, compression, etc.) are left to the next-higher layer in the application stack, they are not implemented by the HTTP server itself. Apart from general focus this design keeps the server small and light-weight as well as easy to understand and maintain.

Basic Architecture

Akka HTTP server is implemented on top of Akka streams and makes heavy use of them - in its implementation and also on all levels of its API.

On the connection level Akka HTTP supports basically the same interface as Akka streams IO: A socket binding is represented as a stream of incoming connections. The application needs to provide a `Flow[HttpRequest, HttpResponse]` to "translate" requests into responses.

Streaming is also supported for single message entities itself. Particular kinds of `HttpEntity` subclasses provide support for fixed or streamed message entities.

Starting and Stopping

An Akka HTTP server is bound by invoking the `bind` method of the akka.http.Http extension:

```
import akka.http.scaladsl.Http
import akka.stream.ActorFlowMaterializer

implicit val system = ActorSystem()
implicit val materializer = ActorFlowMaterializer()

val serverSource: Source[Http.IncomingConnection, Future[Http.ServerBinding]] =
  Http(system).bind(interface = "localhost", port = 8080)
val bindingFuture: Future[Http.ServerBinding] = serverSource.to(Sink.foreach { connection =>
  // foreach materializes the source
  println("Accepted new connection from " + connection.remoteAddress)
  // ... and then actually handle the connection
}).run()
```

Arguments to the `Http.bind` method specify the interface and port to bind to and register interest in handling incoming HTTP connections. Additionally, the method also allows you to define socket options as well as a larger number of settings for configuring the server according to your needs.

The result of the `bind` method is a `Source[IncomingConnection]` which is used to handle incoming connections. The actual binding is only done when this source is materialized as part of a bigger processing pipeline. In case the bind fails (e.g. because the port is already busy) the error will be reported by flagging an error on the materialized stream. The binding is released and the underlying listening socket is closed when all subscribers of the source have cancelled their subscription.

Connections are handled by materializing a pipeline which uses the `Source[IncomingConnection]`. This source materializes to `Future[ServerBinding]` which is completed when server successfully binds to the specified port. After materialization `ServerBinding.localAddress` returns the actual local address of the bound socket. `ServerBinding.unbind()` can be used to asynchronously trigger unbinding of the server port.

(todo: explain even lower level serverFlowToTransport API)

Request-Response Cycle

When a new connection has been accepted it will be published as an `Http.IncomingConnection` which consists of the remote address, and methods to provide a `Flow[HttpRequest, HttpResponse]` to handle requests coming in over this connection.

Requests are handled by calling one of the `IncomingConnection.handleWithX` methods with a handler, which can either be

- a `Flow[HttpRequest, HttpResponse]` for `handleWith`,
- a function `HttpRequest => HttpResponse` for `handleWithSyncHandler`,
- or a function `HttpRequest => Future[HttpResponse]` for `handleWithAsyncHandler`.

```
import akka.http.scaladsl.model.HttpMethods._
import akka.stream.scaladsl.{ Flow, Sink }

val requestHandler: HttpRequest => HttpResponse = {
  case HttpRequest(GET, Uri.Path("/"), _, _, _) =>
    HttpResponse(
      entity = HttpEntity(MediaTypes.`text/html`,
        "<html><body>Hello world!</body></html>"))

  case HttpRequest(GET, Uri.Path("/ping"), _, _, _) =>
    HttpResponse(entity = "PONG!")

  case HttpRequest(GET, Uri.Path("/crash"), _, _, _) =>
    sys.error("BOOM!")

  case _: HttpRequest =>
    HttpResponse(404, entity = "Unknown resource!")
}

val bindingFuture: Future[Http.ServerBinding] = serverSource.to(Sink.foreach { connection =>
  println("Accepted new connection from " + connection.remoteAddress)

  connection handleWithSyncHandler requestHandler
  // this is equivalent to
  // connection handleWith { Flow[HttpRequest] map requestHandler }
}).run()
```

In this case, a request is handled by transforming the request stream with a function `HttpRequest => HttpResponse` using `handleWithSyncHandler` (or equivalently, Akka stream's `map` operator). Depending on the use case, arbitrary other ways of connecting are conceivable using Akka stream's combinators.

If the application provides a `Flow`, it is also the responsibility of the application to generate exactly one response for every request and that the ordering of responses matches the ordering of the associated requests (which is relevant if HTTP pipelining is enabled where processing of multiple incoming requests may overlap). Using `handleWithSyncHandler` or `handleWithAsyncHandler` or, instead, if using stream operators like `map` or `mapFuture` this requirement will automatically be fulfilled.

Streaming request/response entities

Streaming of HTTP message entities is supported through subclasses of `HttpEntity`. You need to be able to deal with streamed entities when receiving a request as well as when constructing responses. See *HttpEntity* for a description of the alternatives.

(todo): Link to *The Routing DSL* for (un-)marshalling facilities.

Closing a connection

The HTTP connection will be closed when the handling `Flow` cancel its upstream subscription or the peer closes the connection.

You can also use the value of the `Connection` header of a response as described below to give a hint to the implementation to close the connection after the completion of the response.

HTTP Headers

When the Akka HTTP server receives an HTTP request it tries to parse all its headers into their respective model classes. No matter whether this succeeds or not, the connection actor will always pass on all received headers to the application. Unknown headers as well as ones with invalid syntax (according to the header parser) will be made available as `RawHeader` instances. For the ones exhibiting parsing errors a warning message is logged depending on the value of the `illegal-header-warnings` config setting.

Some common headers are treated specially in the model and in the implementation and should not occur in the `headers` field of an HTTP message:

- `Content-Type`: Use the `contentType` field of the `HttpEntity` subclasses to set or determine the content-type on an entity.
- `Transfer-Encoding`: The `Transfer-Encoding` is represented by subclasses of `HttpEntity`.
- `Content-Length`: The `Content-Length` header is represented implicitly by the choice of an `HttpEntity` subclass: A Strict entity determines the Content-Length by the length of the data provided. A Default entity has an explicit `contentLength` field which specifies the amount of data the streaming producer will produce. Chunked and CloseDelimited entities don't need to define a length.
- `Server`: The `Server` header is usually added automatically and it's value can be configured. An application can decide to provide a custom `Server` header by including an explicit instance in the response.
- `Date`: The `Date` header is added automatically and can be overridden by supplying it manually.
- `Connection`: When sending out responses the connection actor watches for a `Connection` header set by the application and acts accordingly, i.e. you can force the connection actor to close the connection after having sent the response by including a `Connection("close")` header. To unconditionally force a connection keep-alive you can explicitly set a `Connection("Keep-Alive")` header. If you don't set an explicit `Connection` header the connection actor will keep the connection alive if the client supports this (i.e. it either sent a `Connection: Keep-Alive` header or advertised HTTP/1.1 capabilities without sending a `Connection: close` header).

SSL Support

(todo)

2.1.4 HTTPS

Is not yet supported.

(todo)

2.2 HTTP Routing

2.2.1 Quick-Start

(todo)

2.2.2 The Routing DSL

Routes

The "Route" is the central concept of the routing DSL since all structures you can build with it are instances of a `Route`. The type Route is defined like this:

```
type Route = RequestContext => Future[RouteResult]
```

It's a simple alias for a function taking a `RequestContext` as parameter and returning a `Future[RouteResult]`.

Generally when a route receives a request (or rather a `RequestContext` for it) it can do one of these things:

- Complete the request by returning the value of `requestContext.complete(...)`
- Reject the request by returning the value of `requestContext.reject(...)` (see *Rejections*)
- Fail the request by returning the value of `requestContext.fail(...)` or by just throwing an exception (see *Exception Handling*)
- Do any kind of asynchronous processing and instantly return a `Future[RouteResult]` to be eventually completed later on

The first case is pretty clear, by calling `complete` a given response is sent to the client as reaction to the request. In the second case "reject" means that the route does not want to handle the request. You'll see further down in the section about route composition what this is good for.

A `Route` can be sealed using `Route.seal` and by supplying a `RejectionHandler` and an `ExceptionHandler` which converts rejection results and exceptional results into appropriate HTTP responses for the peer.

A `Route` can be lifted into a handler to be used with the http-core API using `Route.handlerFlow` or `Route.asyncHandler`.

RequestContext

The request context wraps a request together with additional contextual information to be passed through the route tree. Also, it provides the only way of creating a `RouteResult` by calling one of the above methods.

In addition to the request it contains the `unmatchedPath`, a value that describes how much of the request URI has not yet been matched and instances of several configuration instances like a `LoggingAdapter`, an `ExecutionContext`, and `RoutingSettings`, so that they don't have to be passed around explicitly.

The `RequestContext` itself is immutable but contains several helper methods to create updated versions.

Composing Routes

There are three basic operations we need for building more complex routes from simpler ones:

- Route transformation, which delegates processing to another, "inner" route but in the process changes some properties of either the incoming request, the outgoing response or both
- Route filtering, which only lets requests satisfying a given filter condition pass and rejects all others
- Route chaining, which tries a second route if a given first one was rejected

The last point is achieved with the concatenation operator ~, which is an extension method provided by an implicit in `RouteConcatenation`. The first two points are provided by so-called *Directives*, of which a large number is already predefined by Akka HTTP and which you can also easily create yourself. *Directives* deliver most of Akka HTTP's power and flexibility.

The Routing Tree

Essentially, when you combine directives and custom routes via nesting and the ~ operator, you build a routing structure that forms a tree. When a request comes in it is injected into this tree at the root and flows down through all the branches in a depth-first manner until either some node completes it or it is fully rejected.

Consider this schematic example:

```
val route =
  a {
    b {
      c {
        ... // route 1
      } ~
      d {
        ... // route 2
      } ~
      ... // route 3
    } ~
    e {
      ... // route 4
    }
  }
```

Here five directives form a routing tree.

- Route 1 will only be reached if directives `a`, `b` and `c` all let the request pass through.
- Route 2 will run if `a` and `b` pass, `c` rejects and `d` passes.
- Route 3 will run if `a` and `b` pass, but `c` and `d` reject.

Route 3 can therefore be seen as a "catch-all" route that only kicks in, if routes chained into preceding positions reject. This mechanism can make complex filtering logic quite easy to implement: simply put the most specific cases up front and the most general cases in the back.

2.2.3 Directives

A "Directive" is a small building block to construct arbitrarily complex route structures. Here is a simple example of a route built from directives:

```
val route: Route =
  path("order" / IntNumber) { id =>
    get {
      complete {
        "Received GET request for order " + id
      }
    } ~
      put {
        complete {
          "Received PUT request for order " + id
        }
      }
  }
```

The general anatomy of a directive is as follows:

```
name(arguments) { extractions =>
  ... // inner Route
}
```

It has a name, zero or more arguments and optionally an inner Route. Additionally directives can "extract" a number of values and make them available to their inner routes as function arguments. When seen "from the outside" a directive with its inner Route form an expression of type `Route`.

What Directives do

A directive can do one or more of the following:

- Transform the incoming `RequestContext` before passing it on to its inner Route
- Filter the `RequestContext` according to some logic, i.e. only pass on certain requests and reject all others
- Extract values from the `RequestContext` and make them available to its inner Route as "extractions"
- Complete the request

The first point deserves some more discussion. The `RequestContext` is the central object that is passed on through a route structure (also see *RequestContext*). When a directive (or better the Route it built) receives a `RequestContext` it can decide to pass this instance on unchanged to its inner Route or it can create a copy of the `RequestContext` instance, with one or more changes, and pass on this copy to its inner Route. Typically this is good for two things:

- Transforming the `HttpRequest` instance
- "Hooking in" a response transformation function that changes the `RouteResponse` (and therefore the response).

This means a `Directive` completely wraps the functionality of its inner routes and can apply arbitrarily complex transformations, both (or either) on the request and on the response side.

Composing Directives

As you have seen from the examples presented so far the "normal" way of composing directives is nesting. Let's take another look at the example from above:

```
val route: Route =
  path("order" / IntNumber) { id =>
    get {
      complete {
        "Received GET request for order " + id
      }
    } ~
      put {
        complete {
          "Received PUT request for order " + id
        }
      }
  }
```

Here the `get` and `put` directives are chained together with the ~ operator to form a higher-level route that serves as the inner Route of the `path` directive. To make this structure more explicit you could also write the whole thing like this:

```
def innerRoute(id: Int): Route =
  get {
    complete {
      "Received GET request for order " + id
    }
  } ~
    put {
      complete {
        "Received PUT request for order " + id
      }
    }

val route: Route = path("order" / IntNumber) { id => innerRoute(id) }
```

What you can't see from this snippet is that directives are not implemented as simple methods but rather as stand-alone objects of type `Directive`. This gives you more flexibility when composing directives. For example you can also use the `|` operator on directives. Here is yet another way to write the example:

```
val route =
  path("order" / IntNumber) { id =>
    (get | put) { ctx =>
      ctx.complete("Received " + ctx.request.method.name + " request for order " + id)
    }
  }
```

If you have a larger route structure where the `(get | put)` snippet appears several times you could also factor it out like this:

```
val getOrPut = get | put
val route =
  path("order" / IntNumber) { id =>
    getOrPut { ctx =>
      ctx.complete("Received " + ctx.request.method.name + " request for order " + id)
    }
  }
```

As an alternative to nesting you can also use the *&* operator:

```
val getOrPut = get | put
val route =
  (path("order" / IntNumber) & getOrPut) { id =>
    ctx =>
      ctx.complete("Received " + ctx.request.method.name + " request for order " + id)
  }
```

And once again, you can factor things out if you want:

```
val orderGetOrPut = path("order" / IntNumber) & (get | put)
val route =
  orderGetOrPut { id =>
    ctx =>
      ctx.complete("Received " + ctx.request.method.name + " request for order " + id)
  }
```

This type of combining directives with the `|` and `&` operators as well as "saving" more complex directive configurations as a `val` works across the board, with all directives taking inner routes.

There is one more "ugly" thing remaining in our snippet: we have to fall back to the lowest-level route definition, directly manipulating the `RequestContext`, in order to get to the request method. It'd be nicer if we could somehow "extract" the method name in a special directive, so that we can express our inner-most route with a simple `complete`. As it turns out this is easy with the `extract` directive:

```
val orderGetOrPut = path("order" / IntNumber) & (get | put)
val requestMethod = extract(_.request.method)
val route =
  orderGetOrPut { id =>
    requestMethod { m =>
      complete("Received " + m.name + " request for order " + id)
    }
  }
```

Or differently:

```
val orderGetOrPut = path("order" / IntNumber) & (get | put)
val requestMethod = extract(_.request.method)
val route =
  (orderGetOrPut & requestMethod) { (id, m) =>
    complete("Received " + m.name + " request for order " + id)
  }
```

Now, pushing the "factoring out" of directive configurations to its extreme, we end up with this:

```
val orderGetOrPutMethod =
  path("order" / IntNumber) & (get | put) & extract(_.request.method)
val route =
  orderGetOrPutMethod { (id, m) =>
    complete("Received " + m.name + " request for order " + id)
  }
```

Note that going this far with "compressing" several directives into a single one probably doesn't result in the most readable and therefore maintainable routing code. It might even be that the very first of this series of examples is in fact the most readable one.

Still, the purpose of the exercise presented here is to show you how flexible directives can be and how you can use their power to define your web service behavior at the level of abstraction that is right for **your** application.

Type Safety of Directives

When you combine directives with the `|` and `&` operators the routing DSL makes sure that all extractions work as expected and logical constraints are enforced at compile-time.

For example you cannot `|` a directive producing an extraction with one that doesn't:

```
val route = path("order" / IntNumber) | get // doesn't compile
```

Also the number of extractions and their types have to match up:

```
val route = path("order" / IntNumber) | path("order" / DoubleNumber)    // doesn't compile
val route = path("order" / IntNumber) | parameter('order.as[Int])       // ok
```

When you combine directives producing extractions with the `&` operator all extractions will be properly gathered up:

```
val order = path("order" / IntNumber) & parameters('oem, 'expired ?)
val route =
  order { (orderId, oem, expired) =>
    ...
  }
```

Directives offer a great way of constructing your web service logic from small building blocks in a plug and play fashion while maintaining DRYness and full type-safety. If the large range of *Predefined Directives (alphabetically)* does not fully satisfy your needs you can also very easily create *Custom Directives*.

2.2.4 Rejections

In the chapter about constructing *Routes* the ~ operator was introduced, which connects two routes in a way that allows a second route to get a go at a request if the first route "rejected" it. The concept of "rejections" is used by Akka HTTP for maintaining a more functional overall architecture and in order to be able to properly handle all kinds of error scenarios.

When a filtering directive, like the *-get-* directive, cannot let the request pass through to its inner Route because the filter condition is not satisfied (e.g. because the incoming request is not a GET request) the directive doesn't immediately complete the request with an error response. Doing so would make it impossible for other routes chained in after the failing filter to get a chance to handle the request. Rather, failing filters "reject" the request in the same way as by explicitly calling `requestContext.reject(...)`.

After having been rejected by a route the request will continue to flow through the routing structure and possibly find another route that can complete it. If there are more rejections all of them will be picked up and collected.

If the request cannot be completed by (a branch of) the route structure an enclosing *handleRejections* directive can be used to convert a set of rejections into an `HttpResponse` (which, in most cases, will be an error response).

`Route.seal` internally wraps its argument route with the *handleRejections* directive in order to "catch" and handle any rejection.

Predefined Rejections

A rejection encapsulates a specific reason why a Route was not able to handle a request. It is modeled as an object of type `Rejection`. Akka HTTP comes with a set of predefined rejections, which are used by various *predefined directives*.

Rejections are gathered up over the course of a Route evaluation and finally converted to `HttpResponse` replies by the *handleRejections* directive if there was no way for the request to be completed.

RejectionHandler

The *handleRejections* directive delegates the actual job of converting a list of rejections to its argument, a RejectionHandler, which is defined like this:

```
trait RejectionHandler extends PartialFunction[List[Rejection], Route]
```

Since a `RejectionHandler` is a partial function it can choose, which rejections it would like to handle and which not. Unhandled rejections will simply continue to flow through the route structure. The top-most `RejectionHandler` applied by *runRoute* will handle *all* rejections that reach it.

So, if you'd like to customize the way certain rejections are handled simply bring a custom `RejectionHandler` into implicit scope of *runRoute* or pass it to an explicit *handleRejections* directive that you have put somewhere into your route structure.

Here is an example:

```
import akka.http.scaladsl.model._
import akka.http.scaladsl.server._
import StatusCodes._
import Directives._

implicit val myRejectionHandler = RejectionHandler.newBuilder()
  .handle {
    case MissingCookieRejection(cookieName) =>
      complete(HttpResponse(BadRequest, entity = "No cookies, no service!!!"))
  }
  .result()

object MyApp {
  implicit val system = ActorSystem()
  import system.dispatcher
  implicit val materializer = ActorFlowMaterializer()

  def handler = Route.handlerFlow(`<my-route-definition>`)
}
```

Rejection Cancellation

As you can see from its definition above the `RejectionHandler` handles not single rejections but a whole list of them. This is because some route structure produce several "reasons" why a request could not be handled.

Take this route structure for example:

```
import akka.http.scaladsl.coding.Gzip

val route =
  path("order") {
    get {
```

```
      complete("Received GET")
    } ~
      post {
        decodeRequestWith(Gzip) {
          complete("Received POST")
        }
      }
  }
```

For uncompressed POST requests this route structure could yield two rejections:

- a `MethodRejection` produced by the *-get-* directive (which rejected because the request is not a GET request)
- an `UnsupportedRequestEncodingRejection` produced by the *decodeRequest* directive (which only accepts gzip-compressed requests)

In reality the route even generates one more rejection, a `TransformationRejection` produced by the *-post-* directive. It "cancels" all other potentially existing *MethodRejections*, since they are invalid after the *-post-* directive allowed the request to pass (after all, the route structure *can* deal with POST requests). These types of rejection cancellations are resolved *before* a `RejectionHandler` sees the rejection list. So, for the example above the `RejectionHandler` will be presented with only a single-element rejection list, containing nothing but the `UnsupportedRequestEncodingRejection`.

Empty Rejections

Since rejections are passed around in lists you might ask yourself what the semantics of an empty rejection list are. In fact, empty rejection lists have well defined semantics. They signal that a request was not handled because the respective resource could not be found. Akka HTTP reserves the special status of "empty rejection" to this most common failure a service is likely to produce.

So, for example, if the *path* directive rejects a request, it does so with an empty rejection list. The *-host-* directive behaves in the same way.

2.2.5 Exception Handling

Exceptions thrown during route execution bubble up through the route structure to the next enclosing *handleExceptions* directive, `Route.seal` or the `onFailure` callback of a future created by `detach`.

Similarly to the way that *Rejections* are handled the *handleExceptions* directive delegates the actual job of converting a list of rejections to its argument, an ExceptionHandler, which is defined like this:

```
trait ExceptionHandler extends PartialFunction[Throwable, Route]
```

runRoute defined in *HttpService* does the same but gets its `ExceptionHandler` instance implicitly.

Since an `ExceptionHandler` is a partial function it can choose, which exceptions it would like to handle and which not. Unhandled exceptions will simply continue to bubble up in the route structure. The top-most `ExceptionHandler` applied by *runRoute* will handle *all* exceptions that reach it.

So, if you'd like to customize the way certain exceptions are handled simply bring a custom `ExceptionHandler` into implicit scope of *runRoute* or pass it to an explicit *handleExceptions* directive that you have put somewhere into your route structure.

Here is an example:

```
import akka.http.scaladsl.model.HttpResponse
import akka.http.scaladsl.model.StatusCodes._
import akka.http.scaladsl.server._
import Directives._

implicit def myExceptionHandler =
```

```
  ExceptionHandler {
    case e: ArithmeticException =>
      extractUri { uri =>
        logWarning(s"Request to $uri could not be handled normally")
        complete(HttpResponse(InternalServerError, entity = "Bad numbers, bad result!!!"))
      }
  }

object MyApp {
  implicit val system = ActorSystem()
  import system.dispatcher
  implicit val materializer = ActorFlowMaterializer()

  def handler = Route.handlerFlow(`<my-route-definition>`)
}
```

2.2.6 The PathMatcher DSL

For being able to work with the *PathDirectives* effectively you should have some understanding of the `PathMatcher` mini-DSL that Akka HTTP provides for elegantly defining URI matching behavior.

Overview

When a request (or rather the respective `RequestContext` instance) enters the route structure it has an "unmatched path" that is identical to the `request.uri.path`. As it descends the routing tree and passes through one or more *pathPrefix*/*path* directives the "unmatched path" progressively gets "eaten into" from the left until, in most cases, it eventually has been consumed completely.

What exactly gets matched and consumed as well as extracted from the unmatched path in each directive is defined with the patch matching DSL, which is built around these types:

```
trait PathMatcher[L: Tuple]
type PathMatcher0 = PathMatcher[Unit]
type PathMatcher1[T] = PathMatcher[Tuple1[T]]
```

The number and types of the values extracted by a `PathMatcher` instance is represented by the `L` type parameter which needs to be one of Scala's TupleN types or Unit (which is designated by the `Tuple` context bound). The convenience alias `PathMatcher0` can be used for all matchers which don't extract anything while `PathMatcher1[T]` defines a matcher which only extracts a single value of type `T`.

Here is an example of a more complex `PathMatcher` expression:

```
val matcher: PathMatcher1[Option[Int]] =
  "foo" / "bar" / "X" ~ IntNumber.? / ("edit" | "create")
```

This will match paths like `foo/bar/X42/edit` or `foo/bar/X/create`.

Note: The path matching DSL describes what paths to accept **after** URL decoding. This is why the path-separating slashes have special status and cannot simply be specified as part of a string! The string "foo/bar" would match the raw URI path "foo%2Fbar", which is most likely not what you want!

Basic PathMatchers

A complex `PathMatcher` can be constructed by combining or modifying more basic ones. Here are the basic matchers that Akka HTTP already provides for you:

`String` You can use a `String` instance as a `PathMatcher0`. Strings simply match themselves and extract no value. Note that strings are interpreted as the decoded representation of the path, so if they include a '/' character this character will match "%2F" in the encoded raw URI!

Regex You can use a `Regex` instance as a `PathMatcher1[String]`, which matches whatever the regex matches and extracts one `String` value. A `PathMatcher` created from a regular expression extracts either the complete match (if the regex doesn't contain a capture group) or the capture group (if the regex contains exactly one capture group). If the regex contains more than one capture group an `IllegalArgumentException` will be thrown.

Map[String, T] You can use a `Map[String, T]` instance as a `PathMatcher1[T]`, which matches any of the keys and extracts the respective map value for it.

Slash: PathMatcher0 Matches exactly one path-separating slash (`/`) character and extracts nothing.

Segment: PathMatcher1[String] Matches if the unmatched path starts with a path segment (i.e. not a slash). If so the path segment is extracted as a `String` instance.

PathEnd: PathMatcher0 Matches the very end of the path, similar to `$` in regular expressions and extracts nothing.

Rest: PathMatcher1[String] Matches and extracts the complete remaining unmatched part of the request's URI path as an (encoded!) String. If you need access to the remaining *decoded* elements of the path use `RestPath` instead.

RestPath: PathMatcher1[Path] Matches and extracts the complete remaining, unmatched part of the request's URI path.

IntNumber: PathMatcher1[Int] Efficiently matches a number of decimal digits and extracts their (non-negative) `Int` value. The matcher will not match zero digits or a sequence of digits that would represent an `Int` value larger than `Int.MaxValue`.

LongNumber: PathMatcher1[Long] Efficiently matches a number of decimal digits and extracts their (non-negative) `Long` value. The matcher will not match zero digits or a sequence of digits that would represent an `Long` value larger than `Long.MaxValue`.

HexIntNumber: PathMatcher1[Int] Efficiently matches a number of hex digits and extracts their (non-negative) `Int` value. The matcher will not match zero digits or a sequence of digits that would represent an `Int` value larger than `Int.MaxValue`.

HexLongNumber: PathMatcher1[Long] Efficiently matches a number of hex digits and extracts their (non-negative) `Long` value. The matcher will not match zero digits or a sequence of digits that would represent an `Long` value larger than `Long.MaxValue`.

DoubleNumber: PathMatcher1[Double] Matches and extracts a `Double` value. The matched string representation is the pure decimal, optionally signed form of a double value, i.e. without exponent.

JavaUUID: PathMatcher1[UUID] Matches and extracts a `java.util.UUID` instance.

Neutral: PathMatcher0 A matcher that always matches, doesn't consume anything and extracts nothing. Serves mainly as a neutral element in `PathMatcher` composition.

Segments: PathMatcher1[List[String]] Matches all remaining segments as a list of strings. Note that this can also be "no segments" resulting in the empty list. If the path has a trailing slash this slash will *not* be matched, i.e. remain unmatched and to be consumed by potentially nested directives.

separateOnSlashes(string: String): PathMatcher0 Converts a path string containing slashes into a `PathMatcher0` that interprets slashes as path segment separators. This means that a matcher matching "%2F" cannot be constructed with this helper.

provide[L: Tuple](extractions: L): PathMatcher[L] Always matches, consumes nothing and extracts the given `TupleX` of values.

PathMatcher[L: Tuple](prefix: Path, extractions: L): PathMatcher[L] Matches and consumes the given path prefix and extracts the given list of extractions. If the given prefix is empty the returned matcher matches always and consumes nothing.

Combinators

Path matchers can be combined with these combinators to form higher-level constructs:

Tilde Operator (~) The tilde is the most basic combinator. It simply concatenates two matchers into one, i.e if the first one matched (and consumed) the second one is tried. The extractions of both matchers are combined type-safely. For example: `"foo" ~ "bar"` yields a matcher that is identical to `"foobar"`.

Slash Operator (/) This operator concatenates two matchers and inserts a `Slash` matcher in between them. For example: `"foo" / "bar"` is identical to `"foo" ~ Slash ~ "bar"`.

Pipe Operator (|) This operator combines two matcher alternatives in that the second one is only tried if the first one did *not* match. The two sub-matchers must have compatible types. For example: `"foo" | "bar"` will match either "foo" *or* "bar".

Modifiers

Path matcher instances can be transformed with these modifier methods:

/ The slash operator cannot only be used as combinator for combining two matcher instances, it can also be used as a postfix call. `matcher /` is identical to `matcher ~ Slash` but shorter and easier to read.

? By postfixing a matcher with `?` you can turn any `PathMatcher` into one that always matches, optionally consumes and potentially extracts an `Option` of the underlying matchers extraction. The result type depends on the type of the underlying matcher:

If a `matcher` is of type	then `matcher.?` is of type
`PathMatcher0`	`PathMatcher0`
`PathMatcher1[T]`	`PathMatcher1[Option[T]`
`PathMatcher[L: Tuple]`	`PathMatcher[Option[L]]`

repeat(separator: PathMatcher0 = PathMatchers.Neutral) By postfixing a matcher with `repeat(separator)` you can turn any `PathMatcher` into one that always matches, consumes zero or more times (with the given separator) and potentially extracts a `List` of the underlying matcher's extractions. The result type depends on the type of the underlying matcher:

If a `matcher` is of type	then `matcher.repeat(...)` is of type
`PathMatcher0`	`PathMatcher0`
`PathMatcher1[T]`	`PathMatcher1[List[T]`
`PathMatcher[L: Tuple]`	`PathMatcher[List[L]]`

unary_! By prefixing a matcher with `!` it can be turned into a `PathMatcher0` that only matches if the underlying matcher does *not* match and vice versa.

transform / (h)flatMap / (h)map These modifiers allow you to append your own "post-application" logic to another matcher in order to form a custom one. You can map over the extraction(s), turn mismatches into matches or vice-versa or do anything else with the results of the underlying matcher. Take a look at the method signatures and implementations for more guidance as to how to use them.

Examples

```
// matches /foo/
path("foo" /)

// matches e.g. /foo/123 and extracts "123" as a String
path("foo" / """\d+""".r)

// matches e.g. /foo/bar123 and extracts "123" as a String
path("foo" / """bar(\d+)""".r)

// similar to `path(Segments)`
path(Segment.repeat(10, separator = Slash))

// matches e.g. /i42 or /hCAFE and extracts an Int
path("i" ~ IntNumber | "h" ~ HexIntNumber)
```

```
// identical to path("foo" ~ (PathEnd | Slash))
path("foo" ~ Slash.?)

// matches /red or /green or /blue and extracts 1, 2 or 3 respectively
path(Map("red" -> 1, "green" -> 2, "blue" -> 3))

// matches anything starting with "/foo" except for /foobar
pathPrefix("foo" ~ !"bar")
```

2.2.7 Marshalling

(todo)

2.2.8 Custom Directives

TODO

2.2.9 Predefined Directives (alphabetically)

Directive	Description
-cancelAllRejections-	Adds a `TransformationRejection` to rejections from its inner Route, which cancels ot
-cancelRejection-	Adds a `TransformationRejection` cancelling all rejections equal to a given one
-clientIP-	Extracts the IP address of the client from either the `X-Forwarded-For`, `Remote-Addres`
-complete-	Completes the request with a given response, several overloads
compressResponse	Compresses responses coming back from its inner Route using either `Gzip` or `Deflate` unle
compressResponseIfRequested	Compresses responses coming back from its inner Route using either `Gzip` or `Deflate`, but
-conditional-	Depending on the given ETag and Last-Modified values responds with `304 Not Modifie`
cookie	Extracts an `HttpCookie` with a given name or rejects if no such cookie is present in the requ
decodeRequest	Decompresses incoming requests using a given Decoder
decompressRequest	Decompresses incoming requests using either `Gzip`, `Deflate`, or `NoEncoding`
-delete-	Rejects all non-DELETE requests
deleteCookie	Adds a `Set-Cookie` header expiring the given cookie to all `HttpResponse` replies of its
encodeResponse	Compresses responses coming back from its inner Route using a given Encoder
-entity-	Unmarshalls the requests entity according to a given definition, rejects in case of problems
extract	Extracts a single value from the `RequestContext` using a function `RequestContext =`
-extractRequest-	Extracts the complete request
-failWith-	Bubbles the given error up the response chain, where it is dealt with by the closest *handleExce*
-formField-	Extracts the value of an HTTP form field, rejects if the request doesn't come with a field matc
-formFields-	Same as *-formField-*, except for several fields at once
-get-	Rejects all non-GET requests
getFromBrowseableDirectories	Same as *getFromBrowseableDirectory*, but allows for serving the "union" of several directorie
getFromBrowseableDirectory	Completes GET requests with the content of a file underneath a given directory, renders direct
getFromDirectory	Completes GET requests with the content of a file underneath a given directory
getFromFile	Completes GET requests with the content of a given file
getFromResource	Completes GET requests with the content of a given resource
getFromResourceDirectory	Same as *getFromDirectory* except that the file is not fetched from the file system but rather fro
handleExceptions	Converts exceptions thrown during evaluation of its inner Route into `HttpResponse` replies
handleRejections	Converts rejections produced by its inner Route into `HttpResponse` replies using a given R
-handleWith-	Completes the request using a given function. Uses the in-scope `Unmarshaller` and `Marsh`
-head-	Rejects all non-HEAD requests
-headerValue-	Extracts an HTTP header value using a given function, rejects if no value can be extracted
-headerValueByName-	Extracts an HTTP header value by selecting a header by name
-headerValueByType-	Extracts an HTTP header value by selecting a header by type
-headerValuePF-	Same as *-headerValue-*, but with a `PartialFunction`

Table 2.1 – continued from previous page

Directive	Description
-host-	Rejects all requests with a hostname different from a given definition, can extract the hostnam
-hostName-	Extracts the hostname part of the requests `Host` header value
listDirectoryContents	Completes GET requests with a unified listing of the contents of one or more given directories
logRequest	Produces a log entry for every incoming request
logRequestResult	Produces a log entry for every response or rejection coming back from its inner route, allowin
logResult	Produces a log entry for every response or rejection coming back from its inner route
mapResponse	Transforms the `HttpResponse` coming back from its inner Route
mapResponseEntity	Transforms the entity of the `HttpResponse` coming back from its inner Route
mapResponseHeaders	Transforms the headers of the `HttpResponse` coming back from its inner Route
mapInnerRoute	Transforms its inner Route with a `Route => Route` function
mapRejections	Transforms all rejections coming back from its inner Route
mapRequest	Transforms the incoming `HttpRequest`
mapRequestContext	Transforms the `RequestContext`
mapRouteResult	Transforms all responses coming back from its inner Route with a `Any => Any` function
-mapRouteResultPF-	Same as *mapRouteResult*, but with a `PartialFunction`
-method-	Rejects if the request method does not match a given one
-overrideMethodWithParameter-	Changes the HTTP method of the request to the value of the specified query string parameter
-onComplete-	"Unwraps" a `Future[T]` and runs its inner route after future completion with the future's va
-onFailure-	"Unwraps" a `Future[T]` and runs its inner route when the future has failed with the future's
-onSuccess-	"Unwraps" a `Future[T]` and runs its inner route after future completion with the future's va
optionalCookie	Extracts an `HttpCookie` with a given name, if the cookie is not present in the request extrac
-optionalHeaderValue-	Extracts an optional HTTP header value using a given function
-optionalHeaderValueByName-	Extracts an optional HTTP header value by selecting a header by name
-optionalHeaderValueByType-	Extracts an optional HTTP header value by selecting a header by type
-optionalHeaderValuePF-	Extracts an optional HTTP header value using a given partial function
-options-	Rejects all non-OPTIONS requests
-parameter-	Extracts the value of a request query parameter, rejects if the request doesn't come with a para
-parameterMap-	Extracts the requests query parameters as a `Map[String, String]`
-parameterMultiMap-	Extracts the requests query parameters as a `Map[String, List[String]]`
-parameters-	Same as *-parameter-*, except for several parameters at once
-parameterSeq-	Extracts the requests query parameters as a `Seq[(String, String)]`
pass	Does nothing, i.e. passes the `RequestContext` unchanged to its inner Route
-patch-	Rejects all non-PATCH requests
path	Extracts zero+ values from the `unmatchedPath` of the `RequestContext` according to a
pathEnd	Only passes on the request to its inner route if the request path has been matched completely,
pathEndOrSingleSlash	Only passes on the request to its inner route if the request path has been matched completely o
pathPrefix	Same as *path*, but also matches (and consumes) prefixes of the unmatched path (rather than on
pathPrefixTest	Like *pathPrefix* but without "consumption" of the matched path (prefix).
pathSingleSlash	Only passes on the request to its inner route if the request path consists of exactly one remaini
pathSuffix	Like as *pathPrefix*, but for suffixes rather than prefixed of the unmatched path
pathSuffixTest	Like *pathSuffix* but without "consumption" of the matched path (suffix).
-post-	Rejects all non-POST requests
-produce-	Uses the in-scope marshaller to extract a function that can be used for completing the request
provide	Injects a single value into a directive, which provides it as an extraction
-put-	Rejects all non-PUT requests
rawPathPrefix	Applies a given `PathMatcher` directly to the unmatched path of the `RequestContext`, i
rawPathPrefixTest	Checks whether the unmatchedPath of the `RequestContext` has a prefix matched by a `Pat`
-redirect-	Completes the request with redirection response of the given type to a given URI
-reject-	Rejects the request with a given set of rejections
-rejectEmptyResponse-	Converts responses with an empty entity into a rejection
requestEncodedWith	Rejects the request if its encoding doesn't match a given one
-requestEntityEmpty-	Rejects the request if its entity is not empty
-requestEntityPresent-	Rejects the request if its entity is empty
-requestUri-	Extracts the complete request URI
-respondWithHeader-	Adds a given response header to all `HttpResponse` replies from its inner Route

Table 2.1 – continued from previous page

Directive	Description
-respondWithHeaders-	Same as *-respondWithHeader-*, but for several headers at once
-respondWithMediaType-	Overrides the media-type of all `HttpResponse` replies from its inner Route, rejects if the m
-respondWithSingletonHeader-	Adds a given response header to all `HttpResponse` replies from its inner Route, if a header
-respondWithSingletonHeaders-	Same as *-respondWithSingletonHeader-*, but for several headers at once
-respondWithStatus-	Overrides the response status of all `HttpResponse` replies coming back from its inner Rout
responseEncodingAccepted	Rejects the request if the client doesn't accept a given encoding for the response
-mapUnmatchedPath-	Transforms the `unmatchedPath` of the `RequestContext` using a given function
-scheme-	Rejects a request if its Uri scheme does not match a given one
-schemeName-	Extracts the request Uri scheme
setCookie	Adds a `Set-Cookie` header to all `HttpResponse` replies of its inner Route
textract	Extracts a `TupleN` of values from the `RequestContext` using a function
-hprovide-	Injects a `TupleN` of values into a directive, which provides them as extractions
-unmatchedPath-	Extracts the unmatched path from the RequestContext
-validate-	Passes or rejects the request depending on evaluation of a given conditional expression
-withRangeSupport-	Transforms the response from its inner route into a `206 Partial Content` response if th

2.2.10 Predefined Directives (by trait)

All predefined directives are organized into traits that form one part of the overarching `Directives` trait.

Directives filtering or extracting from the request

MethodDirectives Filter and extract based on the request method.

HeaderDirectives Filter and extract based on request headers.

PathDirectives Filter and extract from the request URI path.

HostDirectives Filter and extract based on the target host.

ParameterDirectives*, *FormFieldDirectives Filter and extract based on query parameters or form fields.

CodingDirectives Filter and decode compressed request content.

MarshallingDirectives Extract the request entity.

SchemeDirectives Filter and extract based on the request scheme.

SecurityDirectives Handle authentication data from the request.

CookieDirectives Filter and extract cookies.

BasicDirectives* and *MiscDirectives Directives handling request properties.

Directives creating or transforming the response

CacheConditionDirectives Support for conditional requests (`304 Not Modified` responses).

ChunkingDirectives Automatically break a response into chunks.

CookieDirectives Set, modify, or delete cookies.

CodingDirectives Compress responses.

FileAndResourceDirectives Deliver responses from files and resources.

RangeDirectives Support for range requests (`206 Partial Content` responses).

RespondWithDirectives Change response properties.

RouteDirectives Complete or reject a request with a response.

BasicDirectives* and *MiscDirectives Directives handling or transforming response properties.

List of predefined directives by trait

BasicDirectives

Basic directives are building blocks for building *Custom Directives*. As such they usually aren't used in a route directly but rather in the definition of new directives.

Directives to provide values to inner routes These directives allow to provide the inner routes with extractions. They can be distinguished on two axes: a) provide a constant value or extract a value from the `RequestContext` b) provide a single value or an HList of values.

- *extract*
- *textract*
- *provide*
- *tprovide*

Directives transforming the request

- *mapRequestContext*
- *mapRequest*

Directives transforming the response These directives allow to hook into the response path and transform the complete response or the parts of a response or the list of rejections:

- *mapResponse*
- *mapResponseEntity*
- *mapResponseHeaders*
- *mapRejections*

Directives transforming the RouteResult These directives allow to transform the RouteResult of the inner route.

- *mapRouteResult*
- *mapRouteResponsePF*

Directives changing the execution of the inner route

- *mapInnerRoute*

Directives alphabetically

extract Calculates a value from the request context and provides the value to the inner route.

Signature

Description The `extract` directive is used as a building block for *Custom Directives* to extract data from the `RequestContext` and provide it to the inner route. It is a special case for extracting one value of the more general *textract* directive that can be used to extract more than one value.

See *Directives to provide values to inner routes* for an overview of similar directives.

Example

```
val uriLength = extract(_.request.uri.toString.length)
val route =
  uriLength { len =>
    complete(s"The length of the request URI is $len")
  }

Get("/abcdef") ~> route ~> check {
  responseAs[String] shouldEqual "The length of the request URI is 25"
}
```

mapInnerRoute Changes the execution model of the inner route by wrapping it with arbitrary logic.

Signature

Description The `mapInnerRoute` directive is used as a building block for *Custom Directives* to replace the inner route with any other route. Usually, the returned route wraps the original one with custom execution logic.

Example

```
val completeWithInnerException =
  mapInnerRoute { route =>
    ctx =>
      try {
        route(ctx)
      } catch {
        case NonFatal(e) => ctx.complete(s"Got ${e.getClass.getSimpleName} '${e.getMessage}'")
      }
  }

val route =
  completeWithInnerException {
    complete(throw new IllegalArgumentException("BLIP! BLOP! Everything broke"))
  }

Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "Got IllegalArgumentException 'BLIP! BLOP! Everything broke'"
}
```

mapRejections Transforms the list of rejections the inner route produced.

Signature

Description The `mapRejections` directive is used as a building block for *Custom Directives* to transform a list of rejections from the inner route to a new list of rejections.

See *Directives transforming the response* for similar directives.

Example

```
// ignore any rejections and replace them by AuthorizationFailedRejection
val replaceByAuthorizationFailed = mapRejections(_ => List(AuthorizationFailedRejection))
val route =
  replaceByAuthorizationFailed {
    path("abc")(complete("abc"))
  }
```

```
Get("/") ~> route ~> check {
  rejection shouldEqual AuthorizationFailedRejection
}
```

mapRequest Transforms the request before it is handled by the inner route.

Signature

Description The `mapRequest` directive is used as a building block for *Custom Directives* to transform a request before it is handled by the inner route. Changing the `request.uri` parameter has no effect on path matching in the inner route because the unmatched path is a separate field of the `RequestContext` value which is passed into routes. To change the unmatched path or other fields of the `RequestContext` use the *mapRequestContext* directive.

See *Directives transforming the request* for an overview of similar directives.

Example

```
def transformToPostRequest(req: HttpRequest): HttpRequest = req.copy(method = HttpMethods.POST)
val route =
  mapRequest(transformToPostRequest) {
    extractRequest { req =>
      complete(s"The request method was ${req.method.name}")
    }
  }

Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "The request method was POST"
}
```

mapRequestContext Transforms the `RequestContext` before it is passed to the inner route.

Signature

Description The `mapRequestContext` directive is used as a building block for *Custom Directives* to transform the request context before it is passed to the inner route. To change only the request value itself the *mapRequest* directive can be used instead.

See *Directives transforming the request* for an overview of similar directives.

Example

```
val replaceRequest =
  mapRequestContext(_.withRequest(HttpRequest(HttpMethods.POST)))

val route =
  replaceRequest {
    extractRequest { req =>
      complete(req.method.value)
    }
  }

Get("/abc/def/ghi") ~> route ~> check {
  responseAs[String] shouldEqual "POST"
}
```

mapResponse Changes the response that was generated by the inner route.

Signature

Description The `mapResponse` directive is used as a building block for *Custom Directives* to transform a response that was generated by the inner route. This directive transforms only complete responses. Use -*mapHttpResponsePart*-, instead, to transform parts of chunked responses as well.

See *Directives transforming the response* for similar directives.

Example

```
def overwriteResultStatus(response: HttpResponse): HttpResponse =
  response.copy(status = StatusCodes.BadGateway)
val route = mapResponse(overwriteResultStatus)(complete("abc"))

Get("/abcdef?ghi=12") ~> route ~> check {
  status shouldEqual StatusCodes.BadGateway
}
```

mapResponseEntity Changes the response entity that was generated by the inner route.

Signature

Description The `mapResponseEntity` directive is used as a building block for *Custom Directives* to transform a response entity that was generated by the inner route.

See *Directives transforming the response* for similar directives.

Example

```
def prefixEntity(entity: ResponseEntity): ResponseEntity = entity match {
  case HttpEntity.Strict(contentType, data) =>
    HttpEntity.Strict(contentType, ByteString("test") ++ data)
  case _ => throw new IllegalStateException("Unexpected entity type")
}

val prefixWithTest: Directive0 = mapResponseEntity(prefixEntity)
val route = prefixWithTest(complete("abc"))

Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "testabc"
}
```

mapResponseHeaders Changes the list of response headers that was generated by the inner route.

Signature

Description The `mapResponseHeaders` directive is used as a building block for *Custom Directives* to transform the list of response headers that was generated by the inner route.

See *Directives transforming the response* for similar directives.

Example

```
// adds all request headers to the response
val echoRequestHeaders = extract(_.request.headers).flatMap(respondWithHeaders)

val removeIdHeader = mapResponseHeaders(_.filterNot(_.lowercaseName == "id"))
val route =
  removeIdHeader {
    echoRequestHeaders {
      complete("test")
    }
  }

Get("/") ~> RawHeader("id", "12345") ~> RawHeader("id2", "67890") ~> route ~> check {
  header("id") shouldEqual None
  header("id2").get.value shouldEqual "67890"
}
```

mapRouteResult Changes the message the inner route sends to the responder.

Signature

Description The `mapRouteResult` directive is used as a building block for *Custom Directives* to transform what the inner route sends to the responder (see *The Responder Chain*).

See *Directives transforming the RouteResult* for similar directives.

Example

mapRouteResponsePF Changes the message the inner route sends to the responder.

Signature

Description The `mapRouteResponsePF` directive is used as a building block for *Custom Directives* to transform what the inner route sends to the responder (see *The Responder Chain*). It's similar to the *mapRouteResult* directive but allows to specify a partial function that doesn't have to handle all the incoming response messages.

See *Directives transforming the RouteResult* for similar directives.

Example

pass A directive that passes the request unchanged to its inner route.

Signature

Description The directive is usually used as a "neutral element" when combining directives generically.

Example

```
Get("/") ~> pass(complete("abc")) ~> check {
  responseAs[String] shouldEqual "abc"
}
```

provide Provides a constant value to the inner route.

Signature

Description The *provide* directive is used as a building block for *Custom Directives* to provide a single value to the inner route. To provide several values use the *tprovide* directive.

See *Directives to provide values to inner routes* for an overview of similar directives.

Example

```
def providePrefixedString(value: String): Directive1[String] = provide("prefix:" + value)
val route =
  providePrefixedString("test") { value =>
    complete(value)
  }
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "prefix:test"
}
```

textract Calculates a Tuple of values from the request context and provides them to the inner route.

Signature

Description The `textract` directive is used as a building block for *Custom Directives* to extract data from the `RequestContext` and provide it to the inner route. To extract just one value use the *extract* directive. To provide a constant value independent of the `RequestContext` use the *tprovide* directive instead.

See *Directives to provide values to inner routes* for an overview of similar directives.

Example

```
val pathAndQuery = textract { ctx =>
  val uri = ctx.request.uri
  (uri.path, uri.query)
}
val route =
  pathAndQuery { (p, query) =>
    complete(s"The path is $p and the query is $query")
  }

Get("/abcdef?ghi=12") ~> route ~> check {
  responseAs[String] shouldEqual "The path is /abcdef and the query is ghi=12"
}
```

tprovide Provides an HList of values to the inner route.

Signature

Description The `tprovide` directive is used as a building block for *Custom Directives* to provide data to the inner route. To provide just one value use the *provide* directive. If you want to provide values calculated from the `RequestContext` use the *textract* directive instead.

See *Directives to provide values to inner routes* for an overview of similar directives.

Example

```
def provideStringAndLength(value: String) = tprovide((value, value.length))
val route =
  provideStringAndLength("test") { (value, len) =>
    complete(s"Value is $value and its length is $len")
  }
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "Value is test and its length is 4"
}
```

CodingDirectives

compressResponse Uses the first of a given number of encodings that the client accepts. If none are accepted the request is rejected with an `UnacceptedResponseEncodingRejection`.

Signature

Description The `compressResponse` directive allows to specify zero to three encoders to try in the specified order. If none are specified the tried list is `Gzip`, `Deflate`, and then `NoEncoding`.

The `compressResponse()` directive (without an explicit list of encoders given) will therefore behave as follows:

`Accept-Encoding` header	resulting response
`Accept-Encoding: gzip`	compressed with `Gzip`
`Accept-Encoding: deflate`	compressed with `Deflate`
`Accept-Encoding: deflate, gzip`	compressed with `Gzip`
`Accept-Encoding: identity`	uncompressed
no `Accept-Encoding` header present	compressed with `Gzip`

For an overview of the different `compressResponse` directives see *When to use which compression directive?*.

Example This example shows the behavior of `compressResponse` without any encoders specified:

This example shows the behaviour of `compressResponse(Gzip)`:

compressResponseIfRequested Only compresses the response when specifically requested by the `Accept-Encoding` request header (i.e. the default is "no compression").

Signature

Description The `compressResponseIfRequested` directive is an alias for `compressResponse(NoEncoding, Gzip, Deflate)` and will behave as follows:

`Accept-Encoding` header	resulting response
`Accept-Encoding: gzip`	compressed with `Gzip`
`Accept-Encoding: deflate`	compressed with `Deflate`
`Accept-Encoding: deflate, gzip`	compressed with `Gzip`
`Accept-Encoding: identity`	uncompressed
no `Accept-Encoding` header present	uncompressed

For an overview of the different `compressResponse` directives see *When to use which compression directive?*.

Example

decodeRequest Tries to decode the request with the specified `Decoder` or rejects the request with an `UnacceptedRequestEncodingRejection(supportedEncoding)`.

Signature

Description The `decodeRequest` directive is the building block for the `decompressRequest` directive.

`decodeRequest` and `decompressRequest` are related like this:

```
decompressRequest(Gzip)          = decodeRequest(Gzip)
decompressRequest(a, b, c)       = decodeRequest(a) | decodeRequest(b) | decodeRequest(c)
decompressRequest()              = decodeRequest(Gzip) | decodeRequest(Deflate) | decodeRequest(No
```

Example

```
val route =
  decodeRequest {
    entity(as[String]) { content: String =>
      complete(s"Request content: '$content'")
    }
  }

Post("/", helloGzipped) ~> `Content-Encoding`(gzip) ~> route ~> check {
  responseAs[String] shouldEqual "Request content: 'Hello'"
}
Post("/", helloDeflated) ~> `Content-Encoding`(deflate) ~> route ~> check {
  responseAs[String] shouldEqual "Request content: 'Hello'"
}
Post("/", "hello uncompressed") ~> `Content-Encoding`(identity) ~> route ~> check {
  responseAs[String] shouldEqual "Request content: 'hello uncompressed'"
}
val route =
  decodeRequestWith(Gzip) {
    entity(as[String]) { content: String =>
      complete(s"Request content: '$content'")
    }
  }

Post("/", helloGzipped) ~> `Content-Encoding`(gzip) ~> route ~> check {
  responseAs[String] shouldEqual "Request content: 'Hello'"
}
Post("/", helloDeflated) ~> `Content-Encoding`(deflate) ~> route ~> check {
  rejection shouldEqual UnsupportedRequestEncodingRejection(gzip)
}
Post("/", "hello") ~> `Content-Encoding`(identity) ~> route ~> check {
  rejection shouldEqual UnsupportedRequestEncodingRejection(gzip)
}
val route =
  decodeRequestWith(Gzip, NoCoding) {
    entity(as[String]) { content: String =>
      complete(s"Request content: '$content'")
    }
  }

Post("/", helloGzipped) ~> `Content-Encoding`(gzip) ~> route ~> check {
  responseAs[String] shouldEqual "Request content: 'Hello'"
}
Post("/", helloDeflated) ~> `Content-Encoding`(deflate) ~> route ~> check {
  rejections shouldEqual List(UnsupportedRequestEncodingRejection(gzip), UnsupportedRequestEncodi
}
Post("/", "hello uncompressed") ~> `Content-Encoding`(identity) ~> route ~> check {
```

```
  responseAs[String] shouldEqual "Request content: 'hello uncompressed'"
}
```

decompressRequest Decompresses the request if it is can be decoded with one of the given decoders. Otherwise, the request is rejected with an `UnsupportedRequestEncodingRejection(supportedEncoding)`.

Signature

Description The `decompressRequest` directive allows either to specify a list of decoders or none at all. If no `Decoder` is specified `Gzip`, `Deflate`, or `NoEncoding` will be tried.

The `decompressRequest` directive will behave as follows:

`Content-Encoding` header	resulting request
`Content-Encoding: gzip`	decompressed
`Content-Encoding: deflate`	decompressed
`Content-Encoding: identity`	unchanged
no `Content-Encoding` header present	unchanged

For an overview of the different `decompressRequest` directives and which one to use when, see *When to use which decompression directive?*.

Example This example shows the behavior of `decompressRequest()` without any decoders specified:

This example shows the behaviour of `decompressRequest(Gzip, NoEncoding)`:

encodeResponse Tries to encode the response with the specified `Encoder` or rejects the request with an `UnacceptedResponseEncodingRejection(supportedEncodings)`.

Signature

Description The directive automatically applies the `autoChunkFileBytes` directive as well to avoid having to load an entire file into JVM heap.

The parameter to the directive is either just an `Encoder` or all of an `Encoder`, a threshold, and a chunk size to configure the automatically applied `autoChunkFileBytes` directive.

The `encodeResponse` directive is the building block for the `compressResponse` and `compressResponseIfRequested` directives.

`encodeResponse`, `compressResponse`, and `compressResponseIfRequested` are related like this:

```
compressResponse(Gzip)           = encodeResponse(Gzip)
compressResponse(a, b, c)        = encodeResponse(a) | encodeResponse(b) | encodeResponse(c)
compressResponse()               = encodeResponse(Gzip) | encodeResponse(Deflate) | encodeResponse
compressResponseIfRequested()    = encodeResponse(NoEncoding) | encodeResponse(Gzip) | encodeRespo
```

Example

```
val route = encodeResponse { complete("content") }

Get("/") ~> route ~> check {
  response should haveContentEncoding(identity)
}
Get("/") ~> `Accept-Encoding`(gzip, deflate) ~> route ~> check {
  response should haveContentEncoding(gzip)
```

```
}
Get("/") ~> `Accept-Encoding`(deflate) ~> route ~> check {
  response should haveContentEncoding(deflate)
}
Get("/") ~> `Accept-Encoding`(identity) ~> route ~> check {
  response should haveContentEncoding(identity)
}
val route = encodeResponseWith(Gzip) { complete("content") }

Get("/") ~> route ~> check {
  response should haveContentEncoding(gzip)
}
Get("/") ~> `Accept-Encoding`() ~> route ~> check {
  rejection shouldEqual UnacceptedResponseEncodingRejection(gzip)
}
Get("/") ~> `Accept-Encoding`(gzip, deflate) ~> route ~> check {
  response should haveContentEncoding(gzip)
}
Get("/") ~> `Accept-Encoding`(deflate) ~> route ~> check {
  rejection shouldEqual UnacceptedResponseEncodingRejection(gzip)
}
Get("/") ~> `Accept-Encoding`(identity) ~> route ~> check {
  rejection shouldEqual UnacceptedResponseEncodingRejection(gzip)
}
```

requestEncodedWith Passes the request to the inner route if the request is encoded with the argument encoding. Otherwise, rejects the request with an `UnacceptedRequestEncodingRejection(encoding)`.

Signature

Description This directive is the building block for `decodeRequest` to reject unsupported encodings.

responseEncodingAccepted Passes the request to the inner route if the request accepts the argument encoding. Otherwise, rejects the request with an `UnacceptedResponseEncodingRejection(encoding)`.

Signature

Description This directive is the building block for `encodeResponse` to reject unsupported encodings.

When to use which compression directive? There are three different directives for performing response compressing with slightly different behavior:

encodeResponse Always compresses the response with the one given encoding, rejects the request with an `UnacceptedResponseEncodingRejection` if the client doesn't accept the given encoding. The other compression directives are built upon this one. See its description for an overview how they relate exactly.

compressResponse Uses the first of a given number of encodings that the client accepts. If none are accepted the request is rejected.

compressResponseIfRequested Only compresses the response when specifically requested by the `Accept-Encoding` request header (i.e. the default is "no compression").

See the individual directives for more detailed usage examples.

When to use which decompression directive? There are two different directives for performing request decompressing with slightly different behavior:

decodeRequest Attempts to decompress the request using **the one given decoder**, rejects the request with an `UnsupportedRequestEncodingRejection` if the request is not encoded with the given encoder.

decompressRequest Decompresses the request if it is encoded with **one of the given encoders**. If the request's encoding doesn't match one of the given encoders it is rejected.

Combining compression and decompression As with all Spray directives, the above single directives can be combined using `&` to produce compound directives that will decompress requests and compress responses in whatever combination required. Some examples:

CookieDirectives

cookie Extracts a cookie with a given name from a request or otherwise rejects the request with a `MissingCookieRejection` if the cookie is missing.

Signature

Description Use the *optionalCookie* directive instead if you want to support missing cookies in your inner route.

Example

```
val route =
  cookie("userName") { nameCookie =>
    complete(s"The logged in user is '${nameCookie.content}'")
  }

Get("/") ~> Cookie(HttpCookie("userName", "paul")) ~> route ~> check {
  responseAs[String] shouldEqual "The logged in user is 'paul'"
}
// missing cookie
Get("/") ~> route ~> check {
  rejection shouldEqual MissingCookieRejection("userName")
}
Get("/") ~> Route.seal(route) ~> check {
  responseAs[String] shouldEqual "Request is missing required cookie 'userName'"
}
```

deleteCookie Adds a header to the response to request the removal of the cookie with the given name on the client.

Signature

Description Use the *setCookie* directive to update a cookie.

Example

```
val route =
  deleteCookie("userName") {
    complete("The user was logged out")
  }

Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "The user was logged out"
```

```
  header[`Set-Cookie`] shouldEqual Some(`Set-Cookie`(HttpCookie("userName", content = "deleted", e
}
```

optionalCookie Extracts an optional cookie with a given name from a request.

Signature

Description Use the *cookie* directive instead if the inner route does not handle a missing cookie.

Example

```
val route =
  optionalCookie("userName") {
    case Some(nameCookie) => complete(s"The logged in user is '${nameCookie.content}'")
    case None             => complete("No user logged in")
  }

Get("/") ~> Cookie(HttpCookie("userName", "paul")) ~> route ~> check {
  responseAs[String] shouldEqual "The logged in user is 'paul'"
}
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "No user logged in"
}
```

setCookie Adds a header to the response to request the update of the cookie with the given name on the client.

Signature

Description Use the *deleteCookie* directive to delete a cookie.

Example

```
val route =
  setCookie(HttpCookie("userName", content = "paul")) {
    complete("The user was logged in")
  }

Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "The user was logged in"
  header[`Set-Cookie`] shouldEqual Some(`Set-Cookie`(HttpCookie("userName", content = "paul")))
}
```

DebuggingDirectives

logRequest Logs the request.

Signature

```
def logRequest(marker: String)(implicit log: LoggingContext): Directive0
def logRequest(marker: String, level: LogLevel)(implicit log: LoggingContext): Directive0
def logRequest(show: HttpRequest => String)(implicit log: LoggingContext): Directive0
def logRequest(show: HttpRequest => LogEntry)(implicit log: LoggingContext): Directive0
def logRequest(magnet: LoggingMagnet[HttpRequest => Unit])(implicit log: LoggingContext): Directi
```

The signature shown is simplified, the real signature uses magnets. [1]

Description Logs the request using the supplied `LoggingMagnet[HttpRequest => Unit]`. This `LoggingMagnet` is a wrapped function `HttpRequest => Unit` that can be implicitly created from the different constructors shown above. These constructors build a `LoggingMagnet` from these components:

- A marker to prefix each log message with.
- A log level.
- A `show` function that calculates a string representation for a request.
- An implicit `LoggingContext` that is used to emit the log message.
- A function that creates a `LogEntry` which is a combination of the elements above.

It is also possible to use any other function `HttpRequest => Unit` for logging by wrapping it with `LoggingMagnet`. See the examples for ways to use the `logRequest` directive.

Use `logResult` for logging the response, or `logRequestResult` for logging both.

Example

```
// different possibilities of using logRequest

// The first alternatives use an implicitly available LoggingContext for logging
// marks with "get-user", log with debug level, HttpRequest.toString
DebuggingDirectives.logRequest("get-user")

// marks with "get-user", log with info level, HttpRequest.toString
DebuggingDirectives.logRequest("get-user", Logging.InfoLevel)

// logs just the request method at debug level
def requestMethod(req: HttpRequest): String = req.method.toString
DebuggingDirectives.logRequest(requestMethod _)

// logs just the request method at info level
def requestMethodAsInfo(req: HttpRequest): LogEntry = LogEntry(req.method.toString, Logging.InfoLe
DebuggingDirectives.logRequest(requestMethodAsInfo _)

// This one doesn't use the implicit LoggingContext but uses `println` for logging
def printRequestMethod(req: HttpRequest): Unit = println(req.method)
val logRequestPrintln = DebuggingDirectives.logRequest(LoggingMagnet(_ => printRequestMethod))

Get("/") ~> logRequestPrintln(complete("logged")) ~> check {
  responseAs[String] shouldEqual "logged"
}
```

logRequestResult Logs request and response.

Signature

```
def logRequestResult(marker: String)(implicit log: LoggingContext): Directive0
def logRequestResult(marker: String, level: LogLevel)(implicit log: LoggingContext): Directive0
def logRequestResult(show: HttpRequest => HttpResponsePart => Option[LogEntry])
                    (implicit log: LoggingContext): Directive0
def logRequestResult(show: HttpRequest => Any => Option[LogEntry])(implicit log: LoggingContext):
```

The signature shown is simplified, the real signature uses magnets. [2]

[1] See The Magnet Pattern for an explanation of magnet-based overloading.
[2] See The Magnet Pattern for an explanation of magnet-based overloading.

Description This directive is a combination of `logRequest` and `logResult`. See `logRequest` for the general description how these directives work.

Example

```
// different possibilities of using logRequestResponse

// The first alternatives use an implicitly available LoggingContext for logging
// marks with "get-user", log with debug level, HttpRequest.toString, HttpResponse.toString
DebuggingDirectives.logRequestResult("get-user")

// marks with "get-user", log with info level, HttpRequest.toString, HttpResponse.toString
DebuggingDirectives.logRequestResult("get-user", Logging.InfoLevel)

// logs just the request method and response status at info level
def requestMethodAndResponseStatusAsInfo(req: HttpRequest): Any => Option[LogEntry] = {
  case res: HttpResponse => Some(LogEntry(req.method + ":" + res.status, Logging.InfoLevel))
  case _                 => None // other kind of responses
}
DebuggingDirectives.logRequestResult(requestMethodAndResponseStatusAsInfo _)

// This one doesn't use the implicit LoggingContext but uses `println` for logging
def printRequestMethodAndResponseStatus(req: HttpRequest)(res: Any): Unit =
  println(requestMethodAndResponseStatusAsInfo(req)(res).map(_.obj.toString).getOrElse(""))
val logRequestResultPrintln = DebuggingDirectives.logRequestResult(LoggingMagnet(_ => printRequest

Get("/") ~> logRequestResultPrintln(complete("logged")) ~> check {
  responseAs[String] shouldEqual "logged"
}
```

logResult Logs the response.

Signature

```
def logResult(marker: String)(implicit log: LoggingContext): Directive0
def logResult(marker: String, level: LogLevel)(implicit log: LoggingContext): Directive0
def logResult(show: Any => String)(implicit log: LoggingContext): Directive0
def logResult(show: Any => LogEntry)(implicit log: LoggingContext): Directive0
def logResult(magnet: LoggingMagnet[Any => Unit])(implicit log: LoggingContext): Directive0
```

The signature shown is simplified, the real signature uses magnets. [3]

Description See `logRequest` for the general description how these directives work. This directive is different as it requires a `LoggingMagnet[Any => Unit]`. Instead of just logging `HttpResponses`, `logResult` is able to log anything passing through *The Responder Chain* (which can either be a `HttpResponsePart` or a `Rejected` message reporting rejections).

Use `logRequest` for logging the request, or `logRequestResult` for logging both.

Example

```
// different possibilities of using logResponse

// The first alternatives use an implicitly available LoggingContext for logging
// marks with "get-user", log with debug level, HttpResponse.toString
DebuggingDirectives.logResult("get-user")

// marks with "get-user", log with info level, HttpResponse.toString
DebuggingDirectives.logResult("get-user", Logging.InfoLevel)
```

[3] See The Magnet Pattern for an explanation of magnet-based overloading.

```
// logs just the response status at debug level
def responseStatus(res: Any): String = res match {
  case x: HttpResponse => x.status.toString
  case _               => "unknown response part"
}
DebuggingDirectives.logResult(responseStatus _)

// logs just the response status at info level
def responseStatusAsInfo(res: Any): LogEntry = LogEntry(responseStatus(res), Logging.InfoLevel)
DebuggingDirectives.logResult(responseStatusAsInfo _)

// This one doesn't use the implicit LoggingContext but uses `println` for logging
def printResponseStatus(res: Any): Unit = println(responseStatus(res))
val logResultPrintln = DebuggingDirectives.logResult(LoggingMagnet(_ => printResponseStatus))

Get("/") ~> logResultPrintln(complete("logged")) ~> check {
  responseAs[String] shouldEqual "logged"
}
```

ExecutionDirectives

handleExceptions Catches exceptions thrown by the inner route and handles them using the specified `ExceptionHandler`.

Signature

Description Using this directive is an alternative to using a global implicitly defined `ExceptionHandler` that applies to the complete route.

See *Exception Handling* for general information about options for handling exceptions.

Example

```
val divByZeroHandler = ExceptionHandler {
  case _: ArithmeticException => complete(StatusCodes.BadRequest, "You've got your arithmetic wro
}
val route =
  path("divide" / IntNumber / IntNumber) { (a, b) =>
    handleExceptions(divByZeroHandler) {
      complete(s"The result is ${a / b}")
    }
  }

Get("/divide/10/5") ~> route ~> check {
  responseAs[String] shouldEqual "The result is 2"
}
Get("/divide/10/0") ~> route ~> check {
  status shouldEqual StatusCodes.BadRequest
  responseAs[String] shouldEqual "You've got your arithmetic wrong, fool!"
}
```

handleRejections Handles rejections produced by the inner route and handles them using the specified `RejectionHandler`.

Signature

Description Using this directive is an alternative to using a global implicitly defined `RejectionHandler` that applies to the complete route.

See *Rejections* for general information about options for handling rejections.

Example

```
val totallyMissingHandler = RejectionHandler.newBuilder()
  .handleNotFound { complete(StatusCodes.NotFound, "Oh man, what you are looking for is long gone
  .result()
val route =
  pathPrefix("handled") {
    handleRejections(totallyMissingHandler) {
      path("existing")(complete("This path exists"))
    }
  }

Get("/handled/existing") ~> route ~> check {
  responseAs[String] shouldEqual "This path exists"
}
Get("/missing") ~> Route.seal(route) /* applies default handler */ ~> check {
  status shouldEqual StatusCodes.NotFound
  responseAs[String] shouldEqual "The requested resource could not be found."
}
Get("/handled/missing") ~> route ~> check {
  status shouldEqual StatusCodes.NotFound
  responseAs[String] shouldEqual "Oh man, what you are looking for is long gone."
}
```

FileAndResourceDirectives

Like the *RouteDirectives* the `FileAndResourceDirectives` are somewhat special in spray's routing DSL. Contrary to all other directives they do not produce instances of type `Directive[L <: HList]` but rather "plain" routes of type `Route`. The reason is that they are not meant for wrapping an inner route (like most other directives, as intermediate-level elements of a route structure, do) but rather form the actual route structure **leaves**.

So in most cases the inner-most element of a route structure branch is one of the *RouteDirectives* or `FileAndResourceDirectives`.

getFromBrowseableDirectories Serves the content of the given directories as a file system browser, i.e. files are sent and directories served as browsable listings.

Signature

Description The `getFromBrowseableDirectories` is a combination of serving files from the specified directories (like `getFromDirectory`) and listing a browseable directory with `listDirectoryContents`. Nesting this directive beneath `get` is not necessary as this directive will only respond to `GET` requests.

Use `getFromBrowseableDirectory` to serve only one directory. Use `getFromDirectory` if directory browsing isn't required.

getFromBrowseableDirectory The single-directory variant of *getFromBrowseableDirectories*.

Signature

getFromDirectory Completes GET requests with the content of a file underneath the given directory.

Signature

Description The `unmatchedPath` of the `RequestContext` is first transformed by the given `pathRewriter` function before being appended to the given directory name to build the final file name.

The actual I/O operation is running detached in a *Future*, so it doesn't block the current thread. If the file cannot be read the route rejects the request.

To serve a single file use `getFromFile`. To serve browsable directory listings use `getFromBrowseableDirectories`. To serve files from a classpath directory use `getFromResourceDirectory` instead.

Note that it's not required to wrap this directive with `get` as this directive will only respond to `GET` requests.

getFromFile Completes GET requests with the content of the given file.

Signature

Description The actual I/O operation is running detached in a *Future*, so it doesn't block the current thread (but potentially some other thread !). If the file cannot be found or read the request is rejected.

To serve files from a directory use `getFromDirectory`, instead. To serve a file from a classpath resource use `getFromResource` instead.

Note that it's not required to wrap this directive with `get` as this directive will only respond to `GET` requests.

getFromResource Completes GET requests with the content of the given classpath resource.

Signature

Description The actual I/O operation is running detached in a *Future*, so it doesn't block the current thread (but potentially some other thread !). If the file cannot be found or read the request is rejected.

To serve files from a classpath directory use `getFromResourceDirectory` instead. To serve files from a filesystem directory use `getFromDirectory`, instead.

Note that it's not required to wrap this directive with `get` as this directive will only respond to `GET` requests.

getFromResourceDirectory Completes GET requests with the content of the given classpath resource directory.

Signature

Description The actual I/O operation is running detached in a *Future*, so it doesn't block the current thread (but potentially some other thread !). If the file cannot be found or read the request is rejected.

To serve a single resource use `getFromResource`, instead. To server files from a filesystem directory use `getFromDirectory` instead.

Note that it's not required to wrap this directive with `get` as this directive will only respond to `GET` requests.

listDirectoryContents Completes GET requests with a unified listing of the contents of all given directories. The actual rendering of the directory contents is performed by the in-scope *Marshaller[DirectoryListing]*.

Signature

Description The `listDirectoryContents` directive renders a response only for directories. To just serve files use `getFromDirectory`. To serve files and provide a browseable directory listing use `getFromBrowsableDirectories` instead.

The rendering can be overridden by providing a custom `Marshaller[DirectoryListing]`.

Note that it's not required to wrap this directive with `get` as this directive will only respond to `GET` requests.

PathDirectives

path Matches the complete unmatched path of the `RequestContext` against the given `PathMatcher`, potentially extracts one or more values (depending on the type of the argument).

Signature

Description This directive filters incoming requests based on the part of their URI that hasn't been matched yet by other potentially existing *pathPrefix* directives on higher levels of the routing structure. Its one parameter is usually an expression evaluating to a `PathMatcher` instance (see also: *The PathMatcher DSL*).

As opposed to the *rawPathPrefix* or *rawPathPrefixTest* directives `path` automatically adds a leading slash to its `PathMatcher` argument, you therefore don't have to start your matching expression with an explicit slash.

The `path` directive attempts to match the **complete** remaining path, not just a prefix. If you only want to match a path prefix and then delegate further filtering to a lower level in your routing structure use the *pathPrefix* directive instead. As a consequence it doesn't make sense to nest a `path` or *pathPrefix* directive underneath another `path` directive, as there is no way that they will ever match (since the unmatched path underneath a `path` directive will always be empty).

Depending on the type of its `PathMatcher` argument the `path` directive extracts zero or more values from the URI. If the match fails the request is rejected with an *empty rejection set*.

Example

```
val route =
  path("foo") {
    complete("/foo")
  } ~
    path("foo" / "bar") {
      complete("/foo/bar")
    } ~
    pathPrefix("ball") {
      pathEnd {
        complete("/ball")
      } ~
        path(IntNumber) { int =>
          complete(if (int % 2 == 0) "even ball" else "odd ball")
        }
    }

Get("/") ~> route ~> check {
  handled shouldEqual false
}

Get("/foo") ~> route ~> check {
  responseAs[String] shouldEqual "/foo"
}

Get("/foo/bar") ~> route ~> check {
  responseAs[String] shouldEqual "/foo/bar"
}
```

```
Get("/ball/1337") ~> route ~> check {
  responseAs[String] shouldEqual "odd ball"
}
```

pathEnd Only passes the request to its inner route if the unmatched path of the `RequestContext` is empty, i.e. the request path has been fully matched by a higher-level *path* or *pathPrefix* directive.

Signature

Description This directive is a simple alias for `rawPathPrefix(PathEnd)` and is mostly used on an inner-level to discriminate "path already fully matched" from other alternatives (see the example below).

Example

```
val route =
  pathPrefix("foo") {
    pathEnd {
      complete("/foo")
    } ~
      path("bar") {
        complete("/foo/bar")
      }
  }

Get("/foo") ~> route ~> check {
  responseAs[String] shouldEqual "/foo"
}

Get("/foo/") ~> route ~> check {
  handled shouldEqual false
}

Get("/foo/bar") ~> route ~> check {
  responseAs[String] shouldEqual "/foo/bar"
}
```

pathEndOrSingleSlash Only passes the request to its inner route if the unmatched path of the `RequestContext` is either empty or contains only one single slash.

Signature

Description This directive is a simple alias for `rawPathPrefix(Slash.? ~ PathEnd)` and is mostly used on an inner-level to discriminate "path already fully matched" from other alternatives (see the example below).

It is equivalent to `pathEnd | pathSingleSlash` but slightly more efficient.

Example

```
val route =
  pathPrefix("foo") {
    pathEndOrSingleSlash {
      complete("/foo")
    } ~
      path("bar") {
        complete("/foo/bar")
```

```
      }
  }

Get("/foo") ~> route ~> check {
  responseAs[String] shouldEqual "/foo"
}

Get("/foo/") ~> route ~> check {
  responseAs[String] shouldEqual "/foo"
}

Get("/foo/bar") ~> route ~> check {
  responseAs[String] shouldEqual "/foo/bar"
}
```

pathPrefix Matches and consumes a prefix of the unmatched path of the `RequestContext` against the given `PathMatcher`, potentially extracts one or more values (depending on the type of the argument).

Signature

Description This directive filters incoming requests based on the part of their URI that hasn't been matched yet by other potentially existing `pathPrefix` or *rawPathPrefix* directives on higher levels of the routing structure. Its one parameter is usually an expression evaluating to a `PathMatcher` instance (see also: *The PathMatcher DSL*).

As opposed to its *rawPathPrefix* counterpart `pathPrefix` automatically adds a leading slash to its `PathMatcher` argument, you therefore don't have to start your matching expression with an explicit slash.

Depending on the type of its `PathMatcher` argument the `pathPrefix` directive extracts zero or more values from the URI. If the match fails the request is rejected with an *empty rejection set*.

Example

```
val route =
  pathPrefix("ball") {
    pathEnd {
      complete("/ball")
    } ~
      path(IntNumber) { int =>
        complete(if (int % 2 == 0) "even ball" else "odd ball")
      }
  }

Get("/") ~> route ~> check {
  handled shouldEqual false
}

Get("/ball") ~> route ~> check {
  responseAs[String] shouldEqual "/ball"
}

Get("/ball/1337") ~> route ~> check {
  responseAs[String] shouldEqual "odd ball"
}
```

pathPrefixTest Checks whether the unmatched path of the `RequestContext` has a prefix matched by the given `PathMatcher`. Potentially extracts one or more values (depending on the type of the argument) but doesn't consume its match from the unmatched path.

Signature

Description This directive is very similar to the *pathPrefix* directive with the one difference that the path prefix it matched (if it matched) is *not* consumed. The unmatched path of the `RequestContext` is therefore left as is even in the case that the directive successfully matched and the request is passed on to its inner route.

For more info on how to create a `PathMatcher` see *The PathMatcher DSL*.

As opposed to its *rawPathPrefixTest* counterpart `pathPrefixTest` automatically adds a leading slash to its `PathMatcher` argument, you therefore don't have to start your matching expression with an explicit slash.

Depending on the type of its `PathMatcher` argument the `pathPrefixTest` directive extracts zero or more values from the URI. If the match fails the request is rejected with an *empty rejection set*.

Example

```
val completeWithUnmatchedPath =
  extractUnmatchedPath { p =>
    complete(p.toString)
  }

val route =
  pathPrefixTest("foo" | "bar") {
    pathPrefix("foo") { completeWithUnmatchedPath } ~
      pathPrefix("bar") { completeWithUnmatchedPath }
  }

Get("/foo/doo") ~> route ~> check {
  responseAs[String] shouldEqual "/doo"
}

Get("/bar/yes") ~> route ~> check {
  responseAs[String] shouldEqual "/yes"
}
```

pathSingleSlash Only passes the request to its inner route if the unmatched path of the `RequestContext` contains exactly one single slash.

Signature

Description This directive is a simple alias for `pathPrefix(PathEnd)` and is mostly used for matching requests to the root URI (`/`) on an inner-level to discriminate "all path segments matched" from other alternatives (see the example below).

Example

```
val route =
  pathSingleSlash {
    complete("root")
  } ~
    pathPrefix("ball") {
      pathSingleSlash {
        complete("/ball/")
      } ~
        path(IntNumber) { int =>
          complete(if (int % 2 == 0) "even ball" else "odd ball")
        }
    }
```

```
Get("/") ~> route ~> check {
  responseAs[String] shouldEqual "root"
}

Get("/ball") ~> route ~> check {
  handled shouldEqual false
}

Get("/ball/") ~> route ~> check {
  responseAs[String] shouldEqual "/ball/"
}

Get("/ball/1337") ~> route ~> check {
  responseAs[String] shouldEqual "odd ball"
}
```

pathSuffix Matches and consumes a suffix of the unmatched path of the `RequestContext` against the given `PathMatcher`, potentially extracts one or more values (depending on the type of the argument).

Signature

Description This directive filters incoming requests based on the part of their URI that hasn't been matched yet by other potentially existing path matching directives on higher levels of the routing structure. Its one parameter is usually an expression evaluating to a `PathMatcher` instance (see also: *The PathMatcher DSL*).

As opposed to *pathPrefix* this directive matches and consumes the unmatched path from the right, i.e. the end.

> **Caution:** For efficiency reasons, the given `PathMatcher` must match the desired suffix in reversed-segment order, i.e. `pathSuffix("baz" / "bar")` would match `/foo/bar/baz`! The order within a segment match is not reversed.

Depending on the type of its `PathMatcher` argument the `pathPrefix` directive extracts zero or more values from the URI. If the match fails the request is rejected with an *empty rejection set*.

Example

```
val completeWithUnmatchedPath =
  extractUnmatchedPath { p =>
    complete(p.toString)
  }

val route =
  pathPrefix("start") {
    pathSuffix("end") {
      completeWithUnmatchedPath
    } ~
      pathSuffix("foo" / "bar" ~ "baz") {
        completeWithUnmatchedPath
      }
  }

Get("/start/middle/end") ~> route ~> check {
  responseAs[String] shouldEqual "/middle/"
}

Get("/start/something/barbaz/foo") ~> route ~> check {
  responseAs[String] shouldEqual "/something/"
}
```

pathSuffixTest Checks whether the unmatched path of the `RequestContext` has a suffix matched by the given `PathMatcher`. Potentially extracts one or more values (depending on the type of the argument) but doesn't consume its match from the unmatched path.

Signature

Description This directive is very similar to the *pathSuffix* directive with the one difference that the path suffix it matched (if it matched) is *not* consumed. The unmatched path of the `RequestContext` is therefore left as is even in the case that the directive successfully matched and the request is passed on to its inner route.

As opposed to *pathPrefixTest* this directive matches and consumes the unmatched path from the right, i.e. the end.

> **Caution:** For efficiency reasons, the given `PathMatcher` must match the desired suffix in reversed-segment order, i.e. `pathSuffixTest("baz" / "bar")` would match `/foo/bar/baz`! The order within a segment match is not reversed.

Depending on the type of its `PathMatcher` argument the `pathSuffixTest` directive extracts zero or more values from the URI. If the match fails the request is rejected with an *empty rejection set*.

Example

```
val completeWithUnmatchedPath =
  extractUnmatchedPath { p =>
    complete(p.toString)
  }

val route =
  pathSuffixTest(Slash) {
    complete("slashed")
  } ~
    complete("unslashed")

Get("/foo/") ~> route ~> check {
  responseAs[String] shouldEqual "slashed"
}
Get("/foo") ~> route ~> check {
  responseAs[String] shouldEqual "unslashed"
}
```

rawPathPrefix Matches and consumes a prefix of the unmatched path of the `RequestContext` against the given `PathMatcher`, potentially extracts one or more values (depending on the type of the argument).

Signature

Description This directive filters incoming requests based on the part of their URI that hasn't been matched yet by other potentially existing `rawPathPrefix` or *pathPrefix* directives on higher levels of the routing structure. Its one parameter is usually an expression evaluating to a `PathMatcher` instance (see also: *The PathMatcher DSL*).

As opposed to its *pathPrefix* counterpart `rawPathPrefix` does *not* automatically add a leading slash to its `PathMatcher` argument. Rather its `PathMatcher` argument is applied to the unmatched path as is.

Depending on the type of its `PathMatcher` argument the `rawPathPrefix` directive extracts zero or more values from the URI. If the match fails the request is rejected with an *empty rejection set*.

Example

```
val completeWithUnmatchedPath =
  extractUnmatchedPath { p =>
    complete(p.toString)
  }

val route =
  pathPrefix("foo") {
    rawPathPrefix("bar") { completeWithUnmatchedPath } ~
      rawPathPrefix("doo") { completeWithUnmatchedPath }
  }

Get("/foobar/baz") ~> route ~> check {
  responseAs[String] shouldEqual "/baz"
}

Get("/foodoo/baz") ~> route ~> check {
  responseAs[String] shouldEqual "/baz"
}
```

rawPathPrefixTest Checks whether the unmatched path of the `RequestContext` has a prefix matched by the given `PathMatcher`. Potentially extracts one or more values (depending on the type of the argument) but doesn't consume its match from the unmatched path.

Signature

Description This directive is very similar to the *pathPrefix* directive with the one difference that the path prefix it matched (if it matched) is *not* consumed. The unmatched path of the `RequestContext` is therefore left as is even in the case that the directive successfully matched and the request is passed on to its inner route.

For more info on how to create a `PathMatcher` see *The PathMatcher DSL*.

As opposed to its *pathPrefixTest* counterpart `rawPathPrefixTest` does *not* automatically add a leading slash to its `PathMatcher` argument. Rather its `PathMatcher` argument is applied to the unmatched path as is.

Depending on the type of its `PathMatcher` argument the `rawPathPrefixTest` directive extracts zero or more values from the URI. If the match fails the request is rejected with an *empty rejection set*.

Example

```
val completeWithUnmatchedPath =
  extractUnmatchedPath { p =>
    complete(p.toString)
  }

val route =
  pathPrefix("foo") {
    rawPathPrefixTest("bar") {
      completeWithUnmatchedPath
    }
  }

Get("/foobar") ~> route ~> check {
  responseAs[String] shouldEqual "bar"
}

Get("/foobaz") ~> route ~> check {
  handled shouldEqual false
}
```

The PathMatcher DSL For being able to work with the *PathDirectives* effectively you should have some understanding of the `PathMatcher` mini-DSL that Akka HTTP provides for elegantly defining URI matching behavior.

Overview When a request (or rather the respective `RequestContext` instance) enters the route structure it has an "unmatched path" that is identical to the `request.uri.path`. As it descends the routing tree and passes through one or more *pathPrefix*/*path* directives the "unmatched path" progressively gets "eaten into" from the left until, in most cases, it eventually has been consumed completely.

What exactly gets matched and consumed as well as extracted from the unmatched path in each directive is defined with the patch matching DSL, which is built around these types:

```
trait PathMatcher[L: Tuple]
type PathMatcher0 = PathMatcher[Unit]
type PathMatcher1[T] = PathMatcher[Tuple1[T]]
```

The number and types of the values extracted by a `PathMatcher` instance is represented by the `L` type parameter which needs to be one of Scala's TupleN types or Unit (which is designated by the `Tuple` context bound). The convenience alias `PathMatcher0` can be used for all matchers which don't extract anything while `PathMatcher1[T]` defines a matcher which only extracts a single value of type `T`.

Here is an example of a more complex `PathMatcher` expression:

```
val matcher: PathMatcher1[Option[Int]] =
  "foo" / "bar" / "X" ~ IntNumber.? / ("edit" | "create")
```

This will match paths like `foo/bar/X42/edit` or `foo/bar/X/create`.

Note: The path matching DSL describes what paths to accept **after** URL decoding. This is why the path-separating slashes have special status and cannot simply be specified as part of a string! The string "foo/bar" would match the raw URI path "foo%2Fbar", which is most likely not what you want!

Basic PathMatchers A complex `PathMatcher` can be constructed by combining or modifying more basic ones. Here are the basic matchers that Akka HTTP already provides for you:

`String` You can use a `String` instance as a `PathMatcher0`. Strings simply match themselves and extract no value. Note that strings are interpreted as the decoded representation of the path, so if they include a '/' character this character will match "%2F" in the encoded raw URI!

`Regex` You can use a `Regex` instance as a `PathMatcher1[String]`, which matches whatever the regex matches and extracts one `String` value. A `PathMatcher` created from a regular expression extracts either the complete match (if the regex doesn't contain a capture group) or the capture group (if the regex contains exactly one capture group). If the regex contains more than one capture group an `IllegalArgumentException` will be thrown.

`Map[String, T]` You can use a `Map[String, T]` instance as a `PathMatcher1[T]`, which matches any of the keys and extracts the respective map value for it.

`Slash: PathMatcher0` Matches exactly one path-separating slash (`/`) character and extracts nothing.

`Segment: PathMatcher1[String]` Matches if the unmatched path starts with a path segment (i.e. not a slash). If so the path segment is extracted as a `String` instance.

`PathEnd: PathMatcher0` Matches the very end of the path, similar to `$` in regular expressions and extracts nothing.

`Rest: PathMatcher1[String]` Matches and extracts the complete remaining unmatched part of the request's URI path as an (encoded!) String. If you need access to the remaining *decoded* elements of the path use `RestPath` instead.

`RestPath: PathMatcher1[Path]` Matches and extracts the complete remaining, unmatched part of the request's URI path.

`IntNumber: PathMatcher1[Int]` Efficiently matches a number of decimal digits and extracts their (non-negative) `Int` value. The matcher will not match zero digits or a sequence of digits that would represent an `Int` value larger than `Int.MaxValue`.

`LongNumber: PathMatcher1[Long]` Efficiently matches a number of decimal digits and extracts their (non-negative) `Long` value. The matcher will not match zero digits or a sequence of digits that would represent an `Long` value larger than `Long.MaxValue`.

`HexIntNumber: PathMatcher1[Int]` Efficiently matches a number of hex digits and extracts their (non-negative) `Int` value. The matcher will not match zero digits or a sequence of digits that would represent an `Int` value larger than `Int.MaxValue`.

`HexLongNumber: PathMatcher1[Long]` Efficiently matches a number of hex digits and extracts their (non-negative) `Long` value. The matcher will not match zero digits or a sequence of digits that would represent an `Long` value larger than `Long.MaxValue`.

`DoubleNumber: PathMatcher1[Double]` Matches and extracts a `Double` value. The matched string representation is the pure decimal, optionally signed form of a double value, i.e. without exponent.

`JavaUUID: PathMatcher1[UUID]` Matches and extracts a `java.util.UUID` instance.

`Neutral: PathMatcher0` A matcher that always matches, doesn't consume anything and extracts nothing. Serves mainly as a neutral element in `PathMatcher` composition.

`Segments: PathMatcher1[List[String]]` Matches all remaining segments as a list of strings. Note that this can also be "no segments" resulting in the empty list. If the path has a trailing slash this slash will *not* be matched, i.e. remain unmatched and to be consumed by potentially nested directives.

`separateOnSlashes(string: String): PathMatcher0` Converts a path string containing slashes into a `PathMatcher0` that interprets slashes as path segment separators. This means that a matcher matching "%2F" cannot be constructed with this helper.

`provide[L: Tuple](extractions: L): PathMatcher[L]` Always matches, consumes nothing and extracts the given `TupleX` of values.

`PathMatcher[L: Tuple](prefix: Path, extractions: L): PathMatcher[L]` Matches and consumes the given path prefix and extracts the given list of extractions. If the given prefix is empty the returned matcher matches always and consumes nothing.

Combinators Path matchers can be combined with these combinators to form higher-level constructs:

Tilde Operator (`~`) The tilde is the most basic combinator. It simply concatenates two matchers into one, i.e if the first one matched (and consumed) the second one is tried. The extractions of both matchers are combined type-safely. For example: `"foo" ~ "bar"` yields a matcher that is identical to `"foobar"`.

Slash Operator (`/`) This operator concatenates two matchers and inserts a `Slash` matcher in between them. For example: `"foo" / "bar"` is identical to `"foo" ~ Slash ~ "bar"`.

Pipe Operator (`|`) This operator combines two matcher alternatives in that the second one is only tried if the first one did *not* match. The two sub-matchers must have compatible types. For example: `"foo" | "bar"` will match either "foo" *or* "bar".

Modifiers Path matcher instances can be transformed with these modifier methods:

`/` The slash operator cannot only be used as combinator for combining two matcher instances, it can also be used as a postfix call. `matcher /` is identical to `matcher ~ Slash` but shorter and easier to read.

`?` By postfixing a matcher with `?` you can turn any `PathMatcher` into one that always matches, optionally consumes and potentially extracts an `Option` of the underlying matchers extraction. The result type depends on the type of the underlying matcher:

If a `matcher` is of type	then `matcher.?` is of type
`PathMatcher0`	`PathMatcher0`
`PathMatcher1[T]`	`PathMatcher1[Option[T]`
`PathMatcher[L: Tuple]`	`PathMatcher[Option[L]]`

`repeat(separator: PathMatcher0 = PathMatchers.Neutral)` By postfixing a matcher with `repeat(separator)` you can turn any `PathMatcher` into one that always matches, consumes zero or more times (with the given separator) and potentially extracts a `List` of the underlying matcher's extractions. The result type depends on the type of the underlying matcher:

If a `matcher` is of type	then `matcher.repeat(...)` is of type
`PathMatcher0`	`PathMatcher0`
`PathMatcher1[T]`	`PathMatcher1[List[T]`
`PathMatcher[L: Tuple]`	`PathMatcher[List[L]]`

`unary_!` By prefixing a matcher with `!` it can be turned into a `PathMatcher0` that only matches if the underlying matcher does *not* match and vice versa.

`transform/(h)flatMap/(h)map` These modifiers allow you to append your own "post-application" logic to another matcher in order to form a custom one. You can map over the extraction(s), turn mismatches into matches or vice-versa or do anything else with the results of the underlying matcher. Take a look at the method signatures and implementations for more guidance as to how to use them.

Examples

```
// matches /foo/
path("foo" /)

// matches e.g. /foo/123 and extracts "123" as a String
path("foo" / """\d+""".r)

// matches e.g. /foo/bar123 and extracts "123" as a String
path("foo" / """bar(\d+)""".r)

// similar to `path(Segments)`
path(Segment.repeat(10, separator = Slash))

// matches e.g. /i42 or /hCAFE and extracts an Int
path("i" ~ IntNumber | "h" ~ HexIntNumber)

// identical to path("foo" ~ (PathEnd | Slash))
path("foo" ~ Slash.?)

// matches /red or /green or /blue and extracts 1, 2 or 3 respectively
path(Map("red" -> 1, "green" -> 2, "blue" -> 3))

// matches anything starting with "/foo" except for /foobar
pathPrefix("foo" ~ !"bar")
```

2.3 HTTP TestKit

Note: This part of the documentation has not yet been written, please stay tuned for updates.